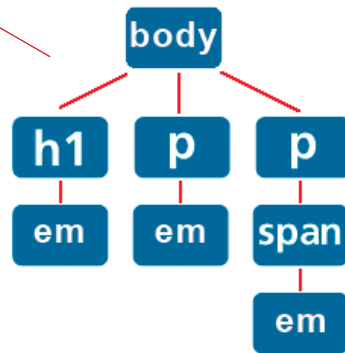


Kapitel 2

Hur CSS fungerar

Lär dig hur CSS hant-
erar dokumenthierarkin



Lär dig hur man forma-
terar specifika element

Lokala (inline) och in-
bäddade stilar jämfört
med länkade stilar

Förstå arv



I föregående kapitel visade jag hur XHTML-kod skapar en strukturell hierarki av taggar. Jag nämnde också hur man kan använda CSS för att formatera dessa taggar så att man exakt kan bestämma layouten och utseendet på element som visas på skärmen, t.ex. text och bilder. Nu är det dags att ta itu med mekanismerna bakom CSS. I slutet av det här kapitlet är du redo att skapa egna format för exemplet med XHTML-kod som vi tittade på i kapitel 1.

Tre sätt att formatera ett dokument

Du kan lägga till CSS-kod till dina webbsidor på tre olika sätt: lokalt (inline), inbäddat (embedded) och länkat från en separat CSS-stilmall. Det klokaste valet när det gäller utveckling av webbplatser är dock att länka XHTML-sidorna till en CSS-stilmall, men vi kommer även att titta på de två andra alternativen eftersom de kan vara användbara medan du utvecklar dina sidor.

En stilmall är en helt och hållet separat fil som enbart innehåller CSS-kod. En stilmall kan delas av ett obegränsat antal XHTML-sidor vilket garanterar att utseendet förblir enhetligt från sida till sida och medger att ändringar som görs av en viss stil omedelbart återspeglas på hela webbplatsen.

Lokala (inline) stilar

Lokala (inline) stilar bifogas till en tagg med hjälp av XHTML-taggen `style`, så här:

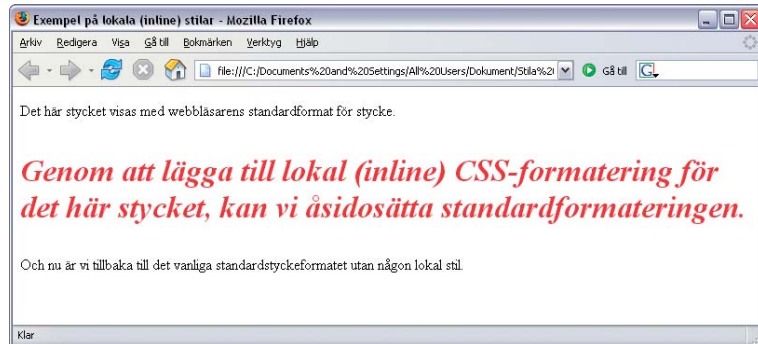
```
<p>Det här stycket visas med webbläsarens standardformat för stycke.</p>
```

```
<p style="font-size: 25pt; font-weight:bold; font-style:italic; color:red;">Genom att lägga till lokal (inline) CSS-formatering för det här stycket, kan vi åsidosätta standardformateringen.</p>
```

```
<p>Och nu är vi tillbaka till det vanliga standardstyckeformatet utan någon lokal stil.</p>
```

Detta ger det som visas i figur 2.1.

Figur 2.1 Lokala stilar appliceras enbart på den tagg de är bifogade till.



Här är ytterligare några saker du behöver känna till om lokala (inline) stilar:

- Deras räckvidd är mycket begränsad. En lokal stil påverkar endast den tagg till vilken den är bifogad.
- Praxisen att använda lokala stilar är helt enkelt ett annat sätt att skriva presentationskod direkt tillsammans med taggarna, precis som vi gjorde förr i tiden. Att lägga till lokala stilar överallt är lika illa för kodens flyttbarhet och redigeringsbarhet som att lägga till (X)HTML-attribut som exempelvis FONT. Lokala stilar bör generellt undvikas.
- Vid de sällsynta tillfällena när man behöver åsidosätta ett format på ett visst ställe och det inte finns något bättre sätt att göra det, kan man skapa en lokal stil utan att behöva känna sig alltför skyldig. Med detta sagt, kan man nästan alltid undvika att använda lokala stilar genom att lägga till ett unikt ID eller en klass till taggen i fråga och sedan skriva motsvarande stil i stilmallen.
- Användning av en lokal stil är ett bra sätt att pröva en stil innan man lägger in den i stilmallen (se avsnittet "Länkade stilmallar" som strax följer). Kom bara ihåg att helt och hållet ta bort `style`-attributet när du fått den effekt du vill ha och klippa ut och klistra in enbart själva stilen i stilmallen. I annat fall kommer den lokala stilen alltid att åsidosätta alla eventuella ändringar du gör via stilmallen för just den taggen, och du riskerar att spendera timmar på att försöka fixa till stilmallen när problemet egentligen gömmer sig i XHTML-koden.
- Lokala stilar prioriteras framför stilar som man definierar med inbäddade stilar (beskrivs härnäst), vilka i sin tur prioriteras framför globala stilar som man definierar i en stilmall. (Se avsnittet "Vad innebär Cascading" senare i det här kapitlet)



De kommenterade CDATA-taggar (`/* */ och /* */`) omger stilarna. Detta förhindrar att de tolkas som XML vilket skulle kunna orsaka tolkningsproblem beträffande tecken som XML förväntar sig att hitta kodade som entiteter. (Kom ihåg att XHTML är XML-baserat.) För exemplen med inbäddade stilar i den här boken har jag inte lagt till CDATA-taggar och jag har aldrig stött på några problem som berott på att jag utelämnat dem. Men du kan bestämma huruvida du vill lägga till dem efter att du läst W3 Schools förklaring angående XML CDATA (www.w3schools.com/xml/xml_cdata.asp).

för mer ingående information om hur konflikter mellan stilar hanteras.)

Inbäddade stilar

Man kan placera en grupp CSS-stilar i head-delen i ett XHTML-dokument – dessa kallas inbäddade stilar. Inbäddade stilar fungerar så här:

```
<head>
<title>Exempel på inbäddade stilar</title>
<meta http-equiv="Content-type" content="text/html; charset=iso-8859-1" />
<meta http-equiv="Content-Language" content="sv" />
<style type="text/css">
/*  */
h1 { font-size: 16pt; font-weight:bold;}
p {color:blue;}
/*  */
</style>
</head>
```

Taggen `style` talar om för webbläsaren att den kommer att träffa på kod som inte är (X)HTML; taggens attribut fastslår att koden är CSS. (Om du vill inkludera JavaScript i dokumentets head-del i stället för CSS, använder du taggen `script` med attributet `"text/JavaScript"`).

Här följer några kommenterar angående inbäddade stilar:

- Räckvidden för inbäddade stilar är begränsad till sidan som innehåller stilarna.
- Om du bara publicerar en enda sida med dessa speciella stilar kan du bädda in stilarna i dokumentets head-del, även om du då egentligen inte separerar stilarna från innehållet; de finns fortfarande i samma dokument. Du får lära dig mer om inbäddade stilar när du arbetar med de enkelsidiga övningsexemplen i det här kapitlet.
- Om man arbetar med flera stilar för en komplex layout, exempelvis ett formulär, är det ibland enklare att skriva stilar som inbäddade stilar i head-delen i dokumentet, så att man inte hela tiden måste växla mellan XHTML-dokumentet och

stilmallen. När sedan allt fungerar kan man flytta stilarna till stilmallsdokumentet och ersätta stilarna i head-delen med en länk till stilmallen.

- Inbäddade stilar prioriteras framför stilmallar, men förlorar mot attribut som man definierar i lokala stilar. (Se avsnittet ”Vad innebär Cascading” senare i det här kapitlet för mer ingående information om hur konflikter mellan stilar hantearas.)

Länkade stilar

Man kan placera stilarna i ett separat dokument, en stilmall, som länkas till flera sidor så att stilarna får global räckvidd (över hela webbplatsen). Stilarna som definieras i en sådan stilmall kan påverka webbplatsens samtliga sidor, inte bara en enskild sida eller tagg. Det här är den enda av de tre metoderna som verkligen separerar presentationsstilarna från koden för strukturen. Om du samlar alla dina CSS-stilar i en stilmall på det här sättet blir det enklare både att designa och redigera en webbplats.

Om man exempelvis behöver göra ändringar som påverkar hela webbsajten (”kunden vill att all brödtext ska vara blå, inte svart”) kan man göra det lika snabbt och smärtfritt som när man modifierar en enskild CSS-stil. Detta är sannerligen mycket enklare än uppgiften att modifiera alla `font`-attribut för varje stycketagg på varje sida på webbplatsen, vilket man var tvungen att göra innan CSS fanns tillgängligt.

