Inhoudsopgave

Deel I

XML

Hoofdstuk 1

XML

1.1 Wat is XML?

XML staat voor eXtensible Markup Language en is ontwikkeld door het World WideWeb Consortium [http://www.w3c.org]. Deze standaard biedt de mogelijkheid om tekstdocumenten te structureren met behulp van labels (*tags*). Deze documenten noemen we XML-documenten. De namen van de labels gebruikt in XML-documenten zijn niet gedefinieerd door de XML-standaard. De XML-standaard legt dus enkel de algemene structuur van een XML-document vast.

XML wordt gebruikt om nieuwe markuptalen te definiëren en noemen we daarom een metataal. Documenten die opgemaakt worden met behulp van deze talen voldoen aan de algemene XML-structuur, maar de nieuwe taal legt de namen van de labels vast en beperkt hun inhoud. De labels geven een betekenis aan de gegevens in een XML-document.

De XML-syntax lijkt een beetje op HTML. Toch zijn er grote verschillen tussen XML en HTML. De HTML-syntax definieert een aantal labels om tekstdocumenten op te maken voor het web. Een XML-markuptaal daarentegen definieert labels om gegevens te structureren en een betekenis te geven. Hoe deze gegevens gepresenteerd moeten worden wordt echter niet bepaald. Dit wordt geïllustreerd in het volgend voorbeeld.

```
<p>Een paragraaf met een woord in <b>vetjes</b>.</p>
```

```
<boek>
    <titel>Java en XML</titel>
    <auteur>
        <voornaam>Jan</voornaam>
        <naam>Janssens</naam>
    </auteur>
</boek>
```

In het eerste (HTML-) voorbeeld bepalen de tags <b> en <p> de opmaak van het stukje tekst. De labels <boek>, <titel>, <auteur>, <voornaam> en <naam> in het tweede voorbeeld maken deel

uit van een XML-taal om boekgegevens (bv. Java en XML) te beschrijven en te structureren. Hoe deze gegevens voorgesteld moeten worden (bv. in een browser) wordt niet bepaald.

1.2 Basisconcepten

Een XML-document kan uit drie stukken bestaan. Dit wordt geïllustreerd in het volgend voorbeeld. Enkel het middelste stuk is verplicht. De andere delen zijn optioneel.

Figuur 1.1: Structuur XML-document

Het eerste deel noemen we de hoofding en het kan de volgende items bevatten.

- De XML-declaratie (zie §1.2.6)
- Verwerkingsinstructies (zie §1.2.7)
- Een verwijzing naar een DTD (zie §2)
- Commentaar (zie §1.2.8)
- Witte ruimte (*white space*, bv. lege lijnen, tabs, ...)

Het tweede deel, ook wel *body* of *corpus* genoemd, bevat de eigenlijke gegevens en bestaat uit het basiselement van het XML-document. Dit element kan andere elementen en tekst bevatten (zie §1.2.1).

Het laatste deel of de epiloog kan uit commentaar, witte ruimte en verwerkingsinstructies bestaan.

1.2.1 Elementen

Elementen zijn de bouwstenen van een XML-document. Een XML-document bevat één basiselement dat alle andere elementen bevat. In het voorgaande voorbeeld is het element *boek* het basiselement.

Een element bevat een deel van de gegevens van het XML-document. De inhoud van een element wordt begrensd door een opentag (bv. *<boek>*) en een sluittag (bv. *</boek>*). De tags bevatten de naam van het element. Deze naam verschaft extra informatie over de inhoud van het element. Bijvoorbeeld de naam *boek* geeft aan dat het element boekgegevens bevat.

Een element kan uitsluitend tekst bevatten

```
<naam>Janssens</naam>
```

of kan als container dienen voor andere elementen.

```
<auteur>
    <voornaam>Jan</voornaam>
    <naam>Janssens</naam>
</auteur>
```

In het geval van een leeg element (bv. *<leeg></leeg>*) kunnen de open- en sluittag samengenomen worden.

```
<leeg />
```

In XML-documenten die gegevens bevatten zullen elementen ofwel tekst ofwel andere elementen en eventueel wat witte ruimte bevatten. In meer beschrijvende XML-documenten zoals bv. rapporten, artikels, webpagina's, ... kan de inhoud van een element zowel tekst als andere elementen zijn. Dit wordt geïllustreerd in het volgend voorbeeld.

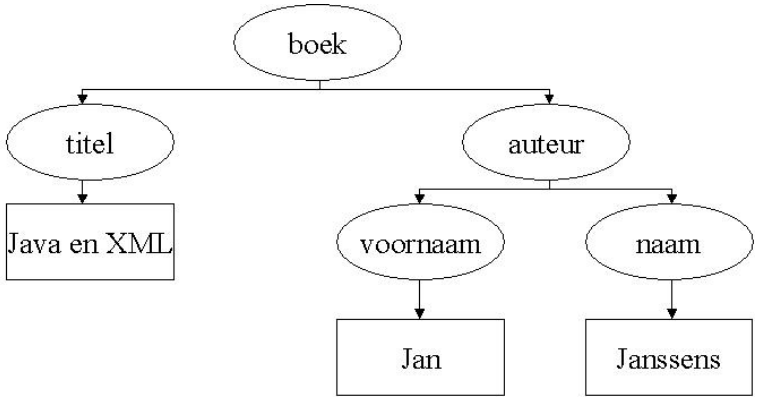
```
<para>XML staat voor eXtensible Markup Language en is
ontwikkeld door het <ulink url="http://www.w3c.org">World
WideWeb Consortium</ulink>. Deze standaard biedt de
mogelijkheid om tekstdocumenten te structureren met behulp
van labels (<foreignphrase lang="en">tags</foreignphrase>).
</para>
```

Volgens de XML-standaard kan een element de volgende zaken bevatten:

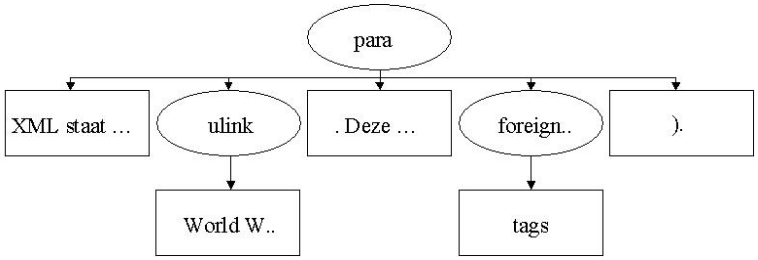
- tekst
- andere elementen
- commentaar (zie §1.2.8)
- verwerkingsinstructies (zie §1.2.7)
- CDATA-stukken (zie §1.2.5)
- entiteiten (zie §1.2.4)

De namen van elementen zijn hoofdlettergevoelig. <*boek*> is dus niet hetzelfde als <*BOEK*> of als <*Boek*>. De namen van element moeten voldoen aan een aantal regels die worden besproken in §1.2.3.

Aangezien een XML-document maar één basiselement heeft en omdat elementen andere elementen kunnen bevatten kan je een XML-document ook beschouwen als een boomstructuur. De boomstructuren van de voorgaande voorbeelden zien er als volgt uit. Elementen worden voorgesteld door ovale structuren, tekst door rechthoeken.



Figuur 1.2: Boomstructuur voor eerste voorbeeld in §1.1



Figuur 1.3: Boomstructuur voor tweede voorbeelde in §1.1

1.2.2 Attributen

Elementen kunnen attributen hebben. Een attribuut is een naam-waardepaar dat toegevoegd wordt aan de opentag van het element. Attributen worden gebruikt om extra informatie over een element toe te voegen.

```
<bestand gebruik="optioneel">
    computer.gif
</bestand>
```

De naam en waarde van een attribuut worden gescheiden door een gelijkheidsteken. Attributen moeten tussen enkele of dubbele aanhalingstekens staan. De naam van een attribuut is uniek binnen één element. Dit betekent dat een element geen twee attributen kan hebben met dezelfde naam. Als een element meerdere attributen heeft dan is de volgorde onbelangrijk. Vanuit XML-perspectief zijn de volgende elementen dus dezelfde.

```
<bestand gebruik="optioneel" type="figuur">
    computer.gif
</bestand>

<bestand type='figuur' gebruik='optioneel'>
    computer.gif
</bestand>
```

De vraag rijst wanneer gebruik je attributen en wanneer kindelementen? Voor deze vraag is er geen eenduidig antwoord en bestaan er geen officiële regels. Er is een algemene consensus die zegt dat elementen gebruikt worden om data te bevatten. Attributen geven dan extra informatie over de data van het element. Een attribuut stelt dus metadata van een element voor.

Een ander argument dat de keuze tussen attributen en element bepaalt, is de vaststelling dat een attribuut maar één waarde heeft. Data bestaande uit meerdere waarden of data die hiërarchisch gestructureerd is, kan je niet beschrijven met attributen.

1.2.3 XML-namen

Namen die voorkomen in een XML-document, bv. namen van elementen of attributen, bestaan uit letters, cijfers, een liggend streepje (_), een koppelteken (-) of een punt (.) Het eerste karakter moet een letter of een liggend streepje zijn.

Namen mogen dus geen spaties of andere 'witte ruimte' bevatten. Bovendien mag een naam niet met het woordje 'xml' (in kleine of grote letters of een combinatie van beide) beginnen. Een dubbele punt (:) kan wel voorkomen, maar heeft een speciale betekenis (zie §1.3).

1.2.4 Entiteiten

In een XML-document zijn er vijf tekens die niet kunnen voorkomen in de inhoud van een element of in de waarde van een attribuut omdat ze een speciale betekenis hebben. De tekens $<$ en $>$ bepalen het begin en einde van een tag. De waarde van een attribuut wordt begrensd door enkele of dubbele aanhalingstekens. Om deze tekens toch te kunnen gebruiken in een tekst worden er entiteiten voorzien. Een XML-parser zal bij het lezen van het XML-document de entiteiten vervangen door de gewenste tekens. Entiteiten starten met een '&' en eindigen op een ';'. De ampersant is dus het vijfde teken met een speciale functie.

Voor de bovenstaande vijf tekens zijn er voorgedefinieerde entiteiten.

- < (<)
- & (&)
- > (>)
- " (")
- ' (')

Het is ook mogelijk om zelf entiteiten te declareren. De declaratie maakt deel uit van de DTD (zie §2) en bepaalt de naam van de entiteit. Je gebruikt de entiteit door een '&' voor en een ';' na de naam te plaatsen. Als je in de DTD van het XML-document twee entiteiten *public* en *servlets* definieerde, dan kan je ze bijvoorbeeld als volgt gebruiken.

```
&public;
&servlets;
```

Zelfgedeclareerde entiteiten zijn handig als je een XML-document in verschillende stukken wilt opdelen. Ook als bepaalde stukken XML herhaaldelijk voorkomen kan een entiteit nuttig zijn.

1.2.5 CDATA-stukken

Als een XML-document stukken XML- of HTML-broncode bevat, dan moeten alle speciale tekens in die tekststukken vervangen worden door entiteiten. Dit is uiteraard onhoudbaar voor langere stukken broncode. Hier biedt een CDATA-sectie een oplossing.

Een CDATA-stuk wordt door een XML-parser niet beschouwd als 'markup', maar als ruwe tekst. De parser gaat het stuk dus niet interpreteren als XML. Je kan dus probleemloos <, >, &, " en ' gebruiken. Een CDATA-sectie begint met *<![CDATA[* en eindigt met *]]>*.

Een typisch voorbeeld van het gebruik van CDATA-stukken zijn cursussen of handleidingen die geschreven worden in een XML-taal en die tekstuele XML-voorbeelden bevatten. Gebruik CDATA-sectie

```
<para>De XML-syntax lijkt een beetje op HTML. Dit wordt
geïllustreerd in het volgend voorbeeld.
    <example>
        <programlisting><![CDATA[
<boek>
    <titel>Java en XML</titel>
    <auteur>
        <voornaam>Jan</voornaam>
        <naam>Janssens</naam>
    </auteur>
</boek>
        ]]></programlisting>
    </example>
</para>
```

De inhoud van het CDATA-stuk zal letterlijk weergegeven worden in de cursustekst. De andere XML-delen daarentegen zullen geformateerd worden tot opgemaakte tekst.

1.2.6 XML-declaratie

De eerste lijn van een XML-document is de XML-declaratie. Deze declaratie bepaalt volgens welke versie van XML het document opgesteld is. Verder kan ze de karakterset van het document bepalen en aangeven of er een externe DTD (zie §2) gelezen moet worden. De XML-declaratie ziet er als volgt uit.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

Een XML-document is een tekstdocument. De codering van dit tekstdocument wordt bepaald door het *encoding*-attribuut. Dit attribuut is optioneel, standaard is de waarde *UTF-8* (unicode - 8 bits). In het bovenstaande voorbeeld wordt de *Latin-1*-codering gebruikt.

Als het *standalone*-attribuut de waarde *no* heeft, kan het nodig zijn dat de parser een externe DTD leest om de waarde voor bepaalde delen van het XML-document te bepalen (bv. standaardwaarden voor attributen, zie §2). Dit attribuut is optioneel en heeft standaard de waarde *no*.

1.2.7 Verwerkingsinstructies

Met verwerkingsinstructies (*processing instructions*) wordt informatie doorgespeeld aan bepaalde applicaties of programma's die het XML-document lezen.

Een verwerkingsinstructie begint met *<?* en eindigt op *?>*. De syntax is als volgt

```
<?target pseudoattributen?>
```

De string *target* bepaalt de applicatie of het type applicatie waarvoor de verwerkingsinstructie informatie heeft. Deze informatie bestaat uit naam-waardeparen die dezelfde syntax hebben als attributen. De string *target* moet een geldige XML-naam zijn (zie §1.2.3).

De meeste gekende verwerkingsinstructie is *xml-stylesheet*. Deze opdracht koppelt XML-documenten aan *stylesheets*. Browsers weten zo hoe ze het XML-document aan de lezer moeten tonen.

```
<?xml-stylesheet href="boeken.css" type="text/css"?>
```

De verwerkingsinstructie in het bovenstaande voorbeeld is bedoeld voor browsers die het XML-document tonen. Als er geen *stylesheet* gedeclareerd is toont de browser het XML-document als een boomstructuur. Deze verwerkingsinstructie vertelt de browser dat er een *stylesheet* beschikbaar is in het bestand *boeken.css*. Het type van de *stylesheet* is CSS (§4.2).

1.2.8 Commentaar

De syntax van XML-commentaar is dezelfde als in HTML. Commentaar begint met <!– en eindigt op –>. Overal in een XML-document mag commentaar staan, behalve in een tag of in een ander commentaarstuk.

```
<!-- Hier staat commentaar -->
```

1.2.9 Goedgevormde XML-documenten

Elk XML-document moet goed gevormd (*well formed*) zijn. Dit betekent dat het moet voldoen aan de XML-specificatie opgesteld door het W3C. De afkorting W3C staat voor World Wide Web Consortium. Dit consortium stelt specificatie op voor allerlei webstandaarden. De XML-specificatie omvat o.a. de volgende regels.

- Elk element heeft een begin- en eindtag, eventueel wordt de verkorte notatie gebruikt.
- De namen van elementen zijn correct XML-namen.
- Elementen vormen een strikte boomstructuur met één basiselement.
- De waarden van attributen staan tussen enkele of dubbele aanhalingstekens.
- De attributen van één element hebben verschillende namen.
- Commentaar en verwerkingsinstructies staan niet in tags.
- De tekens < en & komen niet voor in de tekstinhoud van een element of attribuut. Gebruik de voorgedefinieerde entiteiten waar nodig.
- ...

Er bestaan verschillende manieren om na te gaan of een XML-document goed gevormd is. De eenvoudigste is het XML-document openen in een browser met XML-ondersteuning. Als het document goed gevormd is, dan wordt het getoond in de browser. Anders toont de browser een foutboodschap. Een alternatief is het gebruik van een XML-editor. Tot slot hebben de meeste programmeertalen XML-parsers zodat het mogelijk is om programmatorisch een XML-document te controleren.

1.3 Namenruimtes (*Namespaces*)

1.3.1 Waarom namenruimtes?

Soms bestaan XML-documenten uit elementen van verschillende XML-talen. Een XHTML-document bijvoorbeeld kan zowel SVG-tekeningen als MathML-vergelijkingen bevatten. XHTML is een herformulering van HTML naar de XML-standaarden. SVG staat voor Scalable Vector Graphics en is

een XML-taal om tweedimensionale figuren te beschrijven. MathML is een XML-taal die gebruikt wordt om wiskundige uitdrukkingen te beschrijven.

In dit geval kan het zijn dat er meerdere elementen in het document voorkomen met dezelfde naam, maar met een andere betekenis. Zo betekent het element *set* in SVG iets anders dan in MathML. In het eerste geval wordt het gebruikt om attributen een waarde te geven, in het tweede geval stelt het een verzameling voor.

Een ander voorbeeld is een XML-document dat een catalogus beschrijft. Een catalogus bestaat uit boeken, CD's, ... Voor boeken en CD's zijn er aparte XML-talen gedeclareerd en beide talen bevatten het element *titel*. Ook hier kan het nodig zijn om een onderscheid te maken tussen beide soorten *titel*-elementen.

Om bovenstaande conflicten te voorkomen wordt er gebruik gemaakt van namenruimtes (*namespaces*). Namenruimtes dienen om onderscheid te maken tussen elementen en attributen uit verschillende XML-talen met een andere betekenis, maar met dezelfde naam. Bovendien worden ze ook gebruikt om verwante elementen en attributen te groeperen zodat programma's ze makkelijk kunnen herkennen (zie bijvoorbeeld §4).

1.3.2 Syntax

Elke namenruimte wordt gekenmerkt door een URI (Uniform Resource Indentifier). Deze URI moet niet verwijzen naar een bestaande webpagina. Hij dient enkel om de namenruimte uniek te identificeren.

Als het XML-document gebruik maakt van namenruimtes dan is de naam van een element de combinatie van de URI van de namenruimte en de naam van het element in de namenruimte. Beide delen worden gescheiden door een ':'.

```
<http://www.zwiftbooks.com:book>
    <http://www.zwiftbooks.com:isbn>0-596-0058-8
    </http://www.zwiftbooks.com:isbn>
<http://www.zwiftbooks.com:book>
```

Er zijn nog twee andere methodes om namenruimtes te realiseren. De eerste koppelt een prefix met de URI van de namenruimte. De tweede methode zal voor een element en zijn kindelementen een standaard namespace bepalen.

Prefixen

In deze methode bestaat een XML-naam uit twee delen. Het eerste deel is de prefix en het tweede de locale naam. Beide delen worden van elkaar gescheiden door een *:*. De prefix kan je beschouwen als een afkorting voor de URI van de namenruimte. De volledige naam (prefix + : + locale naam) wordt *qualified name* genoemd.

In het volgend voorbeeld wordt geïllustreerd hoe de prefix met de URI van de namenruimte verbonden wordt.

```
<zbooks:book xmlns:zbooks="http://www.zwiftbooks.com">
    <zbooks:isbn>0-596-0058-8</zbooks:isbn>
</zbooks:book>
```

De koppeling wordt verwezenlijkt door een attribuut *xmlns:prefix* toe te voegen aan een element. Hierbij is *prefix* de afkorting die je wilt gebruiken voor de namenruimte. De afkorting is gekend binnen het element waar het gedeclareerd werd. In de begintag van een element kunnen meerdere afkortingen van namenruimtes gedefinieerd worden. Het element kan maar hoeft dus niet tot één van die namenruimtes te behoren. Het volgend voorbeeld illustreert dit.

```
<catalogus xmlns:zbooks="http://www.zwiftbooks.com"
  xmlns:mcds="http://www.music.com">
    <zbooks:book >
        ...
    </zbooks:book>
    <mcds:cd>
        ...
    </mcds:cd>
</catalogus>
```

De standaard namespace

Een andere manier om de namenruimte van een element te bepalen, is een standaard namenruimte te definiëren. Deze methode wordt bijvoorbeeld gebruikt als alle kinderen van een element tot de namenruimte van het element behoren.

```
<books xmlns="http://www.zwiftbooks.com">
    <book >
        <isbn>0-596-0058-8</isbn>
        <title>Inleiding tot XML</title>
        ...
    <book>
    ...
</books>
```

De standaard namespace van een element wordt bepaald door de waarde van het attribuut *xmlns*. Alle elementen binnen dit element waarvoor geen namenruimte wordt opgegeven (via prefix of standaard namenruimte) behoren tot de namenruimte van het element. In het bovenstaande voorbeeld behoren de elementen *books*, *book*, *isbn* en *title* tot namenruimte ‘http://www.zwiftbooks.com’.

