Orientation in Objects

) Modernes Webdesign mit CSS 3)

Version 2.0.0 (01.09.2017, Autor: Frank Bongers, Webdimensions.de)

Weinheimer Straße 68
68309 Mannheim
http://www.oio.de

Inhaltsverzeichnis

1. Die Rollen von HTML und CSS

Modernes Webdesign erfolgt auf der Grundlage von HTML-Dokumenten und CSS-Formatierung. Dabei sind **Struktur** (logischer und hierarchischer Aufbau der Information) und **Darstellung** (Präsentation der Information) streng voneinander getrennt.

Hierbei übernimmt **HTML5** die Aufgabe der Strukturierung. Inhalte sollten semantisch, d.h. ihrer Aufgabe im Dokument entsprechend benannt werden. Hierzu sind für die Textstruktur Elemente wie Überschrift `<h1>`, `<h2>`, etc. oder Textabsatz `<p>` einzusetzen. Für Auszeichnungen innerhalb von Fließtext sind semantische Inlineelemente wie `<strong>`, `<em>` zuständig. Logische Dokumentbereiche, die als Grundlage eines Seitenlayouts dienen, können durch Abschnittcontainer `<div>` mit sprechenden Identifiernamen (mit `id`-Attribut) gekennzeichnet werden.

Die Präsentationsaufgabe obliegt allein **CSS**. Sämtliche Elemente und Attribute aus XHTML, die als Präsentationsanweisungen dienen, haben in modernen Webseiten daher nichts mehr zu suchen. (dies betrifft Elemente zur Textformatierung, wie `<font>`, sowie Attribute zur Textausrichtung oder Definition von Rahmen, Farben und Hintergrundbildern, wie `align`, `border`, `bgcolor`, `background`).

2. Grundlagen zu CSS

Das Akronym CSS steht für „Cascading Style Sheets“. CSS ist ein **Textformat** (Typ: `"text/css"`).

Das Prinzip von CSS ist einfach:

An die Bestandteile eines HTML-Dokuments, also die Elementcontainer und ihre Inhalte werden Darstellungsanweisungen gebunden. Die Anweisungen betreffen Schriftart, und -größe, aber auch Farbe, Hintergrund oder Abmessungen eines Elements. Die „Bindung“ erfolgt über eine **Mustererkennung**, der im einfachsten Fall der **Bezeichner eines Elements** zugrundeliegt.

Die folgende Anweisung gilt für <p>-Elemente:

```
p { font-family:arial; }
```

Die Anweisung besagt, dass der Inhalt aller <p>-Container des Dokuments mit dem Schrifttyp **Arial** darzustellen ist.

In CSS-Stylesheets dürfen Kommentare gesetzt werden, die den *mehrzeiligen*, aus C bekannten Kommentaren entsprechen. Zeilenumbrüche im Kommentar sind daher möglich:

```
/* dies ist ein Kommentar */
```

✔ **Keine einzeiligen Kommentare!**
Einzeilige Kommentare existieren in CSS *nicht*. (Der verlockende Kommentar per Doppelslash analog zu JavaScript ist ein Syntaxfehler.)

Erklärende Kommentare können am Beginn eines eingebundenen Stylesheets verwendet werden, um dessen Einsatzgebiet zu nennen oder Copyrightvermerke unterzubringen, oder können in längeren Stylesheets als Gliederung oder Erklärung eingesetzt werden.

```
/* lokale Regeln: */
   p   {
      font-family:arial;
   }
```

Ebenso möglich ist das Auskommentieren kompletter Regeln

```
/* diese Regel ist auskommentiert:
   h1  {
     font-size:2em;
   }
*/
```

…oder einzelner Properties eines Regelsatzes:

```
   div {
         font-family:arial;
         font-size:0.8em;
     /*  font-weight: bold; */
         color: #ff0000;
  }
```

✔ Man stößt auf leicht auf das Problem, dass versehentlich eine (verbotene) Verschachtelung der mehrzeiligen Kommentarblöcke

auftritt, sobald komplette Regeln *und* Einzeleigenschaften auskommentiert werden.

Bei Properties verfährt man daher sinnvollerweise anders:

Um eine **Teilregel** vorübergehend zu deaktivieren, setzt man *vor* den Propertynamen (beispielsweise) einen Unterstrich. Dies nutzt den Umstand, dass „unbekannte“ Eigenschaften übersprungen werden (aus Gründen der Abwärtskompatibilität).

```
div { font-family:arial;
      font-size:0.8em;
     _font-weight: bold;   /* deaktiviert */
      color: #ff0000;
      }
```

2.1. Einbindung von CSS

Da CSS ebenso wie HTML jedoch ein reines ASCII-Textformat ist, kann man sehr wohl CSS und HTML in ein und derselben Datei unterbringen. Hierfür muss der CSS-Code jedoch in einem HTML-Container untergebracht werden, innerhalb dessen der Browser die Anweisungen auch interpretiert. Diesen Zweck erfüllt der `<style>`-Container, der stets im `<head>`-Container der HTML-Seite untergebracht wird. Dort befindliche CSS-Anweisungen werden als **Dokumentstyles** bezeichnet.

Alternativ darf ein einzelnes HTML-Element auch unmittelbar CSS-Anweisungen zugewiesen bekommen, die dann in dessen `style`-Attribut unterzubringen sind. Da die Anweisungen quasi „vor Ort“ stehen, bezeichnet man sie als **Inlinestyles**.

Problemlos unterbringen kann man CSS-Anweisungen in einer externen Datei. Um diese mit der HTML-Seite in Verbindung zu bringen, wird ausgehend vom HTML-Dokument auf die externe CSS-Datei verwiesen. Dies geschieht mit Hilfe des `<link>`-Elements. Solche CSS-Anweisungen werden, weil sie außerhalb des HTML-Dokuments liegen, als **externe Styles** bezeichnet. Anstelle einer Verlinkung existieren Mechanismen, um CSS-Anweisungen aus einer externen CSS-Datei (virtuell) in ein anderes Dokument zu kopieren – dies wird in diesem Zusammenhang als „Import“ bezeichnet. Solche CSS-Anweisungen werden als **importierte Styles** bezeichnet.

Was ist ein CSS-Stylesheet?

Wie Sie eben erfahren haben, können CSS-Anweisungen zwar an verschiedenen Stellen stehen, wirken aber dennoch auf ein und dasselbe Dokument. Unter einem „Stylesheet“ versteht man die **Summe aller** auf ein Dokument wirkenden **CSS-Regeln**, wobei eventuelle Widersprüche zwischen einzelnen Befehlen bereinigt sind. Beteiligte Regeln können sowohl im Dokument selbst stehen, oder aus externen Quellen stammen.

✔ Das „Stylesheet“ ist also im Grunde ein „virtuelles Konstrukt“ im Arbeitsspeicher des Browsers.

Styleanweisungen können zu einem HTML-Element, abhängig vom Ort ihrer Deklaration, in den vier, eben vorgestellten Varianten in Beziehung stehen, die im Folgenden eingehender diskutiert werden. Die Anweisungen werden, je nach Art der Einbindung, hierarchisch unterschiedlich gewichtet:

als Inlinestyle

als Dokumentstyle

als externes importiertes Stylesheet

als externes verlinktes Stylesheet

2.1.1. Inlinestyle

Die Styleanweisung wird mittels des `style`-Attributs unmittelbar im Starttag des zu stylenden Elements eingefügt.

```
<p style="font-family:arial">
   Dies wird in Arial dargestellt.</p>
```

Der Inlinestyle gilt lokal ausschließlich für den Inhalt des Elements, in dem er definiert ist. Er kann verwendet werden, um gezielt einen globaler gesetzten Style zu widerrufen oder zu modifizieren. Den gleichen Effekt könnte man (einmalig) mittels eines ID-Selektors oder (wiederholbar) mit einem Klassenselektor erzielen.

✔ Es wird von der Verwendung des `style`-Attributs abgeraten, da dies die Trennung von Struktur (HTML) und Präsentation (CSS) unterläuft.

2.1.2. Dokumentstyle

Ein Dokumentstyle wird innerhalb des `<style>`-Containers im Dokumentkopf definiert und gilt global für das gesamte Dokument. Die Trennung zwischen Inhalt und Präsentation ist besser als beim Inlinestyle, die Möglichkeit der unmittelbaren Mehrfachverwendung von Styles für eine größere Zahl von Dokumenten jedoch nicht gegeben.

```
<style type="text/css">
        p {font-family:arial}
</style>
```

In **HTML5** ist der Einsatz des `<style>`-Containers gegenüber HTML 4 vereinfacht worden. Das bislang obligatorische `type`-Attribut kann entfallen. Dies reduziert das voranstehende Beispiel auf:

```
<style>
        p {font-family:arial}
</style>
```

2.1.3. Externes importiertes Stylesheet

Anstatt eine Stylesheet-Datei über einen Link mit dem Dokument zu verknüpfen, kann diese (wirkungsgleich) auch in den Dokumentstyle importiert werden.

Der `@import`-Anweisung wird ein URL übergeben, der zum Basis-URL der aufrufenden Datei aufgelöst wird. Die Syntax lautet

```
@import url("mein_style.css");
```

Besitzt das aufrufende Stylesheet den Basis-URL

```
"http://www.beispiel.de/styles/"
```

… so wird der URL in diesem Fall aufgelöst zu

```
"http://www.beispiel.de/styles/mein_style.css"
```

Eingesetzt wird die `@import-Anweisung` in der Regel im `<style>`-Container. Dies gibt die Möglichkeit, dort auch weitere lokale Styles zu definieren. In diesem Fall **muss** die `@import`-Anweisung **zuerst** stehen.

```
<style type="text/css">
   @import url("mein_style.css");
```

```
   p {font-family:arial;}
</style>
```

Ebenso ist es möglich, mehrere Stylesheet *nacheinander* zu importieren:

```
<style type="text/css">
   @import url("mein_style.css");
   @import url("mein_anderer_style.css");
</style>
```

2.1.4. Externes verlinktes Stylesheet

Für die Verknüpfung eines Stylesheets mit einem Dokument wird das `<link>`-Element verwendet (im Dokumentkopf des HTML-Dokuments):

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

Das `type`-Attribut kann in HTML5 weggelassen werden. Der Browser kann aus dem `rel`-Attribut die Art der Datei bereits erschließen:

```
<link rel="stylesheet" href="file.css">
```

Import von Styleanweisungen in Stylesheetdateien

Die `@import`-Anweisung kann auch direkt in einer Stylesheet-Datei auftreten – in diesem Fall **muss** sie ebenfalls **vor** den restlichen Styleregeln gesetzt werden:

```
/* Dateiname: screen.css */
   @import url("globale_regeln.css");

/*lokale Regeln: */
   p {font-family:arial;}
```

2.1.5. Kaskade, Präferenz und Spezifizität

Im Rahmen des Gesamtstylesheets, das auf das Dokument angewendet wird, **addieren** sich die CSS-Regeln aller Quellen.

- ✔ Kaskade: Sofern es bei den Eigenschaftszuweisungen keine Konflikte gibt, wird auf einen selektierten Elementcontainer die Summe *aller* für ihn in Frage kommenden Regeln angewendet:

So erhält ein Element mit Klassenattribut zunächst alle Präsentationsvorschriften, die auf seinen Element*typ* zutreffen und *anschließend* auch jene, die sich aus der Bindung an eine Klasse ergeben.

2.1.6. Ermittlung der Präferenz in der Präferenzkaskade

Wenn es daran geht, festzustellen, welche Regel bzw. welche *Eigenschaften* welcher Regeln auf ein Element anzuwenden sind, wird folgendes Prinzip angewendet:

Spezifizität: Bei Regeln aus gleichrangigen Quellen gilt die *passgenauere*, das heißt, diejenige mit dem spezifischeren Selektor

Reihenfolge: Stammen zwei Regeln *gleicher* Passgenauigkeit aus *gleichrangigen* Quellen, so gilt die *zuletzt stehende* (d.h. auch die zuletzt importierte, oder diejenige aus der zuletzt verlinkten Quelle)

Medientyp: Ist für eine Regel ein Medientyp angegeben, so wird sie nur mit Regeln *gleichen Medientyps* verglichen und mit diesen angewendet.

3. Selektoren

Selektoren dienen dazu, eine CSS-Regel an ein Element oder eine Gruppe von Elementen zu binden. Sie stehen daher im Zentrum der Funktionalität von Stylesheets.

✔ Man unterscheidet **einfache** und **zusammengesetzte** Selektoren, sowie **Kontext**- und **Pseudo**-Selektoren.

3.1. Einfache Selektoren

Ein „einfacher“ Selektor besteht aus einem einzelnen Token, das den Selektor definiert.

3.1.1. Universeller Selektor

Anwendbar auf: alle Elemente. Der universelle Selektor * erfasst Elemente beliebigen Namens. Sein Einsatz macht hauptsächlich innerhalb von zusammengesetzten Selektoren oder Kontextselektoren Sinn. Die folgende Regel ordnet für alle Elemente die Schriftart Arial an:

```
* {font-family:arial;}
```

3.1.2. Typ-Selektor

Anwendbar auf: alle Elemente. Der Typ-Selektor (Element-Selektor) zielt global im gesamten Dokument auf Elemente des genannten Bezeichners. Die folgende Stylesheetregel betrifft alle <p>-Container eines Dokuments:

```
p   {font-family:arial; font-size:0.8em; }
```

3.1.3. Klassenselektor

Anwendbar auf: alle Elemente mit `class`-Attribut. Definiert eine CSS-Klasse, die einem Element über das Attribut `class` zugewiesen werden kann. Beachten Sie den Punkt, der dem Klassennamen vorangestellt wird:

```
.wichtig {border:solid 2px red;}
```

Freie Klassen

Eine Klasse ohne vorangestellten Typ-Selektor wirkt auf jedes Element mit entsprechendem `class`-Attributwert.

✔ Einer solchen „freien“ Klasse ist also „quasi“ der universelle Selektor vorangestellt, der in diesem Zusammenhang jedoch entfallen kann:

```
*.wichtig
```

Beide folgenden Elemente werden erfasst:

```
<p class="wichtig">Dies hier ist wichtig!</p>

<div class="wichtig">Dies hier ist auch wichtig.</div>
```

Gebundene Klassen

Wird dem Klassenselektor *unmittelbar* ein Typ-Selektor vorangestellt, so wird die Wirkung der Klasse auf Elemente dieses Typs begrenzt.

```
p.wichtig {border:solid 2px red;}
```

Im folgenden Beispiel wird also *nur* der Inhalt des <p>-Containers, *nicht* aber der des -Containers erfasst:

```
<p class="wichtig">Dies hier ist wichtig!</p>

<div class="wichtig">Dies hier wird nicht
gestylt.</div>
```

Zuweisung mehrerer Klassen an einen Elementcontainer

Einem Element können *mehrere Klassen* zugewiesen werden, die *gleichzeitig* angewendet werden. Hierfür wird im `class`-Attribut eine durch Leerzeichen getrennte **Liste** mit Klassenbezeichnern angegeben:

```
<p class="hinweis wichtig">
 Dies ist ein wichtiger Hinweis.</p>
```

✔ Bei Konflikten zwischen den Eigenschaften der anzuwendenden Klassen zählt nicht die *Reihenfolge* ihrer Nennung im `class`-Attribut, sondern die der Deklaration im Stylesheet – Vorrang hat der Propertywert aus der *zuletzt* definierten Klasse.

Wahl des Bezeichners für Stylesheetklassen

Die im Prinzip frei wählbaren Klassennamen dürfen beliebig Buchstaben (a-z, A-Z), Ziffern (0-9) und dem Bindestrich (-) bestehen. Sie müssen mit einem Buchstaben beginnen, Sonderzeichen (vorsichtshalber sollte man in diesem Zusammenhang auch Umlaute dazu zählen) sind verboten.

✔ Der Unterstrich _ ist als Teil eines Bezeichners durch die CSS-Spezifikation zwar erlaubt, Klassenbezeichner mit Unterstrich werden jedoch in älteren Browsern nicht allgemein unterstützt und sollten vermieden werden.

3.1.4. ID-Selektor

Anwendbar auf: alle Elemente mit `id`-Attribut. Der ID-Selektor wirkt auf genau ein Element, das ein `id`-Attribut mit dem übergebenen Bezeichner besitzt.

✔ Der ID-Selektor wird in der Regel verwendet, um *Dokumentbereiche* oder logische Elemente zu stylen, die sich im Dokument *nicht* wiederholen (andernfalls greift man auf Klassen zurück).

Als Selektor wird dem ID-Bezeichner einfach die Raute # vorangestellt:

```
#einleitung { }
```

erfasst ein beliebiges Element, das den `id="einleitung"` besitzt.

Es ist zulässig, zur **Verdeutlichung** den Elementnamen des Containers, das den ID trägt, mit in den Selektor zu nehmen:

```
p#einleitung { }
```

erfasst einen <p>-Container, der den `id="einleitung"` besitzt. Da der Identifier eindeutig ist, ist dies prinzipiell nicht erforderlich, kann aber die Lesbarkeit des Stylesheets erhöhen. Die Spezifizität des Selektors erhöht sich ebenfalls geringfügig (aufgrund des Typselektors).

3.2. Pseudoklassen

Pseudoklassen zielen wie gewöhnliche Selektoren auf Elementcontainer, wählen ihr Zielelement jedoch anhand des aktuellen *Zustandes des Elements* aus. In CSS3 kommen als Gruppe die „strukturellen“ Pseudoklassen hinzu, die die Position des Elements im DOM auswerten.

Die Schreibweise einer Pseudoklasse ist diese:

```
:selektorname
```

… wobei in der Regel ein Typselektor vorangestellt wird:

```
typname:selektorname
```

Mehrere Pseudoklassen können kombiniert werden, wobei die Reihenfolge der Selektoren irrelevant ist. Das Zielelement muss allerdings *beide* Zustände *gleichzeitig* aufweisen:

```
typname:selektor1:selektor2
```

3.2.1. Pseudoklassen für Useraktionen

Diese Gruppe der Pseudoklassen reagiert auf **Zustandsänderungen** eines Elements, beispielsweise, ob es den Hover-Zustand annimmt oder verliert.

Name	**Beschreibung**
:hover	Die Maus hovert über dem Elementbereich
:target	Das Element ist Ziel eines Fragment-Links
:focus	Der User hat den Fokus auf das Element gesetzt
:active	Der User hat das Element (durch Klick o.ä.) aktiviert

:target

Die `:target`-Pseudoklasse (neu in CSS3) erfasst ein Element, das aktuell Ziel eines Links ist, wobei es sich um einen, auf einen ID zielenden Fragmentidentifier handeln wird. Folgende Regel gilt für ein Element #beispiel, auf das der aktuelle URL verweist:

```
#beispiel:target {
     border:3px solid red;
}
```

Die Target-Pseudoklasse wird aktiv, sobald das Target als Fragment-Identifier im URL des aktuellen Dokuments erscheint (also ein Link aktiviert wurde, der auf das Zielelement zeigt). Etwa so:

```
http://www.meinefirma.de/docs/test.html#beispiel
```

✔ Die **Möglichkeiten**, die sich ergeben, sind vielfältig. Seitenelemente können beispielsweise hervorgehoben oder eingeblendet werden, indem Links auf der Seite als Interface eingesetzt werden.

3.2.2. Pseudoklassen für Links

Noch aus CSS1 stammen die **Pseudoklassen für Links**, mit denen sich besuchte und nicht besuchte Links unterscheiden lassen.

Name	Beschreibung
:link	Alle noch nicht verfolgten Links
:visited	Alle Links, die auf eine besuchte Ressource zeigen

3.2.3. Pseudoklassen für Userinterfaces (Formulare)

Neu in CSS3 sind bestimmte, speziell für **Formularelemente** geeignete Pseudoklassen, die beispielsweise den deaktivierten Zustand eines Eingabefeldes erfassen können.

Name	Beschreibung
:enabled	Formularfeld ist aufnahmefähig geschaltet worden
:disabled	Formularfeld ist deaktiviert
:checked	Checkbox/Radiobutton wurde ausgewählt
:indeterminate	Zustand des Inputs ist undefiniert

3.2.4. Strukturelle Pseudoklassen (CSS3)

Die **strukturellen Pseudoklassen** erlauben eine Selektion von Knoten anhand ihrer Position im Dokument – meist in Relation zum Elternknoten (beispielsweise soll ein Knoten eines Typs gewählt werden, wenn er das „dritte Kindelement“ seines Elternknotens ist).