Du kan länka din stilmall till så många XHTML-sidor som du vill med en enda kodrad i head-delen i varje XHTML-sida:

```
<link href="my_style_sheet.css" media="screen" rel="stylesheet" type="text/css" />
```

Stilarna tillämpas sedan på koden för varje sida när sidan laddas.

Observera att i taggen `link` ovan, definieras attributet `media` som `screen`, vilket innebär att stilen är utformad för bildskärmen, vilket i det här fallet betyder webbläsaren. (Vissa användaragenter söker efter särskilda mediaattribut som bättre passar deras visningskapacitet – möjligheterna här är exempelvis: `all`, `projection`, `handheld`, `print`, `aural`. Gå till webbplatsen W3 Schools (www.w3schools.com/css/css_mediatypes.asp) för att se en komplett lista.)

En webbläsare som laddar sidan använder den stilmall som taggen `link` anger. Men genom att lägga till ännu en `link`-tagg där attribu-

tet `media` definieras som `"print"`, kan man tillhandahålla en andra stilmall som webbläsaren använder i samband med utskrift. En stilmall specialutformad för utskrift kan eventuellt dölja navigeringselement och andra element som inte fyller någon funktion när innehållet ska visas på papper.

Om du skapar en andra stilmall för utskrift kan taggen se ut så här:

```
<link href="my_style_sheet_print.css" media="print"
rel="stylesheet" type="text/css" />
```

Nu när du vet vad stilmallar är ska vi titta på regler för hur man skriver stilmallar och hur begrepp som arv, exakthet (specificity) och Cascading styr hur dessa regler påverkar koden.

Vad är Cascading Style Sheets?

Låt oss dela upp frågan i två delar: Vad är stilmallar och vad innebär "cascading"? Jag svarar direkt på första frågan, och även om jag nyss har berört svaret kommer jag att tala mer om "cascading" senare i det här kapitlet.

En *stilmall* är helt enkelt en textfil med filnamnstillägget `.css`.

En stilmall är en lista med CSS-regler. Varje regel definierar en viss stil som tillämpas på XHTML-koden; en regel kan definiera teckenstorleken för texten i ett stycke, tjockleken på en kantlinje runt en bild, placeringen av en rubrik, en bakgrundsfärg osv. Många av de sofistikerade typografi- och layoutfunktioner som återfinns i layoutprogram som exempelvis Adobe InDesign kan nu emuleras på webbsidor med hjälp av CSS. Webbdesigners har slutligen fått total kontroll över layouten på sina sidor utan att behöva ta till nödlösningar som tabeller eller användningen av GIF-bilder för att skapa mellanrum.

Hur en CSS-regel är uppbyggd

Vi börjar med att tala om hur man skriver CSS genom att titta på en enkel CSS-regel. Här följer en regel som gör all text i alla stycken i dokumentet röd:

```
p {color:red;}
```

Om du har den här XHTML-koden:

```
<p>Den här texten är mycket viktig</p>
```

kommer sålunda stycketexten att bli röd.

En CSS-regel består av två delar: en *selektor*, som talar om vilken tagg regeln gäller – ett stycke, i exemplet ovan – och en *deklara-*



En tagg i XHTML-kod är omgiven av vinkelparenteser men i CSS-kod skrivs taggen utan vinkelparenteser.

tion, som fastslår vad som händer när regeln appliceras – i exemplet ovan visas alltså texten med röd färg. Deklarationen ovan består av två element: en *egenskap*, som talar om vad som ska påverkas – i exemplet textens färg – och ett *värde*, som talar om vad som angivits för egenskapen – i vårt fall värdet rött. Betrakta noga nedanstående figur, figur 2.2, så att du är absolut klar över dessa fyra termer. Jag använder dem ofta i fortsättningen.

Figur 2.2 En CSS-regel består av två huvudelement och två under-element.



Skriva CSS-regler

Den här grundstrukturen med selektor och deklaration kan utökas på tre sätt:



CSS kräver fullständig exakthet; saknas ett semikolon kan detta göra att CSS helt och hållet ignorerar en hel regel.



Du kanske undrar vilka andra värden som finns för egenskaper som exempelvis `font-size` och `color`. Du vill kanske veta om du kan ange en färg med hjälp av RGB (röd, grön, blå) i stället för ett färgnamn (svaret är: Ja, det kan du!). Men ha tålamod medan jag fokuserar på att visa dig hur selektorer fungerar. Senare i det här kapitlet går jag närmare in på deklarationsdelen av en regel.

Flera deklamationer kan placeras inom en regel.

```
p {color:red; font-size:12px; line-height:15px;}
```

Nu är vår stycketext röd, 12 pixlar på höjden och radavståndet är 15 pixlar. (Pixlar är, som du nog vet, de små punkterna som bygger upp visningsytan på skärmen.)

Observera att varje deklaration slutar med ett semikolon som separerar den från nästa. Det sista semikolonet före den avslutande klammerparentesen är valfritt, men jag brukar alltid lägga till det så att jag eventuellt kan lägga till fler deklamationer senare utan att behöva komma ihåg att lägga till det då.

Flera selektorer kan placeras i grupp. Om du exempelvis vill att texten för taggarna `h1` till och med `h6` ska vara blå och fet kan du förstås vara ambitiös och skriva:

```
h1 {color:blue; font-weight:bold;}
```

```
h2 {color:blue; font-weight:bold;}
```

```
h3 {color:blue; font-weight:bold;}
```

Och så vidare.

Men du behöver inte göra det. Selektorerna kan grupperas inom en regel på följande sätt:

```
h1, h2, h3, h4, h5, h6 {color:blue; font-weight:bold;}
```

Mycket enklare! Se bara till att placera ett komma efter varje selektor utom den sista. Mellanslagen är valfria, men de gör koden mer lättläst.

Flera regler kan appliceras på samma selektor. Om du, efter att ha skrivit föregående regel, bestämmer dig för att taggen `h3` även ska vara kursiv, kan du skriva en regel till för `h3` på följande sätt:

```
h1, h2, h3, h4, h5, h6 {color:blue; font-weight:bold;}
```

```
h3 {font-style:italic;}
```



Om du inte har använt XHTML tidigare, notera att `span` är en neutral behållare precis som `div` och att den inte har några standardattribut; `span` har med andra ord ingen effekt på koden förrän du uttryckligen formaterar den. Det är användbart för att koda element i koden som har någon betydelse för dig och som inte definieras av XHTML. Men om ditt dokument befinner sig i en miljö där det inte kan använda din stilmall kommer `span`-elementen inte att ha någon effekt på presentationen. Till skillnad från `div` som är ett blockelement och tvingar fram en ny rad är `span` ett lokalt (*inline*) element och tvingar inte fram någon ny rad. Som standard resulterar *strong* i fet stil och *em* i kursiv stil, men du kan förstås använda CSS för att omformatera dem om du vill.

Ange vilka taggar inom dokumenthierarkin ska påverkas

Om du har glömt vad dokumenthierarki är sedan slutet av föregående kapitel kan du kanske läsa om avsnittet ”Dokumenthierarki: Möt XHTML-familjen” i kapitel 1 nu, så att jag slipper upprepa mig i onödan.

Använda kontextuella selektorer

Om du skriver en regel där du bara använder taggnamnet som selektor är varje tagg av den typen måltavla. Genom att skriva exempelvis:

```
p {color:red;}
```

får varje stycke röd text.

Men om man vill att enbart ett visst stycke ska vara rött? För att ange mer selektivt vilka taggar som ska påverkas kan man använda kontextuella selektorer. Här följer ett exempel:

```
div p {color:red;}
```

Nu visas enbart stycken inom taggarna `div` med röd text.

Som du kan se i exemplet ovan använder kontextselektorer mer än ett taggnamn (i det här fallet `div` och `p`) i selektorn. Taggen som finns närmast deklARATIONEN (i det här fallet taggen `p`) är den tagg som du siktar på. Den andra taggen eller taggarna talar om var måltaggen ska finnas i koden för att den ska påverkas. Låt oss titta lite närmare på detta.

Titta på den här kodsnutten:

```
<h1>Kontextselektorer är <em>mycket</em> selektiva.</h1>
```

<p>Det här exemplet visar hur man siktar på en specifik tagg med hjälp av dokumenthierarkin.</p>

<p>Taggar behöver bara vara arvingar (descendants) i den ordning som anges i selektorn i ; andra taggar kan finnas däremellan och selektorn fungerar ändå.</p>

Båda styckena i koden ovan innehåller taggarna em, så om du skulle skriva en regel som skulle påverka vilken em-tagg som helst inom p-taggar, skulle den regeln tillämpas på båda styckena. Men om du skriver en regel som ska tillämpas enbart på em-taggar inom span-taggar inuti p-taggar, tillämpas den endast på det andra stycket.