Attributen en namenruimtes

Als een element tot een bepaalde namespace behoort betekent dit niet automatisch dat het attribuut ook tot die namespace behoort. Vaak behoren de attributen niet tot een namespace. Indien een attribuut

tot een namespace behoort dan moet zijn naam een prefix hebben.

Hoofdstuk 2

Document Type Definition

2.1 Wat is een DTD?

Een DTD of Document Type Definition bepaalt een XML-taal. Het legt de structuur van een XML-document dat in die taal is opgesteld vast.

DTD's bepalen welke elementen en entiteiten gebruikt mogen worden, waar elementen voorkomen in het document en in welke volgorde. Verder bepalen ze ook wat de inhoud is van een element en welke attributen het kan hebben.

Een XML-document wordt geldig (*valid*) genoemd als de structuur overeenstemt met structuur die bepaald werd in de opgegeven DTD.

Om een XML-document te valideren kan je gebruik maken van een XML-editor of van een publieke validator op het internet. Bijvoorbeeld één van de volgende.

- http://www.stg.brown.edu/service/xmlvalid/
- http://www.cogsci.ed.ac.uk/%7Erichard/xml-check.html

Verschillende programmeertalen voorzien ook parsers waarmee je XML-documenten kan valideren.

2.2 Syntax

De DTD die het document beschrijft kan opgenomen worden in het XML-bestand zelf, kan een apart bestand vormen (met de extensie *.dtd*) of kan publiek beschikbaar zijn via het web.

2.2.1 Document Type Declaration

Een geldig XML-document moet een verwijzing bevatten naar de DTD waaraan het voldoet. Dit noemen we de *document type declaration*. De syntax van deze declaratie hangt af van waar deze DTD zich bevindt: in het XML-document, in een bestand of op een publieke URI.

```
<?xml version="1.0"?>
<!DOCTYPE basiselement [
    declaraties
]>
<basiselement>
    ...
</basiselement>

<?xml version="1.0"?>
<!DOCTYPE basiselement SYSTEM "bestand">
<basiselement>
    ...
</basiselement>

<?xml version="1.0"?>
<!DOCTYPE basiselement PUBLIC "naam" "url">
<basiselement>
    ...
</basiselement>
```

De verwijzing naar de DTD maakt deel uit van de hoofding van het XML-document (zie ook §1.2). Ze bevat de naam van het basiselement (*root element*) van het XML-document en de DTD of een verwijzing naar zijn locatie. Het is ook mogelijk een externe DTD uit te breiden met een aantal interne declaratie zoals in het volgend voorbeeld.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book SYSTEM "docbook\docbookx.dtd" [
    <!ENTITY xml SYSTEM "xml.xml">
    <!ENTITY dtd SYSTEM "dtd.xml">
]>
<book lang="nl">
    <title>XML</title>
    &xml;
    &dtd;
</book>
```

Docbook is een XML-taal die ontwikkeld werd om informaticaboeken in te schrijven. In dit voorbeeld bevindt de DTD zich in een relatieve subdirectory *docbook*. Deze DTD beschrijft o.a. het basiselement *book*. Deze externe DTD wordt nog uitgebreid met twee entiteiten die het mogelijk maken om het XML-document in stukken op te splitsen.

DTD's in externe bestanden worden het meest gebruikt. Interne DTD's zijn soms handig in ontwerpfase of om een externe DTD uit te breiden. Van publieke DTD's wordt vaak een lokale kopie voorzien omdat dit sneller is bij het valideren en geen netwerkverbinding vraagt.

2.2.2 Beschrijving van elementen

Elk element in een geldig XML-document moet gedeclareerd worden in zijn bijhorende DTD. De syntax van een elementdeclaratie is de volgende.

```
<!ELEMENT naamelement inhoud>
```

In de declaratie wordt de naam van het element bepaald en zijn inhoud. De naam is een geldige XML-naam (zie §1.2.3). De inhoud van een element kan zowel tekst, andere elementen of een combinatie van beide zijn.

Een element met enkel tekst

In dit geval mag het element enkel tekst bevatten. De inhoud is dan het sleutelwoord *#PCDATA* tussen ronde haakjes. *PCDATA* is de afkorting van *parsable character data*. Het element *isbn*, dat een isbnnummer van een boek voorstelt, kan als volgt gedeclareerd worden.

```
<!ELEMENT isbn (#PCDATA)>
```

Een element met nul, één of meerdere kindelementen

Een element kan ook nul, één of meerdere kindelementen hebben. De volgorde van deze kindelementen kan opgelegd worden. De volgende voorbeelden illustreren dit. Hierbij wordt gebruik gemaakt van de achtervoegsels *?*, * en +. Deze achtervoegsels geven aan hoeveel keer het kindelement kan voorkomen. Is er geen achtervoegsel dan komt het element juist één keer voor. De betekenis van de verschillende achtervoegsels is

? : nul of één keer

***** : nul of meerdere keren

+ : één of meerdere keren

```
<!ELEMENT books (book*)>
<!ELEMENT author (name, initial?, firstname+)>
```

In dit voorbeeld kan een *books*-element nul of meerdere *book*-elementen bevatten. Een *author*-element heeft één *name*-element, eventueel gevolgd door een *initial*-element en vervolgens minstens één *firstname*-element.

De volgorde van de *name*-, *initial*- en *firstname*-elementen wordt bepaald door deze declaratie. Het volgend voorbeeld toont hoe een keuze waarbij de volgorde van de elementen onbelangrijk is gedefinieerd kan worden. Hierbij wordt gebruik gemaakt van de |-operator. Een *catalogus*-element bestaat uit nul of meerdere kindelementen. Deze elementen kunnen *boek*-, *cd*-, *tijdschrift*-, *dvd*- of *video*-elementen zijn. De volgorde waarin ze voorkomen speelt geen rol.

```
<!ELEMENT catalogus (boek | cd | tijdschrift | dvd | video)*>
```

Een leeg element

Een leeg element heeft geen inhoud. In dat geval kunnen de open- en sluittag samengenomen worden (zie §1.2.1). De declaratie van een leeg element heeft de volgende structuur.

```
<!ELEMENT query EMPTY>
```

Een element met tekst en kindelementen.

In beschrijvende XML-documenten (bv. boeken) kan de inhoud van een element een combinatie van tekst en kindelementen zijn (zie ook §1.1). De declaratie van het element *para* in het voorbeeld §1.1 zou dan als volgt kunnen zijn.

```
<!ELEMENT para (#PCDATA | ulink | foreignphrase)*>
```

De bovenstaande declaratie bepaalt dat het element *para* bestaat uit een willekeurig aantal elementen *ulink* en *foreignphrase* in gelijk welke volgorde, eventueel afgewisseld met tekststukken.

2.2.3 Beschrijving van attributen

Ook de attributen van de elementen in een geldig XML-document moeten gedeclareerd worden in de bijhorende DTD. Voor elk element met attributen moet een declaratie van het type *ATTLIST* bestaan. De syntax is de volgende.

```
<!ATTLIST naamelement
     attr1 type1 standaard1
     attr2 type2 standaard2
     attr3 type3 standaard3
     ...
>
```

Hierbij zijn *attr1*, *attr2*, *attr3*, ... de namen van de attributen van een element *naamelement*. Voor elk attribuut wordt een type opgegeven en een standaarddeclaratie. De standaarddeclaratie bepaalt onder andere of een attribuut vereist is.

Merk op dat de bovenstaande declaratie opgesplitst mag worden in verschillende declaraties van de volgende structuur.

```
<!ATTLIST naamelement attr type standaard>
```

Hier volgen een aantal voorbeelden van attribuutdeclaraties.

```
<!ATTLIST query
    isbn NMTOKEN #REQUIRED
    zipcode (75230 | 75233 | 75235) #REQUIRED>
<!ATTLIST reply
    time CDATA #REQUIRED
    price NMTOKEN #REQUIRED>
```

Het element *query* heeft twee attributen *isbn* en *zipcode*. Beide attributen zijn vereist. Het attribuut *zipcode* kan slechts drie waarden aannemen: 75230, 75233 of 75235. Ook het element *reply* heeft twee vereiste attributen. De types *NMTOKEN* en *CDATA* worden besproken in §2.2.3.

Attribuuttypes

In welgevormde XML-documenten kan de waarde van een attribuut elk stukje tekst zijn dat geen speciaal teken bevat zoals bijvoorbeeld ¡ (zie §1.2.4). Met behulp van DTD's kan je de inhoud van een attribuut beperken. Een aantal van de beschikbare attribuuttypes zijn

- CDATA
- NMTOKEN
- Een opsomming
- ID
- IDREF

CDATA Een attribuut van het type CDATA bestaat uit willekeurige tekst. Deze tekst kan bestaan uit alle mogelijke tekens die toegelaten zijn in een welgevormd XML-document. Enkel speciale tekens zoals $<$ en aanhalingstekens zijn niet toegelaten. In het onderstaand voorbeeld is het attribuut *time* van het type CDATA.

```
<!ATTLIST reply time CDATA #REQUIRED>
```

Een mogelijk geldig *reply*-element is dan (op voorwaarde dat ook het *price*-attribuut gedeclareerd is)

```
<reply time="2 hours" price="45.00" />
```

NMTOKEN Attributen van het type NMTOKEN bestaan uit lettertekens, cijfers en de leestekens _, -, . en :. Dit soort attributen mag geen "witte ruimte"bevatten. Het attribuut *price* in het bovenstaande voorbeeld kan als NMTOKEN gedeclareerd worden.

```
<!ATTLIST reply price NMTOKEN #REQUIRED>
```

Een voorbeeld van een geldig *reply*-element.

```
<reply time="2 hours" price="45.00" />
```

Een opsomming Dit type is geen XML-sleutelwoord, maar is een lijst van mogelijke waarden die het attribuut kan aannemen. De verschillende waarden in de lijst worden gescheiden door een rechte streep. Elke waarde in de lijst moet van het type NMTOKEN zijn. In het onderstaand voorbeeld kan het attribuut *zipcode* slechts drie waarden aannemen.

```
<!ATTLIST query
    isbn NMTOKEN #REQUIRED
    zipcode (75230 | 75233 | 75235) #REQUIRED>
```

Een voorbeeld van een geldig *query*-element.

```
<query isbn="0-596-00058-8" zipcode="75233" />
```

ID Een attribuut van het type ID is een XML-naam (zie §1.2.3) die uniek is in het XML-document. Dit betekent dat geen enkel ander attribuut van het type ID dezelfde waarde mag hebben. Elk element kan maximaal één attribuut van het type ID hebben. Let op aangezien XML-namen niet met een cijfer mogen beginnen, kan een ID geen getal zijn.

In het volgend voorbeeld wordt bepaald dat elk *kunstenaar*-element een uniek attribuut *id* heeft.

```
<!ATTLIST kunstenaar id ID #REQUIRED>
```

Een voorbeeld van een *kunstenaar*-element is dan

```
<kunstenaar id="KDC">
    ...
</kunstenaar>
```

In hetzelfde XML-document mag er geen ander *kunstenaar*-element voorkomen met hetzelfde *id*-attribuut.

IDREF Attributen van het type IDREF verwijzen naar een attribuut van het type ID van een ander element in het XML-document. Deze attributen zijn dus ook XML-namen en ze worden gebruikt om relaties te leggen tussen elementen.

Een kunstwerk kan gelinkt worden met een kunstsoort en een kunstenaar met behulp van ID- en IDREF-attributen.

```
<!ATTLIST kunstwerk id ID #REQUIRED categorie IDREF #REQUIRED
    kunstenaar IDREF #REQUIRED>
```

Een voorbeeld van een *kunstwerk*-element is dan.

```
<kunstwerk id="KW1" categorie="SCHILDERKUNST" kunstenaar="KDC">
...
</kunstwerk>
```

Attribuutstandaarden

Naast het type van een attribuut wordt in de declaratie ook een standaard opgenomen. Er zijn vier mogelijkheiden voor deze standaard.

- #IMPLIED
- #REQUIRED
- #FIXED
- Een stukje tekst

#IMPLIED In dit geval is het attribuut optioneel. Een element kan of kan geen waarde hebben voor dit attribuut. Er is geen standaardwaarde voorzien.

#REQUIRED Als de standaard #REQUIRED is, dan moet elke instantie van dit element een waarde hebben voor het attribuut. Ook hier is geen standaardwaarde voorzien.

#FIXED De attribuutwaarde is een constante. Deze waarde hoeft niet expliciet vermeld te zijn als attribuut van een element. Maar als het attribuut een waarde heeft dan moet het de opgegeven waarde zijn.

De volgende declaratie bepaalt dat elk element *biography* een attribuut *xmlns:xlink* heeft met de waarde *http://www.w3.org/1999/xlink* ook als dit niet expliciet vermeld is.

```
<!ATTLIST biography xmlns:xlink CDATA
    #FIXED "http://www.w3.org/1999/xlink">
```

Een stukje tekst In dit geval is het stukje tekst de standaardwaarde voor het attribuut. De waarde van het attribuut kan opgeven worden. Is dit niet zo dan heeft het attribuut de standaardwaarde.

We illustreren dit met een voorbeeld. Een element *beschrijving* beschrijft een leeftijdscategorie. We veronderstellen dat als de beschrijving geen *taal*-attribuut heeft, dat de beschrijving in het Nederlands is. In het andere geval moet de taal expliciet opgegeven worden. De attribuut declaratie is dan als volgt

```
<!ATTLIST beschrijving taal NMTOKEN "NL">
```

Mogelijke geldige *beschrijving*-elementen zijn dan

```
<beschrijving>Kleuters (drie- tot zesjarigen)</beschrijving>
<beschrijving taal="NL">Peuters (één- tot driejarigen)
</beschrijving>
<beschrijving taal="EN">Toddlers</beschrijving>
```

2.2.4 Beschrijving van entiteiten

In §1.2.4 bespraken we reeds het gebruik van entiteiten. De vijf voorgedefinieerde entiteiten gebruik je om <, >, &, ' en " te kunnen gebruiken in de inhoud van een element of in de waarde van een attribuut.

Entiteiten worden ook gebruikt om XML-documenten op te delen of als afkorting van stukken XML die frequent voorkomen in het document.

De declaratie van een entiteit maakt deel uit van de DTD. De syntax is de volgende.

```
<!ENTITY naamEntiteit "waardeEntiteit">
```

De waarde van de entiteit, in de syntax voorgesteld door *waardeEntiteit*, moet een welgevormd stuk XML zijn. Overal waar je *waardeEntiteit* zou tikken, kan je nu *&naamEntiteit;* schrijven. Stukken XML die herhaaldelijk voorkomen, moeten zo slechts éénmaal getikt worden. In het volgend voorbeeld wordt een entiteit *public* gedeclareerd. Deze entiteit is een afkorting van *<literal role='keyword'>public</literal>*. In de interne of externe DTD wordt de volgende declaratie opgenomen.

```
<!ENTITY public "<literal role='keyword'>public</literal>">
```

Telkens je *<literal role='keyword'>public</literal>* moet schrijven, kan je dit korter schrijven gebruik makend van de entiteit. Merk op dat de waarde van de entiteit geen dubbele aanhalingstekens kan bevatten.

```
&public;
```

Het is ook mogelijk om de inhoud van de entiteit in een apart bestand op te nemen. Bijvoorbeeld om het XML-bestand op te delen in verschillende stukken. In dat geval is de syntax van de entiteitdeclaratie de volgende.

```
<!ENTITY naamEntiteit SYSTEM "PAD">
```

Hierbij is *PAD* een URI die de plaats van het XML-stuk, dat de waarde van de entiteit is, aangeeft. Dit kan een URL zijn, maar ook een absolute of relatieve padnaam van een bestand. In het volgend voorbeeld bevat het bestand *servlets.docbook* een hoofdstuk van een cursus geschreven in de XML-taal docbook. Dit bestand bevindt zich in de subdirectory *webtech\docbook*.

```
<!ENTITY servlets SYSTEM "webtech\docbook\servlets.docbook">
```

Op de plaats waar het hoofdstuk in de cursus moet komen wordt het volgende geschreven.

```
&servlets;
```

Hoofdstuk 3

XML Schema

3.1 Inleiding

Met behulp van een DTD kan de structuur van een XML-document vastgelegd worden. Voor sommige applicaties is het echter nodig om de structuur van de XML-documenten precieser te bepalen en om meer datatypes te voorzien dan mogelijk is met behulp van een DTD. Om aan deze noden te voldoen werd XML Schema ontwikkeld door het W3C. Net zoals een DTD beschrijft een XML Schema de structuur van XML-documenten. In tegenstelling tot de DTD-syntax is XML Schema een XML-taal. Een XML Schema-document is dus een XML-document dat moet voldoen aan een bepaalde syntax.

3.2 Koppeling XML-document en XML-schema

De link tussen een XML-document en het schema dat zijn structuur vastlegt kan op verschillende manieren gerealiseerd worden. In de eerste twee gevallen wordt in het basiselement expliciet verwezen naar de URL van het schemadocument. In de laatste twee gevallen kent enkel de parser van het XML-document het XML-schema.

- Als het element niet tot een namenruimte behoort, dan heeft het een attribuut *xsi:noNamespaceSchemaLocation* die de URL van het bijhorende schema bevat. Hierbij is *xsi* de afkorting voor de namenruimte bepaald door de URL *http://www.w3.org/2001/XMLSchema-instance*. In het voorbeeld wordt de structuur van het element *books* vastgelegd door het schema in het bestand *boeken.xsd*. Het element *books* behoort niet tot een namenruimte.

  ```
  <books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="boeken.xsd">
  ...
  </books>
  ```

- Als het element wel tot een namenruimte behoort, dan wordt naar het bijhorende schema verwezen met het attribuut *xsi:schemaLocation*. Dit attribuut bepaalt de URL van het bijhorende

schema. Als het element *books* tot de namenruimte *http://www.zwiftbooks.com* behoort dan wordt de link met het schema in het bestand *boeken.xsd* als volgt gelegd.

```
<books xmlns="http://www.zwiftbooks.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="boeken.xsd">
...
</books>
```

- De parser kan de opdracht krijgen om een gegeven document te valideren ten opzichte van een opgegeven schema.
- De parser probeert zelf op basis van de namenruimte van het element het schema te bepalen.

3.3 Structuur van een XML Schema

Een XML-schema is een XML-document dat behoort tot de namenruimte *http://www.w3.org/2001/XMLSchema*. Het XML-schema beschrijft de structuur van één of meerdere XML-documenten. Het bestand dat het schema bevat heeft meestal de extensie *xsd* of *xs*. Het basis-element van een schema is het element *schema*. Een schema heeft dus de volgende structuur.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
</xsd:schema>
```

In het bovenstaande voorbeeld wordt verondersteld dat de elementen die gedeclareerd worden in het schema tot de standaardnamenruimte behoren. Om de elementen gedeclareerd binnen een schema te associëren met een namenruimte moet deze expliciet worden opgegeven met het attribuut *targetNamespace*. In het volgend voorbeeld behoren de globaal gedeclareerde elementen tot de namespace *www.zwiftbooks.com*. Elementdeclaraties zijn globaal als ze een kindelement zijn van het *schema*-element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.zwiftbooks.com">
...
</xsd:schema>
```

In het *schema*-element worden onder andere elementen en types gedeclareerd (zie §3.4 en §3.6). Een aantal elementen, die kindelementen van het *schema*-element kunnen zijn, zijn

- *element*. Dit wordt gebruikt om elementen te declareren.
- *simpleType*. Hiermee worden datatypes gedefinieerd die de tekstinhoud van een element of de waarde van een attribuut beperken.
- *complexType*. Met dit element kan men gegevenstypes declareren die elementen beschrijven met kindelementen of met attributen of met zowel kindelementen als attributen.

3.4 Elementdeclaraties

Een elementdeclaratie bepaalt de naam van het element en het type van het element. Types kunnen bestaande types zijn of zelfgedeclareerde types (zie §3.6). Een type kan eenvoudig (*simple*) zijn of complex. Eenvoudige types beschrijven de tekstinhoud van een element of de waarde van een attribuut. Complexe types worden gebruikt voor elementen met kindelementen, met attributen of met beiden.

Het ISBN-nummer van een boek kan bijvoorbeeld beschreven worden door een element *isbn* dat als volgt gedeclareerd is.