Name	Beschreibung
:nth-child()	Element ist n-ter Kindknoten
:nth-last-child()	Element ist n-ter Kindknoten, von hinten gezählt
:nth-of-type()	Element ist n-ter eines Typs
:nth-last-of-type()	Element ist n-ter eines Typs, von hinten gezählt
:first-child	Element ist erster Kindknoten
:last-child	Element ist letzter Kindknoten
:first-of-type	Element ist erstes eines Typs
:last-of-type	Element ist letztes eines Typs
:only-child	Element ist einziger Kindknoten

:only-of-type	Element ist einziges eines Typs
:empty	Element ist leer
:contains("text")	Textinhalt enthält Substring “text”

:nth-child()

```
:nth-child(a*n + b)
```

…wobei a, b Ganzzahlen sein müssen. Sowohl positive als auch negative Werte oder die 0 ist erlaubt (wenn auch nicht immer sinnvoll). So wird diese Pseudoklasse eingesetzt:

```
:nth-child(2*n)
```

…für geradzahlige Children (passt auf 0,2,4,6...)

```
:nth-child(2*n + 1)
```

…für ungeradzahlige Children (passt auf 1,3,5...). Ist a gleich 0, so wird direkt ein Child benannt:

```
:nth-child(3) // dritter Childknoten
```

:nth-last-child()

…wie zuvor, allerdings wird von hinten gezählt

:first-child

entspricht :nth-child(1)

:last-child

entspricht :nth-last-child(1)

:only-child

entspricht :first-child:last-child

:only-of-type

entspricht :first-of-type:last-of-type

:empty

Ein Element, das keine Kindknoten besitzt (darunter fällt auch Text).

```
<p></p>      ==  p:empty

<p>xx</p>    !=  p:empty
```

:contains("textbeispiel")

bezieht sich auf den geplätteten Inhalt, berücksichtigt also keine Tags! Selektiert wird der Container, nicht die Fundstelle.

3.2.5. Die Negations-Pseudoklasse :not() (CSS3)

Die Negation nimmt einen Selektor entgegen und passt auf Elemente, die dem übergebenen Selektor nicht entsprechen.

Name	Beschreibung
:not(sel)	Negiert den übergebenen Selektor *sel*

:not()

```
*:not(p)
```

… alle Elementknoten des Dokuments mit Ausnahme der P-Container.

```
p:not(.einleitung)
```

… alle P-Container, die nicht die Klasse `.einleitung` besitzen

3.3. Attributbezogene Selektoren

Auch die sogenannten **Attributselektoren** wählen Elementcontainer innerhalb des Dokuments aus, prüfen dabei jedoch deren *Attributkontext*.

✔ Attributelektoren zählen, auch wenn sie mit einem Typselektor kombiniert sind, wie alle anderen bislang behandelten Selektoren, zu den einfachen Selektoren.

Gewählt werden Elemente, die entweder ein bestimmtes Attribut besitzen, ohne dass dessen Wert eine Rolle spielt, oder Elemente, deren genanntes Attribut gleichzeitig einen bestimmten Wert besitzt.

E[att]	Attributselektor („E besitzt Attribut att“)
E[all=wert]	Attributselektor („E besitzt Attribut att“ mit Wert „wert“)
E[att~=val]	Attributselektor („E besitzt Attribut att als Listentyp“, das den Teilwert „wert“ enthält)
E[att|=val]	Attributselektor („E besitzt Attribut att“, dessen Wert mit „wert-“ beginnt); speziell für Sprachbezeichner

Attributselektoren in CSS3 (Teil 1)

Test auf Existenz eines Attributs

Anwendbar auf: alle Elemente mit entsprechendem Attribut

```
[attributname]
```

Anm.: eigentlich als Abkürzung für `*[attributname]`

✔ Ein „freier“ Attributselektor gilt für *alle* Elemente, für die ein Attribut mit entsprechendem Namen **explizit** gesetzt ist (d.h. im Quelltext existiert).

Die Wirkung kann durch Kombination mit einem Typselektor auf Elemente mit bestimmtem Namen beschränkt werden:

```
tagname[attributname]
```

Dieser Selektor erfasst alle Elemente vom Typ `tagname`, die über ein Attribut `attributname` egal welchen Wertes verfügen.

Beispiel:

Ein Link *mit Accesskey* kann hervorgehoben werden:

```
a[accesskey] { background-color:#ff9999; }
```

Test auf konkreten Attributwert

Der Attributwert kann dabei ebenfalls berücksichtigt werden

```
[attributname="attributwert"]
```

Dies gilt für alle Elemente, die ein Attribut `attributname` besitzen, wenn dieses den Wert `attributwert` enthält. Der Wert wird als Zeichenkette mit Begrenern übergeben und darf daher Leerzeichen enthalten.

Beispiel:

Diese Anweisung hebt Bilder mit leerem `alt`-Attribut hervor:

```
img[alt=""] {border:solid 2px red;}
```

Substring-matching Attribut-Selektoren

Die folgenden Attribut-Selektoren berücksichtigen **Teilstrings** (nicht „Teilwerte“ eines Listentypes!) eines Attributwertes.

E[att^=val]	Attributselektor („E besitzt Attribut att“, dessen Wert mit dem Teilwert „wert“ beginnt)
E[att$=val]	Attributselektor („E besitzt Attribut att“, dessen Wert mit dem Teilwert „wert“ endet)
E[att*=val]	Attributselektor („E besitzt Attribut att“, dessen Wert den Substring „wert“ enthält)

Attributselektoren in CSS3 (Teil 2)

Beispiele:

Diese Anweisung hebt Links hervor, deren URL mit „http“ beginnt (also in der Regel externe Links):

```
a[href^="http"] { color:red; }
```

Diese Anweisung hebt Bilder vom Typ „.jpg“ hervor:

```
img[src$=".jpg"] { border: 1px solid green; }
```

3.4. Zusammengesetzte Selektoren

Mehrere Einzelselektoren können zu einem zusammengesetzten Selektor **gruppiert** werden, und so das zu selektierende Element genauer bestimmen (dies betrifft meist dessen Kontext im Quelltext).

✔ Die zwischen den Einzelselektoren stehenden Zeichen bezeichnet man als *Kombinatoren*.

Als Kombinator dient im einfachsten Fall ein Leerzeichen (*Descendant Selektor*), aber auch andere Zeichen werden eingesetzt, wie + , > und ~.

✔ **Merke:** Es wird stets ausschließlich das Element selektiert, dass im zusammengesetzen Selektor *ganz rechts* steht. Die anderen Teilselektoren definieren lediglich den Kontext für dessen Selektion.

3.4.1. Der Descendant Selektoren

Anwendbar auf: alle Elemente.

```
vorfahre abkoemmling
```

Dieser zusammengesetzte Selektor erfasst alle Elemente vom Typ `abkoemmling`, die (irgendwo oberhalb in der Hierarchie) einen *Vorfahren* vom Typ `vorfahre` besitzen. Die Einzelselektoren werden durch mindestens ein Leerzeichen getrennt.

Beispiel:

```
a  b
```

erfasst <b> in <a>, wenn folgende Struktur vorliegt

```
<a>
  <b> ... </b>
</a>
```

aber *auch*, wenn beispielsweise gilt

```
<a>
  <c>
    <b> ... </b>
  </c>
<a>
```

Die hierarchische Lage des <b>-Knotens innerhalb von <a> ist unerheblich – die Regel gilt also für Kindknoten genauso, wie für beliebig tief gestaffelte Abkömmling (Kindknoten sind eine Untermenge der Abkömmlinge).

✔ **Achtung:**
Ein beliebter *Flüchtigkeitsfehler* besteht im versehentlichen Einfügen eines Leerzeichens in einen Selektor, was dessen Bedeutung vollständig verändert. Statt:

```
p.beispiel
```

steht etwa versehentlich

```
p .beispiel
```

Wirkung: Ziel ist nun jedes beliebige Element der Klasse „beispiel“ in einem P-Container, wo eigentlich „P-Container der Klasse ‚beispiel‘“ gemeint waren.

3.4.2. Der Child Selektor >

Anwendbar auf: alle Elemente

```
elternelement > kindelement
```

Dieser zusammengesetzte Selektor erfasst alle Elemente vom Typ `kindelement`, die ein Elternelement vom Typ `elternelement` besitzen.

Zwischen den Einzelselektoren steht das Kombinatorsymbol >, das von Leerzeichen umgeben sein darf (die Leerzeichen dürfen entfallen, da das Kombinatorzeichen nicht Teil eines Elementnamens sein kann).

Beispiel:

```
a > b
```

erfasst <b> in <a>, wenn folgende Struktur vorliegt

```
<a>
  <b> ... </b>
</a>
```

nicht jedoch, wenn beispielsweise gilt

```
<a>
  <c>
    <b> ... </b>
  </c>
<a>
```

... da der Childselektor *nur* für die **unmittelbar** in <a> enthaltenen <b> gilt.

3.4.3. Der Adjacent Sibling Combinator +

Anwendbar auf: alle Elemente

```
vorgaenger + nachfolger
```

Dieser zusammengesetzte Selektor erfasst ein Element `nachfolger`, wenn ihm ein Element `vorgaenger` in Quelltextreihenfolge **direkt** vorangeht (hierunter versteht man, dass der andere Container des Vorgängers *unmittelbar* vor dem Starttag des Nachfolgers geschlossen wird).

Zwischen den Einzelselektoren steht das Kombinatorsymbol +.

Beispiel:

```
a + b
```

erfasst <b> *nach* <a>, wenn folgende Struktur vorliegt:

```
<a> ... </a>
<b> ... </b>
```

✔ Einfacher ausgedrückt, geht es um den „unmittelbar folgenden Geschwisterknoten“ im gleichen hierarchischen Kontext (Elternknoten).

3.4.4. Der General Sibling Combinator ~

Anwendbar auf: alle Elemente

```
vorgaenger ~ nachfolger
```

Dieser zusammengesetzte Selektor erfasst **alle Elemente** vom Typ `nachfolger`, wenn ihnen ein Element `vorgaenger` auf Geschwisterebene in Quelltextreihenfolge vorangeht.

Zwischen den Einzelselektoren steht das Kombinatorsymbol ~.

```
h1 ~ p
```

Erfasst **alle** P-Container, die einem H1 auf gleicher Ebene folgen:

Es ist irrelevant, ob zwischen den selektierten Elementen auf gleicher Ebene noch Elemente anderen Typs erscheinen (beispielsweise `<div>`-Container, wie hier):

```
<h1>General Sibling Combinator</h1>
<p>wird selektiert</p>
<div>nicht selektiert</div>
<p>wird selektiert</p>
<p>wird selektiert</p>
<div>nicht selektiert</div>
<div>nicht selektiert</div>
```

3.5. Gruppierung von Selektoren

Anwendbar auf: alle Selektoren.

Soll die gleiche Regel für mehrere Selektoren gelten, so können diese **gruppiert** werden.

✔ Hierfür werden die Einzelselektoren zu einer *kommagetrennten Liste* zusammengefügt.

Im einfachsten Fall ist es eine Liste aus Typselektoren. Hier wird für die Überschriften `<h1>`, `<h2>` und `<h3>` die gleiche Schrift angeordnet:

```
h1, h2, h3 {font-family:arial;}
```

Die Gruppierung von Selektoren ist nicht auf einfache Selektoren beschränkt, sondern funktioniert genauso für zusammengesetzte Selektoren, Klassen, ID- und Pseudo-Selektoren etc.:

```
/* drei Klassenselektoren gruppiert: */
.klasseA, .klasseB, .klasseC:hover {...}


/* zwei ID-Selektoren und ein Descendant-Selektor: */
#idA, #idB, div span {...}
```

3.6. Über Selektor-Performance

Der Browser bewertet Selektoren für zum Rendern anstehende Knoten von **rechts nach links**. Es wird also zunächst der an Ende stehende Ausdruck auf den Knoten geprüft, danach weiter nach links gegangen und dessen Kontext entsprechend des Selektors betrachtet.

Ein Beispiel:

```
div#content p.teaser b { ... }
```

Bei der Bewertung des Ausdrucks müssen folgende Schritte erfolgen:

1. `div#content p.teaser` **`b`**
   Ist der zu stylende Knoten ein <b>-Tag?

2. `div#content p`**`.teaser`** `b`
   Besitzt eines seiner Ancestor-Elemente die Klasse `.teaser`?

3. `div#content` **`p`**`.teaser b`
   Ist dieses Element ein <p>-Tag?

4. `div`**`#content`** `p.teaser b`
   Hat eines der Ancestor-Elemente den ID `#content`?

5. **`div`**`#content p.teaser b`
   Ist dieses Element ein `<div>`-Tag?

... in diesem Fall wird der betrachtete Knoten mit dem Regelsatz belegt.

Es ist zwar richtig, dass komplexe Selektoren mehr Aufwand bei der Bewertung bedeuten, die Forderung nach flachen Selektorhierarchien oder gar die Beschränkung auf Klassen (die entsprechend auch im Dokument vergeben werden müssen!) ist jedoch übertrieben.

Gemäß einer Untersuchung von Steve Sauders beträgt der **Gewinn in der Renderperformance** durch entsprechende Optimierung **maximal 50ms**. (Diese Zahl stammt aus 2009; seitdem ist die Performace der Renderingengines eher besser geworden.)

✔ **Artikel:** http://www.stevesouders.com/blog/2009/03/10/performance-impact-of-cssselectors/

4. Spezifizität

Der Zuordnung einer CSS-Formatierungsregel an einen Elementcontainer dient ihr Selektor. Dieser arbeitet dabei anhand eines Mustervergleichs (*pattern matching*), wobei jede gefundene Übereinstimmung eines Musters auf ein Element angewendet wird. Dadurch addieren sich gegebenenfalls die Eigenschaftszuweisungen mehrerer, auf denselben Elementcontainer passenden Regeln.

Im Fall eines Konflikts zwischen zugewiesenen Eigenschaften ermittelt sich die dominierende CSS-Eigenschaft danach, wie *spezifisch* das Muster ihres Selektors formuliert ist.

4.1. Ermittlung der Spezifizität der Selektoren

Die Gewichtung („Spezifizität“) eines Selektors misst sich an seinen Anteilen von Typ- Klassen- und Id-Selektoren.

✔ **Gruppe A:**
die Zahl der ID-Selektoren im Selektor

✔ **Gruppe B:**
die Zahl der Attribut- und Klassenselektoren sowie der Pseudoklassen

✔ **Gruppe C:**
die Zahl der Typselektoren. Der universelle Selektor wird nicht berücksichtigt.

Die Formulierung der *Spezifizität* wird getrennt nach drei, nach Rang geordneten, Gruppen *A-B-C* vorgenommen. Hierbei wird *pro Gruppe* die Zahl der betreffenden Selektoren summiert – es wird also *keine* übergreifende Gesamtsumme gebildet.

Ein einzelner Klassenselektor überwiegt daher einen (fiktiven) zusammengesetzten Selektor mit beliebig vielen Typselektoranteilen. Dasselbe gilt für ID-Selektoren in Relation zu Klassenselektoren.

Selektor (Beispiel)	**ID**	**Klasse** **Pseudoklasse** **Attribut**	**Typ**
h1	0	0	1
div p	0	0	2
.beispiel	0	1	0
.test.blau	0	2	0
#d1	1	0	0

Im Falle zusamengesetzter Selektoren werden die Anteile einzeln in den Rubriken aufgerechnet. Die Typanteile bei zusammengesetzen Selektoren fallen dementsprechend nicht stark ins Gewicht, können jedoch das „Zünglein an der Waage“ bedeuten.

Selektor (Beispiel)	**ID**	**Klasse** **Pseudoklasse** **Attribut**	**Typ**
p.test	0	1	1
div#d1	1	0	1

In der Bewertung zählen Attributselektoren und Pseudoklassen und Pseudoelemente gleichrangig zu den Klassenselektoren.

Selektor (Beispiel)	**ID**	**Klasse** **Pseudoklasse** **Attribut**	**Typ**
.beispiel	0	1	0
[title]	0	1	0
:hover	0	1	0

Diese Regelung kann dazu führen, dass *sehr unterschiedlich aussehende* Selektoren *dieselbe* Spezifizität besitzen.

In diesem Falle erfolgt eine Konfliktregelung wieder entsprechend der **Reihenfolge** der Deklarationen im Stylesheet.

Selektor (Beispiel)	**ID**	**Klasse** **Pseudoklasse** **Attribut**	**Typ**
a:hover	0	1	1
a.extern	0	1	1
a[title]	0	1	1

Der universelle Selektor *, die Spezifizierung von Attributselektoren auf einen Wert (egal wie) sowie die Combiner-Symbole >, + oder ~ werden bei der Verechnung der Spezifizität nicht berücksichtigt.

Auch dies kann die Bewertung eines Selektors verschleiern.

Selektor (Beispiel)	**ID**	**Klasse** **Pseudoklasse** **Attribut**	**Typ**
*	0	0	0
*.beispiel	0	1	0
* .beispiel	0	1	0
[title="Ein Titel"]	0	1	0
*#d1	1	0	0
div#d1	1	0	1
#d1 p	1	0	1
#d1 > p	1	0	1
#d1 * p	1	0	1
.test.blau	0	2	0
.test .blau	0	2	0

4.1.1. Erhöhung der Spezifizität

Um einen Selektor „künstlich“ spezifischer zu machen, kann man den ID-Selektor des logischen Dokumentbereichs hinzunehmen, in dem sich das Zielelement befindet.

✔ Dies hilft bei der Konfliktlösung, sofern ungewollt eine CSS-Anweisung zur Anwendung kommt, die für Elemente an anderer Position im Dokument gedacht ist.

Statt allgemein zu schreiben:

```
/* eigentlich auf <p> im Inhaltsbereich anzuwenden: */
 p { ... }
```

…kann jederzeit spezifischer wie folgt formuliert werden:

```
/* Erhöhung der Spezifizität mit ID-Selektor */
div#inhalt  p  { ... }
```

Hinweis:
Zu beachten ist, dass die Verwendung von ID-Selektoren wohlüberlegt erfolgen sollte!
Für modulares CSS gelten IDs als riskant und von ihrem Einsatz wird abgeraten. In diesem Falle wird eine möglichst flache Selektorhierarchie abgestrebt, im Extremfall (BEM oder SMACSS) in Form weitgehender Beschränkung auf Klassenselektoren.

4.1.2. Probleme mit Spezifizität

Sobald ID-Selektoren Teil eines zusammengesetzen Selektors sind, können Probleme bei Erweiterungen des Stylesheets auftreten. Dies liegt an der hohen Spezifizität der entsprechenden Selektoren, die durch andere schwer zu überschreiben sind. Ein Klassiker ist ein Link im Inneren eines Bereichs, der wiederum in einem durch eine CSS-Klasse gekennzeichnetem Container speziell dargestellt werden soll:

```
#content .intro a {
   color: green;
}
```

Für Links soll allgemein eine Modifierklasse die Hintergrundfarbe verändern können (grün), gleichzeitig aber die Textfarbe neu setzen (weiß), um den Kontrast zu erhalten:

```
a.aktiv {
   text-decoration: none;
   background-color: green;
   color: white;
   padding: 0.25em;
}
```

Wendet man diese Klasse an, so kann die grüne Textfarbe aufgrund der Spezifizität des anderen Selektors nicht überschrieben werden. Es gibt zwei Lösungen für das Problem:

1. Das Flag !important

2. Erhöhung des Spezifizität des Modifier-Selektors

Die Lösung mit `!important` funktioniert, verhindert aber, dass diese Eigenschaft eventuell auf irgendeinem anderen Wege wieder überschrieben werden kann.

```
/* 1) Zwangsoverride durch !important-Flag */
a.aktiv {
   text-decoration: none;
   background-color: green;
   color: white !important;
   padding: 0.25em;
}
```

✔ Für Modifier-Klassen ist der Ansatz zu rechtfertigen, aufgrund der Annahme, dass die Klasse **in jedem Fall** wirken soll, sobald sie eingesetzt wird. Das Flag muss ggfs. für jedes Property einzeln gesetzt werden. Ansonsten ist vom Einsatz von `!important` abzuraten.

Nennen wir `!important` ruhig den „Holzhammer“ des CSS-Designers :-) ... also nur als *ultima ratio* einzusetzen!

Alternativ bleibt nur, einen entsprechend spezifischen Selektor für die Regel zu definieren. Meist wird man diesen *zusätzlich* zur allgemeinen (einfachen Klasse) definieren.

```
/* zusätzliche alternative Selektoren: */
#content .intro a.aktiv,
a.aktiv {
   text-decoration: none;
   background-color: green;
   color: white;
   padding: 0.25em;
}
```

In wenig komplexen Fällen ist gegen diesen Ansatz nichts einzuwenden. Soll der Modifier aber auch für Links in *anderen* Bereichen, die durch IDs gekennzeichnet sind, angewendet werden (`#sidebar`, `#footer` oder ähnlich), müssten entsprechend weitere Varianten von Selektoren hinzugefügt werden.

5. Regeln – die “rule sets”

Das Stylesheet besteht aus einer Abfolge von *Formatierungsregeln*, so genannten „rule sets“. Jede Regel zerfällt in zwei Teile, den *Selektor* und den in geschweifte Klammern eingeschlossenen *Deklarationsblock*, der die Formatierungsanweisungen enthält.

✔ Die Formatierung muss einem Elementcontainer **eindeutig** zugewiesen werden können. Diese Zuordnung erfolgt über ein Vergleichsmuster – gebildet durch den vorangestellten *Selektor*.

Abstrakt kann man eine Formatierungsregel so formulieren:

```
selektor { propertydeklaration }
```

Jede Eigenschaftsdeklaration ist zusammengesetzt aus **Propertybezeichner** und **Propertywert** – getrennt durch einen Doppelpunkt. Leerzeichen um den Doppelpunkt sind erlaubt.

```
selektor {property-name : property-value;}
```

Das Ende der Deklaration wird duch ein abschließendes **Semikolon** gekennzeichnet.