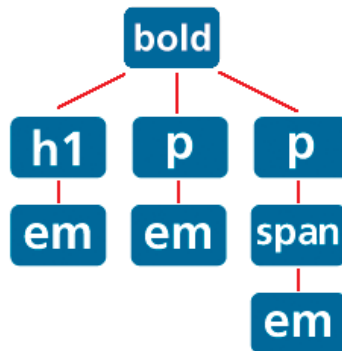
Figur 2.3 visar hur den här koden ser ut med enbart webbläsarens standardformatering.

Figur 2.3 Här har enbart webbläsarens standardformatering tillämpats.



Figur 2.4. visar kodens hierarki.

Figur 2.4 Hierarkin för kodexemplet nyss.



Den här hierarkiuppställningen visar vilken tagg som finns inuti vilken. Om du skriver stilen:

```
em [{color:green;}]
```

för koden tidigare genom att lägga till den mellan taggarna style i head-delen i mallen (som kan hämtas på bokens webbplats på adressen www.pagina.se/bok.html?6360910X) visas – enligt vad

du lärde dig tidigare i avsnittet ”Skriva CSS-regler” tidigare i det här kapitlet – all text i `em`-taggarna med grön färg, se figur 2.5.

Figur 2.5 I det här exemplet visas all text inom `em`-taggarna med grön färg.



Men om du vill vara mer selektiv? Låt säga att du vill att endast `em`-texten (den kursiva texten) i styckena ska vara grön. Då skriver du regeln så här:

```
p em {color:green;}
```

vilket skulle ge resultatet som visas i figur 2.6.

Figur 2.6 Genom att lägga till en kontextselektor kan du få regeln att påverka enbart stycken, inte rubriken.



Eftersom du lät `em` föregås av ett `p` i selektorn, påverkar nu regeln enbart `em`-taggar inom `p`-taggar; `em`-taggen i `h2`-taggen påverkas inte längre. Observera att man skriver mellanslag mellan kontextselektorer, inte kommatecken på det sätt du såg tidigare i exemplet med grupperade selektorer.

Regler med grupperade selektorer gör att regeln appliceras på alla listade taggar, medan regler med kontextselektorer enbart appliceras på den sist listade taggen och endast om selektorn som föregår den visas i samma ordning någonstans i hierarkin ovanför den. Det har ingen betydelse hur många taggar som finns däremellan.

På grund av detta påverkas `em`-taggen inom `span`-taggen av den här regeln. Även om den inte är barn till `p`-taggen, appliceras ändå regeln eftersom den är en arvinge (descendant) till `p`-taggen. (Repetera gärna avsnittet ”Dokumenthierarki: Möt XHTML-familjen” i kapitel 1, om du tycker det här är knepigt.)

Här följer ett exempel på hur du kan använda flera taggar i selektorn för att ännu mer specifikt ange vad som ska påverkas:

```
p spam em {color:green;}
```

Detta ger resultatet som visas i figur 2.7.

Figur 2.7 Med tre element i selektorn kan du mycket specifikt ange vilken text som ska vara grön.



Regeln anger nu att endast en `em`-tagg inom en `span` inom en `p`-tagg ska påverkas; du anger ett mycket specifikt sammanhang i vilket regeln ska fungera och det är enbart en tagg som uppfyller detta villkor. I en kontextselektor som den här, kan du lista så många selektorer som du behöver för att se till att den tagg du vill modifiera påverkas.

Men det blir svårare om du endast vill att ordet ”specifik” ska påverkas. Som du såg i figur 2.5 påverkar regeln `p em {color:green;}` `em`-taggarna i båda styckena och du kan inte enbart sikta på just den här specifika taggen med hjälp av en kontextselektor av standardtyp. Här behövs en *barnselektor* (child selector).

Arbeta med barnselektorer

I kapitel 1 nämnde jag att en barntagg är en direkt arvinge till en omslutande tagg. Om du vill skriva en regel så att den tagg du vill påverka *måste* vara barn till en viss tagg, kan du göra det på följande sätt:

```
p>em {color:green;}
```

Nu har du lyckats sikta på ordet ”specifik” utan att påverka någon annan `em`-text eftersom ordet ”specifik” finns i en `em`-tagg som är barn till `p`-taggen, vilket inte är fallet med orden ”ordning som anges”, se figur 2.8.

Innan du kastar ifrån dig boken i iveren att börja använda barnselektorer i CSS-koden är det viktigt att du känner till att i skrivande stund ignoreras de fullkomligt av Internet Explorer för Windows (men Internet Explorer för Macintosh implementerar dem korrekt). Det finns emellertid sätt att kringgå detta i situatio-



Symbolen `>` används mellan två selektorer för att indikera ”barn till”.

ner när man behöver använda en barnselektor. Som du strax ska få se kan man med hjälp av klasser och ID-selektorer ange att en viss enskild tagg ska påverkas, men för att använda dem behöver man lite extra kod.

Figur 2.8 En barnselektor gör det möjligt att välja ut enbart ordet ”specifik” i den här koden.



Så tills det finns en version av Internet Explorer som kan tolka barnselektorer får vi huvudsakligen använda barnselektorer för att skapa variationer i stilmallen för att kringgå kompatibilitetsproblemen i Internet Explorer. Vi kommer att använda dem på det sättet i kapitlen som följer.

Lägga till klasser och ID-selektorer

Hittills har du sett att när man har en regel med en selektor som bara anger ett taggnamn som exempelvis `p` eller `h1`, appliceras regeln på varje förekomst av den taggen. Du har också sett att man – för att vara mer specifik i urvalsprocessen – kan använda kontextuella selektorer för att specificera taggar inom vilka måltaggar ska finnas.

Men man kan även sikta på specifika områden i dokumentet genom att lägga till ID-selektorer och klasser till taggarna i XHTML-koden. ID-selektorer och klasser erbjuder ännu en metod för att formatera ett dokument – en metod som kan fungera utan hänsyn till dokumenthierarkin.

Enkel användning av en klass

Låt oss använda den här kodsnutten för att illustrera hur man kan använda en klass:

```
<h1 class="specialtext">Det här är en rubrik med <span>samma klass</span> som det andra stycket</h1>
```

```
<p>Den här taggen har ingen klass.</p>
```

```
<p class="specialtext"> När en tagg identifierats med en klass, kan vi välja att påverka den <span>oavsett</span> dess position i hierarkin.</p>
```



När man skriver en klasselektor börjar man med att skriva en punkt.

Figur 2.9 Här använder jag en klasselektor för att applicera fet teckenstil på två olika taggar.

Observera att jag har lagt till klassattributet `specialtext` till två av dessa taggar. Låt oss nu applicera dessa stilar på koden där `specialtext` ska formateras med fet stil, se figur 2.9:

```
p {font-family: Helvetica, sans-serif;}
.specialtext {font-weight:bold;}
```



Dessa regler resulterar i att båda styckena visas med teckensnittet Helvetica (eller med webbläsarens standardteckensnitt av typen sanseriff om Helvetica inte finns tillgängligt) och att stycket med klassen `specialtext` visas med Helvetica och fet teckenstil. Observera att taggen `span`, som inte har några standardattribut, inte gör någonting alls eftersom jag inte uttryckligen formaterade den.

Kontextuella klasselektorer

Om du endast vill formatera ett stycke med klassen, skapar du en selektor som kombinerar taggnamnet och klassen enligt följande, se också figur 2.10

```
p {font-family: Helvetica, sans-serif;}
.specialtext {font-weight:bold;}
p.specialtext {color:red;}
```

Figur 2.10 Genom att kombinera ett taggnamn och ett klassnamn, kan man göra selektorn mer specifik.



Det här är en annan typ av kontextuell selektor eftersom klassen måste finnas i ett styckesammanhang för att regeln ska tillämpas.

Man kan gå ett steg längre och skriva följande, se även figur 2.11:

```
p {font-family: Helvetica, sans-serif;}
.specialtext {font-weight:bold;}
p.specialtext {color:red;}
p.specialtext span {font-style:italic;}
```

Figur 2.11 Genom att lägga till en andra selektor, kan man ännu mer exakt bestämma vilken tagg som ska formateras.



Nu är ordet ”oavsett” fett *och* kursivt eftersom det finns i en `span`-tagg som finns i ett stycke med klassen `specialtext`, precis som regeln anger. Om du även vill att den här regeln ska påverka `span` i taggen `h1`, kan man modifiera den på två sätt. Det enklaste är att inte associera klassen med någon specifik tagg, se figur 2.12:

```
.specialtext span {font-style:italic;}
```

Figur 2.12 Med en mindre specifik selektor påverkas även `span`-texten i rubriken.



Orden ”samma klass” i rubriken visas nu också med kursiv stil. Genom att ta bort det inledande `p`-taggnamnet i selektorn tar man bort kravet att klassen ska hänga ihop med någon speciell tagg, vilket gör att båda `span`-taggarna påverkas. Regeln säger att `span`-taggen kan vara en arvinge till alla taggar med klassen `specialtext` eftersom vi inte specificerat någon tagg.