```
<xsd:element name="isbn" type="xsd:string"/>
```

Het type *string* maakt deel uit van de namenruimte *http://www.w3.org/2001/XMLSchema*. Deze namenruimte bevat alle types die beschikbaar zijn in DTD's en ook de meeste datatypes die voorkomen in moderne programmeertalen zoals strings, integers, datums, tijd, ... Een aantal van de ingebouwde types zijn *string*, *integer*, *double*, *boolean*, *time*, *ID*, *NMTOKEN*, *duration*, *time* ... Alle ingebouwde types zijn eenvoudige types.

3.5 Attribuutdeclaraties

Een attribuutdeclaratie bestaat uit de naam van het attribuut en het type. Attribuuttypes zijn steeds eenvoudig. Deze types kunnen deel uitmaken van de XML Schema-namenruimte of zelfgedeclareerd zijn. Attributen kunnen onmiddellijke kinderen zijn van het *schema*-element of kunnen voorkomen binnen element- of typedeclaraties. In het volgend voorbeeld worden twee attributen gedeclareerd. Het attribuut *isbn* is van het bestaande type *NMTOKEN* en het attribuut *zipcode* is van een zelfgedeclareerd type *zips*.

```
<xsd:attribute name="isbn" type="xsd:NMTOKEN"/>
<xsd:attribute name="zipcode" type="zips"/>
```

3.6 Typedeclaraties

Er zijn twee soorten types: eenvoudige types en complexe types. Eenvoudige types beschrijven tekst of waarden. Ze kunnen gebruikt worden als type voor attributen en elementen met enkel tekstinhoud. Complexe types beschrijven elementen met kinderen, met attributen of met beiden.

3.6.1 Eenvoudige types

De eenvoudige types kunnen in een aantal categorieën ingedeeld worden.

- Bestaande types uit de namenruimte XML Schema
- Beperkingen van bestaande types. Dit zijn meestal beperkingen van strings of getallen. Voor strings kunnen deze beperkingen bijvoorbeeld het volgende zijn: een patroon (*regular expression*) vastleggen, de lengte beperken, keuze uit een aantal waarden, ... Bij getallen kan men bijvoorbeeld een interval bepalen.
- Een unie van eenvoudige types
- Een lijst, vergelijkbaar met een *array* in een programmeertaal.

Een eenvoudig type wordt beschreven door het element *simpleType*. Dit element kan voorkomen als kind van de elementen *schema*, *element* en *attribute*. We bespreken hier enkel eenvoudige types die beperkingen zijn van een ander eenvoudig type.

In dit geval is het kindelement van *simpleType* het element *restriction*. Het element *simpleType* heeft een attribuut *name* die de naam van het type vastlegt. Het attribuut *base* van *restriction* bepaalt het type waarvan vertrokken wordt.

```
<xs:simpleType name="zips">
    <xs:restriction base="xs:string">
        <xs:enumeration value="75230"/>
        <xs:enumeration value="75233"/>
        <xs:enumeration value="75235"/>
    </xs:restriction>
</xs:simpleType>

<xsd:simpleType name="diplomatype">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="HOBU|UNIV|MO|HOKT"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="initiaaltype">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-z]|[A-Z]\."/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="postcodetype">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[1-9][0-9]{3}"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="abs180">
    <xsd:restriction base="xsd:decimal">
        <xsd:fractionDigits value="2"/>
        <xsd:minInclusive value="-180"/>
        <xsd:maxInclusive value="180"/>
    </xsd:restriction>
```

```
</xsd:simpleType>
```

Afhankelijk van het basistype kunnen een aantal kindelementen van *restriction* gebruikt worden om het nieuwe type te definiëren. De betekenis van de meeste van deze elementen spreekt voor zich. Ze hebben allen een attribuut *value* dat de waarde van de beperking bepaalt.

- *length*, *minLength* en *maxLength*
- *pattern*
- *enumeration*
- *maxInclusive* en *maxExclusive*
- *minInclusive* en *minExclusive*
- *totalDigits*
- *fractionDigits*

In het geval van *pattern* stelt het attribuut *value* een reguliere uitdrukking (*regular expression*) voor die het type declareert.

3.6.2 Complexe types

Elementen van een complex type kunnen kindelementen en attributen hebben. Het type van een attribuut is nooit complex. Een complex type kan anoniem zijn. Dit betekent dat het type geen naam heeft en dus slechts één keer voorkomt. Het volgend voorbeeld toont een anoniem type en een type met naam.

```
<xs:element name="query" type="queryType"/>
<xs:complexType name="queryType">
    <xs:attribute name="isbn" type="xs:NMTOKEN"/>
    <xs:attribute name="zipcode" type="zips"/>
</xs:complexType>
```

of

```
<xs:element name="query">
    <xs:complexType>
        <xs:attribute name="isbn" type="xs:NMTOKEN"/>
        <xs:attribute name="zipcode" type="zips"/>
    </xs:complexType>
</xs:element>
```

In beide gevallen is het element *query* leeg en heeft twee attributen *isbn* en *zipcode*. In het eerste geval zou dit type nog kunnen gebruikt worden om een ander element te declareren.

Een complex type beschrijft elementen met kinderen en/of attributen. De declaratie van de attributen komt na de declaratie van de kindelementen. Het element *complexType* declareert het complexe type.

Als de volgorde van de kindelementen belangrijk is, dan gebruikt men het element *sequence* bij de beschrijving van de kindelementen, anders is dat het element *choice*.

```
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="initial" type="xs:NMTOKEN"
          minOccurs="0" maxOccurs="1"/>
      <xs:element name="firstname" type="xs:token"
          minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

De attributen *minOccurs* en *maxOccurs* bepalen hoeveel keer het element minimaal en maximaal kan voorkomen. Standaard hebben deze attributen de waarde 1. Als een element een onbeperkt aantal keer kan voorkomen dan is de waarde van *maxOccursunbounded*. Zijn de elementen *name*, *initial* en *firstname* globaal gedeclareerd, dan wordt ze niet meer lokaal gedefinieerd. Het attribuut *ref* verwijst naar de globale declaraties in het document.

```
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" />
      <xs:element ref="initial" minOccurs="0"
          maxOccurs="1"/>
      <xs:element ref="firstname" minOccurs="1"
          maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

In het volgend voorbeeld wordt een element *meting* beschreven. Meting stelt de waarneming van een hemellichaam (ster, planeet, ...) voor. De lokatie en het tijdstip van de meting worden voorgesteld door de elementen *plaats* en *tijdstip*. De waarde van de meting is de inhoud van het element *hemelcoordinaat*. Verder heeft dit element nog drie attributen. Deze stellen het toestel waarmee de meting gebeurde, de persoon die de meting uitvoerde en het waargenomen hemellichaam voor. Alle drie de attributen zijn van het type *IDREF*. Ze verwijzen naar de unieke identificatie van elementen *toestel*, *persoon* en *hemellichaam*.

```
<xsd:element name="meting">
    <xsd:complexType>
        <xsd:sequence>
```

```
            <xsd:element ref="plaats"/>
            <xsd:element ref="tijdstip"/>
            <xsd:element ref="hemelcoordinaat"/>
        </xsd:sequence>
        <xsd:attribute name="toestel" type="xsd:IDREF"/>
        <xsd:attribute name="persoon" type="xsd:IDREF"/>
        <xsd:attribute name="hemellichaam"
            type="xsd:IDREF"/>
    </xsd:complexType>
</xsd:element>
```

Een voorbeeld van een *meting*-element.

```
<meting toestel="TELESC1" persoon="P1" hemellichaam="TTAN">
    <plaats>
        ...
    </plaats>
    <tijdstip>
        ...
    </tijdstip>
    <hemelcoordinaat>
        ...
    </hemelcoordinaat>
</meting>
```

Een complex type als uitbreiding of beperking van een ander type

Een complex type kan ook een uitbreiding of beperking zijn van een ander type. We onderscheiden hierin twee types: types die elementen met kindelementen beschrijven en types die elementen met enkel tekstinhoud beschrijven. In het laatste geval gaat het over elementen zonder kindelementen, maar met attributen. Zonder attributen zouden het gewoon elementen van een eenvoudig type zijn.

Elementen met tekstinhoud en attributenElementen zonder kinderen, maar met attributen zijn een uitbreiding van een eenvoudig type. Aan het type worden attributen toegevoegd. In het volgend voorbeeld wordt een element *directeur* beschreven. De inhoud van dit element is een *string*, de naam van de directeur. Het element heeft ook een attribuut *wnd* dat aangeeft of de directeur waarnemend is. Dit attribuut kan enkel de waarde *ja* of *neen* aannemen en is van het zelf gedefinieerde type *janeentype*. Bovendien is dit attribuut optioneel en heeft het standaard de waarde *neen*.

```
<xsd:element name="directeur">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute default="neen" name="wnd"
            type="janeentype" use="optional"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

De standaardwaarde van het attribuut *use* is *optional*. Is een attribuut vereist dan moet *use* de waarde *required* aannemen.

Het complexe type (*complexType*) heeft enkel tekst als inhoud (vandaar *simpleContent*) en er worden één of meerdere attributen toegevoegd met de elementen *extension* en *attribute*.

Elementen met kindelementen Een complex type kan een ander type uitbreiden door er elementen of attributen aan toe te voegen. In het geval van een beperking kunnen elementen of attributen verwijderd worden, hun aantal kan beperkt worden, ... In dit geval is de inhoud van het element complex. Het kindelement van *complexType* is *complexContent*. Het kindelement van *complexContent* is dan *extension* of *restriction*. Dit wordt geïllustreerd met twee voorbeelden.

```
<xs:complexType name="personName">
  <xs:sequence>
    <xs:element name="title" minOccurs="0"/>
    <xs:element name="forename" minOccurs="0"
        maxOccurs="unbounded"/>
    <xs:element name="surname"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="extendedName">
  <xs:complexContent>
    <xs:extension base="personName">
      <xs:sequence>
        <xs:element name="generation" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Het type *extendedName* is een uitbreiding van *personName* met een element *generation*. Het type *simpleName* is een beperking van het type *personName*. Het element *title* is verwijderd en er kan maar één element *forname* meer zijn.

```
<xs:complexType name="simpleName">
  <xs:complexContent>
    <xs:restriction base="personName">
      <xs:sequence>
        <xs:element name="forename" minOccurs="1"
            maxOccurs="1"/>
        <xs:element name="surname"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

3.6.3 'qualified' en 'unqualified'

Als een elementdeclaratie een kindelement is van het *schema*-element dan noemen we deze declaratie globaal. Een elementdeclaratie kan ook voorkomen binnen een andere element- of typedeclaratie. In dat geval noemen we deze declaratie lokaal. In het volgend voorbeeld is de declaratie van het element *isbn* lokaal.

```
<xsd:element name="book">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="isbn"
                type="xsd:string"/>
            ...
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Standaard behoren alle elementen die lokaal gedeclareerd worden en alle attributen niet tot de namenruimte bepaald door het attribuut *targetNamespace* van het *schema*-element. In het Engels zegt men dat deze elementen en attributen *unqualified* zijn. Elementen en attributen die wel tot de namenruimte behoren zijn dan *qualified*. Standaard zijn dat de elementen en attributen die globaal gedeclareerd zijn, m.a.w. hun declaraties waren onmiddellijk kinderen van het *schema*-element.

De attributen *elementFormDefault* en *attributeFormDefault* van het *schema*-element bepalen respectievelijk of elementen die lokaal gedeclareerd zijn en attributen al dan niet tot de namenruimte behoren. De mogelijke waarden van deze attributen zijn *qualified* en *unqualified*. De standaardwaarde is voor beide *unqualified*. Moeten lokaal gedeclareerd elementen toch tot de namenruimte behoren, dan heeft het *schema*-element de volgende vorm.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.zwiftbooks.com"
    elementFormDefault="qualified">
...
</xsd:schema>
```

3.7 Schema of DTD?

Met behulp van DTD's is basisvalidatie van een XML-document mogelijk. Zo kan bepaald worden welke elementen en attributen voorkomen, hoe elementen genesteld worden, wat het type van een attribuut is, wat de standaardwaarde van een attribuut is, ... Voor beschrijvende XML-documenten is dit vermoedelijk voldoende.

XML-documenten die gegevens voorstellen hebben vaak nood aan preciesere controle over de inhoud van een element of de waarde van een attribuut. De ingebouwde datatypes van een DTD zijn dan niet voldoende. Met behulp van een XML schema is het mogelijk om complexere gegevenstypes te declareren, om specifiekere beperkingen op elementen te leggen, om types van elkaar af te leiden, ... Bovendien voorziet XML schema een groot gamma aan ingebouwde types.

DTD's voorzien ook geen expliciete ondersteuning voor namenruimtes. Schema's daarentegen maken het gebruik van verschillende namenruimtes en de bijhorende validatie een stuk makkelijker.

Hoofdstuk 4

Extensible Stylesheet Language Transformations

4.1 XSL, XPath, XSL-FO en XSLT

Zoals reeds opgemerkt in hoofdstuk 1 wordt XML niet gebruikt om gegevens op te maken, maar wel om gegevens te structuren. Om de gegevens uit een XML-documenten te presenteren werden er andere technologieën ontwikkeld.

XSL is de afkorting van *Extensible Stylesheet Language Family* en is een standaard om XML-documenten te presenteren en te transformeren. XSL bestaat uit drie delen

- XSL Transformations (XSLT)
- XPath
- XSL Formating Objects (XSL-FO)

XSLT is een XML-taal om XML-documenten om te vormen naar een ander formaat. Dit formaat kan terug een XML-taal zijn, maar dit hoeft niet. Andere formaten zijn bv. HTML, Latex, PDF, ...

XPath is een taal waarmee stukken uit een XML-document geselecteerd worden of waarmee verwezen wordt naar bepaalde delen uit een XML-document.

XSL-FO is het XML-formaat voor drukwerk. In XSL-FO wordt de layout van tekstpagina's beschreven.

4.2 Cascading Style Sheets (CSS)

Een eerste technologie die gebruikt kan worden om XML-documenten op te maken is *Cascading Style Sheets (CSS)*. CCS bepaalt de opmaak van de elementen in een XML-document. Er worden geen transformaties uitgevoerd.

CSS voegt stijlkenmerken toe aan de inhoud van het XML-documenten. Zo kan je met CSS bv. de inhoud van bepaalde elementen in vetjes of in een bepaalde kleur tonen. Maar het is bv. niet mogelijk om met CSS de volgorde van de elementen aan te passen.

4.2.1 CSS Syntax

CSS is geen XML-taal. De CSS-syntax is echter zo eenvoudig dat dat geen probleem is. Een CSS-document bestaat uit een lijst van elementen waarvoor bepaalde stijlkenmerken opgegeven worden, zoals kleur, lettertype, positionering, ...

In het volgend voorbeeld wordt een XML-document dat de beschrijving van twee boeken bevat opgemaakt door de CSS *boeken.css*.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="boeken.css"?>
<!DOCTYPE books SYSTEM "boeken.dtd">
<books>
    <book>
        <isbn>0-596-00058-8</isbn>
        <title>XML in a Nutshell</title>
        <author>
            <name>Harold</name>
            <firstname>Elliotte</firstname>
            <firstname>Rusty</firstname>
        </author>
        <author>
            <name>Means</name>
            <initial>W.</initial>
            <firstname>Scott</firstname>
        </author>
    </book>
    <book>
        <isbn>0-201-77641-3</isbn>
        <title>XML, Web Services, and the
            Data Revolution</title>
        <author>
            <name>Coyle</name>
            <initial>P.</initial>
            <firstname>Frank</firstname>
        </author>
    </book>
</books
```

Het bijhorende CSS-document is dan.

```
books {
    margin-left: 0.5cm
}

book {
    display: list-item;
    list-style-position: inside
}

title {
    font-style: italic
}
```

Als een browser het XML-document opent, dan gebruikt die het CSS-bestand om het XML-document te tonen. Het resultaat van het bovenstaande voorbeeld in een browser is als volgt.

Figuur 4.1: Resultaat

Algemeen bestaat een CSS-document uit verschillende elementbeschrijvingen van de volgende vorm.

```
naamTag1, ..., naamTagN {
    eig1: waarde1;
    eig2: waarde2;
    ...
    eigM: waardeM
}
```

Hierbij zijn *naamTag1*, ..., *naamTagN* de namen van elementen met dezelfde opmaak. De opmaak wordt vastgelegd door de kenmerken *eig1*, ..., *eigM* een waarde te geven. Een overzicht van alle eigenschappen en hun mogelijke waarden vind je terug in de CSS-specificatie ([CSSb]).

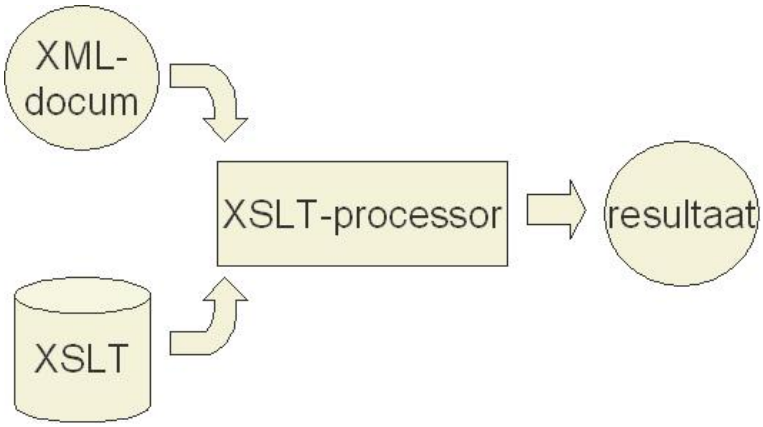
De link tussen het XML-document en het bijhorende CSS-bestand wordt bepaald door een verwerkingsinstructie (*processing instruction*) *xml-stylesheet* in het begin van het XML-document. Het attribuut *type* bepaalt het type van de *style sheet* en het attribuut *href* de locatie.

```
<?xml-stylesheet type="text/css" href="boeken.css"?>
```

4.3 XSLT

XSLT is een XML-technologie waarmee men regels kan vastleggen om een XML-document te transformeren naar een ander (eventueel XML-) document. Met deze technologie is het o.a. mogelijk om elementen te sorteren of te filteren, om nieuwe informatie toe te voegen, ... Een XSLT-document bestaat uit sjablonen (*templates*) die de transformatie beschrijven. XSLT is een XML-taal en gebruikt dus de XML-syntax.

Een XSLT-processor maakt op basis van een XML-document en een XSLT-bestand een nieuw document aan. Sommige webbrowsers hebben een ingebouwde XSLT-processor, maar ook alleenstaande applicaties worden gebruikt als XSLT-processors.



Figuur 4.2: Principe XSLT-transformatie

4.3.1 Sjablonen (*templates*)

Elk sjabloon bestaat uit een patroon en een beschrijving van de te genereren uitvoer. Een XSLT-processor vergelijkt de elementen en andere knopen in het XML-document met de patronen van de sjablonen. Indien het element of de knoop voldoet aan het patroon, dan wordt op basis van het sjabloon een stuk uitvoer gegenereerd.

XSLT gebruikt XPath om knopen in een XML-documenten te identificeren en patronen te beschrijven.

Voorbeeld XSLT-document

In dit voorbeeld maken we een HTML-document op basis van het XML-document uit §4.2. We gebruiken hiervoor het volgende XSLT-document.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
    <html>
        <head>
            <title>Overzicht boeken</title>
        </head>
        <body>
            <xsl:apply-templates/>
        </body>
    </html>
</xsl:template>

<xsl:template match="books">
    <ul>
        <xsl:apply-templates/>
    </ul>
</xsl:template>

<xsl:template match="book">
    <li>
        <xsl:value-of select="title"/>
    </li>
</xsl:template>
</xsl:stylesheet>
```

Het resultaat van de transformatie van het XML-document uit §4.2 met de bovenstaande sjablonen is een HTML-document. Deze HTML-pagina bevat een lijstje met de titels van de boeken uit het oorspronkelijke XML-document.

Figuur 4.3: Resultaat XSLT-transformatie

Resultaat XSLT-transformatie

Structuur XSLT-document

Een XSLT-document is een XML-document en begint dus met de XML-declaratie (zie §1.2.6). Het basiselement van een XSLT-document is het *stylesheet*- of *transform*-element. Deze elementen zijn

synoniemen.

Alle XSLT-elementen, en dus ook *stylesheet* en *transform* behoren tot de namenruimte (*namespace*) . Voor deze namenruimte wordt standaard de afkorting *xsl* gebruikt. Naast het *xmlns*-attribuut moet het basiselement ook het attribuut *version* met de waarde *1.0* hebben.

Sjablonen bepalen de vorm van de uitvoer van een XSLT-transformatie. Elk sjabloon in het XSLT-document wordt voorgesteld door een *xsl:template*-element. Het *match*-attribuut van dit element bepaalt de elementen of knopen waarop het sjabloon toegepast moet worden. De waarde van het *match*-attribuut is een XPath-uitdrukking. Deze uitdrukking is een patroon dat één of meerdere knopen in het XML-document identificeert. In het voorbeeld zijn de patronen namen van elementen (*books* en *book*) en het begin van het XML-document (*/*).