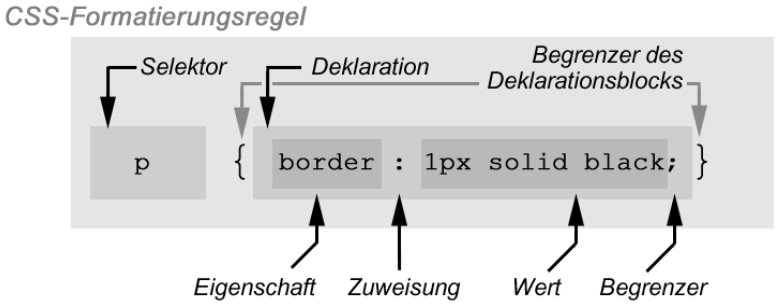


Abb.: Schematischer Aufbau einer CSS-Formatierungsregel

Einzelregeln für den *gleichen* Selektor addieren sich:

```
selektor { property-name_1 : property-value;}

selektor { property-name_2 : property-value;}
```

… und können zu einem Deklarationsblock zusammengefasst werden. Es dürfen beliebig viele, gleichzeitig gültige Anweisungen auftreten:

```
selektor {
          property-name_1: property-wert_1;
          property-name_2: property-wert_2;
}
```

Achtung: Für das *letzte Property* vor der schließenden geschweiften Klammer kann das Semikolon im Prinzip entfallen. Man sollte es sich jedoch zur Gewohnheit machen, stets auch diese *letzte* Anweisung mit Semikolon zu beenden, um Flüchtigkeitsfehler bei der *Erweiterung* von Regelsätzen zu vermeiden!

Warnung vor vergessenen Semikolons:
Wird das Setzen eines Semikolon im Inneren eines Blocks *unterlassen*, so wird die entsprechende Zuweisung *nicht erkannt* und meist auch die auf sie *folgende* CSS-Anweisung nicht ausgeführt!

5.1. Formatierung von Regelsätzen

Folgende Grundregeln haben sich für das Schreiben von CSS-Code als nützlich erwiesen:

- Öffnende Klammer in der Zeile des Selektors
- Jede Deklaration mit Semikolon abschließen
- Ein Leerzeichen nach dem Seminkolon
- Deklaration nicht mit Tab sondern mit vier Leerzeichen einrücken

5.2. Reihenfolge von Properties in Regeln

Im Grunde ist die Abfolge der Eigenschaftenaufzählung im Rahmen einer Regel völlig irrelevant für deren Funktion, sofern hierdurch keine Reihenfolgenkonflikte auftreten.

✔ Man muss darauf achten, dass Multiproperties (Sammeleigenschaften wie `font` oder `border` etc.) versehentlich durch ihre Defaultwerte vorausgehende, durch Einzelproperties festgelegte Eigenschaften überschreiben können. Hier gilt, dass stets zuerst das Multiproperty gesetzt werden muss, danach bei Bedarf die Einzelproperties.

Ansonsten sollte man sich ein Schema für die Reihenfolge der Definitionen überlegen, dass einen Regelsatz übersichtlich und für andere Entwickler nachvollziehbar macht. Bewährt hat sich folgende Faustregel:

1. Boxeigenschaften
2. Bordereigenschaften
3. Backgroundeigenschaften
4. Texteigenschaften
5. Sonstige

6. CSS-Eigenschaften

Die Formulierung der Regeln zur Präsentation eines Elements, das durch einen Selektor erfasst wird, geschieht mittels *CSS-Eigenschaften*, auch als „Properties“ bezeichnet. Diesen wird jeweils ein Wert zugeordnet.

- Ein **Regelsatz** kann sich aus beliebig vielen Wertzuweisungen an Eigenschaften zusammensetzen.

- Nur die jeweils auf das Zielelement *anwendbaren* Eigenschaften treten in Kraft, andere (nicht zutreffende oder nicht existente) werden nicht beachtet (es gibt keine Fehlermeldungen).

- Innerhalb einer Regel kann eine Eigenschaft mehrfach mit einem Wert belegt werden. Dies ist kein Fehler; allerdings gilt nur die letzte Zuweisung.

Die CSS-Eigenschaften werden in Folge nach *Anwendungsgebiet* aufgelistet. (In den Listen stehen diejenigen CSS3-Eigenschaften in eckigen Klammern, die noch nirgends unterstützt werden.)

- **Stetiger Wandel:** Schlagen Sie gegebenenfalls die Unterstützung einer gewünschten Eigenschaft für Ihren Zielbrowser **hier** nach:

  `http://www.caniuse.com/`

6.1.1. Textproperties

Diese Eigenschaften dienen zur **Textformatierung** von Textblöcken oder innerhalb von Fließtext.

Zu ihnen gehören die Eigenschaften `direction`, `letter-spacing`, `line-height`, `text-align`, `text-decoration`, `text-indent`, `text-transform`, `unicode-bidi`, `vertical-align`, `white-space` und `word-spacing`.

Neu in CSS3: `text-shadow, text-emphasis, text-justify, [text-outline], text-overflow, [text-wrap], word-break, word-wrap`

6.1.2. Fontproperties

Diese Eigenschaften steuern die **Schriftdarstellung**, also Schriftgröße, -art und ähnliches.

Zum gleichzeitigen Setzen mehrerer Eigenschaften dient das *Sammelproperty* `font`.

Die restlichen Eigenschaften dieser Gruppe sind deren Untereigenschaften: `font-family`, `font-size`, `font-style`, `font-variant`, `font-weight`.

Neu in CSS3: `font-size-adjust, [font-stretch]`

6.1.3. Farben und Hintergründe

In CSS können allen Elementen Hintergrundfarben und -bilder zugewiesen werden. Die **Schriftfarbe** gilt als „Vordergrundfarbe“ und wird mit dem Eigenschaftsnamen `color` bezeichnet.

Ein Hintergrundbild wird als URL-Ressource eingebunden. Auch in dieser Gruppe existiert ein Sammelproperty `background`.

Die restlichen Eigenschaften dieser sind dessen Untereigenschaften: `background-attachment`, `background-color`, `background-image`, `background-position`, `background-repeat`.

Neu in CSS3: `background-clip, background-origin, background-size, opacity`

6.1.4. Boxproperties

Aus der Sicht von CSS wird jeder Blockcontainer als „Box“ betrachtet, die in vier Bereiche (Inhalt, Padding, Border, Margin) eingeteilt werden kann.

Hierfür stehen folgende Eigenschaften zur Verfügung: `padding`, `margin`, `border` (mit Untereigenschaften `border-color`, `border-style` und `border-width`), `height`, `width` (Innenbreite!), `max-height`, `max-width`, `min-height` und `min-width`.

Neu in CSS3: `overflow-x, overflow-y, [overflow-style], border-radius, border-image, box-shadow, resize`

6.1.5. Positioning-Properties

Die Gruppe der **Positioning-Properties** fasst alle CSS-Eigenschaften zusammen, die Positionierung von Elementcontainern und weitere, damit zusammenhängende Aspekte betreffen.

Dies sind die Eigenschaften `bottom`, `clear`, `clip`, `float`, `left`, `overflow`, `position` (mit den Werten `static`, `relative`, `absolute`, `fixed`), `right`, `top`, `visibility` und `z-index`.

Neu in CSS3: –

6.1.6. Displayproperties

Die Gruppe der **Displayproperties** besteht nur aus einer Eigenschaft, `display`, die entscheidet, ob ein Elementcontainer als Blockelement (`display:block`), Inlineelement (`display:inline`) oder gar nicht (`display:none`)dargestellt werden soll.

✔ Hierbei werden gegebenenfalls auch die **Grundeigenschaften** eines HTML-Elements überschrieben.

Desweiteren existieren zahlreiche weitere Werte zur Darstellung als Inlineblock, Tabellenelement oder Listenelement.

Neu in CSS3: –

6.1.7. Listenstyle Properties

Diese Gruppe umfasst verschiedene Properties zum Stylen von Listen und Listenelementen.

Die Sammeleigenschaft `list-style` ist unterteilt in die Untereigenschaften `list-style-type`, `list-style-image` und `list-style-position`.

Neu in CSS3: –

6.1.8. Tabellen-Properties

In Zusammenhang mit „natürlichen“ HTML-Tabellenelementen und beliebigen Elementen, die per `display`-Property mit `display:table` oder `display:inline-table` CSS-Tabellenelementeigenschaften zugewiesen bekommen haben, existieren weitere CSS-Eigenschaften, um die Darstellung exakter zu steuern.

Dies sind die Properties `border-collapse`, `border-spacing`, `caption-side`, `empty-cells` und `table-layout`.

Neu in CSS3: –

6.1.9. Neue Eigenschaftsgruppen in CSS3

In CSS3 werden der CSS-Landschaft neue Konzepte und damit Eigenschaftsgruppen hinzugefügt. Die wichtigsten von diesen sollen hier kurz aufgelistet werden – zum Teil werden sie im Rahmen dieses Seminars näher betrachtet:

2D/3D Transform Properties:
`transform-origin, transform-style, perspective, perspective-origin, backface-visibility`

Transition Properties:
`transition, transition-property, transition-duration, transition-timing-function, transition-delay`

Animation Properties:
`@keyframes, animation, animation-name, animation-duration, animation-timing-function, animation-delay, animation-iteration-count, animation-direction, animation-play-state`

Multi-column Properties:
`column-count, [column-fill], column-gap, column-rule, column-rule-color, column-rule-style, column-rule-width, column-span, column-width, columns`

✔ **Hinweis:** Es existieren noch weitere Gruppen und Eigenschaften in CSS3, die in diesem Seminar jedoch nicht behandelt werden, da sie mangels Unterstützung noch nicht für den Praxiseinsatz taugen.

6.2. Werte und Maßeinheiten

Als Werte der CSS-Eigenschaften werden häufig *Schlüsselbegriffe* (wie „left“, „center“, „top“, „bottom“, „lowercase“, „thin“ etc.) verwendet. Hier als Beispiel die Textausrichtung:

```
text-align: right;
```

Für die Einbindung von Hintergrundbildern erfolgt eine *Ressourcenangaben als URI*:

```
background-image: url('blau_horiz.gif');
```

Ansonsten erfolgen Angaben als numerische Größenangaben zusammen mit einer **Maßeinheit**.

✔ Dies sind in der Regel *Längenangaben*, die auch zur Kennzeichnung von Abmessungen oder Schriftgrößen dienen.

Ein weiterer benötigter Wertetyp sind *Farbangaben*, die als Schlüsselwort oder Farbcode erfolgen können.

6.2.1. Längenangaben

Abmessungen können in CSS in Form von Längenwerten mit Maßeinheit bestimmt werden. Entgegen der Gepflogenheit in HTML ist stets eine Maßeinheit zu benennen. Beim Wert 0 darf diese entfallen.

Man unterscheidet zwischen *absoluten* und *relativen* Längenwerten:

- Absolute Maßeinheiten bestimmen eine **exakte Darstellungsgröße**, die an genormte Einheiten gebunden ist

- Relative Maßeinheiten bestimmen eine **Darstellungsgröße in Abgängigkeit** von äußeren Bedingungen, wie die festgelegte Basisgröße eines Ausgabemediums oder die eines Bezugselements.

Absolute Größenangaben

Absolute Größenangaben sind geeignet für Ausgabemedien mit vorhersehbaren (konstanten) Randbedingungen, beispielsweise den Ausdruck von Dokumenten (festgelegte Papiermaße).

Zu den absoluten Maßeinheiten in diesem Sinne zählen:

cm („Zentimeter“), mm („Millimeter“), in („Inch/Zoll“), pt („Punkt“), pc („Pica“)

- **Hinweis:** Verwenden Sie diese Einheiten *nicht* für die Bildschirmausgabe. Ziehen Sie sie jedoch für die Erstellung eines *Druckstylesheets* in Betracht.

Relative Größenangaben

Relative Größenangaben sind geeignet für Ausgabemedien mit wechselnden Randbedingungen, wie die Bildschirmausgabe. Hier sind neben den Nutzereinstellungen und Browserdefaults (Basisgrößen) auch Abmessungen und Auflösung (Pixelgröße) des Bildschirmes oder Browserfenstergröße zu berücksichtigen.

Zu den relativen Maßeinheiten in diesem Sinne zählen die Einheiten:

em („M-Breite“), ex („x-Höhe“), px („Pixel“), [zahl]% („Prozentwert“)

✔ **Hinweis:** Verwenden Sie *bevorzugt* Prozentangaben oder em-Breiten zur Bemaßung von Schrift und Layoutcontainern. Dies erleichtert flexibles, anpassungsfähiges Layout. Von der Verwendung der ex-Höhe möchten wir an dieser Stelle abraten, da deren Umsetzung durch die Browser ungleichmäßig und unvorhersehbar ist.

Prozentangaben

Prozentangaben können *positiv* oder *negativ* sein. Das Prozentzeichen steht ohne dazwischenstehenden Leerraum hinter der numerischen Angabe. Eine Prozentangabe ist *stets relativ*, beruht also auf einer *Bezugsgröße*, abgeleitet entweder aus dem übergeordneten Elementcontainer, oder (seltener) aus einer anderen Eigenschaft desselben Elementcontainers.

Beispielsweise richtet sich eine Prozentangabe für die Zeilenhöhe `line-height` nach der Schriftgröße `font-size`, wie sie innerhalb des Elementcontainers gilt. Eine Prozentangabe für Höhe oder Breite eines Containers hingegen wird an der Höhe oder Breite des *umgebenden Containers* (des Elternelements) gemessen.

Beispiel:

```
p      { width: 70%; }

p.eng  { line-height: -10%; }
```

6.2.2. Farben

In CSS können Farben verschiedenen Elementeigenschaften zugewiesen werden, wobei man zwischen drei grundsätzlichen Methoden der Farbzuweisung und zwei von diesen abgeleiteten unterscheiden kann.

Farbname

Es kann einer der vordefinierten Farbnamen verwendet werden. In CSS 2.1 sind *siebzehn Farbnamen* als Grundfarben festgelegt, deren Verwendung sicher ist.

```
p { background-color: red }
```

✔ **Speziellere Farbnamen sind problematisch:**
Weitere 130 Farbbezeichner sind in CSS3 für die restlichen gängigen Farbtöne definiert. Diese entsprechen größtenteils den durch Netscape eingeführten Farbnamen, die aber inzwischen auch in SVG 1.0 und als so genannte “X11”-Farben spezifiziert sind. Da die Unterstützung durch Browser noch immer nicht in allen Einzelfällen gewährleistet ist, sind Angaben in Hexadezimal- oder RGB-Form vorzuziehen.

Hexadezimalfarbangabe

Die Farbangabe wird in die drei Grundfarbanteile (Rot Grün Blau) der additiven Farbsynthese aufgeschlüsselt als Hexadezimalzahlpärchen RRGGBB vorgenommen, wobei der Angabe das Hashsymbol # vorangestellt wird:

```
p { background-color: #aaccff }
```

Hexadezimalfarbangabe, gekürzt

Sind jeweils *beide* Hexadezimalzahlen aller drei *Farbkomponenten* identisch, so ist das Fortlassen des jeweils *zweiten* Zeichens als Verkürzung gestattet. Statt (abstrakt) #RRGGBB kann also #RGB geschrieben werden:

```
/* entspricht: #aaccff */
p { background-color: #acf }
```

RGB-Wert, numerisch

Die RGB-Werte können, anstatt in hexadezimaler, auch direkt in dezimaler Form übergeben werden, wobei sie mir dem Ausdruck `rgb()` zu klammern sind. Für jede Farbkomponente sind Werte zwischen 0 (Minimalwert) und 255 (Maximalwert) möglich. Die Angaben für die Einzelkomponenten müssen durch Komma getrennt werden:

```
p { background-color: rgb(255, 0, 0) }    /* rot */
```

RGB-Wert, prozentual

Alternativ zu einer numerischen Angabe, können die Anteile der Komponenten auch als Prozentangabe übergeben werden, die sich dabei auf den Maximalwert 255 bezieht:

```
p { background-color: rgb(100%, 0%, 0%) }  /* rot */
```

RGBA-Wert, numerisch

Als vierte Komponente kann zu einem RGB-Wert auch noch ein „Alphakanal“-Wert zwischen 0 und 1 für die *Transparenz* definiert werden.

✔ Hierbei bedeutet der Wert 0 Volltransparenz, der Wert 1 Vollopazität.

Man erhält eine RGBA-Angabe wie folgt:

```
/* rot, halbtransparent */
p { background-color: rgba(255, 0, 0, 0.5) }
```

7. Neuerungen aus CSS3

Lange Zeit konnte CSS3 als „Baustelle“ betrachtet werden, da sich die Vielzahl an Unterspezifikationen in unterschiedlichen Phasen der Fertigstellung befanden und die Unterstützung in der Browserlandschaft lückenhaft und teilweise nicht standardkonform war.

Dies ist mittlerweile nicht mehr der Fall. Die meisten in Folge vorgestellten Properties sind generell verwendbar.

Ein Problem ist der **Internet Explorer**, da dieser durch seine Bindung an das Betriebssystem nicht allgemein upgradefähig ist. So ist Windows 7 auf IE11 (also Stand 2013) festgeschrieben. Die neueren Edge-Browser (IE12 und neuer) von Windows 10 unterstützen CSS inzwischen hingegen ebensogut wie Firefox, Chrome und Safari.

✔ **Konsultieren Sie im Zweifelsfall:** www.caniuse.com

7.1. Vendor prefixes für CSS3-Eigenschaften

In der Übergangsphase ließen die Browserhersteller ihre Implementierungen aus Kompatibilitätsgründen nur mit sogenannten Vendor-Prefixes zu. Folgende Prefixes waren/sind in Gebrauch; einige hiervon können mangels Verbreitung des entsprechenden Browsers vernachlässigt werden.

Prefix	**Hersteller / Browser**
-moz-	Mozilla Foundation / Gecko Browser (Firefox, Camino, Flock)
-webkit-	Google Chrome, Safari, Nokia S60 und andere mobile Browser
-ms-	Microsoft (Internet Explorer)
-o-	Opera Software (Opera, Opera mini, Opera mobile)
-khtml-	Konqueror Browser
-wap-	WAP Forum

Lediglich für die Unterstützung (sehr viel) älterer Browser sind diese Prefixes noch relevant. Da CSS3 inzwischen weithin implementiert ist (mit Ausnahme einiger weniger Aspekte im Bereich 3D-Animation) kann meist

einfach die W3C-Standardbezeichnung eines Properties eingesetzt werden.

7.2. CSS3 Opacity

Gerade die Transparenz von Boxen ist ein wichtiges Gestaltungsmittel. CSS3 geht für seine Definition dessen vom Gegenteil aus, der **"Opazität"**, wobei eine mit dem Wert 0 belegte Box „voll-transparent", also durchsichtig ist, eine mit dem Wert 1.0 „voll-opak", also undurchsichtig. Letzteres entspricht dem Defaultzustand.

Durch eine transparente oder teiltransparente Box kann das dahinter liegende Element wahrgenommen werden, was der Rendering-Engine bei Überlagerung teiltransparenter Boxen einiges an Arbeit abverlangt.

```
img.bild1 { opacity:0.25; }
img.bild2 { opacity:0.50; }
img.bild3 { opacity:0.75; }
img.bild4 { opacity:1.0; }
```

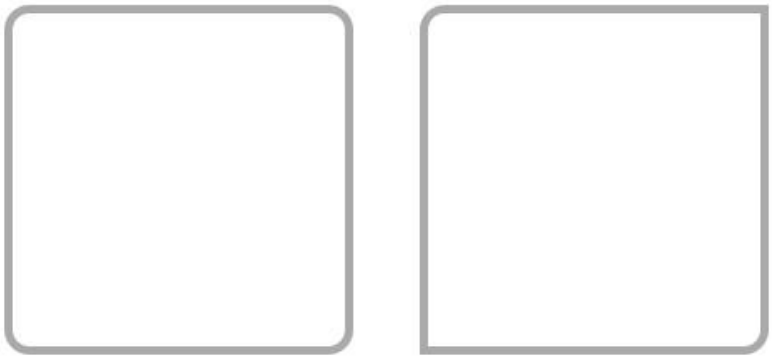
✔ Opazität wird von allen aktuellen Browsern unterstützt; ältere Internet Explorer können das Verhalten nur mit Alpha-Filtern nachbilden.

7.3. CSS3 Border-Radius – Runde Ecken

Runde Ecken bei Boxen zählen zum aktuellen Designhype, sind aber, dezent angewendet, optisch durchaus angenehm.