Fördelen med det här tillvägagångssättet är att man kan använda en klass utan att behöva ta hänsyn till vilken tagg den tillhör, så man kringgår hierarkins arvsregler när man gör på det här sättet.

Nackdelen är att andra taggar som du inte har för avsikt att formatera också kan påverkas eftersom den här modifierade regeln är mindre specifik än tidigare. Låt säga att du senare lägger till en `span`-tagg inuti en annan tagg som redan har klassen `specialtext`, exempelvis så här:

```
<div class="specialtext">I den här div-taggen kan, <span>eller kan inte,</span>
```

```
span-taggen formateras.</div>
```

Texten inom den här `span`-taggen kommer också att kursiveras, vilket kanske inte var en önskvärd effekt, se figur 2.13.

Figur 2.13 Ju mindre specifik selektorn är, desto större risk att andra taggar påverkas på ett sätt som inte var meningen.



Om du inte vill formatera `span`-taggen i den här nya `div`-taggen kan du använda ett annat, mer fokuserat tillvägagångssätt som innebär att du använder grupperade selektorer enligt följande, se även figur 2.14:

```
p.specialtext span, h1.specialtext span {font-style:italic;}
```

Figur 2.14 Genom att använda två grupperade regler kan du välja att påverka specifika taggar.



Nu påverkas enbart de två taggarna ifråga och den nya taggen påverkas inte. De grupperade reglerna pekar inte ut just den `span`-taggen eftersom den är en arvinge till en `div`, men om du använder

det enklare och mindre specifika tillvägagångssättet med `.special-text span`, påverkas också den.

Det här kan tyckas vara mycket att tänka på när man bara formaterar ett enkelt fyra raders exempel som det här, men när du arbetar med en stilmall som kanske är hundratals rader lång, måste du hålla dessa förhållanden i minnet. Vi återkommer till detta i senare kapitel.

ID-selektorer

ID-selektorer skrivs på liknande sätt som klasser, med den skillnaden att man använder #-symbolen (bräddgårdstecken eller hash) för att indikera dem. Som du nog kommer ihåg, indikeras klasser med en punkt (.).

```
#specialtext {font-weight:bold;}
```

Om ett stycke har en ID-selektor enligt följande:

```
<p id="uniktext">Det här är den speciella texten</p>
```

ser motsvarande kontextuella selektor ut så här:

```
p#uniktext {några CSS-regler placeras här}
```

Förutom detta, fungerar ID-selektorer på samma sätt som klasser och allt i beskrivningen ovan av klasser gäller på samma sätt för ID-selektorer. Så vad är då skillnaden?

Skillnaden mellan ID-selektorer och klasser

Hittills har de aspekter på klasser och ID-selektorer som vi har tittat på fått det att se ut som om de är utbytbara – vi har använt dem båda två för att identifiera en viss tagg i koden. Men en ID-selektor är kraftfullare än en klass, ungefär på samma sätt som damen är starkare än bonden i schack. (Du får se hur sant detta är när vi tittar på begreppet ”exakthet” (specificity) i avsnittet ”Vad innebär Cascading?” senare i det här kapitlet.) För att ta den här damen/bonden-analogin ett steg längre kan, enligt reglerna för XHTML, enbart en enda instans av en viss ID-selektor (t.ex. `id="huvudmeny"`) finnas på en sida, men en klass (t.ex. `class="kursivstycke"`) kan förekomma många gånger.

Om du vill identifiera en unik del av sidans kod, exempelvis den primära navigeringsmenyn som du vill att en speciell uppsättning CSS-regler ska peka på, använd då en ID-selektor. För att identifiera ett antal stycken på en sida som alla ska ha samma stil, använd en klass.

Du kan även använda en ID-selektor för att aktivera ett javascript vid en tagg (exempelvis för att aktivera en DHTML-animering när användaren placerar muspekaren på en länk). Ni JavaScript-fantaster vill kanske veta att `id`-attributet ersätter det förlegade attributet `name` (vilket XHTML-valideringen anger som ogiltigt) som tidigare användes för samma ändamål. Det är särskilt viktigt att du ser till att JavaScript-relaterade ID-selektorer förekommer endast en gång på en sida, i annat fall kan javascriptet bete sig oberäkneligt.

Använd klasser med måtta

Vanligtvis bör man använda ID-selektorer och klasser sparsamt. Rätt användning är att placera dem i `div`-taggar som innehåller huvudavsnitten av koden och sedan komma åt taggarna inom dem med hjälp av kontextuella selektorer som börjar med ID-selektor- eller klassnamnet.

Vad man vill undvika är något som Jeffrey Zeldman beskriver som "classitis – en kods variant av mässlingen", där man bifogar en unik klass eller ID-selektor till i princip varje tagg i koden och sedan skriver en regel för var och en av dem. Det här är bara ett steg ifrån att ladda koden med `FONT`-taggar och annan ovidkommande kod. Den gode doktor Zeldman har botat mig och många andra från den här otrevliga åkomman. Om du redan sitter fast i ovanan slänga in klasser i samband med varje tagg, som de flesta av oss gör när vi entusiastiskt ger oss på CSS, kan du ta en titt på kodexemplet i kapitel 1 utifrån vad du nyss läst i det här kapitlet. Du ser då att man enkelt kan ange stilar för varje tagg utan att lägga till fler ID-selektorer eller klasser.

Om du använder ID-selektorer och klasser för att identifiera endast huvudavsnitten i koden – och taggar som inte specifikt kan pekars på med kontextuella taggbaserade selektorer – kan det aldrig bli helt galet. Detta har även den fördelen att det gör stilmallen enklare och renare.

Man kan visserligen använda flera `id`-attribut på en sida, men vart och ett av dem måste ha ett unikt värde (namn) som identifierar det. Du kan applicera en viss klass på så många taggar som du behöver. Låt oss nu ta en snabbtitt på några andra selektorer som du kanske inte använder lika ofta som kontextuella selektorer, klasser och ID-selektorer, men som ändå erbjuder stora möjligheter.

Den universella selektorn

Selektorn `*` (asterisk) betyder ”allt”, så om du skriver:

```
* {color:green;}
```

i stilmallen blir all text grön utom på de ställen där du med hjälp av andra regler specificerat att den ska ha någon annan färg. Ett annat intressant användningsområde för den här selektorn är som en slags invertering av barnselektorn – alltså en icke-barnselektor, om du så vill:

```
p * em {font-weight:bold;}
```

Här väljs alla `em`-taggar som är åtminstone barnbarn, men inte barn, till taggen `p`. Det har ingen betydelse vilken `em`-taggens föräldertagg är.

Den intelligande syskonselektorn

Den här regeln väljer en tagg som följer en viss syskontagg (sibling tag). Syskontaggar finns på samma nivå i kodhierarkin – de har alltså samma föräldertagg. Här följer ett exempel:

```
h1 + p {font-variant:small-caps;}
```

Applicerar vi den här regeln på koden:

```
<div>
  <h1>Allt om syskonselektorer</h1>
  <p>Det måste finnas åtminstone två selektorer, och ett +-tecken
    före den sista.</p>
  <p>Målselektorn kommer att påverkas endast om den också är
    ett syskon till, och föregås av, selektorn före +-tecknet.</p>
</div>
```

blir resultatet det som visas i figur 2.15, eftersom endast det första stycket föregås av taggen `h1` som är ett syskon.

Figur 2.15 Syskonselektorer fungerar med utgångspunkt från vilken tagg som föregår dem i koden och båda måste vara nästlade på samma nivå. Det här är en av de knepigaste selektorerna att förstå.



Som du kan se uppfyller taggen `p` som följer efter `h1` villkoret i regeln och visas därför med kapitäl. Den andra `p`-taggen som inte finns intill `h1`, påverkas inte. (Det här gäller inte för Internet Explorer för Windows.)

Attributselektorer

Attributselektorer använder taggens attribut. De används framför allt i XML men kan även användas i XHTML.

Den här regeln:

```
img[title] {border: 2px solid blue;}
```

gör att en `img` med en `title`-tagg på följande sätt:

```

```

omges av en blå, 2 pixlar bred kantlinje. Det har ingen betydelse vilket värdet för `title`-attributet är, huvudsaken är att det finns ett. Du skulle kunna använda en sådan stil för att indikera för användaren att om han/hon pekar på bilden så visas ett skärmtips (en popup-text som genereras av `title`-attributet). (Det är en vanlig praxis att duplicera `alt`- och `title`-attributvärdena – `<alt>`-taggens text visas om bilden inte kan laddas ner eller läses upp av ett uppläsningssprogram, och `title` gör att ett skärmtips visas när användaren pekar på bilden.)