De inhoud van het *xsl:template*-element bepaalt de vorm van de uitvoer. Het sjabloon voor een *book*-element in het voorbeeld beschrijft dat voor elk *book*-element de HTML-tag *
*, de inhoud van het *titel*-element van het huidige *book*-element en de eind-tag *
* naar de uitvoer geschreven moet worden.

Het element *output* bepaalt het formaat van de gegenereerde uitvoer. De *value-of* en *apply-templates* elementen worden beschreven in §4.3.2.

De inhoud van het *template*-element bestaat voor een deel uit instructie-elementen (zie §4.3.2) die behoren tot de namenruimte ”http://www.w3.org/1999/XSL/Transformën voor een deel uit XML-stukken die integraal naar de uitvoer geschreven worden.

Aangezien het hele XSLT-document een goedgevormd XML-document moet zijn, moeten de HTML-elementen die deel uitmaken van de sjablonen voldoen aan de XML-syntax.

XSLT-processors

Een XSLT-processor voert de transformatie van een XML-document op basis van een XSLT-document uit. Het resultaat kan een XML-document, een HTML-document, ... zijn. Een expliciete link tussen het XML-document en het XSLT-document is niet nodig. Het moet mogelijk zijn om een zelfde XML-document naar verschillende documenten te transformeren gebruik makend van verschillende XSLT-documenten. Bijvoorbeeld een transformatie naar HTML en een transformatie naar XSL-FO, waarmee PDF gegenereerd kan worden.

Enkel indien het XML-document door een browser getransformeerd wordt is er een link met het XSLT-document nodig. Zoals bij CSS (zie §4.2) wordt deze link gerealiseerd met behulp van een verwerkingsinstructie.

```
<?xml-stylesheet type="text/xsl" href="boeken.xsl"?>
```

4.3.2 Instructie-elementen

Instructie-elementen zijn elementen uit de namenruimte “http://www.w3.org/1999/XSL/Transform” die voorkomen binnen een *template*-element. Deze elementen hoeven geen directe kinderen van het *template*-element te zijn.

xsl:value-of

Dit element bepaalt de tekst van een XPath-uitdrukking en plaatst die op de uitvoer. In het geval van een element is dat de inhoud van het element. Voor een attribuut is dat de waarde.

```
<xsl:template match="book">
    <li>
        <xsl:value-of select="title"/>
    </li>
</xsl:template>
```

Het bovenstaande sjabloon schrijft voor elk *book*-element het volgend stukje HTML naar de uitvoer. Hierbij is *titel huidig boek* de inhoud van het *title*-element van het *book*-element waarop het sjabloon toegepast wordt.

```
    <li>
        titel huidig boek
    </li>
```

xsl:apply-templates

Standaard leest een XSLT-processor een XML-document van voor naar achteren. Voor elk element dat de XSLT-processor tegenkomt voert de processor het sjabloon uit. Is er geen sjabloon beschikbaar dan wordt de inhoud van het element naar de uitvoer geschreven of worden de sjablonen van de kindelementen uitgevoerd. Dit betekent dat de sjablonen uitgevoerd worden volgens de volgorde van de elementen in het XML-document.

Met een sjabloon kan je echter de volgorde van het doorlopen van het XML-document veranderen. Zo werd in §4.3.1 voor een *book*-element enkel de inhoud van het *title* uitgeschreven. De andere kindelementen van een *book*-element werden genegeerd.

Met het element *xsl:apply-templates* kan je expliciet de volgorde van het uitvoeren van de sjablonen bepalen. Het *select*-attribuut bepaalt dan voor welke knopen het sjabloon uitgevoerd moet worden. Is dit attribuut niet aanwezig dan wordt het sjabloon voor elk kindelement uitgevoerd.

Pas het volgende stukje XSLT toe op het XML-bestand met boeken uit §4.2. Het basiselement van het XML-bestand was *books*. Dit element bestond uit een aantal *book*-elementen die een titel, auteurs, ... hadden.

```
<xsl:template match="books">
    <ul>
        <xsl:apply-templates/>
    </ul>
</xsl:template>

<xsl:template match="book">
    <li>
        <xsl:apply-templates select="title"/>
    </li>
</xsl:template>
```

Het resultaat van dit stukje XSLT is dat het sjabloon voor het element *books* wordt uitgevoerd: de tag *
* uitschrijven, vervolgens de sjablonen van alle kindelementen (lees: alle *book*-elementen) uitvoeren en dan de tag *
* uitschrijven.

Voor elke *book*-element wordt het sjabloon uitgevoerd: de tag *
* uitschrijven, vervolgens de sjablonen van alle *title*-elementen uitvoeren en dan de tag *
* uitschrijven. Afhankelijk van het sjabloon voor het *title*-element kan het resultaat bv. het volgende zijn. ument. Deze HTML-pagina bevat een lijstje met de titels van de boeken uit het oorspronkelijke XML-document.

Figuur 4.4: Uitvoer stukje XSLT

Als een sjabloon verschillende keren uitgevoerd moet worden voor verschillende elementen of knopen, dan is de volgorde waarin dat gebeurt de volgorde van de knopen in het document. Om de sjablonen in een andere volgorde uit te voeren kan je gebruik maken van het *xsl:sort*-element (§4.3.2). Dit element is dan een kindelement van het *xsl:apply-templates*-element.

xsl:for-each

Met behulp van het *xsl:foreach*-element is het mogelijk om een zelfde actie voor een reeks knopen uit te voeren. De knopen waarop deze actie uitgevoerd wordt, worden bepaald door het *select*-attribuut. De waarde van dit attribuut is een XPath-uitdrukking die een aantal knopen identificeert.

De geselecteerde knopen worden behandeld in de volgorde waarin ze in het XML-document voorkomen. Om deze volgorde te veranderen kan je gebruik maken van het *xsl:sort* (zie §4.3.2)-element.

In het volgend voorbeeld kan een *author*-element meerdere *firstname*-elementen bevatten. Het sjabloon voor een *author*-element voert het sjabloon voor de verschillende *firstname*-elementen uit en schrijft tussen de uitvoer van twee sjablonen een spatie naar de uitvoer.

```
<xsl:template match="author">
    <xsl:for-each select="firstname">
        <xsl:apply-templates select="."/>
        <xsl:text> </xsl:text>
    </xsl:for-each>
</xsl:template>
```

Merk op dat '.' een XPath-uitdrukking is voor de huidige knoop. In het geval van het voorbeeld is dat een *firstname*-element.

xsl:sort

Het *xsl:sort*-element is een kindelement van een *xsl:apply-templates* (zie §4.3.2)-element of een *xsl:for-each* (zie §4.3.2)-element. Het bepaalt de volgorde waarin de sjablonen of de acties uitgevoerd worden. Deze sortering kan alfabetisch of numeriek zijn. Wil je op meerdere aspecten sorteren, dan moet je meerdere *xsl:sort*-elementen gebruiken.

Dit element heeft o.a. de volgende attributen.

select bepaalt de knoop, het element, het attribuut, ... waarop gesorteerd wordt.

data-type bepaalt het type van de knoop waarop gesorteerd wordt. Standaard wordt er alfabetisch gesorteerd (waarde is dan *text*). Om numeriek te sorteren moet dit attribuut de waarde *number* hebben.

order bepaalt de sorteervolgorde. Standaard is dat oplopend (waarde van het attribuut is dan *ascending*). Om aflopend te sorteren moet dit attribuut de waarde *descending* hebben.

In het volgend voorbeeld wordt voor elke kindelement *book* van het element *books* het sjabloon uitgevoerd. De volgorde waarin de sjablonen worden uitgevoerd is afhankelijk van het *isbn*-element. Het sjabloon wordt eerst toegepast op het boek met het (alfabetisch) kleinste isbn-nummer, enz.

```
<xsl:template match="books">
    <ul>
        <xsl:apply-templates select="book">
        <xsl:sort select="isbn"/>
        </xsl:apply-templates>
    </ul>
</xsl:template>
```

xsl:if

De XSLT-syntax heeft ook een aantal selectiestructuren. Het instructie-element *xsl:if* is er één van. Deze constructie is vergelijkbaar met een *if*-structuur zonder *else*-clausule uit een programmeertaal. De algemene syntax is de volgende.

```
<xsl:if test="voorwaarde">
    opdrachten
</xsl:if>
```

Hierbij is de waarde van het attribuut *test* (*voorwaarde*) een logische uitdrukking in XPath die al dan niet vervuld is. De XSLT-opdrachten *opdrachten* worden uitgevoerd als de voorwaarde vervuld is.

Stel dat je een XML-document hebt waarbij de structuur bepaald wordt door de volgende DTD.

```
<!ELEMENT catalog (cd*)>
<!ELEMENT cd (price, title, artist)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

Het volgende stukje XSLT zorgt er dan voor dat enkel voor de CD's waarvan de prijs groter is dan *10*, de inhoud van het *title*- en het *artist*-element uitgeschreven wordt.

```
<xsl:for-each select="catalog/cd">
    <xsl:if test="price &gt; 10">
        <tr><td>
            <xsl:value-of select="title"/>
        </td><td>
            <xsl:value-of select="artist"/>
        </td></tr>
    </xsl:if>
</xsl:for-each>
```

Merk op dat de voorwaarde gebruik maakt van *>* ipv van $>$ omdat het XSLT-document een goed-gevormd XML-document moet zijn.

xsl:choose

Een andere selectie-structuur is het element *xsl:choose*. Deze structuur selecteert nul of één geval van een reeks alternatieven. De syntax is als volgt.

```
<xsl:choose>
    <xsl:when test="geval1">opdrachten1</xsl:when>
    <xsl:when test="geval2">opdrachten2</xsl:when>
    ...
```

```
     <xsl:when test="gevalN">opdrachtenN</xsl:when>
    <xsl:otherwise>opdrachtenAnders</xsl:otherwise>
</xsl:choose>
```

Het *xsl:choose*-element bestaat uit één of meerdere *xsl:when*-elementen eventueel gevolgd door één *xsl:otherwise*-element. Elk *xsl:when*-element heeft een *test*-attribuut. De waarde van dit attribuut (respectievelijk *geval1*, *geval2*, ...) is een logische XPath-uitdrukking.

De opdrachten van het eerste *when*-element waarvan de voorwaarde vervuld is, worden uitgevoerd. Indien voor geen enkele *when*-element de voorwaarde waar is, dan worden de opdrachten in het *xsl:otherwise*-element uitgevoerd indien dat element aanwezig is.

xsl:text

Met het *xsl:text*-element kan je tekst naar de uitvoer schrijven. Dit kan handig zijn om spaties of andere witte ruimte (*whitespace*) op te nemen in het resultaat. Aangezien een XSLT-document een XML-document is, worden spaties, ... normaal genegeerd.

In het volgend voorbeeld wordt het *xsl:text*-element gebruikt om een spatie te plaatsen tussen de inhoud van verschillende *firstname*-elementen.

```
<xsl:for-each select="firstname">
    <xsl:apply-templates select="."/>
    <xsl:text> </xsl:text>
</xsl:for-each>
```

xsl:variable

Het element *xsl:variable* wordt gebruikt om een naam met een bepaalde waarde te verbinden. Deze waarde kan dan elders in het XSLT-document opgeroepen worden met behulp van de gegeven naam. Merk op dat het geen variabele is zoals in een programmeertaal. De waarde kan namelijk niet aangepast worden en dus eigenlijk meer het equivalent van een constante in een programmeertaal. De waarde kan van gelijk welk XPath-type zijn: string, getal, verzameling knopen, ...

De syntax van het *xsl:variable*-element:

```
<xsl:variable name=?naam? select=?waarde?/>
```

of

```
<xsl:variable name=?naam?>
    ...
</xsl:variable>
```

In het eerste geval heeft het *xsl:variable*-element een *select*-attribuut en moet het een leeg element zijn. De waarde is dan de waarde van het *select*-attribuut (een XPath-uitdrukking). In het tweede geval is de waarde de inhoud van het *xsl:variable*-element.

De waarde van de 'variabele' kan in een uitdrukking elders in het XSLT-document opgeroepen worden. Hiervoor wordt een uitdrukking van de vorm *$naam* gebruikt. *naam* is de waarde van het *name*-attribuut.

In het volgend voorbeeld worden twee 'variabelen' gedeclareerd. De eerste is het aantal *book*-elementen in het element *books*. De tweede is het getal *10*. Het stukje XSLT schrijft uit hoeveel *book*-elementen er zijn. Als minder zijn dan 10 dan wordt dit ook gemeld.

```
<xsl:variable name="aantal" select="count(books/book)"/>
<xsl:variable name="min">10</xsl:variable>
<xsl:text>Er zijn </xsl:text>
<xsl:value-of select="$aantal"/>
<xsl:text> boeken in de catalogus. </xsl:text>
<xsl:if test="$aantal &lt; $min">
    <xsl:text>Er zijn te weinig boeken.</xsl:text>
</xsl:if>
```

xsl:param en xsl:with-param

Met de elementen *xsl:param* en *xsl:with-param* is het mogelijk om een parameter mee te geven aan een sjabloon. Het element *xsl:param* declareert de parameter in het sjabloon en geeft hem een standaardwaarde. Het element *xsl:with-param* wordt gebruikt om de parameter mee te geven bij het oproepen van het sjabloon.

De syntax van het *xsl:param*-element is analoog aan de syntax van het *xsl:variable*-element.

```
<xsl:param name="naam" select="waarde"/>
```

of

```
<xsl:param name="naam">
    ...
</xsl:param>
```

De waarde van het attribuut *name* bepaalt de naam van de parameter. De standaardwaarde van de parameter is in het eerste geval de waarde van het *select*-attribuut en in het tweede de inhoud van het *xsl:param*-element. Als het *xsl:param*-element een *select*-attribuut heeft, dan moet het een leeg element zijn. Dit attribuut is een XPath-uitdrukking. De waarde van de parameter is het resultaat van deze uitdrukking.

Het *xsl:param*-element is een kindelement van een *xsl:template*-element of eventueel van het basiselement. In het laatste geval is de parameter beschikbaar voor meerdere sjablonen.

Om een parameter mee te geven aan een sjabloon wordt het *xsl:with-param*-element gebruikt.

```
<xsl:apply-templates ...>
    <xsl:with-param name="naam" select="waarde"/>
</xsl:apply-templates>
```

Hierbij is *naam* de naam van de parameter die gedeclareerd werd in het sjabloon (m.b.v. het *xsl:param*-element). De waarde van de parameter, de XPath-uitdrukking *waarde* in het voorbeeld, vervangt de standaardwaarde van de parameter die vastgelegd werd door het *xsl:param*-element.

Ook hier moet het *xsl:with-param*-element leeg zijn als het een *select*-attribuut heeft. Een alternatief voor bovenstaande code is.

```
<xsl:apply-templates ...>
    <xsl:with-param name="naam">
        ...
    </xsl:with-param>
</xsl:apply-templates>
```

De inhoud van het element *xsl:with-param* bepaalt nu de waarde van de parameter.

Wordt een sjabloon opgeroepen zonder parameter door te geven, dan heeft de parameter de standaardwaarde bepaald door het *xsl:param*-element.

De waarde van de parameter wordt in het sjabloon opgeroepen door een uitdrukking van de vorm *$naam*. Hierbij is *naam* de naam van de parameter (de waarde van het *name*-attribuut van het element *xsl:param*).

In het volgend voorbeeld veronderstellen we dat er verschillende servers met foto's beschikbaar zijn via het internet. De gegevens van deze servers worden bewaard in een XML-bestand van de volgende vorm.

```
<?xml version="1.0"?>
<servers>
    <server>
        <url>http://www.mijnfoto.be</url>
        <foto bron="IMG001">Mijn huis</foto>
        <foto bron="IMG002">Kinderen</foto>
        <foto bron="IMG003">Poezen</foto>
        <foto bron="IMG004">Tuin</foto>
        <foto bron="IMG005">Vakantie in Praag</foto>
        <foto bron="IMG006">Scoutskamp</foto>
        <foto bron="IMG007">Sneeuwpret</foto>
    </server>
    <server>
        <url>http://www.bedreigdedieren.org</url>
        <foto bron="IMG001">Bengaalse tijger</foto>
        <foto bron="IMG002">Panda</foto>
        <foto bron="IMG003">Koala</foto>
```

```
        <foto bron="IMG004">Wolf</foto>
    </server>
</servers>
```

De bijhorende DTD is dan

```
<!ELEMENT servers (server*)>
<!ELEMENT server (url,foto*)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT foto (#PCDATA)>
<!ATTLIST foto bron NMTOKEN #REQUIRED
```

De bedoeling is om voor XML-bestanden van bovenstaande vorm een HTML-pagina te genereren die een overzicht geeft van de beschikbare foto's en waarop je kan doorklikken naar de foto's op de servers. In de twee kolom wordt de URL van de foto nog eens vermeld. De link in de eerste kolom klikt door naar de URL getoond in de tweede kolom. (Zie figuur 4.5)

De volgende sjablonen voor *server* en *foto* realiseren deels de omzetting van het XML-bestand naar het HTML-bestand.

```
<xsl:template match="server">
    <xsl:variable name="host" select="url"/>
    <table border="1">
        <tr><th colspan="2">
            <xsl:value-of select="$host"/>
        </th></tr>
        <xsl:apply-templates select="foto">
            <xsl:with-param name="host"
                select="$host"/>
        </xsl:apply-templates>
    </table> <br></br>
</xsl:template>

<xsl:template match="foto">
    <xsl:param name="host"/>
    <xsl:variable name="url">
        <xsl:value-of select="$host"/>
        <xsl:text>/</xsl:text>
        <xsl:value-of select="@bron"/>
    </xsl:variable>
    <tr><td>
        <![CDATA[<a href="]]>
        <xsl:value-of select="$url"/>
        <![CDATA[">]]>
        <xsl:value-of select="."/>
        <![CDATA[</a>]]>
    </td><td>
    <xsl:value-of select="$url"/>
    </td></tr>
</xsl:template>
```

De *CDATA*-stukken in het bovenstaande voorbeeld worden gebruikt om een attribuut toe te voegen aan het HTML-element *a*. Een beter alternatief is het gebruik van het *attribute*-element zoals geïllustreerd in de volgende code.