Hier ein einfaches Beispiel, das alle vier Ecken einer Box gleichmäßig rundet (folgende Abbildung, links):

```
#beispiel {
  border: 5px solid #aaa;
  border-radius: 15px;
}
```



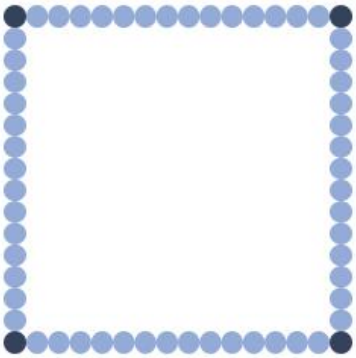
Auch ein individuelles Runden einzelner Ecken ist möglich (siehe Abbildung, rechts). Hier wird die obere linke und die untere rechte Ecke einzeln angesprochen. Die anderen beiden Ecken bleiben „eckig“. Man wird wohl eher die Schriebweise mit vier Werten einsetzen:

```
#beispiel2 {
  border: 5px solid #aaa;

  border-top-left-radius: 15px;
  border-top-right-radius: 0px;
  border-bottom-right-radius: 15px;
  border-bottom-left-radius: 0px;

  /* alternativ ist dies auch so möglich: */
  border-radius: 15px 0 15px 0;
}
```

7.4. CSS3 Border Image

Mittels der `border-image`-Eigenschaft kann eine Grafik zum Hinterlegen des Rahmens einer Box eingesetzt werden. (Auch im Internet Explorer 10 wird diese Eigenschaft derzeit noch nicht unterstützt.)

✔ Die Grafik wird je nach vorliegender `border-width` skaliert und entsprechend der Breite der Box nahtlos gekachelt, wobei die Ecken der Grafik auf die der Box gelegt werden.



```
#beispiel  {
  border-width:15px;
   border-image:url(border.png) 30px/30px round;
}
```

Handelt es sich bei der Grafik um einen Farbverlauf, so kann diese, anstelle einer Kachelung, auch gedehnt werden:

```
#beispiel2 {
  border-width:15px;
  border-image:url(border2.png) 30 30 stretch;
}
```

7.5. CSS3 Box Shadows

Ein interessantes Spielzeug für den Designer ergibt sich auch aus der Möglichkeit, eine Box mit **Schlagschatten** zu versehen, wobei Richtung, Farbe und Weichzeichnung des Schattens festgelegt werden können.

✔ Diese Eigenschaft kann auch *direkt* einem Bild zugewiesen werden. Bei Kombination mit Opacity wird der Schatten ebenfalls transparent!

```
#beispiel{

 /* Werte: xPos yPos blur color */

  box-shadow: 5px 5px 7px #888;
}
```

7.6. CSS3 Background Images

Oft beklagt wurde, dass Hintergrundgrafiken einem Container zwar zugewiesen, aber nicht weiter beeinflusst (beispielsweise skaliert) werden konnten.

Eine solche **Skalierung** ist mit CSS3 nunmehr möglich (derzeit noch nicht umgesetzt in Mozilla-Browern und IE).

Auch die Zuweisung **mehrerer Hintergrundbilder** an einen Container ist Teil der Spezifikation und wird bereits von einigen Browsern unterstützt (Firefox ab 3.6, Safari/Chrome, Opera ab 10.5, Internet Explorer ab 9.0).

7.6.1. background-size

Die Abmessungen einer Hintergrundgrafik werden festgelegt, indem als erster Wert die zu verwendende Breite, als zweiter die Höhe der zu skalierenden Grafik übergeben werden.

Die in der folgenden Abbildung verwendete Grafik hat (wie der grau umrandete Container, dem sie als Hintergrund dient) die Dimensionen 415 x 375 Pixel. Sie wird auf 200 x 150 herunterskaliert und zentriert.


```
#beispiel {
      background:url(globus.jpg) center no-repeat;
      background-size: 200px 150px;
}
```

7.6.2. Multiple Background Images

Mehrfache Hintergrundbilder können nun einfach durch **Aneinanderreihung** bei der Zuweisung festgelegt werden.

✔ Hierbei steht ein Komma zwischen den einzelnen Anweisungen.

```
#beispiel {
background:url(topleft.jpg) top left no-repeat,
           url(bottomleft.jpg) bottom left no-repeat,
           url(bottomright.jpg) bottom right no-repeat,
           url(topright.jpg) top right no-repeat;
}
```

Im oberen Beispiel werden einem Container **vier Grafiken** für die vier Ecken zugewiesen.

7.7. CSS3 Gradients

Anstelle einer Hintergrundgrafik oder einer Farbe kann einer Box auch die Definition eines **Farbverlaufs** (Gradient) als Hintergrund zugewiesen werden.

✔ Dies ist besonders für **Mobilanwendungen** sinnvoll, da so plastisch wirkende Oberflächen (Buttons) erzeugt werden können, ohne Grafiken zu Hilfe zu nehmen.

7.7.1. CSS 3 Linear Gradient

Relativ überschaubar ist die Erzeugung eines einfachen, also „linearen“ Farbverlaufs. Hierfür wird (W3C, Mozilla) die Ursprungskante der Box genannt, sowie Ausgangs- und Zielfarbe.

Dieses Beispiel erzeugt einen linearen Farbverlauf.

```
#beispiel {
  /* Firefox */
  background:
    -moz-linear-gradient(top, #f50 0%, #fff 100%);

  /* Webkit, alt (Safari bis 4.3) */
  background:
    -webkit-gradient(linear, left top, left bottom,
                     from(#f50), to(#fff));

  /* Webkit, neu (Safari ab 5,1, Chrome) */
  background:
    -webkit-linear-gradient(top, #f50 0%, #fff 100%);

  /* nach W3C-Standard: */
  background:
     linear-gradient(to bottom, #000 0%, #fff 100%);
}
```

✔ Ältere Webkit-Browser handhaben die Syntax anders – hier muss zunächst der Gradienten**typ** angegeben werden, anschließend Ausgangs- und Zielpunkt sowie Start- und Endfarbe. Auch die Eigenschaft selbst heißt hier nur `gradient` und nicht `linear-gradient`.

Neuere Webkit-Browser (Safari ab 5.1 und Chrome ab Version 10) unterstützen auch die „normale“ Syntax – analog zu Firefox und dem W3C-Standard (eben nur mit Präfix).

Inzwischen kann man unbesorgt den W3C-Standard verwenden, was das Beispiel reduziert auf:

```
#beispiel {
  background:
     linear-gradient(to bottom, #000 0%, #fff 100%);
}
```

7.7.2. CSS 3 Radial Gradient

Ebenfalls möglich, aber unbequemer in der Handhabung, sind **radiale Farbverläufe**. Hier muss ein Ursprungspunkt gesetzt werden, ab dem der Verlauf gerechnet werden soll.


```
#beispiel {
  background:
      radial-gradient(50% 50%, farthest-side,
                           #f50, #fff);
}
```

Tool für CSS-Gradienten

Ein Online-Tool zur Erstellung browserübergreifender Gradienten ist der *Ultimate CSS Gradient Generator*:

- http://www.colorzilla.com/gradient-editor/

7.8. CSS3 Text Shadow

Eher zum dezenten Einsatz, möglicherweise aber für Überschriften einsetzbar, bietet sich in Form von **CSS-Textschatten** ein weiteres Gestaltungsmittel an.

✔ Die Eigenschaft entspricht in ihrer Handhabung dem Box-Shadow:
`text-shadow: xPos yPos blurRad color;`


```
h1#nr1 {
  text-shadow: 5px 5px 5px #aaa;
}

h1#nr2 {
  color:#900;
  text-shadow: 8px 8px 3px #e88;
}

h1#nr3 {
  text-shadow: 1px 1px 5px #fff;
  background-color:#000;
}

h1#nr4 {
  color:white;
  text-shadow: 2px 2px 8px #555;
}
```

7.9. CSS3 Embedding @font-face

Ein lange gewünschtes, auch bereits seit langem konzipiertes, jedoch nicht gleichmäßig bzw. erfolgreich implementiertes, Feature betrifft das Einbetten sogenannter **Webfonts**.

In CSS3 kommt das bereits für CSS2 vorgesehene `@font-face` quasi in die „Neuvorlage“.

✔ Mit `@font-face` muss die gewünschte **Schriftdatei** eingebettet werden, bevor sie eingesetzt wird. (Beachten Sie in jedem Fall auch die *Copyrights*, die gegebenenfalls auf den Schriftdateien liegen!)

Hier ein Beispiel mit der Einbettung eines True Type Fonts (.ttf):

```
@font-face {
  font-family: Jacinto;  src: url('jacinto.ttf');
}

#beispiel h1 {
  font-family: Jacinto; font-size: 3.2em;
}
```

Auf die so erzeugten Texte können auch andere CSS-Eigenschaften, wie Textschatten angewendet werden:

```
h1#nr2 {
  font-family: Jacinto;
  font-size: 3.2em; color:#900;
  text-shadow: 8px 8px 3px #e88;
}
```

Auch andere Formate stehen zur Verfügung, wie **Opentype** (.otf), **Embedded Open Type** (.eot) oder **SVG-Fonts** (.svg, .svgz).

✔ Denken Sie beim Einsatz in Überschriften daran, das `font-weight`-Property gegebenenfalls auf „normal“ zurückzusetzen, um mit der gewählten Schrift auch die gewünschte Optik zu erzielen!

```
@font-face {
  font-family: Ink;
  src: url('ink.otf');
}

#beispiel h1 {
  font-family: Ink;
  font-size: 7em;
  font-weight:normal;
  padding:5px;
}
```

7.10. CSS3 Multiple Columns

Mit CSS3 rückt nun auch **mehrspaltiger Satz** in erreichbare Nähe. Hierbei ist nicht direkt mehrspaltiges Layout gemeint, sondern vielmehr die Definition einer Textbox, die sich über mehrere Spalten erstreckt und ihren Inhalt bei Bedarf in die Nachbarspalte überfließen lässt.

✔ Dies entspricht den Verhalten gegenüber „verlinkten“ Textboxen bei bekannten DTP-Anwendungen wie Quark Express.

7.10.1. CSS 3 Multiple Columns (width)

Die Spalten können per `column-width` über ihre Breite definiert werden. Mit `column-gap` wird der Abstand zwischen den Spalten festgelegt.

```
beispiel  p {
 text-align: justify;
 column-width: 15em;
 column-gap: 2em;
}
```

Hierbei richtet sich die Anzahl der Spalten nach dem im Fenster des Useragents verfügbaren Platz:

…ist das Browserfenster schmaler, werden **weniger Spalten** eingesetzt, damit deren vorgeschriebene Breite eingehalten wird:

7.10.2. CSS 3 Multiple Columns (count)

Anstelle einer Spaltenbreite kann über `column-count` auch eine **Spaltenzahl** genannt werden. Hier wird über `column-rule` auch noch eine vertikale Trennlinie eingefügt. Deren Eigenschaftsdefinition gleicht der einer normalen Border.

```
#beispiel  p {
 text-align: justify;
 column-count: 3;
 column-gap: 1.5em;
 column-rule: 1px solid #aaa;
}
```

In diesem Fall ist es die **Spaltenzahl**, die forciert wird.

Steht in der Horizontalen weniger Platz zur Verfügung, werden die Spalten schmaler und entsprechend länger. Ihre Anzahl bleibt jedoch konstant.

7.11. CSS3 Transform

Einige weitere Darstellungsmöglichkeiten, die CSS3 bietet, werden unter dem Oberbegriff „Transformations“ zusammengefasst.

✔ Dies ist in dem Sinne zu verstehen, dass die Präsentation ausgehend von einer „regulären“ Präsentation (in der Regel die Darstellung im Flow) modifiziert, sprich „transformiert“ wird.

Die umliegenden Elemente verhalten sich zum transformierten Element, als ob dessen Präsentation unverändert gebleiben wäre.

✔ Das Verhalten der Umgebung eines transformierten Elements ähnelt somit derjenigen auf ein **relativ positioniertes Element**, das gegenüber Nachbarelementen de facto seine Ursprungsposition im Flow behauptet, obwohl es verschoben wird.

Die Möglichkeiten der Transformation beinhalten:

- **Translation**
  Das Element wird aus seiner Urprungsposition verschoben (bekommt dabei jedoch nicht den Status „positioniert“).

- **Rotation**
  Das Element wird aus seiner Ursprungslage um seinen geometrischen Mittelpunkt rotiert.

- **Skewing**
  Die Umrissbox des Elements wird parallelperspektivisch verzerrt.

- **Skalierung**
  Die Umrissbox und der Inhalt des Elements werden skaliert (vergrößert oder verkleinert).

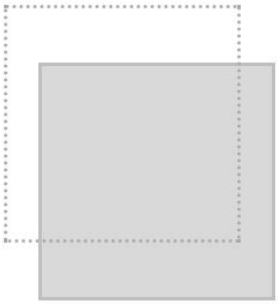
Bis auf die Rotation können alle diese Transformation entweder in zwei Dimensionen (X, Y) erfolgen, oder auf eine Dimension (X oder Y) beschränkt werden.

Es existieren die Funktionen

```
rotate(winkel)

translate(xDiff, yDiff)

translateX(xDiff)
translateY(yDiff)

scale(xFaktor, yFaktor)

scaleX(xFaktor)
scaleY(yFaktor)

skew(xWinkel, yWinkel)

skewX(xWinkel)
skewY(yWinkel)
```

7.11.1. CSS3 Transform – Translate

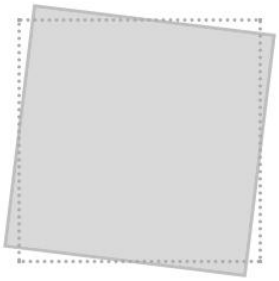
Eine Translation entpricht einer **Verschiebung** der Präsentation des Objekts (Ursprungsposition gepunktet angedeutet):



```
#beispiel  {
  transform: translate(30px, 50px);
}
```

7.11.2. CSS3 Transform – Rotate

Die Rotation “verdreht” die Präsentation des Objekts. Positive Winkel drehen im Uhrzeigersinn, negative Winkel in der Gegenrichtung:



```
#beispiel  {
  transform: rotate(7deg);
}
```

✔ Eine Rotation um den geometrischen Mittelpunkt der Box ist Default. Der Rotationsmittelpunkt kann allerdings mittels `transform-origin` geändert werden)

7.11.3. CSS3 Transform – Skew

Die `skew()`-Methode verzerrt das Objekt, indem parallele Seitenlinien der Umrissbox um den Winkelbetrag gekippt werden. Die Mitte der Seitenlinie bildet den Kippmittelpunkt. Für ein Skewing auf der X-Achse wird die Y-Achse gekippt und umgekehrt.

Für das hier gezeigte **X-Skewing** wird die Y-Achse um 25 Grad *im* Uhrzeigersinn verdreht (nicht etwa entgegengesetzt; da die Y-Achse nach *unten* gerichtet ist!):

```
#beispiel  {
  transform: skew(-25deg);
}
```

Eigentlich werden hier zwei Werte (für X- und Y-Achse) übergeben, wobei der y-Wert implizit 0 ist, hier also entfallen darf.

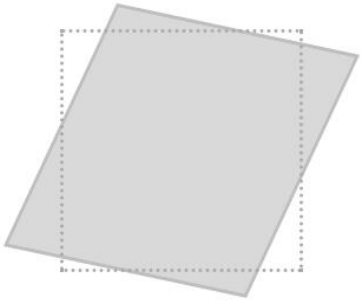
Die explizite Übergabe eines Einzelwertes für X geschieht durch die Methode `skewX()`:

```
#beispiel  {
  transform: skew(-25deg);
}
```

Ein **Y-Skewing** mit demselben Wert hat folgende Wirkung (hier wird die X-Achse gedreht – diesmal gegen den Uhrzeigersinn):

```
#beispiel  {
  transform: skewY(-25deg);
}
```

Wird die `skew()`-Methode mit zwei Werten eingesetzt, so erhält man Ergebnisse wie das folgende:

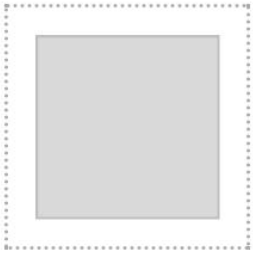


```
#beispiel  {
  transform: skew(-25deg, 12deg);
}
```

7.11.4. CSS3 Transform – Scale

Mit der `scale()`-Methode wir ein Objekt **skaliert**. Wird nur ein Wert übergeben, so erfolgt eine proportionale Skalierung (d.h der nicht übergebene y-Wert wird gleich dem x-Wert gesetzt).

✔ **Achtung:** Die Skalierung setzt am Mittelpunkt an. Der Ursprung der Box (linke obere Ecke) bleibt deshalb nicht an Ort und Stelle!



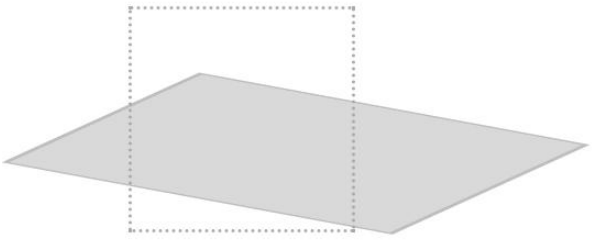
```
#beispiel  {
  transform: scale(0.75);
}
```

Werden `scale()` zwei Werte übergeben, so bezeichnet der erste die Skalierung in X-Richtung, der zweite diejenige in Y-Richtung. Beachten Sie, dass die **Borderbreite mitskaliert** wird:

```
#beispiel  {
  transform: scale(1.75, 0.5);
}
```

7.11.5. Mehrfach-Transformationen

Auf ein Objekt können auch mehrere Transformationen gleichzeitig – das bedeutet im Grunde: nacheinander – angewandt werden, indem der `transform`-Eigenschaft eine **Reihe** von Methoden übergeben wird.



```
#beispiel  {
   transform: rotate(7deg)
              scale(1.75, 0.5)
```

```
                translate(30px, 50px)
                skew(-25deg, 12deg);
}
```

Zwischen den Translationsschritten steht kein Komma!

8. Resets

Eine Reihe von HTML-Elementen, darunter auch der Textabsatzcontainer <p> sowie alle Überschriftencontainer, besitzen vordefinierte Außenabstände (*margins*), die für einen Abstand zum Vorgänger- oder Folgecontainer sorgen.

Sofern Sie diese Grundeigenschaften nicht beibehalten, sondern individuell festlegen möchten, ist es denkbar, diese zunächst global auf Null zu setzen. Das gleiche gilt für Innenabstände (*paddings*), die für einige Elemente in der Darstellung einiger Browser ebenfalls ungleich Null sind.

8.1. Nullung der Default-Margins

Für eine globale Nullung der Default-Margins der HTML-Präsentation setzen Sie folgende CSS-Anweisung an den Beginn Ihres Stylesheets:

```
*  {
  margin: 0;
  padding: 0;
}
```

Der Stern, der hier als Selektor dient, ist ein Platzhalter für einen beliebigen Typ-Selektor, erfasst also alle Elemente ungeachtet ihres Bezeichners. Die so erfassten Elemente sind in der folgenden Tabelle aufgelistet.

Element	**Defaultmargin**	**Element**	**Defaultmargin**
<body>	8px	<p>	1.12em 0
<h1>	0.67em 0	<blockquote>	1.12em 40px
<h2>	0.75em 0	<form>	1.12em 0
<h3>	0.83em 0	<fieldset>	1.12em 0
<h4>	1.12em 0	,	1.12em 0 1.12 em 40px

<h5>	1.5em 0 1	<menu>, <dir>	1.12em 0 1.12 em 40px
<h6>	1.67em 0	<dl>	1.12em 0 1.12 em 40px

Defaultmargins gemäß W3C-Spezifikation (tatsächliche Umsetzung variiert)

✔ Beachten Sie, dass Sie nun selbst dafür Sorge tragen müssen, dass im Rahmen des Layouts erforderliche Abstände definiert werden. (Im Gegenzug kommen Ihnen allerdings nicht unvermuteterweise Defaulteigenschaften ins Gehege.)

Die Nullung der Paddings vereinheitlicht das Verhalten der verschiedenen Browser gegenüber den Elementen – laut den offiziellen Spezifikationen existierten zwar keine Default-Paddings (obwohl sich `<body>` und `<td>` sowie einige weitere Elemente browserspezifisch so verhalten), es wird jedoch in keinem Fall Schaden angerichtet.

8.2. Universeller Reset

Berücksichtigt man außer Margin und Padding vernünftigerweiser auch noch weitere Eigenschaften mit Defaults, so gelangt man beim Versuch einer Vereinheitlichung über die Browserlandschaft bei etwas folgendem Ansatz:

```
* {
  vertical-align: baseline;
  font-weight: inherit;
  font-family: inherit;
  font-style: inherit;
  font-size: 100%;
  border: none;
  padding: 0;
  margin: 0;
}
```

Zusätzlich könnte (aber Vorsicht!) auch das `outline`-Property genullt werden. Dies geht jedoch zu Lasten der Barrierefreiheit.

8.3. Defaults für Überschriften, Absätze und Tabellen

Ohne in allen Dingen ins Detail zu gehen, genügen oft eine Handvoll Regeln, um die Lage bezüglich der meistverwendeten Elemente zu klären.

```
body {
  padding: 5px;
```

```
}

h1, h2, h3, h4, h5, h6,
p, pre, blockquote, form, ul, ol, dl {
  margin: 20px 0;
}

li, dd, blockquote {
  margin-left: 40px;
}

table {
  border-collapse: collapse;
  border-spacing: 0;
}
```

Die oben gelisteten Regeln, sowie der davor gezeigte universelle Reset gehen auf die Arbeit von *Chris Poteet* zurück.