Man kan också mycket specifikt ange vilket värde attributet ska ha. Regeln:

```
img[alt="Stylin logo"] {border: 2px solid blue;}
```

placerar bara en kantlinje runt bilden om bildens `alt`-attribut är "Stylin logo"; det vill säga om koden för bilden ser ut ungefär så här:

```

```

Den här regeln görs mer användbar om den låter dig specificera enbart första tecknen i attributvärdet, men den "gemensamma" delen av attributet måste separeras från den "annorlunda" delen av attributet med ett bindestreck. Som om du noggrant försett dina `img`-taggar med attribut som de här:

```

```

```

```

kan du därmed välja dem genom att lägga till ett vertikalt streck (Alt Gr+>-tangenten) i regeln på följande sätt:



Texterna i alt-taggarna för bilderna i exemplen är avsiktligt mycket kortfattade för att de ska bli tydliga. Men med tanke på tillgänglighet bör man alltid skriva alt-texter som är meningsfulla för användare som inte kan se bilden.

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

Den här regeln skulle förresten också påverka det här exemplet:

```

```

även om det här exemplets alt-tagga inte har det med bindestreck försedda tillägget i värdet.

Sammanfattning av selektorer

Hittills har du sett att man kan tilldela CSS-regler på flera olika sätt: genom att använda taggselektorer, genom att använda klass- och ID-selektorer, genom att använda selektorer som är en kombination av båda och till och med genom att välja vad som ska påverkas med utgångspunkt från de attribut som är bifogade till taggen.

En vanlig aspekt på dessa selektorer är att samtliga siktar på att påverka någonting i koden – ett taggnamn, en klass, ett ID-värde, ett attribut eller ett attributvärde. Men vad händer om du vill att en stil ska tillämpas när en viss händelse inträffar, exempelvis när användaren pekar på en länk? Du är alltså kort och gott ute efter ett sätt att kunna applicera regler som baseras på händelser. Nu har du antagligen redan förstått att det är precis vad vi kommer att titta på härnäst.

Pseudoklasser

Den här benämningen har de fått därför att de är klasser som egentligen inte är bifogade till taggar i koden. Pseudoklasser gör att regler appliceras när vissa händelser inträffar. Den vanligaste händelsen är att användaren pekar eller klickar på någonting. Men de nyare webbläsarna (tråkigt nog inte Internet Explorer; åtminstone inte utan att lägga till en speciell JavaScript-funktion) är det enkelt att få vilket objekt som helst på skärmen att svara på en rollover – alltså när man rör muspekaren över någonting, vilket också ibland kallas att "hovra". En pseudoklass kan exempelvis användas för att få en kantlinje att visas runt en bild när muspekaren förs över den.

Pseudoklasser för länkar

Pseudoklasser används mest i samband med hyperlänkar (a-taggar) och möjliggör saker som att länkens färg ändras eller att understrykningen försvinner när muspekaren förs över länken.

Det finns fyra pseudoklasser för länkar eftersom länkar alltid befinns i något av följande fyra stadier:

Link. Länken bara finns där och väntar på att någon ska klicka på den.

Visited. Användaren har klickat på länken.

Hover. Användaren har placerat muspekaren på länken.

Active. Användaren håller just på att klicka på länken.

Här är motsvarande pseudoklasselektorer för dessa stadier (selektorn `a` används tillsammans med några exempeldeklarationer):

```
a:link {color:black;}
a:visited {color:gray;}
a:hover {text-decoration:none;}
a:active {color:navy;}
```



Det tydliga kolonet (:) i selektorn ropar (nåja, indikerar) "Jag är en pseudoklass!"

Vi sparar resonemanget om lämpliga länkfärger och länkbeteenden till senare och konstaterar nu bara att enligt deklARATIONERNA ovan är länkar ursprungligen svarta (och som standard understrukna). När muspekaren förs över dem försvinner understrykningen och de förblir svarta eftersom ingen färg har definierats för det här stadiet (hover-stadiet). När användaren trycker ned musknappen på länken (active-stadiet) blir den blå och därefter visas den alltid (eller rättare sagt, till dess att uppgiften om den här länkens URL-adress försvinner från webbläsarens historik eller tas bort av användaren) med grå färg. När man använder dessa pseudoklasselektorer har man fullständig kontroll över en länks utseende och beteende vid de fyra stadierna.

Allt detta är ju bra nog, men det verkligt fantastiska kommer när du börjar använda dessa pseudoklasser för länkar som en del i kontextuella selektorer. Du kan då skapa varierande utseenden och beteenden för olika grupper av länkar i din design – exempelvis länkar i navigeringsfält, sidfötter, rutor och text. Vi ska titta närmare på hur man använder dessa pseudoklasser för att formatera länkar och andra saker senare i boken, men för tillfället nöjer vi oss med att konstatera följande innan vi går vidare:

Man måste inte definiera alla fyra stadierna. Om du enbart vill definiera stadierna `link` och `hover`, går det bra. Ibland är det inte befogat att ha länkar som visar att de använts.

En webbläsare kan strunta i en del av dessa regler om man inte placerar dem i den ordning som visas ovan: `link`, `visited`, `hover`,

active. Kanske det cyniska utropet ”LoVe-HA!” kan hjälpa dig att komma ihåg ordningen.

Du kan använda vilken selektor som helst tillsammans med dessa pseudoklasser, inte bara `a`, för att skapa alla typer av rollovereffekter. Exempelvis:

```
p:hover {background-color: gray;}
```

Ovanstående kod kommer att ... nej förresten, jag tror inte att jag behöver tala om för någon som är så smart som du vad som då kommer att hända med ditt stycke.

(Som jag nämnde tidigare bör du komma ihåg att äldre webbläsare enbart stöder rollovereffekter på ankarlänkar och även att senaste versionen av Internet Explorer behöver lite hjälp på traven i det här fallet, något du får se senare i boken.)

Andra användbara pseudoklasser

Ändamålet med pseudoklasser är att simulera klasser som lagts till i koden när särskilda omständigheter inträffar. Vi har tittat på hur de tillämpas som svar på användarens åtgärder, att han/hon exempelvis klickar eller pekar, men de kan även tillämpas utifrån att vissa villkor uppfylls i koden.

:FIRST-CHILD

Där x är ett taggnamn

```
x:first-child
```

Den här pseudoklassen påverkar det första barnelementet med namnet `x`. Om exempelvis den här regeln:

```
div.weather strong:first-child {color:red;}
```

appliceras på den här koden:

```
<div class="weather">
```

```
  Det är <strong>mycket</strong> varmt och <strong>otroligt</strong> fuktigt.
```

```
</div>
```

visas `mycket` med röd teckenfärg medan `otroligt` inte gör det.

:FOCUS

Där x är ett taggnamn

```
x:focus
```

Ett element som exempelvis ett textfält i ett formulär sägs ha ”focus” när användaren klickar i det; det är där tecknen visas när användaren skriver. Koden:

```
input:focus {border: 1px solid blue;}
```

placerar en blå kantlinje runt ett sådant fält när användaren klickar i det.

Pseudoelement

Pseudoelement har den effekten att extra kodelement dyker upp i dokumentet som genom ett trollslag, trots att du faktiskt inte lagt till någon extra kod. Här kommer några exempel.

Den här pseudoklassen:

Där x är ett taggnamn — `x:first-letter`

```
p:first-letter {font-size:300%; float:left;}
```

gör att du t.ex. kan skapa en stor anfangeffekt i början av ett stycke.

Den här pseudoklassen:

Där x är ett taggnamn — `x:first-line`

gör att du kan formatera första raden i (vanligtvis) ett stycke med text. Exempelvis koden:

```
p:first-line{font-variant:small-caps;}
```

resulterar i att första raden visas med kapitäl. Om man har en flytande layout där radlängden ändras i enlighet med webbläsarfönstrets storlek, ändras automatiskt ordens formatering på det sätt som krävs för att enbart första raden ska formateras på det här sättet.

Dessa två pseudoklasser:

`x:before` och `x:after`

gör att angiven text läggs till före och efter ett element, så den här koden:

```
<h1 class="age">25</h1>
```

och de här stilarna:

```
h1.age:before {content:"Ålder: "}
```

```
h1.age:after {content:" år gammal."}
```

resulterar i att texten ”Ålder: 25 år gammal.” visas. Observera att mellanslagen som lagts till i innehållssträngarna inom citattecken ser till att utmatningen visas med mellanslag på rätt sätt. Dessa två selektorer är särskilt användbara när taggens innehåll ska genereras som ett resultat av en databasfråga. Om allt resultatet innehåller är



Det finns fyra andra pseudoklasser. Den första är `:lang`, vilken appliceras på element med en specifik språkkod. De andra tre är `:left`, `:right` och `:first`, vilka appliceras på medier som innehåller sidor (utskrift) i stället för på innehåll som visas i en webbläsare. De används sällan och stöds dåligt av webbläsarna så jag går inte närmare in på dem här.