```
<tr><td>
    <a>
        <xsl:attribute name="href">
            <xsl:value-of select="$url"/>
        </xsl:attribute>
        <xsl:value-of select="."/>
    </a>
</td><td>
<xsl:value-of select="$url"/>
</td></tr>
```

Figuur 4.5: Voorbeeld HTML-uitvoer

Hoofdstuk 5

XML Path Language (XPath)

XPath is een taal om bepaalde stukken van een XML-documenten te selecteren. XPath is zelf geen XML-taal. XPath beschouwt een XML-document als een boomstructuur met knopen. Aan de hand van allerlei criteria, zoals de naam van een element, zijn positie, zijn inhoud, de waarde van een attribuut, ... is het mogelijk om knopen in die boom te identificeren.

Zoals reeds besproken in hoofdstuk 4 gebruikt XSLT XPath-uitdrukkingen om bijvoorbeeld elementen te selecteren waarop een sjabloon toegepast moet worden. Ook andere technologieën zoals XForms en XPointer maken gebruik van XPath.

Naast knopen van een XML-document kan het resultaat van een XPath-uitdrukking een getal, een string of een logische waarde (*boolean*) zijn. Eenvoudige rekenkundige uitdrukkingen en stringmanipulaties zijn dus ook mogelijk.

5.1 De boomstructuur van een XML-document

Voor XPath is een XML-document een boom van knopen. Sommige knopen kunnen andere bevatten. Er is juist één basisknoop (*root node*) die alle andere bevat. XPath is een taal om knopen en verzamelingen van knopen te selecteren in die boom. XPath kent zeven types knopen.

- De basisknoop
- Elementen
- Tekst
- Attributen
- Commentaar
- Verwerkingsinstructies

- Namenruimtes

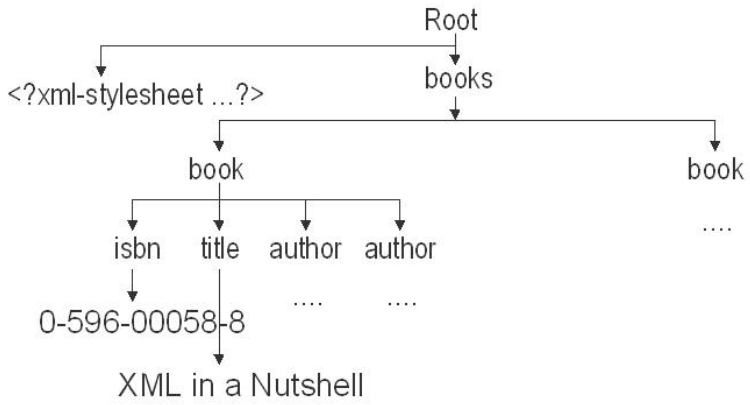
DTD's, entiteiten en CDATA-stukken zijn geen knopen voor XPath en kunnen dus niet geïdentificeerd worden met een XPath-uitdrukking. In de volgende voorbeelden wordt de boomstructuur van een aantal XML-documenten getoond.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="boeken.xsl"?>
<!DOCTYPE books SYSTEM "boeken.dtd">
<books>
    <book>
        <isbn>0-596-00058-8</isbn>
        <title>XML in a Nutshell</title>
        <author>
            <name>Harold</name>
            <firstname>Elliotte</firstname>
            <firstname>Rusty</firstname>
        </author>
        <author>
            <name>Means</name>
            <initial>W.</initial>
            <firstname>Scott</firstname>
        </author>
    </book>
    <book>
        <isbn>0-201-77641-3</isbn>
        <title>XML, Web Services, and the
            Data Revolution</title>
        <author>
            <name>Coyle</name>
            <initial>P.</initial>
            <firstname>Frank</firstname>
        </author>
    </book>
</books>
```

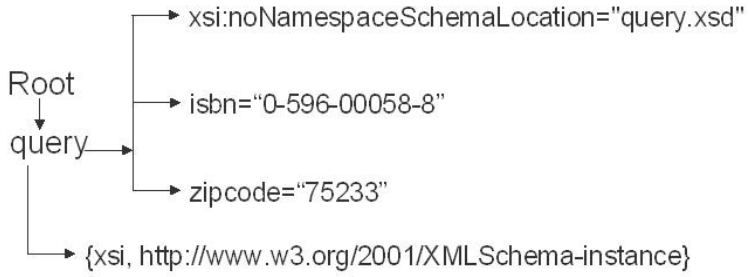
```
<?xml version="1.0" encoding="UTF-8"?>
<query xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="query.xsd"
     isbn="0-596-00058-8" zipcode="75233"/>
```

Merk op dat de basisknoop *NIET* het basiselement van het XML-document is. De basisknoop omvat het volledige XML-document: basiselement, eventuele verwerkingsinstructies of commentaar voor of na het basiselement. De verwerkingsinstructie *xml-stylesheet* uit het eerste voorbeeld is een kind van de basisknoop.

De attributen *xmlns* en *xmlns:prefix* worden niet beschouwd als attribuutknopen. De boom bevat ook de namenruimte die gekend is in het *query*-element.



Figuur 5.1: XPath-boom books



Figuur 5.2: XPath-boom query

5.2 Paden

Een zeer belangrijk XPath-uitdrukking is een pad (*location path*). Een pad identificeert nul, één of meerdere knopen in een XML-document. De types van de geselecteerde knopen zijn één of meerdere types zoals beschreven in §5.1. Een pad bestaat uit verschillende stappen (*location steps*) verbonden met een /. Elke stap wordt uitgevoerd t.o.v. een bepaalde knoop, de contextknoop (*context node*) genoemd. Deze paden zijn op het eerste zicht vergelijkbaar met paden in een bestandssysteem.

5.2.1 Basispad (*Root Location Path*)

Het eenvoudigste pad is '/' en selecteert de basisknoop van het XML-document. Dit is een absoluut pad. In welke context je het basispad ook opgeeft, het identificeert steeds de basisknoop van het XML-document. (Vergelijkbaar met de *root* van een Unix-bestandssysteem.)

In het volgend voorbeeld gebruikt XSLT het basispad om een sjabloon te maken voor het volledige XML-document.

```
<xsl:template match="/">
```

```
    <html>
        <head>
            <title>Overzicht boeken</title>
        </head>
        <body>
            <xsl:apply-templates/>
        </body>
    </html>
</xsl:template>
```

5.2.2 Kindelement-stap (*Child Element Location Steps*)

De kindelement-stap bestaat uit de naam van een element en selecteert alle elementen met deze naam in de huidige context. Bijvoorbeeld de stap *book* identificeert alle kindelementen *book* van de contextknoop. Welke elementen geselecteerd worden hangt af van de contextknoop. Een kindelement-stap is dus steeds een relatief pad.

Bijvoorbeeld, in de contextknoop *books* in het eerste voorbeeld van §5.1, zal het pad *book* verwijzen naar twee *book*-elementen. Was de contextknoop echter /, de basisknoop van het document, dan selecteert het pad *book* niets.

In XSLT is de contextknoop voor een XPath-uitdrukking, bv. als waarde van een *select*-attribuut, die knoop die afgehandeld wordt. In het onderstaande voorbeeld is de contextknoop voor het pad *name*, het *author*-element waarvoor het sjabloon wordt uitgevoerd.

```
<xsl:template match="author">
    <xsl:apply-templates select="name" />
</xsl:template>
```

5.2.3 Attribuut-stap (*Attribute Location Steps*)

Om het attribuut van een element te selecteren gebruik je een @ gevolgd door de naam van het attribuut. Het element moet dan de contextknoop zijn. Om in het tweede voorbeeld van §5.1 het attribuut *isbn* van het element *query* te selecteren kan je de volgende paden gebruiken.

- @*isbn* als de context het element *query* is
- */query/@isbn* in elke context

Met het volgende stukje XSLT is het mogelijk om de waarde van het *isbn*-attribuut van het *query*-element uit te schrijven.

```
<xsl:template match=?query">
    <xsl:value-of select=?@isbn" />
</xsl:template>
```

5.2.4 Stap naar commentaar, tekst of verwerkingsinstructie

Aangezien er naast de basisknoop, elementen en attributen ook knopen in de boomstructuur van een XML-document voorkomen die commentaar, tekst of verwerkingsinstructies voorstellen, moeten die ook geselecteerd kunnen worden met een XPath-uitdrukking. Tekst- en commentaar-knopen hebben geen naam en worden respectievelijk met de volgend stappen geïdentificeerd.

- *text()*
- *comment()*

Het volgende pad selecteert alle ISBN-nummers, als tekstknopen van de XML-boom, in het eerste voorbeeld van §5.1.

```
/books/book/isbn/text()
```

Bij verwerkingsinstructies is het mogelijk om alle verwerkingsinstructies in een bepaalde context te selecteren of om enkel die verwerkingsinstructies te identificeren met een bepaalde naam.

- *processing-instruction()*
- *processing-instruction('target')* (*target* is de naam van de te selecteren verwerkingsinstructies)

5.2.5 Andere padelementen

Naast de stappen besproken in de voorgaande paragrafen kunnen paden nog uit andere aspecten bestaan zoals bv. jokertekens.

Jokertekens (*wildcards*)

Met jokertekens is het mogelijk om verschillende element- of andere knopen tegelijk te selecteren. De drie mogelijke jokertekens zijn.

- *: selecteert elke elementknoop in de context. De naam van het element is willekeurig.
- *node()*: selecteert alle knopen in de context. Deze knopen kunnen van gelijk welk type zijn: elementen, attributen, tekst, commentaar, namenruimtes of verwerkingsinstructies.
- @*: selecteert alle attributen van de huidige context.

Het is ook mogelijk om alleen elementen of attributen uit een bepaalde namenruimte te selecteren. In dat geval maakt ook de afkorting (*prefix*) van de namenruimte deel uit van het pad.

- *prefix:**: selecteert elke elementknoop van een bepaalde namenruimte in de contextknoop.
- @*prefix:**: selecteert alle attributen van een bepaalde namenruimte in de huidige contextknoop.

Meerdere mogelijkheden

Soms wil je meerdere types elementen of attributen identificeren, maar niet alle types. Bijvoorbeeld, je wilt een XSLT-sjabloon maken voor de elementen *isbn* en *title*, maar niet voor de elementen *author*, *firstname*,... In dat geval kan je paden combineren met een rechte streep (—). In de volgende voorbeelden worden alle *isbn*- en *title*-elementen geselecteerd.

```
isbn|title
```

```
/books/book/isbn|title
```

Speciale tekens

Zoals bij paden in bestandssystemen hebben één punt (.) en twee punten (..) een speciale betekenis voor paden in XPath. Ook twee schuine strepen (//) hebben een speciale betekenis.

De contextknoop selecteren met . Een punt (.) betekent de contextknoop. In XSLT wordt die vaak gebruikt om de waarde van de huidige knoop te bepalen. In het volgend voorbeeld wordt een sjabloon beschreven voor *title*-elementen. De inhoud van de *title*-elementen wordt naar de uitvoer geschreven als kind van het HTML-element *i*. Het zal dus als schuine tekst getoond worden in een browser.

```
<xsl:template match="title">
    <i>
        <xsl:value-of select="." />
    </i>
</xsl:template>
```

Nakomelingen selecteren met // Twee schuine strepen (//) selecteren knopen uit alle nakomelingen van de contextknoop en de contextknoop zelf. In het begin van XPath-uitdrukking selecteert het alle nakomelingen van de basisknoop (*root node*).

De uitdrukking *//book* selecteert alle *book*-elementen in het XML-document. Alle *id*-attributen in een XML-document identificeren kan met de uitdrukking *//@id*. Het volgende voorbeeld selecteert alle *name*-elementen die voorkomen in de huidige knoop (als kind, kleinkind, achterkleinkind, ...)

```
.//name
```

Selecteer het ouderelement met .. Twee punten (..) betekenen de ouder van de huidige knoop. In het volgende voorbeeld worden alle elementen met een attribuut *id* geïdentificeerd.

```
//@id/..
```

5.2.6 Voorwaarden

Een XPath-uitdrukking kan meer dan één knoop identificeren. Soms is het nodig om deze selectie nog te verfijnen. Dit kan met een voorwaarde. Elke stap in een pad kan een voorwaarde hebben die de selectie knopen, bepaald door de stap, kan verkleinen. Deze voorwaarde is een logische uitdrukking en wordt gecontroleerd voor elke knoop uit de selectie. Is de voorwaarde niet voldaan (het resultaat is *false*), dan wordt de knoop verwijderd uit de selectie.

In het volgend voorbeeld wordt het basiselement *query* geïdentificeerd waarvan de waarde van het attribuut *zipcode* de waarde *75233* heeft.

```
/query[@zipcode=75233]
```

```
//book[title="XML"]
//name[.='Ongenae']
```

In de bovenstaande voorbeelden worden respectievelijk alle *book*-elementen geselecteerd waarvan het kindelement *title* de inhoud *XML* heeft en alle *name*-elementen waarvan de inhoud *Ongenae* is.

Merk op dat de waarde van een element de inhoud van het element is en dat voor strings zowel enkele (') als dubbele (") aanhalingstekens gebruikt kunnen worden. Dit is vaak zinvol als een XPath-uitdrukking gebruikt wordt als waarde van een attribuut, die tussen dubbele aanhalingstekens staat.

```
<xsl:apply-templates select="//name[.='Ongenae']" />
```

In de bovenstaande voorbeelden wordt de relationele operator = gebruikt. Naast deze operator ondersteunt XPath ook nog de operatoren $<$, $>$, $<=$, $>=$, =, *or* en *and*. De uitdrukking *//person[@born<=1976]* identificeert alle *person*-elementen waarvan de numerieke waarde van het *born*-attribuut kleiner is dan of gelijk aan 1976. Merk op dat als je deze uitdrukking gebruikt in een XML-document (bv. XSLT), je het $<$-teken moet vervangen door *<*.

Elke stap in een pad kan een voorwaarde hebben. De uitdrukking //person[@born¡1950]/name[firstname="Alan"] selecteert eerst alle *person*-elementen waarvan de waarde van het attribuut *born* kleiner is dan 1950. Van al deze *person*-elementen selecteert ze de *name*-elementen die een kindelement *firstname* hebben met inhoud *Alan*.

5.2.7 Lange padnamen

Tot nu toe hebben we verkorte paden gebruikt. Deze zijn makkelijker om te schrijven, korter en vertrouwder. Dit zijn ook de paden die het best zijn voor XSLT. Er bestaat echter ook een niet-verkorte syntax voor paden in XPath. Deze syntax is langer, maar minder cryptisch en flexibeler.

Elke stap in een pad heeft twee vereiste delen, een as en een test om knopen te identificeren, en één optioneel stuk, een voorwaarde. De as bepaalt de richting waarin knopen geselecteerd worden vanuit de contextknoop. De test bepaalt welke knopen uit die bepaalde richting geïdentificeerd moeten worden. De voorwaarde kan de selectie nog verfijnen.

In een verkort pad worden de as en de knopentest gecombineerd. In de niet-verkorte vorm worden as en test gescheiden door twee dubbele punten (::). De uitdrukkingen

```
books/book/isbn
query/@isbn
```

worden in niet-verkorte vorm

```
child::books/child::book/child::isbn
child::query/attribute::isbn
```

De as *child* bepaalt dat er enkel naar kindelementen van de contextknoop gekeken wordt. Met de as *attribute* worden alle attributen van de huidige knoop geselecteerd. Naast deze twee assen zijn er nog drie die we reeds in verkorte vorm bespraken.

- *parent* (..)
- *self* (.)
- *descendant-or-self* (//)

De andere assen hebben geen verkorte vorm.

ancestor	Alle voorouders (elementen) van de contextknoop (niet inbegrepen).
ancestor-or-self	Alle voorouders van de contextknoop en de contextknoop zelf.
namespace	Alle namenruimtes in de huidige knoop. Deze namenruimtes kunnen gedefinieerd zijn in de huidige knoop of in één van zijn voorouders.
descendant	Alle nakomelingen van de huidige knoop (kinderen, kinderen van kinderen, ...), behalve de huidige knoop zelf.
following-sibling	Alle knopen na de huidige knoop en kinderen van dezelfde ouder als de contextknoop. Attributen en namenruimtes hebben geen siblings (broers of zussen).
preceding-sibling	Alle knopen voor de huidige knoop en kinderen van dezelfde ouder als de contextknoop. Attributen en namenruimtes hebben geen siblings (broers of zussen).
following	Alle knopen na de huidige knoop, maar geen attributen of namenruimtes.
preceding	Alle knopen voor de huidige knoop, maar geen attributen of namenruimtes.

5.3 Andere XPath-uitdrukkingen

Tot nu toe hebben we ons geconcentreerd op paden. Paden zijn XPath-uitdrukkingen die een verzameling knopen van een XML-documenten selecteren. Naast deze paden zijn er ook XPath-uitdrukkingen die strings, getallen of logische waarden voorstellen. Bijvoorbeeld,

```
@born < 1950
'Ongenae'
75332
@zipcode=75332
```

Deze uitdrukkingen zijn geen verzamelingen van knopen en kunnen dus niet gebruikt worden in het *match*-attribuut van een *xsl:template*-element. Ze kunnen echter wel gebruikt worden als waarde van het *select*-attribuut van een *xsl:value-of* -element of in voorwaarden.

5.3.1 Datatypes

XPath kent drie andere datatypes naast knopen: getallen, strings en logische waarden.

Getallen

Er zijn geen echte gehele getallen in XPath. Alle getallen worden voorgesteld door de IEEE 64-bit codering (equivalent met het type *double* in Java).

Voor dit gegevenstype zijn vijf rekenkundige operatoren beschikbaar: + (optelling), - (verschil), * (product), *div* (deling) en *mod* (rest bij deling). Deze operatoren gedragen zich zoals in Java.

De volgende XSLT-opdracht schrijft de eeuw waarin iemand geboren is naar de uitvoer. De waarde van het attribuut *born* is het geboortejaar.

```
<xsl:value-of select=''(@born-(@born mod 100)) div 100 + 1''/>
```

Strings

Strings in XPath bestaan uit reeksen van Unicode-lettertekens. Strings staan tussen enkele of dubbele aanhalingstekens. Deze tekens mogen geen deel uitmaken van de string. Als een string dubbele aanhalingstekens bevat moet hij tussen enkele aanhalingstekens staan en omgekeerd.

Met de operatoren = en *!*= kan je twee strings vergelijken.

Logische waarden

Een logische waarde heeft twee mogelijke statussen: *true* en *false*. Deze twee sleutelwoorden zijn niet beschikbaar in XPath. Wel zijn er twee functies die de rol van deze constanten overnemen: *true()* en *false()*.

Meestal verkrijg je logische waarden als resultaat van een vergelijking tussen twee getallen of strings. Deze kunnen gecombineerd worden met de operatoren *and* en *or*. De negatie wordt gerealiseerd met de functie *not(...)*

Logische waarden komen meestal voor in de voorwaarden van paden en in het *test*-attribuut van sommige XSLT-elementen (bv. *xsl:if*).

5.3.2 Functies

XPath heeft een aantal functies. Deze sectie geeft hiervan een beperkt overzicht. Er bestaan geen procedures in XPath. Het resultaat van de functie is van één van de bestaande types: logische waarde, getal, strings of verzameling van knopen.

Knopenfuncties

Met knopenfuncties worden functies bedoeld die ofwel als argument ofwel als resultaat een verzameling van knopen hebben.

position()	bepaalt het volgnummer van de huidige knoop in de context.
last()	bepaalt het aantal knopen in de huidige context. (Het volgnummer van de laatste knoop)
count(...)	telt het aantal knopen in het argument. Het argument is een verzameling knopen.
id(...)	bepaalt een verzameling knopen. Deze verzameling bevat alle elementen in het XML-document met één van de gespecifieerde ID's. Het argument is een string bestaande uit ID's gescheiden door spaties.

In het onderstaand voorbeeld wordt het sjabloon voor een element *jaar* beschreven. Een jaar kan meerdere *vak*-elementen hebben. Het *xsl:for-each*-element overloopt deze elementen. De functie *position()* geeft het volgnummer van het huidige *vak*-element weer dat afgehandeld wordt door de lus. Het bepaalt met andere woorden het hoeveelste *vak*-element dit element is binnen het *jaar*-element.

```
<xsl:template match="jaar">
    <xsl:for-each select="vak">
        <xsl:value-of value="position()"/>
    </xsl:for-each>
</xsl:template>
```

String-functies

XPath bevat een aantal functies voor eenvoudige stringmanipulaties zoals de lengte bepalen, samenvoegen, ...

concat(...,...)	voegt zijn argumenten samen tot één string. Deze functie verwacht tenminste twee argumenten.
contains(...,...)	gaat na of het tweede argument een deel (substring) is van het eerste. Deze functie is hoofdlettergevoelig.
starts-with(...,...)	controleert of het eerste argument begint met het tweede argument.
string(...)	converteert het argument naar een string.
string-length(...)	bepaalt de lengte van het argument.

Logische functies

true()	geeft steeds de waarde *true* terug. De symbolische constante *true* bestaat niet in XPath.
false()	geeft steeds de waarde *false* terug. De symbolische constante *false* bestaat niet in XPath.
not(...)	neemt de negatie van het argument. *true* wordt *false* en omgekeerd.
boolean(...)	converteert het argument naar een logische waarde. Alle getallen behalve 0 en NaN worden *true*. Lege verzamelingen van knopen worden *false*. Alle strings behalve de lege string zijn *true*.

Numerieke functies

XPath voorziet een aantal eenvoudige numerieke functies.

ceiling(...)	bepaalt de kleinste gehele waarde groter dan of gelijk aan het argument.
floor(...)	bepaalt de grootste gehele waarde kleiner dan of gelijk aan het argument.
number(...)	converteert het argument naar een getal.
round(...)	bepaalt de gehele waarde dichtst bij het argument.

Hoofdstuk 6

XML en programmeermodellen

In dit hoofdstuk bespreken we een aantal gangbare programmeermodellen om XML-documenten te parsen en te manipuleren.

6.1 Mogelijke modellen

XML-documenten zijn gestructureerde en gelabelde tekstdocumenten die op verschillende manieren verwerkt kunnen worden. Programma's kunnen XML beschouwen als tekst, als een stroom van gebeurtenissen, als een boomstructuur, ...

Tekstverwerkers zoals vi, emacs, notepad, wordpad, ... maar ook programma's zoals sed, grep, awk, ... behandelen XML als tekst. Deze programma's geven geen speciale betekenis aan de XML-labels (*tags*).

6.1.1 XML-parser

Programma's die de inhoud van een XML-document willen begrijpen, gebruiken een *XML-parser* om het document te lezen. Deze parser interpreteert het document en zal het opdelen in elementen, attributen, ... Stukje bij beetje geeft hij de inhoud van het document door aan de applicatie. Indien de parser vaststelt dat het document niet welgevormd is, zal hij het parsen onderbreken en een fout melden.