✔ Siehe: http://www.siolon.com/blog/browser-reset-css/

8.4. Einsatz von Normalisierungsstyles

Eine bequemere Praxis besteht darin, ein bewährtes **Normalisierungs-Stylesheet** einzusetzen. Hier bieten sich eine Reihe verschiedener Ansätze an. Die Auswahl ist im Grunde Geschmackssache.

8.4.1. NormalizeCSS

Am beliebtesten ist **NormalizeCSS** von *Nicolas Gallagher*, das auch im Rahmen von PureCSS, Bootstrap oder als Teil des HTML5 Boilerplate-Projekts eingesetzt wird:

✔ Siehe: `http://necolas.github.com/normalize.css/`

8.4.2. ResetCSS

Historisch älter, aber gleichwertig ist **ResetCSS** von *Eric Meyer*, der auch federführend bei der Erstellung der CSS-Spezifikationen war:

✔ Siehe: `http://meyerweb.com/eric/tools/css/reset/`

8.4.3. Tantek's CSS Reset

Ein anderer bekannter Name in der CSS-Welt ist *Tantek Celik*, auf den (seinerzeit noch erforderlich) eine Reihe bekannter IE-CSS-Hacks zurückgehen. Auch wirft seinen CSS-Reset in den Ring

✔ Siehe: `http://tantek.com/log/2004/undohtml.css`

9. Layoutmodelle

Der Browser beachtet beim Rendering des HTMLs mittels CSS verschiedene **Layoutmodi**. Ein wichtiger Teil hiervon ist der Flowmodus, in dem wir anhand des Renderings Block- und Inlineelemente unterscheiden, sowie Elemente die nicht gerendert werden. Auch für Tabellen und Listen existieren Renderingverhalten.

Der Kernpunkt an dieser Stelle ist, dass das Renderingverhalten einem Element gegenüber nicht in Stein gemeisselt ist, sondern durch das display-Property über CSS beschrieben und gesteuert werden kann.

display

CSS-Level	CSS 1; erweitert in CSS 2 und CSS 3
Werte	none, inline, block, inline-block, flex, inline-flex, grid, inline-grid, table, inline-table, table-header-group, table-footer-group, table-column-group, table-column, table-row-group, table-row, table-cell, table-caption, list-item, run-in, inherit

Das `display`-Property legt mit einer großen Anzahl wählbarer Werte die Art und Weise der Darstellung eines Elementcontainers fest.

✔ Hierbei können auch grundsätzliche Eigenschaften von HTML-Elementen, wie deren Darstellung als Block- oder Inlineelemente überschrieben werden.

Hier die vier wichtigsten Werte:

- `display: none`

Die Anzeige des Elements wird **vollständig unterdrückt** und dieses aus dem Flow genommen. Nachfolgende Inhalte rücken an die frei gewordene Position nach.

`display: block` (Default für Blockelemente)

Das Element erhält die Eigenschaften eines **Blockelements**. Dies bewirkt einen Zeilenumbruch vor und nach den Containergrenzen und die vollständige Anwendbarkeit des CSS-Boxmodells.

`display: inline` (Default für Inlineelemente)

Das Element erhält die Eigenschaften eines **Inlineelements**.

`display: inline-block`

Das Element unterstützt das komplette Block-Box-Modell, behält aber gewisse Eigenschaften eines Inlineverhaltens, indem es keinen vorausgehenden und nachfolgenden Laoutumbruch fordert.

9.1. Boxmodell

✔ Gilt für Elemente mit Block- oder Inlineeigenschaften, solange sie im Flow sind (d.h. nicht gefloated oder positioniert sind).

Das CSS-Boxmodell ist im Besonderen auf Blockelemente anwendbar (als „Block-Box“) – für Inlineelemente (als „Inline-Box“) ist es nur eingeschränkt gültig (hier gelten Top- und Bottom-Margin nicht und Padding hat keinen Einfluss auf die Boxhöhe).

Aus der Sicht von CSS wird jeder Blockcontainer als „Box“ betrachtet, die in vier Bereiche (Inhalt, Padding, Border, Margin) eingeteilt werden kann.

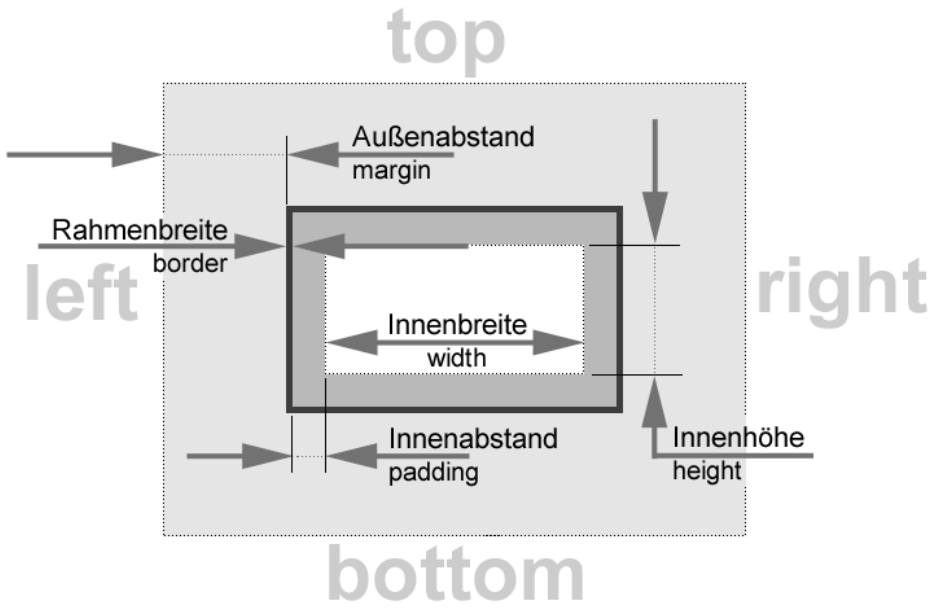


Abb.: Das CSS-Boxmodell für Blockelemente

9.1.1. Containerinhalt

Die Größe dieses Bereichs ergibt sich aus den Abmessungen des Inhalts, beispielsweise also aus Textmenge, Schriftgröße und Zeilenhöhe, wie sie aus fonttypischen Ober- und Unterlängen resultiert.

✔ Ansonsten können die Innenabmessungen mit `width` und `height` explizit festgelegt werden, wobei dies Vorrang vor Anforderungen des Inhalts besitzt (dieser kann im Endergebnis aus dem Container hinausragen!).

height

CSS-Level	CSS 1
Werte	[laenge], [prozent], auto, inherit

Das `height`-Property bestimmt die Innenhöhe eines Elements. Die hier übergebene Abmessung bezieht sich auf den **inneren Bereich** der Box – eventuell angegebene Margins, Border und Paddings erhöhen den Platzbedarf des Gesamtgebildes entsprechend.

`height: auto` (Default)

setzt die Innenhöhe auf den aufgrund Höhe des Inhalts errechneten Wert. Die Box passt sich an den Inhalt an; es tritt kein Overflow auf.

`height: [laenge]`

setzt die Innenhöhe auf den übergebenen Wert, auch wenn der Inhalt geringere oder größere Höhe besitzt. Ist der Inhalt höher, so ragt er nach unten aus dem Containerbereich heraus und es tritt ein Overflow auf.

`height: [prozent]`

setzt die Innenhöhe unabhängig vom Inhalt auf den übergebenen Prozentwert, der sich an der *Höhe des Elternelements* misst.

```
<p style="height:25px; border-style:solid;
 border-width:1px;border-color:black;">
Lorem ipsum... </p>

<p style="height:50px; border-style:solid;
 border-width:1px;border-color:black;">
Lorem ipsum .... </p>
```

width

CSS-Level	CSS 1
Werte	[laenge], [prozent], auto, inherit

Das `width`-Property bestimmt die Innenbreite eines Elements. Die hier übergebene Abmessung bezieht sich, analog zum `height`-Property ebenfalls auf den **inneren Bereich** der Box. Margin, Border und Padding erhöhen die Gesamtbreite entsprechend.

`width: auto` (Default)

setzt die Innenbreite auf den durch die Breite des Inhalts erzwungenen Wert, bzw. die Gesamtbreite auf die durch das Elternelement zur Verfügung gestellte Breite und bricht Textinhalt entsprechend um.

`width: [laenge]`

setzt die Innenbreite auf den übergebenen Wert und bricht Textinhalt entsprechend um. Ist die Höhe nicht mit `“auto”` festgelegt und ist der Inhalt höher als die zur Verfügung gestellte Höhe, so ragt er nach unten aus dem Containerbereich heraus (sofern nicht `overflow:hidden` angegeben ist).

✔ **Overflow nach rechts:**
Wegen des automatischen Umbruchs ragen nur nicht-teilbare Elemente (Grafiken, längere Einzelworte) seitlich aus einem Container beschränkter Breite heraus. Um den automatischen Umbruch für Text zu unterdrücken könnte jedoch `white-space: nowrap;` gesetzt werden.

```
width: [prozent]
```

setzt die Innenbreite auf den übergebenen Prozentwert, der sich an der Breite des durch das *Elternelement* zur Verfügung gestellten Raums misst und bricht Textinhalt entsprechend um.

```
<p style="width:200px; height:50px; border-style:solid;
 border-width:1px;border-color:black;">
 Lorem ipsum ...</p>
```

max-height

CSS-Level	CSS 2
Werte	[laenge], [prozent], none, inherit

Das `max-height`-Property bestimmt die maximale *Höhe*, die der **Innenbereich** des Containers dynamisch in Abhängigkeit vom zur Verfügung stehenden Raum oder der Höhe der Inhalte annehmen darf. Überschreitet die Menge der Inhalte das Maß, tritt ein Overflow auf.

max-width

CSS-Level	CSS 2
Werte	[laenge], [prozent], none, inherit

Das `max-width`-Property bestimmt die maximale *Breite*, die der **Innenbereich** des Containers dynamisch in Abhängigkeit vom zur Verfügung stehenden Raum annehmen darf.

min-height

CSS-Level	CSS 2
Werte	[laenge], [prozent], inherit

Das `min-height`-Property bestimmt die minimale *Höhe*, die der **Innenbereich** des Containers nicht unterschreiten darf.

min-width

CSS-Level	CSS 2
Werte	[laenge], [prozent], inherit

Das min-width-Property bestimmt die minimale *Breite*, die der **Innenbereich** des Containers nicht unterschreiten darf.

9.1.2. Padding – Leerraum um den Inhalt

Der `padding`-Bereich (Innenabstand) entspricht dem Cellpadding in HTML-Tabellenzellen, gehört also zum inneren Bereich, bzw. Hintergrund des Elements. Die Maße des Paddings sind für alle vier Seiten (`top`, `right`, `bottom`, `left`) getrennt wählbar.

padding

CSS-Level	CSS 1
Werte	[laenge], [prozent], inherit

Das `padding`-Property legt den Innenabstand des Textbereichs eines Containers von seiner (gedachten) Grenze fest, das so genannte „Padding“.

Es können zwischen *einem* und *vier* Werten übergeben werden, wobei sowohl Längen- als auch Prozentangaben wie auch eine Mischung aus beidem möglich sind. Prozentangaben beziehen sich auf die Breite des Elternelements.

Achtung: Negative Werte werden als 0 interpretiert!

```
padding: [wert]
```

Wird **ein Wert** übergeben, so bezieht sich dieser auf *alle vier* Seiten des Containers und weist allen den gleichen Wert zu.

```
padding: [wert] [wert]
```

Werden **zwei Werte** übergeben, so bezieht sich der erste auf das *obere und untere* Padding, der zweite auf das Padding *links und rechts*.

```
padding: [wert] [wert] [wert]
```

Die Übergabe von **drei Werten** legt mit dem ersten Wert das *obere*, mit dem zweiten das *linke und rechte*, sowie mit dem *dritten das untere* Padding fest

```
padding: [wert] [wert] [wert] [wert]
```

Vier Werte legen, in dieser Reihenfolge, das *obere*, das *rechte*, das *untere* und das *linke* Padding fest (im Uhrzeigersinn ab oben).

```
p { padding: 1em; }

p { padding:0.2em 1.5em }

p { padding:1em 0.5em 0.5em 2em }
```

9.1.3. Border – ein Rahmen um den Innenraum

Das `border`-Property definiert einen Rahmen um den Innenraum (Containerinhalt mit Padding), der selbst ebenfalls noch zum Inhaltsbereich zählt. Breite, Farbe und Erscheinungsbild des Rahmens sind für alle vier Seiten getrennt wählbar.

border

CSS-Level	CSS 1
Anwendung	Blockelemente; Inlineelemente nur bedingt
Werte	[border-style] [border-width] [border-color], inherit

Das Sammelproperty `border` erhält (in *nicht festgelegter* Reihenfolge) drei Angaben zu Aussehen (`border-style`), Breite (`border-width`) und Farbe (`border-color`) des Rahmens.

Die drei übergebenen Werte wirken gleichmäßig auf alle vier Seiten des Rahmens.

```
<p style="border:solid 1px black">
Lorem ipsum ...</p>

<p style="border:solid 4px #dddddd">
Lorem ipsum ...</p>
```

```
<p style="border:dotted 1px black">
Lorem ipsum ...</p>

<p style="border:dotted 8px #aaaaaa">
Lorem ipsum ...</p>
```

9.1.4. Margin – der Leerraum um das Element

Der Marginbereich (Außenabstand) bestimmt den Abstand zu Nachbarcontainern. Der Außenabstand kann für jede Seite einzeln festgelegt werden. Er zählt *nicht* zum Inhaltsbereich des Containers, ist also nicht über das `background-color`-Property einzufärben. (Vielmehr „erbt“ er die Hintergrundfarbe des umgebenden Blocks.)

✔ **Nochmals:** Die Bereiche „Containerinhalt“, „Padding“ und „Border“ bilden zusammen den *Inhaltsbereich* des Containers, auf den Hintergrundfarben und -grafiken angewendet werden.

Im normalen Flowlayout findet ein „collapsing“ der Margins benachbarter Elemente statt. Das kleinere Margin geht hierbei im größeren auf, sodass der Abstand zwischen den Boxen der beiden Elemente stets dem größeren Margin entspricht.

margin

CSS-Level	CSS 1
Werte	[laenge], [prozent], inherit

Das `margin`-Property legt den Außenabstand des Inhaltsbereichs eines Containers fest. Die Randbedingungen entsprechen denen bei Padding, auch was die Anzahl der zu übergebenden Werte angeht.

Achtung: Marginwerte dürfen auch **negativ** sein und führen auf der entsprechenden Seite gegebenenfalls zu einer Überlagerung mit dem Nachbarcontainer.

✔ Mögliche **Defaultwerte** sind elementabhängig. Textabsätze und Überschriften besitzen beispielsweise einen Defaultmargin nach unten.

```
p { margin:0px; }
```

```
p { margin:0.2em 1.5em; }

p { margin:-1.2em 0em 0.5em; }
```

9.1.5. Sichtbarkeit

visibility

CSS-Level	CSS 2
Werte	visible, hidden, collapse, inherit

Das `visibility`-Property steuert die Sichtbarkeit eines Elementcontainers. Dieser nimmt, unabhängig von den vorgenommenen Einstellungen, stets Raum an der ihm angewiesenen Position ein (d.h. das Element bleibt im Flow).

✔ Für Teiltransparenz verwenden Sie stattdessen `opacity`!

- **visibility: visible**
  Das Element ist explizit auf “sichtbar” gesetzt, wird daher also angezeigt.

✔ *Hinweis:* Dies gilt auch dann, wenn das Elternelement auf “hidden” gesetzt ist. Die Hintergrundeigenschaften des versteckten Elternelements vererben sich in diesem Falle *nicht* an das sichtbare Kindelement (auch bei Inlineelementen).

- **visibility: hidden**
  Das Element ist explizit auf „Verstecken“ gesetzt, wird daher also verborgen. Hinter dem Element befindliche Inhalte oder Hintergründe werden sichtbar; das Element nimmt jedoch den ihm zugewiesenen Raum ein und beeinflusst so, falls es nicht absolut positioniert ist, den Flow von Nachfolgeelementen.

✔ Auf Tabellenelemente angewendet, bewirkt die Einstellung “hidden” nur das Verstecken der Zelleninhalte, nicht jedoch von Teilen der Tabellenstruktur. Verwenden Sie hierfür “collapse”

opacity

CSS-Level	CSS 3
Werte	[numeric, 0 - 1]

Das `opacity`-Property steuert die Transparenz eines Elementcontainers. Hierbei bedeutet der Wert 0 Volltransparenz, der Wert 1 volle Opazität (Default). Der Wert bezieht sich sowohl auf die Inhalte, als auch auf Hintergrund, Border oder Box-Shadow des betreffenden Containers.

✔ Das Element bleibt im Flow, auch wenn es augeblendet ist.

10. Float und Position

Durch **Positionierung** oder **Floating** kann ein Element dem normalen Flow entnommen werden. Floating führt dazu, dass ein gefloatetes Element an den Rand seines umliegenden Containers (des Elternelements) verschoben wird.

✔ Für Floater findet **kein Collapsing der Margins** statt.

Absolute und fixierte Positionierung entnimmt das betreffende Element (für absolute oder fixe Positionierung) dem Flow. Ein absolut positioniertes Element kann nun frei über dem Layout platziert werden (gegenüber seinem Bezugscontainer) und dieses ggfs. überlagern. Ein fixiertes Element wird am Viewport orientiert.

✔ Margins absolut und fixiert positionierter Element collapsen nicht.

Ein relativ positioniertes Element verbleibt hingegen im Flow und behält sein vom Flow her bekanntes Marginverhalten. Es ist anschließend (in seinen Flowabmessungen) frei im Layout verschiebbar.

10.1. Floating Properties

Das Floaten wird durch nur zwei Properties bestimmt, nämlich `float`, das Floatverhalten auslöst oder unterbindet und `clear`, das das Verhalten von Folgeelemente (laut Quelltextreihenfolge) gegenüber Floatern regelt.

float

CSS-Level	CSS 2
Werte	left, right, none, inherit

Das `float`-Property bewirkt das Umfließen eines Elements durch Folgeelemente und nimmt das gefloatete Objekt dabei aus dem Flowkontext. Es verliert dabei seine implizite (Flow-)Breite. Auch seine Margins kollapsen nicht mehr in die Nachbarmargins.

✔ Das gefloatete Element wird auf seiner gesamten Höhe von Folgeinhalten umflossen, sofern keiner der Folgecontainer vor Ende des Floatobjekts ein `clear`-Property nach der entsprechenden Seite besitzt.

- **float:none** (Default)
  Das Element wird nicht gefloatet.

- **float:left**
  Das Element wird nach links gefloatet und von folgenden Inhalten auf der rechten Seite umflossen.

- **float:right**
  Das Element wird nach rechts gefloatet und von folgenden Inhalten auf der linken Seite umflossen.

clear

CSS-Level	CSS 2
Anwendung	Blockelemente
Werte	none, left, right, both, inherit

Das `clear`-Property steht in Zusammenhang mit gefloateten Elementen. Es beendet für den formatierten Elementcontainer das Umfließen eines

vorhergehenden, mittels des `float`-Properties nach links oder rechts gefloateten Containers zur angegebenen Seite hin.

- **clear: none** (Default)
  Das Umfließen des Vorgängercontainers wird nicht unterbrochen.

- **clear: left**
  Das Rechtsumfließen eines mit `float:left` formatierten Vorgängercontainers wird unterbrochen; der Container bricht nach der nächsten freien Position am linken Rand des Elternelements um. Mit `float:right` formatierte Vorgänger werden nach wie vor links umflossen.

- **clear: right**
  Das Linksumfließen eines mit `float:right` formatierten Vorgängercontainers wird unterbrochen; der Container bricht nach der nächsten freien Position am rechten Rand des Elternelements um. Mit `float:left` formatierte Vorgänger werden nach wie vor rechts umflossen.

- **clear: both**
  Das Rechts- oder Linksumfließen eines mit `float:left` oder `float:right` formatierten Vorgängercontainers wird unterbrochen; der Container bricht nach der entsprechend nächsten freien Position am linken oder rechten Rand des Elternelements um.

10.2. Layoutbeispiel mit Float

Füre dieses und die folgenden Layoutbeispiel arbeiten wir mit einer HTML-Struktur wie folgt:

```
...
<div id="wrapper">
   <div id="header">Header</div>
   <div id="contentwrapper">
      <div id="spalte1"> ... </div>
      <div id="spalte2"> ... </div>
      <div id="spalte3"> ... </div>
   </div>
   <div id="footer">Footer</div>
</div>
...
```

✔ Ein spezieller **Contentwrapper** ist nicht in jedem Falle erforderlich, aber allgemein nützlich. Bereits beim Position-Layout bedeutet er eine wesentliche Erleichterung, für das Flexlayout wird er unbedingt benötigt.