Eftersom sökmotorer inte kan tolka pseudo-elementinnehåll (de visas inte i koden) bör du inte använda dessa element för att lägga till viktig innehåll som du vill att en sökmotor ska indexera.

siffran, medger dessa selektorer att du förser uppgiften med ett meningsfullt sammanhang när du visar den för användaren.

Arv

Precis som de slantar som du hoppas få arva en dag efter din rika gamla farbror, innebär arv i CSS att något arvs från förfäder till arvingar. Men här handlar det inte om feta sedelbuntar utan om värdena hos olika CSS-egenskaper.

Du kanske kommer ihåg från beskrivningen om dokumenthierarkin i kapitel 1 att `body`-taggen är förfader till dem alla – alla CSS-formaterade taggar i koden härstammar från den. Om du formaterar `body`-taggen så här:

```
body {font-family: verdana, helvetica, sans-serif; color:blue;}
```

så ärver texten i samtliga textelement i hela dokumentet – tack vare kraften i arvsprincipen i CSS – dessa stilar och visas i blått med teckensnittet Verdana (eller med något av de andra alternativen om Verdana inte finns tillgängligt), oavsett hur långt ner i hierarkin de befinner sig. Effektiviteten är uppenbar; i stället för att behöva specificera det önskade teckensnittet för alla taggar behöver du bara ange det en gång på det här sättet som det primära teckensnittet för hela webbplatsen. Sedan behöver du enbart ange teckensnittsegenskaper för taggar som ska visas med något annat teckensnitt.

Många CSS-egenskaper ärvs på det här sättet, framför allt textattribut. Men många CSS-egenskaper ärvs *inte* eftersom arv i samband med dem inte är befogat. Dessa egenskaper kan i första hand relateras till positionering och visning av boxelement, t.ex. kantlinjer, marginaler och padding (mellanrum, luft). Låt säga att du vill skapa en ruta med text. Du vill kanske göra detta genom att använda en `div` (som du kan tänka på som en rektangulär box) i vilken det finns flera stycken och formatera denna `div` med en kantlinje som är röd och 2 pixlar bred. Men du har inget behov av att alla stycken inom denna `div` automatiskt också skulle få en röd kantlinje. Det får de inte heller – kantlinjeegenskaper ärvs inte. När vi tittar på boxmodellen i kapitel 4 tittar vi också närmare på det här med arv.

Du måste också vara försiktig när du arbetar med relativa storlekar som procentvärden och fyrkanter (`em`). Om du anger att en taggs text ska vara 80 procent och den är arvinge till en tagg vars text också har storleken angiven till 80 procent kommer alltså texten att bli 64 procent (80 procent av 80 procent), vilket kanske inte är



Just nu behöver du bara komma ihåg att stilar som kan relateras till text och textfärg ärvs av elementarvingar. Stilar som kan relateras till boxar som skapats genom att man formaterat `div`-taggar, stycken eller andra element, exempelvis kantlinjer, padding, marginaler och bakgrundsfärger, ärvs däremot inte.

den effekt du var ute efter. I kapitel 3 tittar vi på för- och nackdelar med att ange absolut och relativ storlek.

Vad innebär "cascading"?

Okej, nu har vi tillräckligt med information för en meningsfull diskussion om en av de svåraste aspekterna beträffande CSS att ta till sig – kaskaden. Om det här avsnittet känns för mastigt, hoppa då över det och läs rutan "Charlies enkla sammanfattning om kaskaden" senare i det här kapitlet. Den rutan är en förenklad, kanske lite mindre exakt, version som är tillräcklig till dess att du gjort lite CSS-kodning och verkligen behöver detaljinformationen.

Som namnet antyder innebär kaskaden att stilarna faller nedåt från en nivå i dokumenthierarkin till nästa och dess funktion är att låta webbläsaren bestämma vilken av de många möjliga källorna för en viss egenskap för en viss tagg som är den rätta att använda.

Kaskaden är en kraftfull mekanism. Förstår du den blir det enklare att skriva CSS på det mest ekonomiska och redigerbara sättet och du kan skapa dokument som visas på det sätt som du avsett, och överlåta vederbörlig kontroll över aspekter på dokumentvisningen, exempelvis teckenstorleken, till användare med särskilda behov.

Källor för format

Stilar, eller format, kommer från många platser. Det är inte så svårt att acceptera att det måste finnas en webbläsarformatmall (standardformatmallen) gömd någonstans inuti webbläsaren, eftersom varje tagg fastställer stilar utan att man skriver några. Taggen **h1** t.ex. skapar fet stil, taggen **em** skapar kursiv stil och listor visas med indrag och punkter vid varje listpost, och allt detta sker automatiskt. Man behöver inte formatera något för att den här formateringen ska ske.

Om du har Firefox installerad i datorn kan du söka efter dess standardformatmall. Det är fritt fram att modifiera den på egen risk – om du nu skulle känna för det.

Sedan finns det förstås också användarformatmallen. Användaren kan också skapa en formatmall, även om det är väldigt få som gör det. Den här möjligheten är praktisk för bland annat användare med nedsatt syn eftersom de då kan öka baslinjestorleken för text eller ange att texten ska visas med färger som de lätt kan skilja från varandra. Man kan lägga till en användarformatmall i Internet Explorer i Windows genom att välja Verktyg > Internet-alternativ



Läs mer om standard-formatmallar för webbläsare på Eric Meyers blogg (www.meyerweb.com/eric/thoughts/2004/09/15/emreallyem-undoing-htmlcss).

och klicka på knappen Hjälpmedel. Den här funktionen låter t.ex. användare med nedsatt syn lägga till en stil som exempelvis:

```
body {font-size:200%;}
```

vilken gör att all text visas med dubbla teckenstorleken – allt i enlighet med arvsprincipen! Det här är anledningen till att det är så viktigt att specificera text i relativa storlekar som t.ex. em-måttenheter (em kallas på svenska helfyrkant) i stället för i fasta storlekar som t.ex. punkter, så att man inte åsidosätter sådana ändringar. Vi återkommer till detta intressanta ämne i kapitel 3.

Sedan finns det också författarstilmallar, som är skrivna av dig, författaren. Vi har redan diskuterat källorna till dessa: länkade stilmallar, inbäddade stilmallar högst upp på sidorna och lokala (inline) stilar som är bifogade till taggar.

Här är den ordning i vilken webbläsaren tittar på de olika platserna:



Kaskaden definierar vilken stilmall som webbläsaren tittar på och i vilken ordning, samt vilken stil som "vinner" om en stil definierats på två eller fler ställen.

- Webbläsarens standardformatmall.
- Användarformatmallen.
- Författarstilmallen.
- Inbäddade stilar som skapats av författaren.
- Lokala (inline) stilar som skapats av författaren.

Webbläsaren uppdaterar sina inställningar för varje taggs egenskapsvärden (om sådana definierats), allt eftersom den stöter på dem, och tittar i tur och ordning på varje plats. De är definierade i webbläsarens standardformatmall och webbläsaren uppdaterar alla som också definierats på de andra platserna. Om exempelvis författarstilmallen definierar att taggen `<p>` ska visas med teckensnittet Helvetica samtidigt som en inbäddad stil anger att taggen `<p>` ska visas med teckensnittet Verdana, kommer stycket att visas med teckensnittet Verdana – den inbäddade stilen läses efter författarstilmallen. Men om det inte finns någon stil för stycken vare sig i användarformatmallen eller i författarstilmallen, kommer de att visas med Times, eftersom det är det teckensnitt som specificeras i standardformatmallarna för alla webbläsare.

Det här är grundprincipen bakom "kaskaden", men det finns flera regler som styr hur kaskaden fungerar.



På webbplatsen W3C hittar du mer information om kaskaden och hur den fungerar (www.w3.org/TR/CSS2/cascade.html).

Regler för kaskaden

Förutom att känna till i vilken ordning stilar tillämpas bör du också känna till några regler om hur kaskaden fungerar.

Regel 1: Hitta alla deklARATIONER som gäller varje element och egenskap. När webbläsaren laddar varje sida tittar den på varje tagg på sidan för att se om det finns en regel som matchar den.

Regel 2: Sortera utifrån ordning och vikt. Webbläsaren söker i tur och ordning igenom de fem källorna och fastställer varje matchande egenskap allt eftersom. Om en matchande egenskap definieras senare i ordningen, uppdaterar webbläsaren värdet och gör detta flera gånger om det behövs till dess att alla de fem möjliga platserna för varje taggs egenskaper på sidan har kontrollerats. Vadhelst en egenskap är angiven till i slutet av den här processen, är detta det sätt på vilket den visas.

I tabell 2.1 tittar vi på den här processen för en sida med flera `p`-taggar. Anta att två av dessa `p`-taggar har lokala (inline) stilar som definierar att de ska visas med röd färg.

I det här fallet är texterna i varje `p`-tagg blå, förutom de med den lokala stilen – som ju är röda.