Meestal doen parsers meer dan enkel het omzetten van platte tekst naar betekenisvolle XML. Er zijn parsers om XML-documenten te controleren op welgevormdheid, om XML-documenten te valideren t.o.v. DTD's of XML Schema's, om een XML-boomstructuur op te bouwen, om gebeurtenissen te genereren, ...

6.1.2 XML: een opeenvolging van gebeurtenissen

Principe

Een XML-parser leest een XML-document één keer van het begin tot het einde. Hij pauseert enkel om externe bronnen zoals DTD's, externe entiteiten, ... op te halen. Tijdens het lezen houdt de parser een stuk context bij. Dit is nodig om welgevormdheid te controleren, om het document te valideren, om namenruimtes toe te passen, om standaard attribuutwaarden toe te kennen, ...

Parsers die gebruik maken van gebeurtenissen (*event based parsers*), genereren tijdens het lezen van het XML-document een hele reeks gebeurtenissen. Op deze manier geven ze de informatie van het XML-document door aan de applicatie. Mogelijke gebeurtenissen zijn bijvoorbeeld het openen van een element, het sluiten van een element, de inhoud van een element, ...

Voor het volgend stukje XML

```
<boek><titel>Java en XML</titel><auteur>
<voornaam>Jan</voornaam><naam>Janssens</naam>
</auteur></boek>
```

genereert dit type parser bijvoorbeeld de volgende gebeurtenissen.

```
start element: boek
start element: titel
inhoud:Java en XML
einde element: titel
start element: auteur
start element: voornaam
inhoud: Jan
einde element: voornaam
start element: naam
inhoud: Janssens
einde element: naam
einde element: auteur
einde element: boek
```

In dit voorbeeld zijn er drie type gebeurtenissen: het begin van een element, het einde van een element en de inhoud van het element. De parser genereert die gebeurtenissen en geeft de bijhorende informatie (bv. de naam van het element, de attributen van een element, tekst, ...) mee.

De lijst van mogelijke gebeurtenissen is uitgebreider als er rekening gehouden wordt met namenruimtes, witte ruimte tussen elementen (*white space*: spaties, tabs, ...), maar het principe blijft hetzelfde. Bij het lezen van een XML-document genereert de parser een reeks gebeurtenissen. Elke gebeurtenis correspondeert met een bepaald aspect van een XML-document en bevat de inhoudelijke informatie. De applicatie vangt deze gebeurtenissen op en verwerkt de informatie.

Voor- en nadelen

Parsers die een XML-document beschouwen als een reeks van gebeurtenissen moeten maar een beperkte hoeveelheid informatie bijhouden. Ze moeten eventueel (een deel van) de inhoud van de DTD of het schema kennen en begrijpen, en ze moeten stacks bijhouden voor de elementnamen en namenruimtedeclaraties. Het is echter niet nodig dat ze de inhoud van het volledige XML-document bijhouden. Als ze een gebeurtenis gegenereerd hebben, mogen ze de doorgegeven informatie uit het geheugen verwijderen indien ze die niet meer nodig hebben voor de verwerking van de rest van het document. Dit principe zorgt ervoor dat de hoeveelheid geheugen die de parser nodig heeft minimaal is en dat de parser performant is.

Een nadeel is dat de applicatie die de gebeurtenissen afhandelt complexer is. Deze applicatie moet een context bijhouden en de informatie die het krijgt via de gebeurtenissen naar de juiste plaats leiden. Bovendien moet de applicatie ermee rekening houden dat tijdens het lezen van het document een fatale fout kan optreden. Acties die reeds uitgevoerd werden voordat de fout gebeurde, moeten mogelijks ongedaan gemaakt worden (*rollback*).

Een ander nadeel is dat de applicatie tijdens het afhandelen van een gebeurtenis, geen gebruik kan maken van informatie die later in het XML-document voorkomt. Dit maakt het quasi onmogelijk om gebruik te maken van bv. XPath.

Event-based parsers kunnen gebruikt worden om XML-documenten weg te schrijven naar andere databronnen. Ook indien de informatie van het XML-document door verschillende filters of *pipelines* moet passeren is dit type parser aangewezen. Zelfs parsers die XML-documenten omvormen tot een boomstructuur (zie §6.1.3) maken gebruik van een *event-based parser* om deze boom te construeren.

6.1.3 XML: een boomstructuur

Principe

Een welgevormd XML-document vormt een boomstructuur. Deze boom bestaat o.a. uit het basiselement, tekst, kindelementen, ... Voor deze boomstructuur bestaan verschillende modellen. XPath (zie hoofdstuk 5) bijvoorbeeld gebruikt een boomstructuur die een beetje anders is dan de boomstructuur bepaald door DOM (zie §6.3).

Om XML-documenten als boom te manipuleren in een applicatie zullen ontwikkelaars gebruik maken van een API en een onderliggende bibliotheek die een objectmodel gebruiken om het XML-document voor te stellen. Deze API's stellen het volledige XML-document ter beschikking van de applicatie op voorwaarde dat het parsen gelukt is.

Voor- en nadelen

Het voordeel van de bovenstaande manier van werken is dat applicaties de volledige XML-boom ter beschikking hebben. Het parsen, het opbouwen van de XML-boom en de eventuele rollback bij een

fatale fout worden uitgevoerd door de gebruikte API (en bijhorende implementatie). De applicatie kan door de boom lopen om informatie te zoeken, kan de boom veranderen of uitbreiden. Het gebruik van bv. XPath is dus mogelijk.

Een nadeel van deze manier van werken is dat die vaak een grote hoeveelheid geheugen nodig heeft. Ook het doorzoeken van de boomstructuur kan het aantal uit te voeren opdrachten (en dus processortijd) flink vergroten. Het is dus moeilijk om hiermee grote schaalbare applicaties te maken. Voor een beperkt aantal kleine documenten is dit principe echter wel bruikbaar.

6.2 Simple API for XML (SAX)

De *Simple API for XML (SAX)* is een API om XML-documenten te lezen gebruik makend van gebeurtenissen. Een hele reeks bestaande parsers zoals Xerces, Crimson, ... implementeren de SAX API. SAX was oorspronkelijk een Java API, maar is ondertussen ook beschikbaar voor vele andere objectgeoriënteerde talen (C++, Python, Perl, Eiffel, ...). In tegenstelling tot de andere XML-technologieën die in deze cursus besproken worden, is SAX geen standaard van het *World Wide Web Consortium*. In dit hoofdstuk bespreken we de SAX2 specificatie in Java. Deze specificatie ondersteunt ook namenruimtes.

Een XML-parser die de SAX API implementeert leest één keer het XML-document van voor naar achteren en stuurt tijdens het lezen informatie naar jouw programma. Elke keer dat de parser een begin-tag, een eind-tag, tekst, ... tegenkomt, vertelt hij dat aan jouw programma. Het XML-document wordt in verschillende stukjes aan je programma gegeven. Je kan deze stukjes informatie bewaren in je programma of je kan ze onmiddellijk verwerken.

SAX bestaat vooral uit een verzameling interfaces in het pakket *org.xml.sax*. Verder is er nog een specifiek Java-pakket *javax.xml.parsers* dat gemeenschappelijke, productonafhankelijke interfaces biedt voor verschillende SAX- en DOM-parsers.

6.2.1 XMLReader

De interface *XMLReader* stelt de XML-parser voor. Ze voorziet methodes om een XML-document te parsen en om het parsen te configureren (bv. moet de parser valideren?).

Om een instantie van *XMLReader* te creëren, maken we gebruik van het Java-pakket *javax.xml.parsers*.

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;

try {
  SAXParserFactory factory = SAXParserFactory.newInstance();
  SAXParser parser = factory.newSAXParser();
  XMLReader reader = parser.getXMLReader();
  ...
```

```
} catch (ParserConfigurationException e) {
  ...
} catch (SAXException e) {
  ...
}
```

De klasse *SAXParserFactory* is een abstracte klasse die het mogelijk maakt om van SAX-implementatie te veranderen zonder dat de broncode van jouw programma hoeft te veranderen. Met een *SAXParserFactory* kan je *SAXParser*-objecten aanmaken.

Een instantie van een *SAXParserFactory* verkrijg je met de statische methode *newInstance*. De concrete klasse van deze instantie wordt bepaald door jouw Java-omgeving. Hiervoor bekijkt deze methode de volgende zaken: systeemeigenschappen, configuratiebestanden, meta-informatie in jar-bestanden of de standaardwaarde van jouw Java-platform.

Om een *SAXParser* aan te maken, gebruik je de methode *newSAXParser*. Dit is een abstracte methode van de klasse *SAXParserFactory*. Bij het uitvoeren van deze methode kunnen twee types fouten optreden: er kan geen parser aangemaakt worden die voldoet aan de gevraagde configuratie (*ParserConfigurationException*) of algemene SAX-fouten (*SAXException*). De klasse *SAXException* is de algemene bovenklasse voor fouten die optreden bij het parsen van een XML-document.

SAXParser is een abstracte klasse, die een implementatie van de interface *XMLReader* bevat (*wrapper*). Enkel de interface *XMLReader* en de klasse *SAXException* maken deel uit van de SAX2-specificatie. De andere klassen zijn typisch voor het Java-platform.

Nu kunnen we een XML-document parsen met de verkregen *XMLReader*. Hiervoor gebruiken we de methode *parse*. Deze methode kan een *SAXException* en een *IOException* gooien.

```
try {
  SAXParserFactory factory = SAXParserFactory.newInstance();
  SAXParser parser = factory.newSAXParser();
  XMLReader reader = parser.getXMLReader();
  String uri = ...;
  reader.parse(uri);
  ...
} catch (ParserConfigurationException e) {
  ...
} catch (SAXException e) {
  ...
} catch (IOException e) {
  ...
}
```

De string *uri* stelt een URI voor. Dit kan zowel een bestandsnaam, een URL, ... zijn. De klasse *XMLReader* heeft ook een methode *parse* met als argument een *org.xml.sax.InputSource*. Op deze manier kan er o.a. XML gelezen worden van een *byte stream* of een *character stream*.

De methode *parse* is een synchrone methode. Dit betekent dat de methode pas stopt als het volledige parsen voltooid is. Bovendien gaat het parsen pas verder als een gebeurtenis afgehandeld is.

Als de methode *parse* succesvol is, dan is de teruggeefwaarde *void*. Om informatie te verkrijgen van de parser moet je een *ContentHandler* instellen.

6.2.2 ContentHandler

De interface *ContentHandler* behoort tot het pakket *org.xml.sax*. Om informatie te verkrijgen van de *XMLReader*, implementeer je de interface *ContentHandler* in een eigen klasse. Daarna configureer je de *XMLReader* met een instantie van deze klasse. Tijdens het doorlopen van het XML-document zal de *XMLReader* de methodes van de interface *ContentHandler* oproepen om te vertellen wat er in het XML-document staat. Een overzicht van deze methodes:

```
public void setDocumentLocator(Locator locator);
public void startDocument() throws SAXException;
public void endDocument() throws SAXException;
public void startPrefixMapping(String prefix, String uri)
  throws SAXException ;
public void endPrefixMapping(String prefix)
  throws SAXException;
public void startElement(String namespaceURI, String localName,
  String qualifiedName, Attributes atts) throws SAXException;
public void endElement(String namespaceURI, String localName,
  String qualifiedName) throws SAXException;
public void characters(char[] text, int start, int length)
  throws SAXException;
public void ignorableWhitespace(char[] text, int start,
  int length) throws SAXException;
public void processingInstruction(String target, String data)
  throws SAXException;
public void skippedEntity(String name) throws SAXException;
```

De meeste methodes spreken voor zich. De methodes *startPrefixMapping* en *endPrefixMapping* geven het begin en het einde van een afkorting van een namenruimte aan. Voor de standaardnamenruimte wordt de lege string gebruikt.

De waarde van de argumenten van de methodes *startElement* en *endElement* zijn afhankelijk van een aantal instellingen bv. kent de parser namenruimtes. Het argument *namespaceURI* stelt de URI van de namenruimte voor of de lege string als er geen namenruimte is. Het argument *localName* is de naam van het element in de namenruimte of de lege string als er geen namenruimte is. Het argument *qualifiedName* is de combinatie van de afkorting van de namenruimte (als er één is) en de naam van het element.

Merk op dat de *char*-tabel in de methodes *characters* en *ignorableWhitespace* de inhoud van het volledige XML-document kan zijn. Het tekstblok waarop de gebeurtenis van toepassing is wordt dan bepaald door de argumenten *start* en *length*. Of dit het geval is, hangt af van de gebruikte parser.

Met de methode *setDocumentLocator* kan een SAX-parser een *org.xml.sax.Locator* object ter beschikking stellen aan de applicatie. SAX-parsers hoeven dit niet te doen, maar als ze dit ondersteunen wordt deze methode opgeroepen voor één van de andere methodes van de interface *ContentHandler* wordt opgeroepen. Het *Locator*-object kan dan gebruikt worden tussen de gebeurtenissen *startDocument* en *endDocument*. Dit object geeft de plaats van het einde van de gebeurtenis op, gekenmerkt door een lijn- en kolomnummer.

Als je een eigen *ContentHandler* geïmplementeerd hebt, dan kan je die op de volgende wijze toekennen aan de *XMLReader*.

```
try {
  SAXParserFactory factory = SAXParserFactory.newInstance();
```

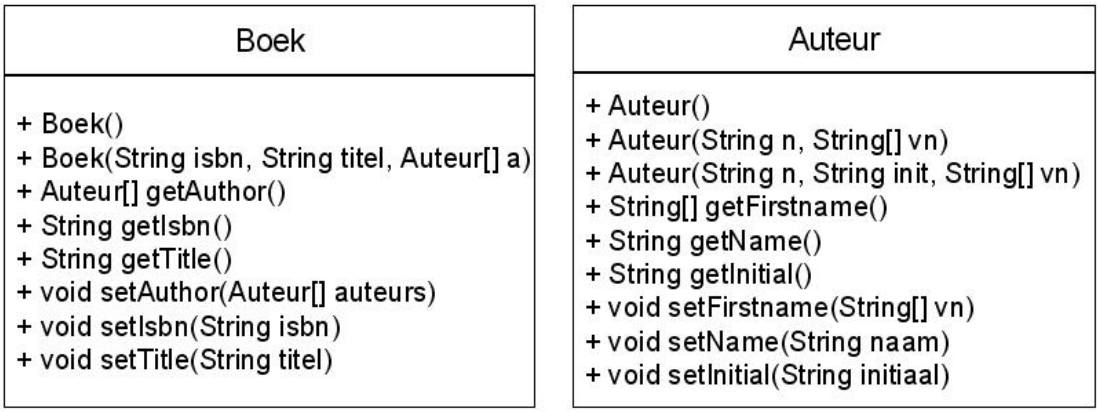
```
  SAXParser parser = factory.newSAXParser();
  XMLReader reader = parser.getXMLReader();
  ContentHandler handler = ...;
  reader.setContentHandler(handler);
  String uri = ...;
  reader.parse(uri);
  ...
} catch (ParserConfigurationException e) {
  ...
} catch (SAXException e) {
  ...
} catch (IOException e) {
  ...
}
```

Voorbeeld

In het volgend voorbeeld maken we een *java.util.ArrayList*-object van *Boek*-objecten op basis van een XML-document dat voldoet aan de volgende DTD.

```
<!ELEMENT books (book*)>
<!ELEMENT book (isbn, title, author+)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name, initial?, firstname+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT initial (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
```

De klassen *Boek* en *Auteur* hebben de volgende structuur.



Vervolgens schrijven we een implementatie van de interface *ContentHandler*. De volgende methodes van deze interface gebruiken we niet en geven we een lege implementatie.

```
public void setDocumentLocator(Locator locator) {  }
public void startDocument() throws SAXException {  }
public void endDocument() throws SAXException { }
public void startPrefixMapping(String prefix, String uri)
  throws SAXException {  }
public void endPrefixMapping(String prefix)
  throws SAXException {  }
public void ignorableWhitespace(char[] text, int start,
  int length) throws SAXException {  }
public void processingInstruction(String target,
  String data) throws SAXException {  }
public void skippedEntity(String name)
  throws SAXException {  }
```

Enkel de gebeurtenissen naar aanleiding van het begin van een element, het einde van een element en de inhoud van een element worden verwerkt.

Als de parser een opentag tegenkomt, dan houden we de naam van het geopende element bij in de variabele *naamLaatsteElement*. Als het element bovendien de naam *book* of *author* heeft, dan maken we respectievelijk een *Boek*- of *Auteur*-object aan. Deze objecten worden voorgesteld door de variabelen *boek* en *auteur*. Het object *boeken* wordt in de constructor geïnitialiseerd als een lege *ArrayList*. Merk op dat de klasse *Catalogus.Util* constanten bevat die de namen van de elementen in het XML-document zijn. De klasse die de interface *ContentHandler* implementeert is *SAXCatalogusHandler*.

```
private ArrayList boeken, auteurLijst, voornamenLijst;
private Auteur auteur;
private Boek boek;
protected String naamLaatsteElement;

public SAXCatalogusHandler() {
  boeken = new ArrayList();
  naamLaatsteElement = null;
  auteurLijst = null;
  voornamenLijst = null;
  auteur = null;
  boek = null;
}

public void startElement(String namespaceURI, String localName,
  String qualifiedName, Attributes atts) throws SAXException {
    naamLaatsteElement = qualifiedName;
    if (qualifiedName.equals(CatalogusUtil.BOOK))
      startBoekElement();
    else if (qualifiedName.equals(CatalogusUtil.AUTEUR))
      startAuteurElement();
}

protected void startBoekElement() {
  boek = new Boek();
  auteurLijst = new ArrayList();
}

protected void startAuteurElement() {
  auteur = new Auteur();
  voornamenLijst = new ArrayList();
}
```

Als de parser tekstinhoud van een element tegenkomt, dan kennen we dat toe aan het juiste object. We vermelden enkel de hulpmethode *setIsbn*, maar de andere methodes zijn analoog.