Mit diesem HTML-Setup kann ein Floatlayout realisiert werden, indem den Spaltencontainern `spalte1` bis `spalte3` das `float`-Property hinzugefügt wird. So ist auch eine Sortierung der Spalten im Viewport möglich (Hier 3-2-1). Allerdings gibt es eine *Nebenwirkung* auf das restliche Layout.

```
#spalte1 {
   background-color: lightblue;
   width: 25%;
   float:right;
}

#spalte2 {
   background-color: lightgreen;
   width: 40%;
   float:right;
}

#spalte3 {
   background-color: pink;
   width: 35%;
   float:left;
}
```

Es ist notwendig, das Verhalten der *Folgeelemente* (hier ist es der Footer) zu regeln, damit kein **Überfließen** durch die Floater erfolgt. Hierfür muss **clear** eingesetzt werden:

```
#footer {
   background-color: #ccc;
   height: 40px;
   clear:both;
}
```

✔ Ein Nachteil des Floarlayouts ist die **fehlende Kopplung der Spaltenlängen**. Jeder Spaltencontainer bestimmt seine Höhe anhand seiner Inhalte unabhängig von den Nachbarspalten.

Dies kann optisch ausgeglichen werden durch Hintergrundeigenschaften (Farben oder Gradienten) am Contentwrapper. Damit dieser die

enthaltenen Floater jedoch als Inhalt *erkennt* (sonst hätte er Höhe 0!), muss er ebenfalls gefloatet werden (und gegebenenfalls eine Breite erhalten):

```
#contentwrapper {
    background-color: yellow;
    border: 3px solid red;
    /* soll Floater-Inhalt bemerken: */
    float: left;
}
```

10.3. Positioning

Die Gruppe der Positioning-Properties fasst alle CSS-Eigenschaften zusammen, die **Positionierung** von Elementcontainern und weitere, damit zusammenhängende Aspekte betreffen.

position

CSS-Level	CSS 2
Werte	static, absolute, relative, fixed, inherit

Das `position`-Property bestimmt die Art, in der Positionsangaben bezüglich des positionierten Elements interpretiert werden.

- **position: static** (Default)
  Positionierung entspricht der des Elements im normalen Dokumentfluss. Angaben zur Positionierung (`left`, `top`, `bottom`, `right`) werden ignoriert.

- **position: relative**
  Positionierung bezieht sich auf die Position des Elements im normalen Dokumentfluss. Angaben zur Positionierung (`left`, `top`, `bottom`, `right`) beziehen sich auf diese **Ursprungsposition**.

- **position: absolute**
  Positionierung mit `left`, `top`, `right` und `bottom` bezieht sich auf einen **Koordinatenursprung**, der *entweder* durch die Kanten des <body>-Elements gebildet wird, oder durch die Kanten eines übergeordneten Elementcontainers, sofern dieser ebenfalls relativ, absolut oder fixiert („fixed“) positioniert ist.

- **position: fixed**
  Der Elementcontainer bleibt beim Scrollen relativ zu seinem Ursprung

fixiert. Die Positionierung mit `left`, `top`, `right` und `bottom` bezieht sich auf die Kanten des Viewports.

10.3.1. Kantenpositionen

Grundsätzlich beziehen sich positionierte Kanten einer Box auf die gleichnamigen Kanten eines **Bezugsrahmens**, also `left` auf dessen linke, `right` auf rechte, `top` auf obere und `bottom` auf untere Kante.

Relativ positionierte Boxen beziehen sich auf ihre *Flowposition*, die sie innehätten, wenn sie nicht positioniert wären. Absolut positionierte Boxen beziehen sich auf den hierarchisch nächsthöheren relativ oder absolut positionierten *Container* oder den Body, fixierte Boxen auf den *Viewport*.

left

CSS-Level	CSS 2
Werte	[laenge], [prozent], auto, inherit

✔ Für diese und die folgenden Eigenschaften können beliebige (positive oder negative) Längenangaben übergeben werden. Prozentwerte beziehen sich auf die Abmessungen des Elternelements.

Bei **relativer Positionierung** bestimmt das `left`-Property den Abstand des linken Randes des positionierten Containers von der Flowposition.

Bei **absoluter Positionierung** bestimmt das `left`-Property den Abstand zwischen der linken Kante des positionierten Elementcontainers und der linken Kante des Bezugs-Containers, der hierfür *absolut* oder *relativ* positioniert sein muss. Ansonsten orientiert sich das positionierte Element am linken Rand des <body>-Elements.

right

CSS-Level	CSS 2
Werte	[laenge], [prozent], auto, inherit

Das Verhalten entspricht dem von `left`, bezieht sich aber auf die rechte Kante.

✔ Positive Werte bewirken daher einen Abstand nach links!

top

CSS-Level	CSS 2
Werte	[laenge], [prozent], auto, inherit

Das Verhalten entspricht dem von `left`, bezieht sich aber auf die obere Kante.

✔ Positive Werte bewirken daher einen Abstand nach unten!

bottom

CSS-Level	CSS 2
Werte	[laenge], [prozent], auto, inherit

Das Verhalten entspricht dem von `left`, bezieht sich aber auf die untere Kante.

✔ Positive Werte bewirken daher einen Abstand nach oben!

10.3.2. Tiefenstaffelung und Überlagerung

Positionierte Elemente überlagern das Flowlayout grundsätzlich, egal, ob es sich um vorausgehende oder nachfolgende Elemente in Quelltextreihenfolge handelt. Eine Überlagerung positionierter Elemente folgt der Quelltextreihenfolge, wobei die nachfolgenden die vorausgehenden Elemente überlagern. Enthält eine positioniert Box weitere positionierte Boxen, so „erben“ sie deren Tiefenstaffelung. Untereinander verhalten sie sich wie oben.

Soll von der impliziten Tiefenstaffelung abgewichen werden, so kann ein `z-index`-Wert vergeben werden.

z-index

CSS-Level	CSS 2
Anwendung	positionierte Elemente
Werte	[numerisch], auto, inherit

Das `z-index`-Property bestimmt die Tiefenstaffelung sich überlagernder, positionierter Elemente. Es können ganze Zahlen oder der Wert „auto“ übergeben werden.

- **z-index: auto** (Default)
  Die Überlagerung von hinten nach vorn entspricht der Folge der Elemente in Quelltextreihenfolge.

- **z-index: [numerisch]**
  Es können beliebige ganze Zahlen übergeben werden. Das Element mit dem höchsten `z-index` steht in der Tiefenstaffelung am weitesten vorne. Wird an zwei Elemente der gleiche `z-index` vergeben, so entscheidet die Quelltextreihenfolge über die Tiefenstaffelung.

✔ Die Abstände zwischen den vergebenen z-Indexen können beliebig sein.

10.4. Layoutbeispiel mit Position

Auch mit Positionierung ist das dreispaltige Layout zu erzielen. Generell wird man jedoch das stabilere Flowlayout vorziehen, das ein Positionslayout Einschränkungen über die Menge der Spalteninhalte mit sich bringt: Es muss feststehen, welche Spalte die höchste sein wird – diese muss dann im Flow bleiben. Die anderen beiden Spalten können auf dem freigewordenen Raum absolut positioniert werden.

✔ Es ist erforderlich, einem positionierten Element Raum zu schaffen, da keine Wechselwirkung zum Flowlayout besteht. Ein Überfließen auf das Flowlayout ist von diesem aus nicht zu behandeln.

Hier erhält Spalte 3 (verlässlich) die meisten Inhalte und wird im Flow belassen (sie kann eine `min-height` erhalten, damit das Layout gewährleistet ist, selbst wenn sie leer ist – die positionierten Spalten dürfen in ihrer Höhe jedoch diesen Wert keinesfalls überschreiten). Den Raum links für die absolut positionierte Spalte schaffen wir durch ein Margin; rechts bleibt er von allene frei.

```
#spalte3 {
    background-color: pink;
    width: 35%;
    min-height: 200px;
    margin-left: 25%;
}
```

Spalte 1 und 2 werden absolut positioniert.

```
#spalte1 {
    background-color: lightblue;
    width: 25%;
    position: absolute;
    top: 0;
}

#spalte2 {
    background-color: lightgreen;
    width: 40%;
    position: absolute;
    right: 0;
    top: 0;
}
```

Wichtig ist es, den Contentwrapper durch relative Positionierung als Bezugsrahmen zu deklarieren. Ansonsten funktioniert Breitenberechnung und Positionierung der positionierten Spalten nicht wie erwartet.

```
#contentwrapper {
    /* Bezugscontainer für Spalten! */
    position: relative;
}
```

Derzeit sind die positionoerten Spalten unterschiedlich lang und sicherlich nicht entsprechend der Flowspalte dimensioniert. Eine Möglichkeit besteht jedoch, die positionierten Spalten auf die Höhe der Flowspalte zu strecken. Dies geschieht einfach dadurch, dass außer der Topkante auch die **Bottomkante** an den Bezugscontainer angelegt wird:

```
#spalte1 {
    background-color: lightblue;
    width: 25%;
    position: absolute;
    top: 0;
    bottom: 0;
}

#spalte2 {
    background-color: lightgreen;
    width: 40%;
    position: absolute;
    right: 0;
    top: 0;
```

```
    bottom: 0;
}
```

11. Tablelayout

Das Layoutmodell „Table“ basiert auf dem üblichen Renderingverhalten des Browsers gegenüber HTML-Tabellen. Gegenüber dem Box-Layout zeigt es Unterschiede beim Behandeln von Paddings und Margins.

Das Tabellenverhalten kann jedem HTML-Element einfach über das `display`-Property zugewiesen werden.

- `display:table-cell`
  Definiert das Element als **Tabellenzelle**. Liegen zwei derart definierte Elemente in Quelltextreihenfolge unmittelbar benachbart, so werden sie als Teil einer Tabellenzeiile betrachtet. Das erste Nichttabellenelement in Folge beendet die Tabellendarstellung.

- `display:table-row`
  Definiert das Element als **Zeilencontainer**. Dies macht gegebenenfalls Sinn, wenn Zellencontainer definiert sind, diese aber nicht in einer einzigen Zeile augegeben werden sollen. In diesem Fall MUSS ein Container in der Rolle des TR bestimmt werden und dieser *darf nur* TD-Container enthalten.

- `display:table`
  Definiert das Element als **Tabellencontainer**. Dies mag nötig sein, wenn gewünscht ist, der Tabelle selbst Eigenschaften zu geben (Breite, Spaltenabstände oder Hintergrundinformationen). Ein TABLE-Container darf nur Elemente enthalten, die entweder als TR oder als TD definiert sind.

- `display:inline-table`
  Definiert das Element als Tabellencontainer, aber mit Inline-Verhalten, sodass dieser in einer Textzeile platziert werden könnte. Ansonsten siehe oben.

Anmerkung:
In der Tat fehlt eine Möglichkeit, ein `<th>`-Element zu deklarieren! Hingegen einsetzbar, aber selten in Verwendung, sind Zuweisungen wie `table-caption` (als `<caption>`), `table-header-group` (als `<thead>`), `table-footer-group` (als `<tfoot>`), `table-row-group` (als `<tbody>`) oder `table-column` (als `<col>`) und `table-column-group` (als `<colgroup>`). So lassen sich auch komplexe HTML-Tabellen mit anderen Containertypen nachbauen.

Gegenüber einem auf dem normalen Boxmodell in Zusammenhang mit Floats basierendem Layout bietet das Tabellenlayout eine Reihe von Vorteilen, jedoch bei geringerer Flexibilität, da nicht von der Quelltextreihenfolge der Container abgewichen werden kann.

✔ **Vorteil 1:** Nach innen gerichtetes Padding

Ein tabellenbasiertes Spaltenlayout lässt sich einfacher bemaßen, da die beabsichtigte Breite nicht mit dem (oft erforderlichen) Padding verrechnet werden muss. Das Spaltenpadding (bzw. für eine Tabellenzelle deren Zellenpadding) ist nach innen gerichtet und muss nicht etwa mit `calc()` umständlich von der zugewiesenen Breite subtrahiert werden.

✔ **Vorteil 2:** Zellen einer Zeile sind gleich hoch

Ein tabellenbasiertes Spaltenlayout kennt naturgemäß keine Probleme mit unterschiedlichen Spaltenhöhen. Alle Elemente einer Zeile sind stets gleich hoch. Auch kann kein versehentliches Overflow auftreten (wie bei Positionlayouts) oder Wechselwirkungen mit Floatern.

Als einziger Nachteil bleibt die geringere Flexibilität, da die als Tabellenzeilen auserkorenen Container stets in Quelltextreihenfolge in ihre Zeile gerendert werden. Anders als beim Floatlayout sind Abweichungen von dieser Reihenfolge nicht möglich. Um ein Spaltenlayout zu erzielen, werden die Container einfach zu Tabellenzellen gemacht:

```
#spalte1 {
    background-color: lightblue;
    width: 25%;
    padding: 5px;
```

```
    display: table-cell;
}

#spalte2 {
    background-color: lightgreen;
    width: 40%;
    padding: 5px;
    display: table-cell;
}

#spalte3 {
    background-color: pink;
    width: 35%;
    padding: 5px;
    display: table-cell;
}
```

Definiert man die Spalten im Layoutbeispiel als Tabellenzellen, so konstruiert der Browser virtuelle Instanzen eines Zeilen- und eines Tabellencontainers (als anonyme Boxen) um die Spalten.

Man kann sich dies wie folgt vorstellen:

```
<div id="contentwrapper">
  <!-- TABLE -->
    <!-- TR -->
      <div id="spalte1"> ... </div>
      <div id="spalte2"> ... </div>
      <div id="spalte3"> ... </div>
    <!-- TR -->
  <!-- TABLE -->
</div>
```

Möchte man zusätzliche Kontrolle über das Layout, so kann man dem Contentwrapper die Rolle der Tabelle zuweisen:

```
#contentwrapper {
    display: table-cell;
}
```

In diesem Fall würde folgende Struktur konstruiert:

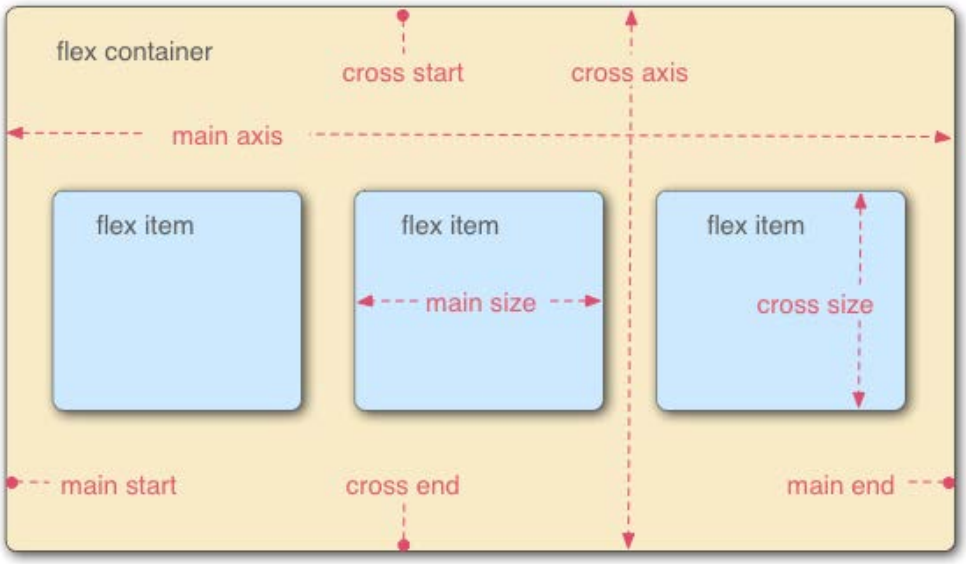
```
<div id="contentwrapper">
  <!-- TR -->
    <div id="spalte1"> ... </div>
    <div id="spalte2"> ... </div>
```

```
    <div id="spalte3"> ... </div>
  <!-- TR -->
</div>
```

12. Flexlayout

Das Flex-Layout verspricht ein wesentliches flexibleres Layoutmodell, als bisher möglich gewesen ist. Hierbei enthält ein **Flexcontainer** sogenannte **Flexitems**, die entweder horizontal (Default) oder vertikal gelayoutet werden können. Flexitems haben (je nach Richtung) automatisch gleiche Höhe bzw. gleiche Breite. Darüberhinaus kann die Layoutreihenfolge der Items beliebig geregelt werden.

Die Lyyoutrichtung wird als **Main Axis** bezeichnet. Die Richtung quer zur Layoutrichtung ist entsprechend die **Cross Axis**. Statt von „links“ und „rechts“ spricht man bei Flexcontainern von **Flex Start** und **Flex End** (für beide Richtung, siehe Grafik).



Wir beschränken uns auf eine Grundbetrachtung des recht umfangreichen Konzepts.

✔ **Die Spezifikation:** http://www.w3.org/TR/css3-flexbox/

Leider ist die Browserunterstützung speziell bei IE noch nicht vollständig gewährleistet. IE10 unterstützt Flex mit Präfix `-ms`, IE11 unterstützt Flex halbwegs, wenn auch mit Bugs. Erst die Edge-Browser unter Windows 10 unterstützen Flex zufriedenstellend. Alle anderen Browser sind flexfähig.

12.1. Der Flexcontainer

Die äußere Box des Flexlayouts wird als **Flexcontainer** bezeichnet. Grundlegend ist auch in diesem Fall das `display`-Property.

`display: flex;` definiert eine Box als Flexcontainer. Genügt oft bereits.

✔ Am Flexcontainer können bei Bedarf eine Reihe weiterer Flexproperties eingesetzt werden. Ansonsten gelten deren Defaultwerte.

flex-direction
Die Eigenschaft **`flex-direction`** (Default: `row`) bestimmt, ob horizontal oder vertikal (`column`) gelayoutet wird. Hierbei ist auch ein Vertauschen der Renderreihenfolge möglich (`row-reverse` bzw. `column-reverse`).

justify-content
Mit der Eigenschaft **`justify-content`** wird das Verhalten geregelt, wenn die Items den Flexcontainer nicht vollständig ausfüllen. Sie können dann z.B. zentriert (`center`), an Anfang oder Ende der Flexbox (`flex-start` (Default) bzw. `flex-end`) oder an beide *Main-Axis-Ränder* des Containers angelegt werden, wobei ein gleichmäßiger Zwischenraum zwischen den Items erzeugt wird (`space-between`). Auch kann der verfügbare Raum um die Items generell gleichmäßig verteilt werden (`space-around`).

align-items
Die Eigenschaft **`align-items`** regelt die Abmessungen der Items „cross-axis“. Im horizontalen Layout ist dies die Höhe. Der Defaultwert `stretch` bringt alle Items auf gleiche Höhe. Alternativ sind die Werte `flex-start`, `flex-end`, `center` oder `baseline` möglich.

flex-wrap
Die Eigenschaft **`flex-wrap`** bestimmt, ob die Items in der Axis-Richtung umbrechen sollen, wenn sie die verfügbare Breite überschreiten (`wrap`) oder nicht (`nowrap`). Der Wert `nowrap` ist Default. In diesem Fall werden die Items so gestaucht, dass sie in die Flexbox passen.

align-content
Mit **`align-content`** wird auf *gewrappte* (mehrzeilig gelayoutete) *Items* eingewirkt, indem deren „Flexlines“ positioniert werden. Hier sind die Werte `center` (Default), stretch, `flex-start`, `flex-end`, `space-between` und `space-around` möglich. Die Wirkung ist vergleichbar der von `justify-content`.

12.2. Die Flexitems

Die Children eines Flexcontainers werden *automatisch* zu **Flexitems** Hier sind im Grunde keine besonderen Maßnahmen erforderlich.

Will man auf die Items jedoch speziell einwirken, so steht eine Reihe von Eigenschaften zur Verfügung.

margin
Die **`margin`**-Eigenschaft hat im Flexkontext besondere Wirkung. So bewirkt `margin:auto` eine horizontale und vertikale Zentrierung innerhalb eines Flexcontainers, da sowohl horizontaler als auch vertikaler Raum um die Items gleichmäßig verteilt wird.

order
Mit **`order`** kann die *Reihenfolge* eines Items gegenüber den anderen Flexitems festgelegt werden. Es wird ein numerischer Wert eingegeben. Die `0` verändert die Position nicht, der Wert `-1` zieht das Item um eine Position nach vorne, der Wert `1` verschiebt es um eine Position nach hinten. Entsprechend verhalten sich die Werte `-2` und `2` usw..

flex
Die Eigenschaft **`flex`** legt die (je nach Achse) Breite oder Höhe eines Items gegenüber den anderen Items fest. Besitzt nur ein Item die Eigenschaft, so wächst es auf die größte mögliche Abmessung (unabhängig vom vergebenen Wert), wohingegen die anderen Items die durch ihre Inhalte vorgegebene Abmessung in Axisrichtung erhalten. Haben alle Items ein `flex`-Property so bestimmt deren **Zahlenverhältnis** das Verhältnis der Abmessungen der Items.