Tabell 2.1 Kaskadexempel

Plats	Tagg	Egenskap	Värde
Standardformatmall	P	color	black
Användarformatmall			
Författarformatmall	P	color	blue
Inbäddade stilar som skapats av författaren			
Lokala (inline) stilar som skapats av författaren	P	color	red

Naturligtvis är det hela inte fullt så enkelt. Sedan tillkommer också deklARATIONENS *vikt*. Man kan definiera en deklARATION som är lika viktig så här:

Observera utrops-tecknet

```
p {color:red !important; font-size:12pt;}
```

Ordet `!important` följer på ett mellanslag efter den stil som du vill göra viktig, men före det avskiljande semikolonet (;).

Den här stilen anger att textens röda färg är viktig och därför visas den också så, även om den deklarerats som någon annan färg läng-

re ner i kaskaden. Tänk efter länge och väl innan du påtvingar användaren en viss stil med hjälp av **!important**, eftersom du då kanske förstör någons personliga formatmall som av goda skäl kanske angivits på just det sättet. Se till att du är absolut säker på att det är viktigt att en sådan stil prioriteras framför andra möjliga formateringar av den taggen.

Charlies enkla sammanfattning om kaskaden

I den här förenklade versionen av reglerna för kaskaden behöver du bara komma ihåg tre saker. Dessa gäller i praktiskt taget alla lägen.

Regel 1: Selektorer med ID-värden vinner över selektorer med klasser, vilka i sin tur vinner över selektorer med enbart taggar.

Regel 2: Om samma egenskap för samma tagg definierats på mer än ett ställe i kaskaden, vinner lokala (inline) stilar över inbäddade stilar, vilka vinner över stilar i stilmallen. Regel 2 förlorar dock mot regel 1 – om selektorn är mer specifik vinner den, var den än finns.

Regel 3: Definierade stilar vinner över ärvda stilar oavsett exakthet (specificity).

Regel 3 kräver dock en liten förklaring. Följande kod:

```
<div id="kaskaddemo">
<p id="arvsfakta">Arvsprincipen är <em>svag</em> i kaskaden.</p>
</div>
```

och den här regeln som har hög exakthet (specificity):

```
2 - 0 - 4 — html body div#kaskaddemo p#arvsfakta {color:blue;}
```

resulterar i att all text, inklusive ordet *svag*, visas med blå färg eftersom **em** ärver färgen från sina föräldrar, **p**-taggen.

När vi lägger till den här regeln för **em**, även om den har mycket låg exakthet (specificity):

```
0 - 0 - 1 — em {color:red}
```

blir **em**-texten röd. Den ärvda stilen åsidosätts av den definierade stilen för **em**, oavsett den höga exaktheten i regeln för stycket den står i.

Tre enkla regler för kaskaden. Mycket enklare, eller hur?

Regel 3: Sortera utifrån exakthet (specificity). Exaktheten (specificity) bestämmer hur specifik en regel är. Jag försökte förbereda dig på den här idén genom att använda ordet specifik på exakt det här sättet många gånger när vi talade om selektorer. Om en stilmall innehåller den här regeln:

```
p {font-size:12px;}
```

och den här regeln:

```
p.stortext {font-size:16px;}
```

och sedan den här koden:

```
<p class="stortext">En bit text</p>
```

blir texten 16 pixlar på höjden eftersom den andra regeln är mer specifik, och alltså vinner.

Det här kan tyckas ganska självklart och lätt att förstå, men vad händer med den kodsnutten om du i stället använder de här stilarna?

```
p {font-size:12px;}
.stortext {font-size:16px;}
```

Båda dessa regler matchar taggen, men klassen vinner och texten blir 16 pixlar. Anledningen till detta är följande: Den numeriska exaktheten i taggselektorn är 1, men klassen har en exakthet på 1-0. Så här räknar man ut exaktheten för en selektor. Det finns ett enkelt poängsystem för varje stil som du placerar i en trevärdes-layout som ser ut så här:

A - B - C

Strecken är avskiljare, inte minustecken. Så här fungerar poängsystemet:

1. Lägg till 1 till A för varje ID i selektorn.
2. Lägg till 1 till B för varje klass i selektorn.
3. Lägg till 1 till C för varje elementnamn (taggnamn).

Resultatet blir ett tresiffrigt tal. (Det är egentligen inte ett tresiffrigt tal, det är bara det att i de flesta fall fungerar det att läsa resultatet som ett tresiffrigt tal. Försök bara förstå att du kan få ett resultat som 0-1-12 och att 0-2-0 ändå är mer exakt (specifikt).

Låt oss titta på exaktheten (specificity) hos dessa exempel:

0-0-1 exakthet = 1	—	p
0-1-1 exakthet = 11	—	p.largetext
1-0-1 exakthet = 101	—	p#largetext
1-0-2 exakthet = 102	—	body p#largetext
1-1-3 exakthet = 113	—	body p#largetext ul.mylist
1-1-4 exakthet = 114	—	body p#largetext ul.mylist li

Varje exempel har en högre exakthet än det föregående.

Regel 4: Sortera utifrån ordningen. Om två regler har exakt samma tyngd, vinner den som finns längst ner i kaskaden.



Exakthet (specificity) är viktigare än ordning, så en mer specifik regel högre upp i kaskaden vinner över en mindre specifik regel längre ner i kaskaden.

Och, kära läsare – det var kaskaden och visst, den är lite knepig att förstå, särskilt om du ännu inte har så stor erfarenhet av CSS. På sidan 56 finns min förenklade version av reglerna för kaskaden, vilka gäller i nittioåtta procent av alla fall. Om du upptäcker att något inte betar sig som du vill när du använder den förenklade versionen hänvisar jag till ovanstående regler.

Regler för deklARATIONER

Så här långt har jag fokuserat på hur man använder selektorer för att påverka taggar, men vi har ännu inte tittat särskilt mycket på den andra halvan av en CSS-regel: deklARATIONEN. Jag har använt flera olika deklARATIONER för att illustrera exemplen med selektorer men jag har bara förklarat dem helt kort. Nu är det dags att titta på deklARATIONER mer i detalj.

Diagrammet som visar strukturen i en CSS-regel tidigare i det här kapitlet (figur 2.2) visar att en deklARATION består av två delar: en egenskap och ett värde. *Egenskapen* fastställer vilken aspekt av elementet som ska påverkas (dess färg, dess höjd osv.) och *värdet* talar om vad egenskapen är angiven till (grön, 12px osv.).

Varje element har ett antal egenskaper som kan ställas in med hjälp av CSS och dessa varierar från element till element. Man kan t.ex. ställa in egenskapen `font-size` för text, men inte för exempelvis en bild. I vart och ett av kapitlen som följer använder jag exempel som hämtats från verkligheten för att visa de egenskaper du kan ställa in för olika element samt de värden du kan ange för dessa egenskaper. Eftersom det endast finns några få typer av värden ska vi titta på dem nu.

Värden kan delas in i tre huvudtyper:

Ord. I exempelvis `font-weight:bold`, är `bold` en typ av värde.

Numeriska värden. Numeriska värden följs vanligtvis av en enhetstyp. I exempelvis `font-size:12px`, är 12 det numeriska värdet och px enhetstypen – pixlar i det här fallet.

Färgvärden. Färgvärden skrivs `color:#336699` där färgen i det exemplet definierats med ett hexadecimalt värde.

Det är inte mycket jag kan berätta om ordvärden som skulle säga dig särskilt mycket förrän du börjar använda dem, eftersom de är specifika för varje element. Men numeriska värden och färgvärden kan endast uttryckas på vissa sätt.

Numeriska värden

Man kan använda numeriska värden för att beskriva längden (jag använder ordet ”längd” generiskt för både höjd, bredd, tjocklek osv.) på alla typer av element. Dessa värden kan delas in i två huvudgrupper: absoluta och relativa.

Absoluta värden (tabell 2.2) beskriver längden i ”verkligheten” (t.ex. 6 cm), i motsats till relativa mått som helt enkelt anges som ett förhållande till någon annan mätbar företeelse (när man säger ”två gånger så lång” är denna måttangivelse relativ till någonting annat).

Tabell 2.2 Absoluta värden

Absolut värde	Enhetsförkortn.	Exempel	Motsv. i cm
Tum	in	height:6in	2,54
Centimeter	cm	height:40cm	1
Millimeter	mm	height:500mm	0,1
Punkter	pt	height:60pt	0,0352
Picas	pc	height:90pc	0,423
Pixlar	px	height:72px	0,0352

När jag skriver CSS som gäller element med fixerad storlek, t.ex. bilder, använder jag enbart måttenheten pixlar. Du väljer förstås själv vad du vill använda, men genomgående i den här boken är pixlar den enda absoluta måttenhet jag använder, utom i stilmallar för utskrift – eftersom papper mäts i centimeter är det klokare att använda samma måttenhet även i samband med layouter för utskrift.