```
public void characters(char[] text, int start, int length)
  throws SAXException {
    if (naamLaatsteElement.equals(CatalogusUtil.ISBN))
      setIsbnTekst(text, start, length);
    else if (naamLaatsteElement.equals(CatalogusUtil.TITLE))
      setTitleTekst(text, start, length);
    else if (naamLaatsteElement.equals(CatalogusUtil.NAME))
      setNameTekst(text, start, length);
    else if (naamLaatsteElement.equals(CatalogusUtil.INITIAL))
      setInitialTekst(text, start, length);
    else if (naamLaatsteElement.equals(CatalogusUtil.FIRSTNAME))
      setFirstnameTekst(text, start, length);
}

protected void setIsbnTekst(char[] text, int start,
  int length) {
    boek.setIsbn(new String(text,start,length));
}
```

Bij het afsluiten van elementen hoeven we enkel rekening te houden met de element *book* en *author*. De methodes *convertAuteur* en *convertVoornaam* dienen om *ArrayList*-objecten om te zetten naar tabellen van *Auteur*- of *String*-objecten.

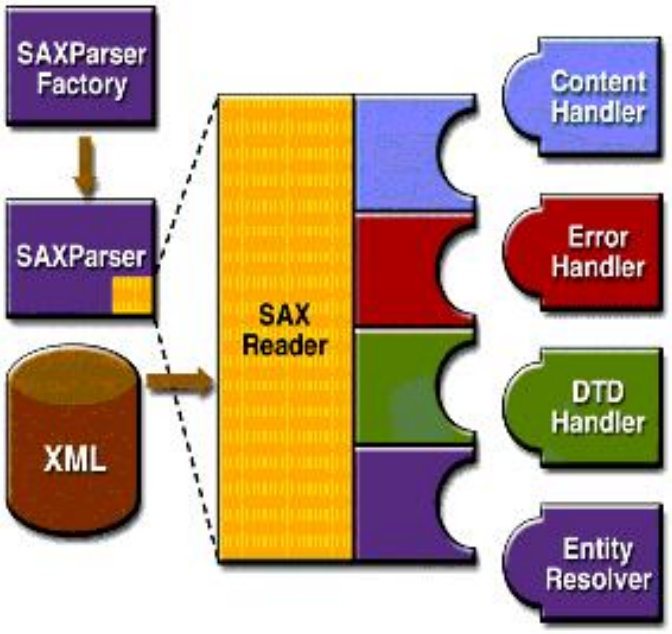
```
public void endElement(String namespaceURI, String localName,
  String qualifiedName) throws SAXException {
    if (qualifiedName.equals(CatalogusUtil.BOOK))
      endBoekElement();
    else if (qualifiedName.equals(CatalogusUtil.AUTEUR))
      endAuteurElement();
}
protected void endBoekElement() {
    boek.setAuthor(CatalogusUtil.convertAuteur(auteurLijst));
    boeken.add(boek);
}

protected void endAuteurElement() {
    auteur.setFirstname(
      CatalogusUtil.convertVoornaam(voornamenLijst));
    auteurLijst.add(auteur);
}
```

In dit SAX-programma bouwden we zelf stukje bij stukje de datastructuur op (in dit geval een *ArrayList* van *Boek*-objecten). Deze datastructuur wordt opgebouwd tijdens het parsen van het programma. Indien er iets misloopt, moet er beslist worden wat er met de reeds verwerkte informatie gebeurt. Dat is vooral belangrijk als we bijvoorbeeld informatie zouden weggeschreven hebben in een databank en we niet willen dat de databank onvolledige data bevat. Het zelf opbouwen van de datastructuur en het voorzien van een eventuele rollback vormt het complexe stuk van een SAX-programma.

Als de opgebouwde datastructuur de volledige informatie van het XML-document bevat, dan is het zinvoller om DOM (zie §6.3) te gebruiken. DOM voorziet namelijk al in onmiddellijk bruikbare datastructuur. Indien je echter slechts een deel van de informatie van het XML-document nodig hebt, dan is SAX wel zinvol.

6.2.3 DefaultHandler

Naast de gebeurtenissen (*events*) die verwerkt worden door een *ContentHandler*, genereert de *XMLReader* nog andere gebeurtenissen. Die houden verband met het verwerken van fouten, DTD's en externe entiteiten. Hiervoor kunnen nog drie andere objecten geregistreerd worden bij een *XMLReader*: een *ErrorHandler*, een *DTDHandler* en een *EntityResolver*. Dit zijn opnieuw drie interfaces waarvoor je een eigen implementatie kan voorzien. In een Java-omgeving kan je dit visueel als volgt voorstellen.

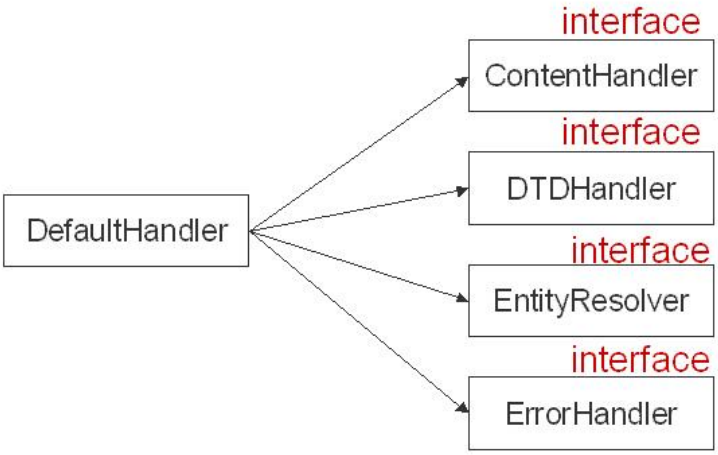


Figuur 6.1: Principe SAX (bron: http://java.sun.com)

In §6.2.2 hebben we een heleboel methodes van de interface *ContentHandler* een lege implementatie gegeven. De SAX2-specificatie biedt echter ook de mogelijkheid om alleen die methodes te implementeren die jouw applicatie nodig heeft. Hiervoor is een standaardimplementatie van de vier interfaces *ContentHandler*, *ErrorHandler*, *DTDHandler* en *EntityResolver* beschikbaar. De klasse *DefaultHandler* voorziet in een standaardimplementatie van deze vier interfaces. Alle methodes werden leeg geïmplementeerd. Dit mechanisme is vergelijkbaar met de *Adapter*-klassen in Java Swing. Je kan dus een afgeleide klasse van *DefaultHandler* maken en enkel die methodes overschrijven die je nodig hebt in jouw applicatie. De klasse *DefaultHandler* maakt deel uit van het pakket *org.xml.sax.helpers*.

Als je gebruik maakt van een *DefaultHandler* ziet je programma er zo uit. In dit geval hoef je de *XMLReader* niet expliciet te gebruiken. De gebruikte *parse*-methode in de onderstaande code is slechts één van de mogelijkheden. Het eerste argument kan i.p.v. een URI-string ook een stream of een *InputSource* zijn. Het tweede argument is wel steeds een *DefaultHandler*.

```
try {
```




```
  SAXParserFactory factory = SAXParserFactory.newInstance();
  SAXParser parser = factory.newSAXParser();
  String bestand = ...;
  DefaultHandler handler = ...;
  parser.parse(bestand, handler);
  ...
} catch (ParserConfigurationException e) {
  ...
} catch (IllegalArgumentException e) {
  ...
} catch (SAXException e) {
  ...
} catch (IOException e) {
  ...
}
```

Als we in §6.2.2 een *DefaultHandler* gebruiken dan is de structuur van deze *handler* als volgt.

```
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;

public class CatalogusDefaultHandler extends DefaultHandler {
  ...

  public void startElement(String namespaceURI, String localName,
    String qualifiedName, Attributes atts) throws SAXException
    { ... }

  public void endElement(String namespaceURI, String localName,
    String qualifiedName) throws SAXException { ... }

  public void characters(char[] text, int start, int length)
    throws SAXException { ... }
}
```

6.2.4 Configuratie van de SAX-parser

Een validerende parser

Een XML-parser zal steeds de welgevormdheid van een XML-document controleren, maar niet alle parsers kunnen het XML-document ook valideren t.o.v. een DTD. Om dit te realiseren moeten we de *SAXParserFactory* zo configureren dat hij een parser aanmaakt die ook kan valideren. Hiervoor gebruiken we de methode *setValidating*.

```
try {
  SAXParserFactory factory = SAXParserFactory.newInstance();
  factory.setValidating(true);
  SAXParser parser = factory.newSAXParser();
  ...
} catch (ParserConfigurationException e) {
  ...
} catch (SAXException e) {
  ...
}
```

Een parser die valideert t.o.v. een XML Schema

Opdat een parser zou kunnen valideren t.o.v. een XML Schema moet de parser namenruimtes kennen. Dit betekent dat de *SAXParserFactory* zo ingesteld moet zijn dat hij parsers aanmaakt die namenruimtes kennen.

```
try {
  SAXParserFactory factory = SAXParserFactory.newInstance();
  factory.setNamespaceAware(true);
  factory.setValidating(true);
  SAXParser parser = factory.newSAXParser();
  ...
} catch (ParserConfigurationException e) {
  ...
} catch (SAXException e) {
  ...
}
```

Aan de aangemaakte parser moet nu nog gemeld worden dat hij niet moet valideren t.o.v. een DTD, maar wel t.o.v. een XML Schema. Om dit te realiseren wordt gebruik gemaakt van de volgende methode van *SAXParser*.

```
public abstract void setProperty(String name,
  Object value) throws SAXNotRecognizedException,
  SAXNotSupportedException
```

Met deze methode kunnen eigenschappen van de onderliggende *XMLReader* ingesteld worden. Eigenlijk roept ze een analoge methode op van *XMLReader*. Een eigenschap bestaat steeds uit een naam/waarde-paar. Waarbij de naam een string is en de waarde een object.

Om te kunnen valideren tegenover een schema moeten de volgende eigenschappen ingesteld worden. Eerst wordt er opgegeven welke schemataal er gebruikt wordt, vervolgens wordt het schema-object doorgegeven.

```
try {
  String schema = ...;
  SAXParser parser = ...;
  parser.setProperty(
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
    "http://www.w3.org/2001/XMLSchema");
  parser.setProperty(
    "http://java.sun.com/xml/jaxp/properties/schemaSource",
    new File(schema));
  ...
} catch (SAXException e) {
  ...
}
```

Vanaf JDK 1.5 bevat het Javaplatform het pakket *javax.xml.validation* om XML-documenten te valideren. Hier wordt het valideren van XML-document en het parsen van een XML-document ontkoppeld. Dit maakt het o.a. mogelijk om andere schemadocumenten te ondersteunen dan DTD's en XML Schema's.

6.3 Document Object Model (DOM)

Het Document Object Model (DOM) definieert een API om XML-documenten te manipuleren als een boomstructuur. DOM is een *W3C Recommendation* die beschrijft hoe hiërarchische documenten in het geheugen bewaard kunnen worden. Deze beschrijving staat los van een specifieke programmeertaal. In de voorbeelden zullen we de programmeertaal Java gebruiken. In dit hoofdstuk bespreken we enkel die aspecten van *DOM Level 2* die betrekking hebben op het verwerken van XML-documenten (DOM Level 2 Core module). DOM voorziet o.a. ook modellen voor HTML-documenten.

DOM bestaat louter uit een verzameling interfaces die onafhankelijk van programmeertaal en besturingssysteem beschreven worden in de *Interface Description Language (IDL)*. Deze interfaces worden dan vertaald naar een specifieke programmeer- of scriptingtaal.

Er zijn twee manieren om toegang te krijgen tot de documentboom: gebruik makend van de algemene interface *Node* of gebruik makend van specifieke interfaces voor elk knooptype. De interface *Node* beschrijft een minimale set van attributen en methodes om de documentboom te manipuleren. Daarnaast zijn er specifieke interfaces die elementen, attributen, ... voorstellen. Zij zijn allen afgeleid van de algemene interface *Node*.

In tegenstelling tot XPath zijn ook *DocumentType*'s, *CDATA*-stukken en entiteiten knopen in de DOM-boom.

6.3.1 Node en algemene interfaces

De *Node*-interface is de basisinterface van alle specifieke interfaces. Hiermee kan je informatie verkrijgen van een knoop zonder zijn exacte type te kennen. De volgende tabellen bevatten een overzicht van alle eigenschappen/attributen van de *Node*-interface en van zijn methodes.

Type	Naam	Beschrijving
DOMString	nodeName	Naam van de knoop
DOMString	nodeValue	Waarde van de knoop (*null* voor document, element, ...; wel een waarde voor attributen, tekst, commentaar, ...)
unsigned short	nodeType	Type knoop, gekenmerkt door constanten
Node	parentNode	Ouderknoop, anders *null* (bv. bij attributen, document)
NodeList	childNodes	Kinderen van de knoop
Node	firstChild	Eerste kind
Node	lastChild	Laatste kind
Node	previousSibling	Vorige buur
Node	nextSibling	Volgende buur
NamedNodeMap	attributes	Lijst van attributen bij een element, anders *null*
Document	ownerDocument	Huidige XML-document
DOMString	namespaceURI	URI van de namenruimte van de knoop
DOMString	prefix	Afkorting van de namenruimte van de knoop
DOMString	localName	Naam van de knoop in de namenruimte

Tabel 6.1: Attributen van de *Node*-interface

Merk op dat alle eigenschappen behalve *nodeValue* en *prefixreadonly* zijn.

Resultaat	Naam	Argument	Beschrijving
Node	insertBefore	Node newChild, Node refChild	newChild invoegen voor refChild
Node	replaceChild	Node newChild, Node oldChild	oldChild vervangen door newChild; resultaat is oldChild
Node	removeChild	Node	Knoop verwijderen
Node	appendChild	Node	Knoop toevoegen op het einde van de lijst van kinderen
boolean	hasChildNodes		Kinderen?
Node	cloneNode	boolean	Kopie knoop met (*true*) of zonder (*false*) nakomelingen
	normalize		Tekstknopen samenvoegen, lege tekstknopen verwijderen
boolean	isSupported	DOMString feature, DOMString version	Gaat na of de implementatie een bepaald kenmerk ondersteunt
boolean	hasAttributes		Zijn er attributen?

Tabel 6.2: Methodes van de *Node*-interface

In de bovenstaande tabellen werden naast *Node* nog de DOM-types *DOMString* (een UTF-16 string), *NodeList* en *NamedNodeMap* gebruikt.

De interface *NodeList* stelt een geordende lijst knopen voor. Deze interface heeft één eigenschap *length* en één methode

```
Node item(Long index)
```

Je kan deze interface dus gebruiken zoals een tabel (*array*).

Voor niet-geordende verzamelingen (bv. de attributen van een element) wordt de interface *NamedNodeMap* gebruikt. Deze interface heeft ook één eigenschap *length*. De volgende tabel beschrijft zijn methodes.

Resultaat	**Naam**	**Argument**	**Beschrijving**
Node	getNamedItem	DOMString	Haal de knoop met opgegeven naam
Node	setNamedItem	Node	Bewaar de knoop onder zijn naam
Node	removeNamedItem	DOMString	Verwijder de knoop met de opgegeven naam
Node	item	unsigned long index	Haal het *index*de item in de lijst
Node	getNamedItemNS	DOMString namespaceURI, DOMString localName	Haal de knoop gekenmerkt door de namenruimte en zijn lokale naam
Node	setNamedItemNS	Node	Bewaar de knoop gekenmerkt door de namenruimte en zijn lokale naam
Node	removeNamedItemNS	DOMString namespaceURI, DOMString localName	Verwijder de knoop gekenmerkt door de namenruimte en zijn lokale naam

Tabel 6.3: Methodes van de *NamedNodeMap*-interface

6.3.2 Specifieke interfaces

De specifieke interfaces, afgeleid van de interface *Node*, zijn de volgende:

- *DocumentType*: verwijzing naar DTD of inline DTD
- *ProcessingInstruction*: verwerkingsinstructie
- *Notation*
- *Entity*: entiteit
- *Document*: XML-document

- *DocumentFragment*: deel van een XML-document
- *Element*: element
- *Attr*: attribuut
- *CharacterData*: tekstblok in een XML-document: commentaar, tekst of CDATA-stuk
- *Comment*: afgeleide interface van *CharacterData*, commentaar, beschikbaarheid afhankelijk van de implementatie
- *EntityReference*: verwijzing naar een entiteit (*&...;*); meestal worden ze vervangen door hun waarde
- *Text*: afgeleide interface van *CharacterData*, tekstknoop
- *CDATASection*: afgeleide interface van *Text*, CDATA-stuk

Enkel de interfaces *Document* en *Element* bespreken we meer in detail.

Document

Een DOM-parser leest het volledige XML-document en maakt daarbij één *Document*-knoop aan die het XML-document voorstelt. Deze interface geeft o.a. toegang tot DTD-informatie en tot het basiselement van het XML-document.

Bovendien biedt deze interface ook een aantal methodes om nieuwe knopen aan te maken. Knopen gecreëerd door een *Document* mogen enkel aan de structuurboom horende bij dit document toegevoegd worden. Enkel met de methode *importNode* kunnen knopen van andere documenten gekopieerd worden. Merk op dat aangemaakte knopen nog geen deel uit maken van de boomstructuur. Daarvoor moeten ze toegevoegd worden aan een knoop.

Tot slot biedt deze interface methodes om elementknopen te selecteren. De volgende tabellen bevatten een overzicht van de eigenschappen en methodes van de interface *Document*. Merk op dat alle eigenschappen *readonly* zijn.

Type	**Naam**	**Beschrijving**
DocumentType	doctype	Verwijzing naar de externe of interne DTD
DOMImplementation	implementation	Implementatie van de DOM-interfaces
Element	documentElement	Basiselement van het XML-document

Tabel 6.4: Attributen van de *Document*-interface

Element

Elementen geven toegang tot de meeste andere items van een XML-document: kindelementen, attributen, tekst, ... De interface *Element* heeft maar één (*readonly*) eigenschap *tagName*, die de naam van het element voorstelt. De tabel bevat een overzicht van de methodes.

6.3.3 DOM en Java

Het pakket *org.w3c.dom* is de vertaling van de DOM-interfaces (DOM level 2 Core API) naar Java. Dit pakket maakt deel uit van het standaard Java-platform sinds JDK 1.4. Bij de omzetting naar Java van attributen van een interface worden de woorden *get*, respectievelijk *set*, voor de naam van het attribuut toegevoegd. De eerste letter van de attribuutnaam wordt een hoofdletter. De namen van de methodes blijven dezelfde.

Andere pakketten die we nodig hebben om Javaprogramma's volgens het DOM-principe te schrijven zijn: *javax.xml.parsers* en *org.xml.sax*. Het pakket *javax.xml.parsers* wordt gebruikt om een XML-parser aan te maken. Van het pakket *org.xml.sax* gebruiken we de klassen die dienen voor foutafhandeling.

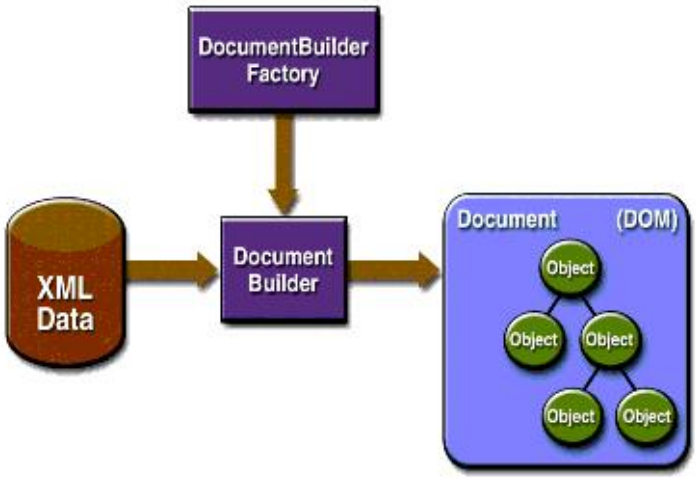
Een XML-document inlezen

In het geval van DOM wordt de XML-parser een *DocumentBuilder* genoemd. Deze parser zal op basis van een XML-document een *Document*-object aanmaken. De interface *Document* is de Java-versie van de interface *Document* besproken in §6.3.2.

Ook hier wordt een *factory* gebruikt om de XML-parser (*DocumentBuilder*) aan te maken. Welke parser je krijgt hangt af van het gebruikte Javaplatform, ingestelde systeemeigenschappen, commandolijnopties, ...

Een *Document*-object aanmaken op basis van een XML-bestand gaat als volgt.

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import java.io.File;
import java.io.IOException;

import org.w3c.dom.Document;

...

    try {
        String bestand = ...;
```



Figuur 6.2: Principe DOM (bron: http://java.sun.com)

```
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(new File(bestand));
    } catch (ParserConfigurationException e) {
        ...
    } catch (SAXParseException e) {
        ...
    } catch (SAXException e) {
        ...
    } catch (IOException e) {
        ...
    }
```

Bij de *DocumentBuilder* kan ook een *ErrorHandler* (zie §6.2.3) geregistreerd worden. Aan de bovenstaande code wordt dan het volgende toegevoegd (voor het parsen).

```
ErrorHandler foutAfhandeling = ...;
builder.setErrorHandler(foutAfhandeling);
```

Ook hier is het mogelijk het XML-document te valideren, al dan niet tegenover een XML-schema, tijdens het inlezen.

```
try {
  String bestand = ...; // naam XML-bestand
  String schema = ...; // naam XML-schemabestand
  DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
  factory.setValidating(true);
  factory.setNamespaceAware(true);
  factory.setAttribute(
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
    "http://www.w3.org/2001/XMLSchema");
  factory.setAttribute(
    "http://java.sun.com/xml/jaxp/properties/schemaSource",
    new File(schema));
  DocumentBuilder builder = factory.newDocumentBuilder();
  ErrorHandler foutAfhandeling = ...
```

```
  builder.setErrorHandler(foutAfhandeling);
  Document document = builder.parse(new File(bestand));
} catch (...) {
  ....
}
```

Zoeken in de XML-boom

In de vorige sectie beschreven we hoe je een *Document*-object kan maken op basis van een XML-bestand. Met behulp van o.a. (de Javavertaling van) de attributen en methodes van de interfaces *Node* (§6.3.1), *Document* (§6.3.2) en *Element* (§6.3.2) kunnen we nu informatie opvragen aan het *Document*-object.

In het voorbeeld uit §6.2.2 werd een *ArrayList* van *Boek*-elementen opgebouwd tijdens het parsen van het XML-document. Om dit te verwezenlijken schreven we een *ContentHandler*. In het geval van DOM hebben we deze informatie beschikbaar in de vorm van een *Document*-object. Dit object weerspiegelt de structuur van het XML-bestand.

Indien nodig kan deze informatie ook naar *Boek*- en *Auteur*-objecten omgezet worden. De volgende stukjes code toont hoe dit kan en illustreert meteen een aantal attributen en methodes van de interfaces *Node*, *NodeList*, *Document* en *Element*.