✔ Die `flex`-Eigenschaft hat ggfs. Vorrang vor einem `width`-Property!

align-self
Die Eigenschaft **`align-self`** bestimmt *individuell pro Item*, wie dieses sich im Flexverbund verhält. Sie überschreibt ggfs. die Zueweisung über

`align-items` am Flexcontainer. Mögliche Werte sind `flex-start`, `flex-end`, `center`, `baseline` und `stretch` (Default).

12.3. Layoutbeispiel mit flex

Um mit der vorliegenden HTML-Struktur und der vorgegebenen Breite der Spaltencontainer ein Flexlayout zu erzielen, das dem Tabellenlayout gleichkommt, genügt eine Zuweisung an den Contentwrapper:

```
#contentwrapper {
   display: flex;
}
```

Für die Spalten hat man ähnliche Vorteile wie beim Tabellenlayout. So bricht das Layout nicht beim Zuweisen eines Paddings. Allerdings lässt sich mit Hilfe des `order`-Properties nun die **Spaltenreihenfolge** beliebig festlegen:

```
#spalte1 {
    background-color: lightblue;
    width: 25%;
    padding: 5px;
}

#spalte2 {
    background-color: lightgreen;
    width: 40%;
    padding: 5px;
    order: 1;
}

#spalte3 {
    background-color: pink;
    width: 35%;
    padding: 5px;
}
```

Hier rückt die zweite Spalte um eine Position nach rechts.

13. Transitions und Animations

Anfangs konnten mit CSS nur **statische Zustände** von Präsentationen beschrieben werden. Dies ändert sich mit CSS3, wo in Form von

Transitionen und **Animationen** Übergänge zwischen Zuständen definiert werden können.

✔ **CSS-Transitions:**
http://www.w3.org/TR/css3-transitions/

✔ **CSS-Animations:**
http://www.w3.org/TR/css3-animations/

13.1. CSS-Transitions

Die einfachere Form der Animation legt die Berechnung von Zwischenwerten beim Übergang von einer Ausgangs- zu einer Zielpräsentation fest. Dies wird als **Transition** bezeichnet.

Hierfür wird der `transition`-Befehl als Teil der **Zielpräsentation** eingefügt; meist als Beschreibung, die an eine Zustandsänderung wie `hover` gebunden ist.

Ausgangsszenario: Eine Box besitzt eine definierte Höhe und Breite:

```
div#box {
   width: 100px;
   height: 100px;
}
```

… die beim Hoverzustand verändert wird:

```
div#box:hover {
   width: 150px;
   height: 150px;
}
```

13.1.1. Transition

Bislang hat der Übergang zwischen Normal- und Hoverzustand **unmittelbar**, wie bei einem Schaltvorgang zu erfolgen.

Dies ändert sich, wenn in der Hover-Definition eine Transition-Anweisung platziert wird:

```
div#box:hover {
   transition: all 0.5s ease-in-out;
   width: 150px;
   height: 150px;
}
```

Die Anweisung besagt, dass der Browser die neuen Werte nicht sofort übernehmen soll, sondern ausgehend von den aktuellen Werten schrittweise **Zwischenwerte** berechnet – und zwar für alle (*all*) hier veränderten Properties.

Die Zwischenschritte werden über einen **Zeitraum** von 0,5 Sekunden (*0.5s*) verteilt, und zwar nicht gleichmäßig, sondern nach einer **Easingkurve** gewichtet.

✔ Die Anweisung *ease-in-out* bewirkt ein langsames Ein- und Auslaufen der Animation.

13.1.2. Re-Transition

Wird der Hoverzustand beendet, so stellt der Browser unmittelbar den Ursprungszustand des Elements wieder her.

✔ Möchte man dies nicht, so kann man **auch für den Ursprungszustand** eine Transition definieren, die zu den Ausgangswerten zurückinterpoliert.

Folgende Anweisungen lassen die Box bei Hover erst wachsen und bei Mouseout wieder schrumpfen:

```
div#box {
 /* Re-Transition zu Ausgangszustand: */
   transition: all 0.3s ease-in-out;
   width: 100px;
   height: 100px;
}

div#box:hover {
 /* Transition zu Hoverzustand: */
   transition: all 0.5s ease-in-out;
   width: 150px;
```

```
   height: 150px;
}
```

Hier erfolgt der Übergang in den Ausgangszustand geringfügig schneller über einen Zeitraum von 0,3 Sekunden.

13.1.3. Delay

Mittels eines optionalen vierten Parameters kann eine **Verzögerungszeit** (*Delay*) festgelegt werden.

So tritt die Re-Transition erst eine halbe Sekunde *nach* dem Mouseout auf (kommt gelegen, um beispielsweise das Einklappen eines Slideout-Menüs zu verzögern):

```
div#box {
/* Re-Transition zu Ausgangszustand: */
   transition: all 0.3s ease-in-out 0.5s;
   width: 100px;
   height: 100px;
}
```

13.2. CSS-Animations

Hinaus über die CSS-Transitions gehen die **CSS-Animations**. Eine *Transition* interpoliert nur zwischen zwei Zuständen. Es gibt daher, abgesehen von den Easing-Funktionen, keine Kontrolle über die Zwischenzustände, die vom Browser berechnet werden – beispielsweise indem der Weg eines translatierten Elements anders als linear zwischen Ausgangs- und Endpunkt festgelegt werden könnte.

Mit CSS-Animations erhöhen sich die Freiheiten durch die Option, sogenannte „Keyframes“ zu definieren, die nichts anderes sind als „Schnappschüsse“ von Zwischenzuständen, denen Zeitpunkte während des Animationsverlaufs zugeordnet werden.

Spielen wir ein Beispiel durch – definieren wir zunächst eine Box mit Höhe und Breite sowie roter, teiltransparenter Hintergrundfarbe. Hierfür verwenden wir einen RGBA-Wert:

```
#beispiel {
   height:150px;
   width:250px;
   border:1px red solid;
```

```
   background-color:rgba(255,0,0,0.7);
}
```

✔ Diese Box soll mittels Hover animiert werden, und zwar ihre Abmessungen. Würden wir eine Transition einsetzen, so müssten Sie nun einen `:hover`-Zielzustand definieren und dort einen Transitionbefehl platzieren.

13.2.1. Bindung einer Animation

Dies funktioniert bei Animationen anders. In der Beschreibung des Hoverzustandes wird nicht der Zielzustand abgelegt, sondern lediglich auf einen (anderswo definierte) **Animationsbezeichner** verwiesen. Nennen wir die Animation „resize“.

Hier wird, der Deutlichkeit halber, zunächst der W3C-Bezeichner `animation` verwendet (Später kommen weitere Anweisungen hinzu):

```
#beispiel:hover {
   animation: resize;
}
```

Die Animationsanweisungen, auf die aus dem Hoverzustand verwiesen wird, definiert man im Rahmen einer `@keyframes`-Anweisung.

13.2.2. Die @keyframes-Anweisung

Diese soll wie folgt geschrieben werden (achten Sie darauf, wie der Name der Animation eingeführt wird – nicht unähnlich einer Funktionsdeklaration):

```
/* lies: „Keyframes für die Animation 'resize'.“ */

@keyframes resize {
    /* hier die Definition der Zwischenwerte */
}
```

Die allgemeine Syntax ist:

```
@keyframes animationname  {
    keyframes-selector  {
      css-styles;
```

```
    }
}
```

Es werden einzelne **Keyframe-Definitionen** beschrieben, die man sich wie Schnappschüsse der Zwischenzustände vorstellen kann. Als `keyframes-selector` dienen entweder Prozentwerte oder die **Schlüsselworte** `from` und `to`, falls man sich auf die Definition von Anfangs und Endzustand beschränken möchte:

```
@keyframes resize {
  /* hier die Definition der Zwischenwerte */
  from {
    height:150px; width:250px;
  }
  to {
    height:200px; width:400px;
  }
}
```

Mit nichts als Anfangs- und Endwerten wäre man jedoch nicht viel weiter als mit CSS-Transformations.

✔ Interessant wird es, sobald man mit **Prozent-Keyframes** arbeitet. Der Wert `0%` entspricht `from`, der Wert `100%` entspricht `to`. Diese beiden Frames *müssen* definiert werden.

Folglich sind noch 99 Zwischenwerte definierbar, die auf entsprechende Zeitpunkte im Animationsverlauf verteilt werden können. Für jeden dieser Punkte können beliebige Präsentationsanweisungen abgelegt werden.

Nehmen wir zur obigen Definition zwei Zwischenwerte hinzu:

```
@keyframes resize {
  /* hier die Definition der Zwischenwerte */
  0% {
    height:150px;  width:250px;
  }
  30% {
    height:180px; width:380px;
  }
  60% {
    height:250px; width:450px;
  }
  100% {
```

```
    height:200px; width:400px;
  }
}
```

13.2.3. Steuerung der Animation

Diese Animation läuft noch nicht. Es fehlen noch Steueranweisungen, die ebenfalls, **zusätzlich** zur Anweisung, die auf `@keyframes` verweist, in die `hover`-Definition platziert werden müssen.

✔ Mindestanforderung ist ein **Duration-Wert** für die Dauer der Animation.

Dieser wird zusammen mit der `@keyframes`-Referenz übergeben:

```
#beispiel:hover {
 /* @keyframes-Definition und Animationsdauer: */
    animation: resize 5s;
}
```

Es können weitere Werte übergeben werden, wobei dies aus Gründen der Übersichtlichkeit besser als **Einzelanweisungen** geschieht.

✔ In diesem Fall muss auch der `@keyframes`-Verweis (*animation-name*) und die Dauer (*duration*) separat festgelegt werden.

Zusätzlich kann man die Zahl der Wiederholungen (*iteration-count*), die Richtung (*direction*) und eine Easing-Funktion (*timing-function*) bestimmen:

```
#beispiel:hover {
 /* Verweis auf die @keyframes-Definition: */
    animation-name: resize;

 /* weitere Anweisungen zur Animation: */
    animation-duration: 1s;
    animation-iteration-count: 4;
    animation-direction: alternate;
    animation-timing-function: ease-in-out;
}
```

Neben den hier vorgestellten Eigenschaften existieren noch Werte zur Verzögerungszeit (*delay*) und Abspielzustand (*play-state*).

14. Media-Queries

Zwar ist CSS (noch immer) keine Programmiersprache, es ziehen jedoch zunehmend Aspekte in die Sprache ein, die sie reaktiver machen.

Mit Hilfe der **Media-Queries** ist es nun möglich, Teile des Stylesheets abhängig von äußeren Umständen, wie den Eigenschaften des Useragents oder des Viewports, an- oder abzuschalten.

✔ **Media Queries** sind die Grundlage des „Responsive Webdesigns“. Ihre Bedeutung ist deshalb nicht zu unterschätzen.

Die Bezeichnung Media „*Query*“ beruht darauf, dass die übliche **Media-Angabe**, wie sie beispielsweise über das `media`-Attribut im Link-Element oder als Media-Angabe in einer Importanweisung erfolgt, durch eine **beschreibende Abfrage** in Form eines Ausdrucks (*expression*) ergänzt werden kann, die vom Useragent beantwortet wird.

✔ Die Abfrage betrifft Eigenschaften (*media features*) des aktuellen Useragents. Zur näheren Eingrenzung kann ein Vergleichs- oder Schwellenwert (*expression*) übergeben werden.

Eine Media-Angabe kann ganze Stylesheetdateien betreffen oder einzelne Gruppen von Regeln innerhalb einer Stylesheetdatei. Der **Query-Aspekt** wird bei den folgenden Beispielen jeweils fett hervorgehoben. Zusätzlich *kann* eine Typangabe des Useragents erfolgen – im Falle der folgenden Beispiele ist dies `screen` oder `print`.

✔ Der Query-Aspekt einer Media-Query hat **abfragenden Charakter**. Der abgefragte Aspekt ist entweder *true* oder *false*.

Abhängig hiervon wird die betroffene CSS-Anweisung ausgeführt.

14.1.1. Media-Angabe für verlinkte Stylesheets

So erfolgt die Media-Angabe für verlinkte Stylesheets

```
<link rel="stylesheet"
      media="screen and (color)"
      href="beispiel.css">
```

Im vorliegenden Falle wird die Datei nur dann in das Gesamt-Stylesheet eingegliedert, wenn das ausgebende Gerät einen Bildschirm mit Farbwiedergabe besitzt (nicht also für Monochromausgabe).

✓ **Um Missverständnissen vorzubeugen:**
Die Media Query verhindert *nicht* das Laden (oder den Import) einer CSS-Datei, sondern lässt lediglich deren Regeln *brachliegen*, solange die Query zu *false* ausgewertet wird.

Achtung:
Es werden stets *alle* CSS-Dateien für den aktuellen Medientyp geladen!

14.1.2. Media-Angabe für Stylesheet-Import

In analoger Weise kann die Angabe in einem Importstatement erfolgen:

```
@import url(beispiel.css) screen and (color);
```

Hinweis:
Auch hier werden die importierten Regeln nur aktiv, wenn die Bedingung erfüllt ist. Der Import (das Laden der CSS-Datei) erfolgt jedoch gleichwohl.

14.1.3. Media-Angabe innerhalb eines Stylesheets

Sollen einzelne Regeln oder Gruppen von Regeln einer Stylesheet-Datei aktiviert oder deaktiviert werden, so werden sie in einen `@media`-Block eingeschlossen:

```
@media handheld and (min-width: 480px) {
```

```
    /* verschiedene Regeln für diesen Fall
    body {
        background-color:#aaa;
    }

}
```

Der `@media`-Block verhält sich wie eine Styleregel, die abhängig von der geprüften Bedingung in Kraft tritt. Beachten Sie das zusätzlich erforderliche Paar aus **geschweiften Klammern**, das die inneren Regeln *umschließen* muss!

✔ Mittels `@media` lassen sich nur **komplette Regeln**, nicht aber Properties innerhalb einer Regel aktivieren.

Tipp:
Bei Stylesheets für **mobile Clients** ist diese Herangehensweise zu empfehlen. Ein Zergliedern in mehrere Dateien bringt keinen Gewinn, da ohnehin alle fraglichen Dateien für den aktuellen Mediatyp (meist ist dies `screen`, seltener wird `handheld` unterstützt) geladen würden.

Es bringt mehr Nutzen, die Zahl der HTTP-Requests zu minimieren (nur eine CSS-Datei) und dafür „brachliegende“ Regeln hinzunehmen.

14.2. Syntax der CSS Media-Angabe

Eine Media-Angabe setzt sich aus verschiedenen Teilen zusammen, nämlich der **Typangabe** und der eigentlichen **Query**. Beide Teile sind optional bzw. können auch einzeln auftreten. Die Query kann **Properties** oder **Features** des Useragents betreffen.

✔ Zwischen Typangabe und Query steht, sofern beide Teile vorliegen, verbindend das **Keyword and** (dazu später mehr). Um das Keyword müssen Leerzeichen stehen.

✔ Mehrere Media-Angaben können durch **Komma** verkettet werden.

14.2.1. Typangabe des Useragents

Eine Media-Angabe kann grundsätzlich den **Ausgabetyp** nennen, auf den sich die Styleanweisung bezieht.

✔ Default ist `all`, also *alle Typen* ohne Ausnahme. In diesem Fall kann die Typangabe entfallen.

Weitere Typen sind (u.a):

`screen`, `print`, `speech`, `handheld`, `projection` ...

Folgende Typangabe beschränkt die Verwendung der einzubindenden Datei auf die Druckausgabe. Das Stylesheet *druck.css* wird nur für die Druckpräsentation des aktuellen Dokuments aktiviert:

```
<link rel="stylesheet"
      media="print"
      href="druck.css">
```

✔ Ein Druckstylesheet wird geladen, wenn der Client die *Druckausgabe* unterstützt, also **nicht erst, wenn der Druck eingeleitet wird**, sondern *unmittelbar* beim Laden des entsprechenden Dokuments.

Mehrere Typangaben werden *kommagetrennt* übergeben. Das Stylesheet ist für *alle* angegebenen Typen aktiv:

```
<link rel="stylesheet"
      media="screen, handheld"
      href="style.css">
```

Gegebenenfalls kann mit jedem Typ der Liste ein eigener Query-Aspekt verknüpft werden:

```
<link rel="stylesheet"
      media="screen and (max-width:480px), handheld"
      href="style.css">
```

✔ Wird ein Query-Aspekt genannt, so wird dieser dem Typ zugeordnet, nicht der gesamten Typliste.

Die Angabe `all` lädt das Stylesheet grundsätzlich für alle Ausgabegeräte. Eine Typangabe erübrigt sich dann im Prinzip. Die folgenden Angaben sind *wirkungsgleich*:

```
<link rel="stylesheet"
      media="all"
      href="universal.css">

<link rel="stylesheet"
      href="universal.css">
```

Ein **unbekannter Media-Typ** wird stets als *false* betrachtet. Eine so gekennzeichnete CSS-Anweisung tritt daher nie in Kraft:

```
<link rel="stylesheet"
      media="unbekannt"
      href="zwecklos.css">
```

✔ **Achtung** - geladen wird eine solche Datei dennoch!

Dasselbe gilt grundsätzlich für Media-Querys, die nicht „conforming“ sind, also nicht anwendbare Properties nennen, falsche Maßeinheiten oder einen ungültigen Media-Typ beinhalten.

✔ Ist eine CSS-Anweisung an eine *Liste* mehrerer Media-Queries gekoppelt, so gilt sie für alle „conforming“ Media-Queries der Liste. Ist eine Query „non-conforming“, so wird sie ignoriert.

14.2.2. Query-Aspekt der Media Query

Eine **Query** ist eine zusätzliche Bedingung, die an den ausführenden Useragent gestellt wird. Dieser beantwortet die Query mit *true* oder *false* und führt entsprechend die bedingte CSS-Anweisung aus.

✔ Jede Query wird individuell in **runde Klammern** gesetzt.

Eine Angabe des Mediatyps ist hierbei nicht notwendigerweise erforderlich. Folgende beiden Anweisungen sind daher identisch:

```
@media all  and (color) { ... }
```

```
@media (color) { ... }
```

✔ Wird *kein* Media-Typ angegeben, so wird automatisch `all` als Typ angenommen. Der Verbinder `and` entfällt.

Sollen mehrere Bedingungen an einen Mediatyp *gleichzeitig* erfüllt sein, so müssen alle Queries durch `and` verknüpft werden:

```
@media screen and (min-width:480px)
              and (orientation:portrait) { ... }
```

14.2.3. Keywords für Media Queries

Im Rahmen der Media-Queries werden **drei Keywords** eingeführt, die zur Formulierung verwendet werden

and

Das Keyword `and` hat verbindenden Charakter. Es steht zwischen der Typangabe und der ersten Query, sowie zwischen allen weiteren folgenden Queries eines Media-Query-Ausdrucks.

```
@media screen and (device-aspect-ratio: 16/9) { ... }
```

not

Das Keyword `not` hat negierenden Charakter. Es steht am Anfang eines Media-Query-Ausdruck und kehrt dessen gesamten Wert um.

Aktiv für den Media-Typ screen:

```
@media screen { ... }
```

Aktiv, wenn der Media-Typ *nicht* screen ist:

```
@media not screen { ... }
```

Aktiv, wenn der Bildschirm *nicht* 16/9-Format besitzt:

```
@media not screen and (device-aspect-ratio: 16/9) { ...
}
```

only

Das Keyword `only` dient als Fallback für ältere Useragents, die keine Media-Queries unterstützen. Von diesen wird es wie ein unbekannter Media-Type angesehen, da es stets am Beginn eines Media-Query-Ausdrucks steht. Aktuelle Useragents *kennen* (und ignorieren!) dieses Keyword.

Folgender Block würde von älteren Useragents ausgeführt, ohne dass aber die *Bedingung* beachtet würde (der Query-Zusatz wird ignoriert):

```
@media screen and (min-width:480px) { ... }
```

Mit dem davorgesetzten `only` führt der **ältere Useragent** den Block nicht aus, der neuere beachtet `only` nicht und wertet die Query aus:

```
@media only screen and (min-width:480px) { ... }
```

✔ **Achtung** - das Keyword `only` verhindert nicht das Laden oder den Import der Datei, sondern ggfs. nur das Inkrafttreten deren *Regeln*.

14.3. Media Features des Useragents

Die Query fragt eine **Eigenschaft des Useragents** (*media feature*) ab.

Hierbei kann es darum gehen, ob diese Eigenschaft grundsätzlich existiert (boolean), oder, ob sie einen bestimmten Wert besitzt (es wird ein Vergleichswert geprüft), oder einen Schwellenwert unter- oder überschreitet.

Existenz:

```
@media screen and (monochrome) { ... }
```

Soll ein Feature dahingehend geprüft werden, ob es grundsätzlich unterstützt wird, so genügt die Angabe des **Featurenamens**. Der Client gibt für nicht unterstützte Features grundsätzlich den Wert 0 zurück, der zu `false` gewertet wird. (Das Feature `monochrome` ist 0, wenn der Client über eine farbfähige Ausgabe verfügt, unabhängig von der Farbtiefe.)

Vergleichswert:

```
@media screen and (device-aspect-ratio: 16/9) { ... }
```

✔ Ein **Vergleichwert** wird direkt in Zusammenhang mit der geprüften Eigenschaft übergeben. Der Operator hierfür ist der **Doppelpunkt**.