Medan absoluta enheter är enkla att förstå kräver relativa värden (tabell 2.3) lite mer ingående förklaring.

Tabell 2.3 Relativa värden

Relativt värde	Enhetsförkortning	Exempel
Helfyrkant	em	height:1.2em
Ex	ex	height:6ex
Procent	% height:	120%

Varför man bör använda måttenheten em för att ange teckenstorlek

Det finns två viktiga fördelar med att använda en metod för relativ storleksbestämning, t.ex. måttenheten em, för att ange teckenstorleken.

- Du kan dra nytta av arvsprincipen genom att deklarerar att body-elementet ska ha storleken 1 em och detta blir då en storleksbestämmande baslinje eftersom texten i alla andra element storleksanpassas i förhållande till den. Eftersom textinnehållet alltid finns inuti andra element, exempelvis `p` och `h4`, skriver du sedan helt enkelt regler som t.ex. anger att `p`-taggen är `.8em` och textlänkar `.7em`. På så sätt etablerar du proportionella storleksförhållanden mellan alla textelementen i din design.

Observera att när du i Internet Explorer anger en em-storlek för body-elementet storleksanpassas automatiskt styckena proportionerligt därefter, men däremot inte `h1-h6`; för dessa måste du uttryckligen ange relativa storlekar (t.ex. `1.1em` för `h1`, `9em` för `h2` osv.), i annat fall visas de med sina respektive standardstorlekar.

Om du senare bestämmer dig för att öka textstorleken genomgående på hela webbplatsen, kan du gå tillbaka till `body`-taggen och ange dess storlek till exempelvis `1.2 em`. Som genom magi ökar då teckenstorleken för all text proportionerligt utifrån detta (med en femtedel i det här fallet) eftersom alla de andra taggarna ärver sin storlek från `body`-taggen.

- Om du inte definierar teckenstorleken med relativa måttenheter, inaktiverar du effektivt funktionerna för att ange teckenstorlek som finns tillgängliga på Visa-menyn i Internet Explorer (men andra webbläsare kan storleksändra absoluta enheter för teckenstorlek) och saboterar därmed för användare med nedsatt synförmåga som förlitar sig på den funktionen för att visa innehållet med en storlek som de kan läsa. Du bör under hela utvecklingsprocessen regelbundet kontrollera att en ökning av teckenstorleken på det här sättet inte förstör sidornas struktur.

Av dessa två anledningar råder jag dig att alltid ange alla teckenstorlekar i måttenheten em i stället för i absoluta måttenheter som t.ex. pixlar. Om du designar en rad med tabbar i ett fixerat vågrätt område finns risken att layouten havererar om texten storleksändras. Men om du är försiktig och utformar layouten med detta i åtanke kan du utveckla sådana komponenter i din design så att de även kan rymma större text när storleken ändras av användaren.

Em och ex är båda måttenheter som används i samband med teckenstorlek. Em (helfyrkant) motsvarar tecknens bredd i ett visst teckensnitt så dess storlek varierar beroende på vilket teckensnitt som används. Ex motsvarar x-höjden i ett visst teckensnitt (har fått den benämningen eftersom det avser höjden på den gemena bokstaven x, med andra ord, mellandelen utan de uppåt- och nedåtgående staplar som förekommer i tecken som exempelvis p och d).

Procentvärden är användbara när man vill bestämma bredden på behållarelement, t.ex. `div`-element, till proportionerna på webbläsarfönstret, vilket är det enda sättet att skapa ”elastiska” layouter som smidigt ändrar storlek allt eftersom användaren ändrar webbläsarfönstrets storlek. Användningen av procenttal är också rätta sättet att få proportionellt radavstånd, vilket är avståndet från baslinjen för en textrad till baslinjen för nästa textrad i ett text-

block med flera rader, exempelvis ett stycke. Du får läsa mer om radavstånd i kapitel 3.

Färgvärden

Du kan använda flera värdetyper för att specificera färg. Välj bland nedanstående alternativ det du föredrar.

Hexadecimalt (#RRGGBB och #RGB). Om du redan är bekant med språk som C++, PHP eller JavaScript känner du också till den hexadecimala notationen för färg. Formatet ser ut så här:

`#RRGGBB`

I det här värdet definierar de två första bokstäverna färgen rött, nästa två grönt och de sista två blått. Datorer använder enheter med två för att räkna istället för basen 10 som vi dödliga är vana vid, och det är därför som det hexadecimala systemet använder basen 16 (2 upphöjt i 4) med hjälp av de 16 tal/bokstäverna 0–9 och A–F. A till F fungerar effektivt för 10 till och med 15. Eftersom färg representeras av ett par av dessa tal med basen 16, finns det 256 (16 x 16) möjliga värden för varje färg eller 16 777 216 kombinationer (256 x 256 x 256) av färger. Du får definitivt de flesta färgalternativen genom att välja den hexadecimala notationen även om du i praktiken kan klara dig med mycket mindre. Du får emellertid svårt (för att inte tala om din bildskärm) att urskilja skillnaden mellan två intilliggande hexadecimala färger. Glöm inte att ange #-symbolen (brädgårdstecken) framför värdet.



De flesta färger är tämligen svåra att gissa sig till vid en första anblick. #7CA9BE är en dunkelt grönbå färg. Men om du bara tittar på första värdet i varje RGB-par, 7, A och B i det här fallet, så ser du att rött ligger något under hälften av 16, det maximala värdet, och grönt och blått ligger högre och har ungefär samma värde. Med tillgång till den informationen är det enklare att göra en kvalificerad gissning om vilken färg värdet representerar.

Rent rött exempelvis, skrivs `#FF0000`, rent grönt är `#00FF00` och rent blått är `#0000FF`.

Man kan även använda följande kortform för hexformatet:

`#RGB`

Om du väljer en färg där varje par består av samma två bokstäver eller siffror, t.ex. `#FF3322` (en starkt röd färg), kan du förkorta formatet till `#F32`.

Procentvärden för RGB (R%, G%, B%). Den här notationen använder ett procentvärde för varje färg på följande sätt:

`R%, G%, B%`

Giltiga värden är 0% till 100%. Även om detta bara ger futtiga 1 miljon möjliga färgkombinationer (100 x 100 x 100) är det mer än tillräckligt för de flesta av oss. Det är också mycket enklare att gissa sig till vilken färg man vill ha i RGB jämfört med hexnotationen.

Skriver man 100%, 0%, 0% innebär det maximalt rött, 0%, 100%, 0% är maximalt grönt och 46%, 76%, 80% ligger nära den dunkla grönbå färg som jag nyss demonstrerade i hex.

Färgnamn (red). Som du sett i alla de föregående färgexemplen i diskussionen om selektorer så kan man också specificera en färg med hjälp av färgnamnet, eller med ett nyckelord för att använda den officiella termen. Men det finns begränsningar; det finns ingen W3C-specifikation som säger exakt hur en webbläsare ska återge en färgspecifikation som olive eller lime. I princip tilldelar varje webbläsartillverkare sina egna (förmodligen hex) värden till varje färgnyckelord. W3C-specifikationen innehåller också bara 16 färger och därför kan man vara säker på att hitta dessa 16 färger i alla webbläsare. Här är de i alfabetisk ordning:

aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, yellow

De flesta moderna webbläsare erbjuder många fler färger (vanligtvis 140) men om du vill använda färgnamnen för att specificera färger kan du bara helt och fullt lita på dessa sexton.

Jag använder vanligtvis hexnotation eftersom jag programmerar och det är så man gör i kodningens snåriga värld. För att bespara dig mödan med att själv behöva blanda till färger kan du besöka adressen www.webcolors.freeserve.co.uk/, som erbjuder färgpaletter – den webbsäkra (se rutan ”Du behöver inte begränsa dig till de webbsäkra färgerna”) och andra – plus en stor översikt om färg.

Du behöver inte begränsa dig till de webbsäkra färgerna

Om du använder Macromedia Dreamweaver eller något annat verktyg för webbutveckling är du van att välja färger från en webbsäker palett. Den innehåller en uppsättning med 216 färger (som endast en ingenjör kunnat konstruera) och består huvudsakligen av klara och mättade färger, vilket begränsar urvalet av mörka och bleka färger. Dessa färger består av tvillinghexpar som exempelvis #3399CC eller #FF99CC och använder endast värdena 0, 3, 6, 9, C och F, så alla färger som du konstruerar och som uppfyller dessa villkor är webbsäkra. Dessa färger är en stor underuppsättning av de 256 färger (40 är reserverade för systemet) som en bildskärm som använder ett 8 bitars VGA-kort kan visa, så i årtal fick vi höra att vi inte skulle använda några andra. Men idag är det mindre än 1 procent (källa: www.thecounter.com) av världens alla surfare som fortfarande använder 8 bitars färg, så du kan lugnt använda vilka färger som helst som du kan skapa med hjälp av metoderna som listas i det här kapitlet.