Die Query wird dann als *false* gewertet, wenn der Client das Feature nicht mit dem genannten Vergleichswert unterstützt. (Das Feature selbst mag unterstützt werden.)

Als Vergleichswert kann ein **Ausdruck** übergeben werden, der einen Zahlenwert ergibt. Hier ist es das Seitenverhältnis des Viewports `device-aspect-ratio`.

In anderen Fällen kann als Vergleichswert ein **Keyword** genannt werden (beispielsweise für das Feature `orientation`).

Schwellenwert:

Wird eine Eigenschaft (in diesem Falle die Viewportbreite `width`) auf einen *Schwellenwert* hin geprüft, so wird je nach Intention entweder das Präfix `min` oder `max` vor den Propertybezeichner gesetzt:

```
/* Viewport breiter als 480px */
@media screen and (min-width: 480px) { ... }

/* Viewport schmaler als 480px */
@media screen and (max-width: 480px) { ... }
```

✔ Ein **Schwellenwert** wird immer mit **min- oder max-Präfix** vor dem eigentlichen Eigenschaftsbezeichner definiert.

Einige Features können durchaus boolsch geprüft werden, oder aber wahlweise mit Vergleichs- bzw. Schwellenwert.

Je nachdem ist die Query unterschiedlich zu formulieren:

```
/* Device ist grundsätzlich farbfähig */
@media screen and (color) { ... }
```

```
/* Device farbfähig mit 3 Bit Farbtiefe */
@media screen and (color: 3) { ... }

/* Device farbfähig mit mindestens 3 Bit Farbtiefe */
@media screen and (min-color: 3) { ... }
```

14.3.1. Media Features: width, height

Die Feature-Prüfung `width` zielt auf die **Viewportbreite**, also den Anzeigebereich des Dokuments. Die Prüfung `height` zielt analog auf die **Viewporthöhe**. Die Präfixe `min` und `max` sind anwendbar.

✔ In beiden Fällen können **keine negativen Werte** als Vergleichs- bzw. Schwellenwerte übergeben werden.

```
@media screen and (min-width: 400px)
              and (max-width: 700px) { ... }
```

Für das Druckmedium beziehen sich die Angaben auf die Breite bzw. Höhe der **Pagebox** (bedruckbare Fläche).

✔ Der **Viewport** ist nicht identisch mit dem Screen, sondern ist die Anzeigefläche, die für das Dokument zur Verfügung steht. Das kann das Browserfenster sein, aber auch ein iFrame-Rahmen.

Screenbezogene Features sind `device-width` und `device-height`.

14.3.2. Media Features: device-width, device-height

Die Feature-Prüfungen `device-width` und `device-height` betreffen die Breite und Höhe des **Renderingbereichs** (mit anderen Worten des Screens). Die Präfixe `min` und `max` sind prinzipiell anwendbar, allerdings können auch konkrete Vergleichswerte genannt werden (die dann auf exakte Übereinstimmung geprüft werden).

✔ In beiden Fällen können **keine negativen Werte** als Vergleichs- bzw. Schwellenwerte übergeben werden.

```
@media screen and (device-width: 800px) { … }
```

Für das Druckmedium beziehen sich die Angaben auf die Breite bzw. Höhe der **Druckseite** (Papierabmessung).

14.3.3. Media Feature: orientation

Das Media-Feature `orientation` bezieht sich auf das **Verhältnis** der Media-Features `width` und `height` zueinander.

Es unterscheidet genau zwei Zustände:

- ✔ Ist *width* **kleiner oder gleich** *height*, spricht man von **Portrait-Modus**
- ✔ Ist *width* **größer als** *height*, spricht man von **Landscape-Modus**

Die Präfixe `min` und `max` sind **nicht** anwendbar.

```
@media handheld and (orientation: portrait) { ... }
```

```
@media handheld and (orientation: landscape) { ... }
```

14.3.4. Media Features: aspect-ratio, device-aspect-ratio

Die Media-Features `aspect-ratio` und `device-aspect-ratio` beschreiben das Verhältnis `width` zu `height` (bezogen auf den *Viewport*) bzw. `device-width` zu `device-height` (bezogen auf den *Screen*), betrachten dieses jedoch als einen Zahlenwert.

- ✔ Die Präfixe `min` und `max` sind daher anwendbar. Üblicherweise wird ein **Ausdruck** anstelle eines konkreten Wertes übergeben.

Hier wird ein **16/9-Schirm** addressiert, der beispielsweise 1280 Pixel Breite und 720 Pixel Höhe besitzt.

```
@media screen and (device-aspect-ratio: 16/9) { ... }
```

Dies ist identisch zu einer Übergabe der *konkreten* Pixelwerte:

```
@media screen and (device-aspect-ratio: 1280/720) {...}
```

... womit jedoch - Achtung! - höher auflösende 16/9-Schirme (z.B. 2560px zu 1440px) nicht erfasst würden.

14.3.5. Media Feature: color

Das Media-Feature `color` nennt die **Farbtiefe** in Bit pro Farbkomponente. Für monochrome Ausgabegeräte hat `color` den Wert 0.

Die Verwendung der Präfixe `min` und `max` ist zulässig.

Diese Query gilt generell für alle Geräte mit Farbdarstellung (`color` gleich 0 wird *false* gewertet, Monochrom-Displays fallen daher beim Test durch):

```
@media all and (color) { ... }
```

Diese Query gilt für alle Geräte deren Farbtiefe *3 Bit* pro Farbkomponente nicht unterschreitet:

```
@media all and (min-color: 3) { ... }
```

14.3.6. Media Feature: color-index

Das Media-Feature `color-index` bezieht sich auf Geräte mit **Color-Lookup-Table** (Clut). Es nennt die Zahl der entsprechenden Einträge (Anzahl der unterscheidbaren Farbtöne). Die Präfixe `min` und `max` können eingesetzt werden.

```
@media all and (min-color-index: 256) { ... }
```

14.3.7. Media Feature: monochrome

Das Media-Feature `monochrome` bezieht sich auf die Größe des Framebuffers pro Display-Pixel für **Monochrome-Displays**. Der Wert ist 1 bzw. größer 1, wenn Framebuffering unterstützt wird. (Die Präfixe `min` und `max` können hier daher ebenfalls eingesetzt werden.)

Diese Query gilt für alle Monochrome-Displays:

```
@media all and (monochrome) { ... }
```

✔ **Anmerkung:**
Für Farbdisplays ist der Wert von `monochrome` gleich 0. Umgekehrt hat wiederum `color` für monochrome Displays den Wert 0.

Farbfähigkeit und Druckstylesheets

Auch zum Steuern von **Druckstylesheets** sind die Media-Features `color` und `monochrome` brauchbar. Auf folgendem Weg können Sie Stylesheets für Ausgabe auf Farbdrucker bzw. SW-Drucker zur Verfügung stellen:

```
<link rel="stylesheet"
      media="print and (color)" href="colorprint.css">

<link rel="stylesheet"
      media="print and (monochrome)" href="print.css">
```

14.3.8. Media Feature: resolution

Das Media-Feature `resolution` bezieht sich auf die **Auflösung**, die das Ausgabegerät bietet. Auch hier können die Präfixe `min` und `max` verwendet werden.

```
@media print and (min-resolution: 300dpi) { ... }
```

15. Responsive Design

Mit dem Begriff **Responsive Webdesign** wird das **selbstanpassende Verhalten** entsprechender Webseiten benannt, deren Darstellung sich automatisch an Features und Typ des ausgebenden Useragents orientiert. Ein „responsives“ Layout funktioniert gleichermaßen auf einem Smartphone, auf einem Tablet oder auf einem Desktop-Rechner - wobei die Darstellung für die jeweilige Umgebung optimiert wird.

- Der Begriff „responsive web design“ wurde vom Webdesigner **Ethan Marcotte** geprägt, der ihn das erste Mal in einem Artikel des Onlinemagazins Alistapart verwendete: `www.alistapart.com/articles/responsive-web-design/`

Eine zentrale Grundlage für Responsive Design sind die im vorigen Abschnitt beschriebenen Media Queries.

15.1. Wann macht ein Responsive Layout Sinn?

Ein responsives, für alle potentiellen Clients zugängliches Layout, ist nicht notwendigerweise die optimale Lösung. Oft ist es am Ende die bessere Wahl, zwei voneinander unabhängige Websites für mobile und stationäre Nutzer anzubieten. Dreh- und Angelpunkt sind die zu präsentierenden **Inhalte** oder die angebotenen **Dienste**.

- Sofern unabhängig vom Aufenthaltsort und Natur des Clients stets **dieselben Inhalte** (oder weitestgehend dieselben) präsentiert werden können, macht es Sinn, über einen responsiven Ansatz nachzudenken.

- Müssen für mobile und stationäre User weitestgehend **unterschiedliche Inhalte** oder Dienste angeboten werden, ist ein Ansatz mit zwei unabhängigen Websites für unterscheidbare Nutzergruppen der geeignetere Ansatz.

15.2. Methodik: Mobile first vs. Desktop first

Ein Kerngedanke des Responsive Designs besteht darin, dass von einem Grundlayout ausgegangen wird, das sich an einem **Extrem** der zu erwartenden auswertenden Clients orientiert (meist wird dies an der **Viewportgröße** festgemacht):

- **Desktop first:**
  Das Layout wird für **Desktopbrowser** optimiert und schrittweise für kleinere Viewports durch Wegnahme von Features aufbereitet (*graceful degradation*).

✔ **Mobile first:**
Das Layout wird für **Smartphone-Ausgabe** optimiert und schrittweise für größere Viewports durch Hinzunahme von Features aufbereitet (*progressive enhancement*).

Abb: Responsive Layout „Mobile First“ (Smartphone- und Desktopansicht)

Zu beachten ist, dass Viewport- und Screengröße nicht dasselbe sind. Unter Viewport ist die „rendering Area“ im Browser zu verstehen, deren Größe von der des Screens abweicht, wenn der Browser nicht im Vollbildschirmmodus betrieben wird. Nicht zum Viewportbereich zählen die Bedienflächen des Browsers (Tabs, URL-Feld, Menü), sodass der Viewport normalerweise stets *kleiner* als die Screenfläche sein wird.

• Einsatz der Media Query im Responsive Layout

Mittels geschickten Einsatzes von Media Queries kann ein für verschiedene Useragents angepasstes Layout mit einem einzigen Stylesheet beschreiben werden. Die Anpassungen erfolgen durch bedingte CSS-Blöcke, die nach Bedarf aktiv werden.

Ebenfalls möglich wären mehrere, durch Queries bedingte CSS-Dateien.
Beim aktuellen Verhalten der Clients (Dateien werden grundsätzlich geladen, wenn der Mediatyp übereinstimmt), bringt dies für den mobilen Einsatz eher Nachteile (mehr Requests).

Beim responsiven Ansatz wird mit einem **Ausgangslayout** gearbeitet, das schrittweise angepasst wird. Die Media Queries prüfen hierfür ein

geeignetes Feature oder eine geeignete Eigenschaft des Useragents - dies wird in den meisten Fällen die Breite des Viewports sein.

Es ist **üblich**, aber in der Spezifikation *nicht vorgeschrieben*, dass ein Useragent die Seite **neu rendert**, wenn sich **Umgebungsaspekte ändern**, wie die Größe des Viewports oder ein Wechsel der Orientierung von Portrait- zu Landscape-Modus.

• Anpassung des Layouts anhand der Viewportbreite

Folgender Code bewirkt einen Wechsel der Hintergrundfarbe des Dokuments bei einem **Resize des Viewports**. Die Punkte, an denen die Farbe umspringt, bezeichnen wir als **Breakpoints**. Ausschlaggebend ist die aktuelle Breite des Viewports.

Die Verwendung des `max`-Prefix bewirkt, dass die Darstellung bei **Unterschreitung** des Breakpointlimits umspringt, weil ab dann der entsprechende bedingte Block aktiv wird.

Hier wird entsprechend vom **Desktopviewport** ausgehend designt. Die Herangehensweise entspricht also dem „Desktop First“-Prinzip:

```
/* Ausgangsfarbe; für Viewports breiter als 960px*/
   body {
      background-color: grey;
   }


/* Breakpoint 960px */
@media screen and (max-width: 960px) {
   body {
      background-color: red;
   }
}

/* Breakpoint 768px */
@media screen and (max-width: 768px) {
   body {
      background-color: orange;
   }
}
```

```
/* Breakpoint 550px */
@media screen and (max-width: 550px) {
   body {
      background-color: yellow;
   }
}

/* Breakpoint 320px */
@media screen and (max-width: 320px) {
   body {
      background-color: green;
   }
}
```

Auch der umgekehrte Weg, also die **Smartphone-Ausgabe** als Ausgangspunkt, ist möglich, indem man die Breakpoints für suksessiv *größer* werdende Viewports setzt. Hier nun also „Mobile First“:

```
/* Ausgangsfarbe; für Viewports schmaler als 320px*/
   body {
      background-color: grey;
   }


/* Breakpoint 320px */
@media screen and (min-width: 320px) {
   body {
      background-color: red;
   }
}

/* Breakpoint 550px */
@media screen and (min-width: 550px) {
   body {
      background-color: orange;
   }
}

/* Breakpoint 768px */
@media screen and (min-width: 768px) {
   body {
      background-color: yellow;
   }
}
```

```
/* Breakpoint 960px */
@media screen and (min-width: 960px) {
   body {
      background-color: green;
   }
}
```

Die Verwendung des `min`-Prefix bewirkt, dass die Darstellung bei **Überschreitung** des Breakpointlimits umspringt, weil ab dann der entsprechende bedingte Block aktiv wird.

16. Überblick über die Spezifikationen

16.1. Spezifikationen zu CSS1

CSS Level 1

W3C Recommendation, 17. Dezember 1996, Revision 11. Januar 1999; Aktueller Standard. Cascading Style Sheets Level Eins; wird in allen aktuellen Browsern unterstützt

- www.w3.org/TR/REC-CSS

16.2. Spezifikationen zu CSS2

CSS Level 2.1 (Level 2 Revision 1)

W3C Working Draft 11. April 2006; Überarbeitung von CSS 2; nimmt als Fehlerbereinigung einige Teile der Spezifikation zurück; wird CSS 2.0 im Endeffekt ablösen. Wird von aktuellen Browsern bereits voll unterstützt.

- www.w3.org/TR/CSS21

CSS Level 2

W3C Recommendation, 12. Mai 1998; Aktueller Standard. Cascading Style Sheets Level Zwei; baut auf Level 1 auf; wird in allen aktuellen Browsern weitestgehend unterstützt.

- www.w3.org/TR/REC-CSS2

16.3. Spezifikationen zu CSS3

CSS Level 3

Seit 2001 arbeitet das W3C an einer Weiterentwicklung von CSS 2, die unter der Bezeichnung CSS 3.0 firmiert.

- **Überblick:**
  www.w3.org/Style/CSS/current-work

Analog zur Modularisierung von XHTML ist CSS 3.0 in verschiedene funktionale Module aufgespalten (es sind, genauer gesagt, rund 20 derartige Module), die jeweils von einer eigenen Arbeitsgruppe betreut werden. Die Spezifikationsteile befinden überwiegend noch in Entwicklung. Sehr weitgehende Teile werden in aktuellen Browsern dennoch bereits unterstützt (Achtung: inkonsistent; stets testen!).

Hier eine Auswahl(!) der interessantesten Spezifikationen:

- **CSS Selectors**
  `www.w3.org/TR/css3-selectors`
  Beschreibt Element-Selektoren in CSS3. CSS1 und CSS2 Selektoren sind mit eingeschlossen und werden um neue Vorschläge erweitert.

- **CSS Media Queries**
  `www.w3.org/TR/css3-mediaqueries/`
  Media Queries stellt eine Erweiterung der @media Regeln von CSS dar sowie des "media" Attributes in HTML. Es werden weitere Parameter hinzugefügt wie Displaygröße, Farbtiefe und Seitenverhältnis.

- **CSS Backgrounds and Borders**
  `www.w3.org/TR/css3-background/`
  Beschreibt Hintergrundfarben und -bilder sowie den Style von Rahmen. Neue Funktionen umfassen die Fähigkeit Hintergrundbilder zu dehnen, Bilder als Rahmen zu verwenden, Ecken von Boxen abzurunden und einer Box einen Schatten außerhalb des Rahmens hinzuzufügen.

- **CSS Basic Box Model**
  `www.w3.org/TR/css3-box/`
  Das Box Model beschreibt das Layout von Inhalten in Block-Ebenen im normalen Ablauf. CSS stellt die Elemente des Dokuments als rechteckige Kästchen dar, welche nacheinander oder ineinander verschachtelt in einer Reihenfolge angeordnet sind, die man flow nennt.

- **CSS Cascading and Inheritance**
  www.w3.org/TR/css3-cascade/
  Gibt an, auf welche Weise Eigenschaftenwerte zugewiesen werden. CSS erlaubt verschiedene StyleSheets, um das Rendering eines Dokuments zu beinflussen.

- **CSS Color**
  www.w3.org/TR/css3-color/
  Spezifiziert die farbbezogenen Aspekte von CSS, einschließlich Transparenz und verschiedenen Notationen für den <color> Wert Typ.

- **CSS Values and Units**
  www.w3.org/TR/css3-values/
  Values and Units beschreiben die geläufigen Werte und Einheiten, die CSS Eigenschaften annehmen können.

- **CSS Text**
  www.w3.org/TR/css3-text/
  Beinhaltet die textbezogenen Eigenschaften von CSS2 (Unterstreichung, Ausrichtung, Text Wrapping usw.) und fügt etliche neue Eigenschaften hinzu; viele für Text in verschiedenen Sprachen.

- **CSS Fonts**
  www.w3.org/TR/css3-fonts/
  Fonts beinhaltet die Einstellungen für Schrift-Anpassungen, wie Emboss- und Outline Effects, Kerning und Smoothing/Anti-Aliasing.

- **CSS Hyperlink Presentation**
  www.w3.org/TR/css3-hyperlinks/
  Behandelt die Präsentationsmöglichkeiten von Hyperlinks und bietet Möglichkeiten zur Kontrolle, welche Hyperlinks aktiv sein sollen und wo das Link-Ziel angezeigt werden soll, wenn der Nutzer den Link anklickt.

- **CSS Line Layout**
  www.w3.org/TR/css3-linebox/
  Line beschreibt die Ausrichtung von Text und anderen Boxen innerhalb einer Zeile. Es erweitert die "vertical-align" Eigenschaft von CSS1 und CSS2 um die Fähigkeit verschiedene Schriften auszurichten.

- **CSS Lists**
  www.w3.org/TR/css3-lists/

Beinhaltet die Eigenschaften um Listen zu gestalten, wie verschiedene Typen von Aufzählungszeichen und Nummerierungssystemen.

- **CSS Multi-column Layout**
  `www.w3.org/TR/css3-multicol/`
  Multi-column Layout stellt neue Eigenschaften vor, um flexibel definierte Spalten mit Inhalt zu füllen.

- **CSS Flexible Box Layout**
  `www.w3.org/TR/css3-flexbox/`
  Das Flexible Box Layout Module definiert die "box" und "inline-box" Schlüsselworte für die "display" Eigenschaft, welche bestimmt ob ein Element als Spalte oder als Zeile eines Kindeleements anzeigt wird.

- **CSS Paged Media**
  `www.w3.org/TR/css3-page/`
  Erweitert die Eigenschaften von CSS2 um neue Eigenschaften, wie fortlaufende Kopf- und Fußzeilen sowie Seitennummern.

- **CSS 2D Transformations Module**
  `www.w3.org/TR/css3-2d-transforms`
  Das 2D Transformations Module beschreibt Eigenschaften zur Rotation, Überführung und weiterer affiner Transformationen einer Box.

- **CSS 3D Transformations Module**
  `www.w3.org/TR/css3-3d-transforms`
  Erweitert die 2D Transformation um eine Perspektivtransformation. Dieses Modul wird in Zusammenarbeit mit der SVG entwickelt.

- **CSS Transitions Module**
  `www.w3.org/TR/css3-transitions`
  Das Transitions Module definiert eine Eigenschaft um die Übergänge zwischen Pseudoklassen zu steuern (z.B., wenn ein Element den :hover Zustand annimmt oder verliert).

- **CSS Animations Module**
  `www.w3.org/TR/css3-animations`
  Beschreibt wie Eigenschaften ihre Werte während einer Präsentation ändern und welche Werte sie über welchen Zeitraum nacheinander annehmen.

17. Literatur

CSS: The Definitive Guide (4th Ed.)
Eric Meyer, Estelle Weyl
(O'Reilly, 2017)

CSS Mastery (3rd Ed.)
Andy Budd, Emil Björklund
(Apress 2016)

Pro CSS3 Layout Techniques
Sam Hampton-Smith
(Apress 2016)

Transitions and Animations in CSS
Estelle Weyl
(O'Reilly, 2016)

CSS: Das umfassende Handbuch
Kai Laborenz
(Rheinwerk Verlag, 2015)

Flexible Boxes: Eine Einführung in moderne Websites
Peter Müller
(Rheinwerk Verlag, 2015)