De methode *getBoekenLijst* maakt een *ArrayList* van *Boek*-objecten aan op basis van het *Document*-object verkregen bij het parsen. De constante *CatalogusUtil.BOOK* stelt de naam van het boekelement voor.

```
private ArrayList getBoekenLijst(Document document) {
    ArrayList boeken = new ArrayList();
    NodeList boekKnopen =
        document.getElementsByTagName(CatalogusUtil.BOOK);
    for (int i = 0; i < boekKnopen.getLength(); i++) {
        Node boek = boekKnopen.item(i);
        boeken.add(maakBoekObject(boek));
    }
    return boeken;
}
```

De methode *maakBoekObject* maakt op basis van de *Element*-knoop verkregen van het *Document*-object een *Boek*-object. De klasse *CatalogusUtil* bevat verschillende constanten die de namen van de element voorstellen. Merk op dat we aan een *Element*-knoop (*kind* in het voorbeeld) niet onmiddellijk de waarde kunnen vragen. We moeten eerst de tekstknoop bepalen en daaraan zijn waarde vragen. De methode *maakAuteurObject* is analoog aan de methode *maakBoekObject*. De methode *convertAuteur* van *CatalogusUtil* zet een *ArrayList* om in een tabel van *Auteur*-objecten.

```
private Boek maakBoekObject(Node boekKnoop) {
    NodeList kinderen = boekKnoop.getChildNodes();
    String isbn = null, title = null;
    ArrayList auteurs = new ArrayList();
    for (int i = 0; i < kinderen.getLength(); i++) {
        Node kind = kinderen.item(i);
        if (kind.getNodeName().equals(CatalogusUtil.ISBN)) {
            isbn = kind.getFirstChild().getNodeValue();
        } else if
```

```
        (kind.getNodeName().equals(CatalogusUtil.TITLE)) {
            title = kind.getFirstChild().getNodeValue();
        } else if
        (kind.getNodeName().equals(CatalogusUtil.AUTEUR)) {
            auteurs.add(maakAuteurObject(kind));
        }
    }
    return new Boek(isbn, title,
        CatalogusUtil.convertAuteur(auteurs));
}
```

Een XML-document aanmaken of veranderen

De interface *DocumentBuilder* kan ook gebruikt worden om een leeg *Document*-object aan te maken.

```
DocumentBuilderFactory factory =
     DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.newDocument();
```

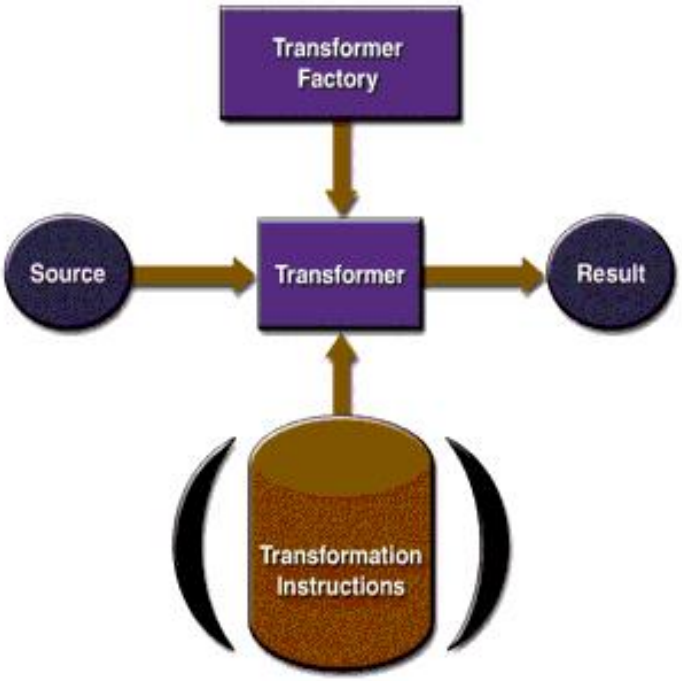
Om knopen toe te voegen aan een *Document*-object maken we gebruik van de interfaces *Node* en *Document*. In het volgend voorbeeld wordt een boekelement toegevoegd aan het basiselement op basis van een *Boek*-object.

```
Boek boek = ...
Document document = ...
Element boekKnoop = document.createElement(CatalogusUtil.BOOK);

Element isbnKnoop = document.createElement(CatalogusUtil.ISBN);
isbnKnoop.appendChild(document.createTextNode(boek.getIsbn()));
boekKnoop.appendChild(isbnKnoop);

Element titleKnoop = document.createElement(CatalogusUtil.TITLE);
titleKnoop.appendChild(document.createTextNode(boek.getTitle()));
boekKnoop.appendChild(titleKnoop);

Auteur[] auteurs = boek.getAuthor();
for (int i = 0; i < auteurs.length; i++) {
    boekKnoop.appendChild(maakAuteurKnoop(auteurs[i], document));
}

document.getDocumentElement().appendChild(
    maakBoekKnoop(boek, document));
```

Alternatieven: JDOM en dom4j

Het DOM-model is eigenlijk ontworpen als een model voor tekstdocumenten zoals artikels en boeken. In §1.2.1 noemden we dat beschrijvende XML-documenten. Bovendien ondersteunt de Java-implementatie van SAX en DOM het gebruik van XML-schema's.

Een nadeel van het feit dat de DOM-interfaces programmeertaalonafhankelijk zijn, is dat ze geen gebruik maken van object-georiënteerde technieken (niet alle programmeertalen zijn object-georiënteerd).



Figuur 6.3: Principe Transformaties (bron: http://java.sun.com)

Indien je een programma in Java schrijft waarin geen beschrijvende XML-bestanden gebruikt worden, enkel eenvoudige XML-datastructuren, en je geen XML-schema's nodig hebt, dan zijn er object-georiënteerde alternatieven: bv. JDOM en dom4j.

6.4 XML-transformaties in Java

Een ander aspect van XML in applicaties zijn transformaties, hierbij wordt het ene XML-formaat omgezet naar een ander. Een van de meest populaire manieren om XML-documenten te transformeren is het gebruik van XSLT (zie §4). In deze sectie wordt beschreven hoe je transformaties kan realiseren vanuit een Java-programma.

Een *Transformer*-object zal een bronobject omvormen tot een resultaatobject volgens een bepaalde transformatie. Om *Transformer*-objecten te verkrijgen wordt gebruik gemaakt van een *factory* namelijk *TransformerFactory*. Het bronobject kan gegenereerd worden op basis van een *XMLReader*, een (DOM) *Node* of een *InputStream*. Het resultaatobject kan een *ContentHandler*, *Node* of een *OutputStream* zijn.

Als het *Transformer*-object niet aangemaakt wordt op basis van transformatie-opdrachten dan kopieert het het bronobject naar het resultaatobject. Een voorbeeld hiervan is het wegschrijven van een *Document*-object naar een bestand.

Er zijn vier transformatiepakketten in de standaard Java API.

javax.xml.transform	Dit pakket bevat de basisklassen en interfaces om transformaties te kunnen uitvoeren in Java.
javax.xml.transform.dom	Dit pakket beschrijft de klassen die DOM-invoer en -uitvoer van een transformatie voorstellen.
javax.xml.transform.sax	Dit pakket beschrijft de klassen die invoer en uitvoer van een transformatie in SAX-formaat voorstellen, maw als een reeks van gebeurtenissen.
javax.xml.transform.stream	Dit pakket definieert klassen die invoer en uitvoer van een transformatie als *streams* voorstellen.

In het volgend voorbeeld wordt een *Document*-object weggeschreven naar een XML-bestand.

```
try {
    Document document = ...;
    String bestand = ...;
    OutputStream out = new FileOutputStream(bestand);
    TransformerFactory tFactory =
        TransformerFactory.newInstance();
    Transformer transformer = tFactory.newTransformer();
    DOMSource source = new DOMSource(document);
    StreamResult result = new StreamResult(out);
    transformer.transform(source, result);
} catch (...) {
    ...
}
```

6.5 JAXP

De *Java API for XML Processing (JAXP)* wordt gebruikt om XML-data te verwerken in Javaprogramma's. JAXP bevat de API's *Simple API for XML Parsing (SAX)* en *Document Object Model (DOM)*. Je kan de XML-gegevens dus beschouwen als een stroom van gebeurtenissen met SAX of je kan een object maken van deze gegevens met DOM. JAXP ondersteunt ook de XSLT-standaard in de XSLT API. Verder is er in JAXP ondersteuning voor namenruimtes.

JAXP is flexibel ontworpen. Je kan naar keuze een SAX-parser, een DOM-parser en een XSLT-processor inpluggen in je applicatie. Uiteraard kan je ook gebruik maken van die parsers die standaard meegeleverd worden met de Java-runtime.

6.6 StAX

Zowel het SAX- als DOM-programmeermodel hebben een aantal nadelen. Het SAX-programmeermodel is zeer efficiënt, maar is statusloos. Dit vereist dat de applicatie zelf statusinformatie opbouwt en bijhoudt. Bovendien stuurt de SAX-parser de applicatie aan. Hij genereert gebeurtenissen die de applicatie moet afhandelen. Het DOM-programmeermodel daarentegen biedt een volledig willekeurige toegang tot het XML-document ten koste van performantie. Om de nadelen van beide modellen weg te werken, is een nieuw model ontwikkeld dat *pull parsing* heet. Dit model probeert om de

goede kanten van het SAX- en DOM-principe te combineren, namelijk eenvoudige toegang tot het XML-document, performantie en efficiëntie.

Het principe van *pull parsing* is het *iterator*-model. Met behulp van de *iterator* kan de applicatie telkens het volgende stukje XML opvragen. Het is dus mogelijk om stukken te negeren, vroegtijdig te stoppen, meerdere documenten tegelijk te parsen, ... Waar bij SAX de parser de applicatie aanstuurt door het genereren van gebeurtenissen, is dat bij *pull parsing* omgekeerd. De applicatie stuurt de parser aan.

De Java-API die dit principe ondersteunt heet *StAX* en staat voor *Streaming API for XML*. Vanaf JDK 1.6 zal deze API deel uitmaken van de Java-omgeving.

6.7 XML Data Binding

Ontwikkelaars die de gegevens van XML-documenten willen gebruiken in hun programma, maar liever niet met expliciete XML-structuur willen werken (bv. via SAX of DOM) kunnen gebruik maken van *XML Data Binding*. Deze werkwijze legt een extra laag die het werken met gebeurtenissen of boomstructuren verbergt.

Met *XML Data Binding* worden XML-documenten omgezet naar objecten en omgekeerd. Dit kan gerealiseerd worden aan de hand van XML-schema's omdat die de structuur van XML-document vastleggen.

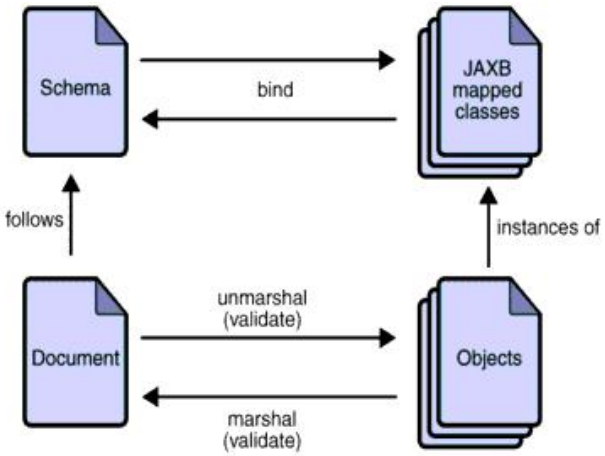
6.7.1 JAXB

Met de *Java Architecture for XML Binding (JAXB)* is het mogelijk om een verbinding te leggen tussen XML schema's en Java-voorstellingen (klassen, ...) van deze gegevens. Deze API biedt methodes om XML-documenten om te zetten naar Java-objecten en omgekeerd.

De volgende figuur toont hoe XML-gegevens gebonden worden met behulp van JAXB.

Het bindingsproces bestaat uit een aantal stappen.

- *Klassen genereren*: Op basis van het XML-schema genereert de *JAXB binding compiler* Java-klassen.
- *Klassen compileren*: De gegenereerde broncode wordt gecompileerd.
- Unmarshal: XML-documenten die voldoen aan het XML-schema worden omgezet naar Java-objecten. Merk op dat deze XML-documenten geen bestanden hoeven te zijn. Ook DOM-structuren, SAX-streams, ... zijn mogelijk.
- *Valideren (optioneel)*: Tijdens het *unmarshalling* kunnen de XML-documenten gevalideerd worden. Omgekeerd is het ook mogelijk om Java-objecten die aangemaakt of veranderd werden in de applicatie te valideren vooraleer ze terug omgezet worden naar XML-documenten.



Figuur 6.4: Principe JAXB (bron: http://java.sun.com)

- *Marshal*: De Java-objecten worden omgezet naar XML-documenten.

Resultaat	**Naam**	**Argument**	**Beschrijving**
Element	createElement	DOMString	Maak een elementknoop met de opgegeven naam
DocumentFragment	createDocumentFragment		Maak een deel van een XML-document
Text	createTextNode	DOMString	Maak een tekstknoop met de opgegeven inhoud
Comment	createComment	DOMString	Maak een commentaarknoop met de opgegeven inhoud
CDATASection	createCDATASection	DOMString	Maak een CDATA-stuk met de opgegeven inhoud
ProcessingInstruction	createProcessingInstruction	DOMString target, DOMString data	Maak een verwerkingsinstructie met de opgegeven inhoud
Attr	createAttribute	DOMString	Maak een attribuut met de opgegeven naam
EntityReference	createEntityReference	DOMString	Maak een verwijzing naar een entiteit met de opgegeven naam
NodeList	getElementsByTagName	DOMString	Bepaal de elementen met de opgegeven naam
Node	importNode	Node, boolean	Importeer een knoop (van een ander document); de *boolean* bepaalt of ook de nakomelingen gekopieerd moeten worden
Element	createElementNS	DOMString namespaceURI, DOMString qualifiedName	Maak een elementknoop met de opgegeven data
Attr	createAttributeNS	DOMString namespaceURI, DOMString qualifiedName	Maak een attribuutknoop met de opgegeven data
NodeList	getElementsByTagNameNS	DOMString namespaceURI, DOMString qualifiedName	Bepaal de elementen die voldoen aan de opgegeven data
Element	getElementById	DOMString	Bepaal een element a.d.h.v. de opgegeven ID

Tabel 6.5: Methodes van de *Document*-interface

Resultaat	**Naam**	**Argument**	**Beschrijving**
DOMString	getAttribute	DOMString	Bepaalt de waarde van het attribuut met de opgegeven naam
void	setAttribute	DOMString name, DOMString value	Stelt de waarde van het attribuut met de opgegeven naam in
void	removeAttribute	DOMString	Verwijdert het attribuut met de opgegeven naam
Attr	getAttributeNode	DOMString	Bepaalt de attribuutknoop met de opgegeven naam
Attr	setAttributeNode	Attr	Voegt een attribuutknoop toe
Attr	removeAttributeNode	Attr	Verwijdert de opgegeven attribuutknoop
NodeList	getElementsByTagName	DOMString	Bepaalt de elementen met de opgegeven naam binnen het huidig element
DOMString	getAttributeNS	DOMString namespaceURI, DOMString localName	Bepaalt de waarde van het attribuut met de opgegeven specificaties
void	setAttributeNS	DOMString namespaceURI, DOMString qualifiedName, DOMString value	Stelt de waarde van een attribuut in
void	removeAttributeNS	DOMString namespaceURI, DOMString localName	Verwijdert het attribuut met de opgegeven specificaties
Attr	getAttributeNodeNS	DOMString namespaceURI, DOMString localName	Bepaalt de attribuutknoop met de opgegeven specificaties
Attr	setAttributeNodeNS	Attr	Voegt een attribuutknoop toe
NodeList	getElementsByTagNameNS	DOMString namespaceURI, DOMString localName	Bepaalt de elementen met de opgegeven specificaties binnen het huidige element
boolean	hasAttribute	DOMString	Heeft het huidige element een attribuut met de opgegeven naam?
boolean	hasAttributeNS	DOMString namespaceURI, DOMString localName	Heeft het huidige element een attribuut met de opgegeven specificaties?

Tabel 6.6: Methodes van de interface *Element*

Hoofdstuk 7

Andere XML-technologieën

7.1 XLink

7.2 XPointer

7.3 XForm

7.4 XHTML

7.5 VoiceXLM

7.6 XQuery

Deel II

Toegang tot gegevensbanken

Hoofdstuk 8

Java Database Connectivity (JDBC)

8.1 Wat is JDBC?

JDBC is de afkorting van Java Database Connectivity. De JDBC API biedt de mogelijkheid om vanuit Java-programma's op een eenvoudige manier toegang te krijgen tot relationele gegevensbanken. Deze API bestaat uit een aantal klassen en interfaces die het mogelijk maken om SQL-opdrachten uit te voeren op een gegevensbank en de resultaten van deze opdrachten te verwerken. Daarvoor voorziet de JDBC API een aantal algemene methoden om toegang te verkrijgen tot SQL-compatibele relationele gegevensbanken. Deze methoden zijn productonafhankelijk en voorzien de meeste standaardacties voor een gegevensbank. Dit maakt het mogelijk om een zelfde applicatie te gebruiken met gegevensbanken van verschillende producenten.

JDBC bestaat uit twee pakketten: *java.sql* dat de basisfunctionaliteiten bevat en *javax.sql* met de extensies.

8.1.1 Verschillende versies

De JDBC API bestaat in drie verschillende versies, waarbij elke versie meer mogelijkheden biedt dan de vorige. Uiteraard omvat elke versie de vorige. Deze tekst behandelt hoofdzakelijk de basisfunctionaliteiten van de verschillende versies van de JDBC API.

JDBC 1.0

De JDBC 1.0 API bevat de basisklassen en interfaces om een verbinding te maken met een gegevensbank, een aantal SQL-opdrachten uit te voeren op die gegevensbank en de resultaten van die opdrachten te verwerken. Deze API maakt deel uit van de JDK 1.1.x.

JDBC 2.0

JDBC 2.0 bestaat uit twee delen: een basispakket en een optioneel pakket. Het basispakket werd opgenomen vanaf de JDK 1.2. Het optioneel deel is intussen deel van het J2SE-platform, versie 1.4. Het basispakket omvat JDBC 1.0 en voegt er een aantal nieuwe functionaliteiten aan toe. Deze nieuwe functionaliteiten zijn o.a. aanpasbare *results sets*, wijzigingen via batch, programmatorisch aanpassen van de gegevensbank, ondersteuning van de SQL3-gegevenstypes.

Het optioneel deel maakt deel uit van het J2EE-platform waarmee professionele serverapplicaties ontwikkeld kunnen worden. Het omvat o.a. transacties over verschillende gegevensbanken, *connection pooling*, JNDI voor gegevensbanken, ...

JDBC 3.0

Versie 3.0 is een uitbreiding van 2.0 en bevat meteen ook alle extensies. Ze beperkt zich in principe niet tot gegevensbanken, maar kan ook gebruikt worden voor andere gegevensstructuren in tabelvorm, zoals rekenbladen en gewone bestanden (*flat files*). Deze versie wordt standaard bij Java 1.4 gebundeld. Een overzicht van de nieuwe functionaliteiten vind je op de JDBC-website van Sun Microsystems JDBC Technology Guide: Getting Started - Appendix A: Summary of New Features

8.2 JDBC-drivers

Om client-applicaties toegang te geven tot een gegevensbankserver voorzien producenten een stel van (productafhankelijke) API's. Vanuit programma's (bv. in C++) kan dan gebruik gemaakt worden van deze API's om gegevens uit de gegevensbank op te halen of aan te passen. De JDBC API biedt een alternatief voor deze productafhankelijke API's, namelijk een API die voor elke SQL-compatibele relationele gegevensbank gebruikt kan worden, onafhankelijk van het gekozen product. Dit betekent dat het mogelijk zou moeten zijn om van gegevensbank te veranderen zonder het programma aan te passen.

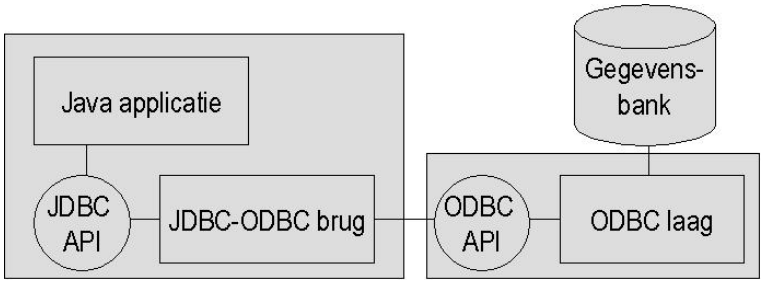
De opdrachten van deze JDBC API moeten echter nog vertaald worden naar opdrachten in de productafhankelijke API. Met andere woorden er moet implementaties voorzien worden voor de interfaces uit de JDBC API. Deze implementatie noemt men een JDBC *driver*. Deze drivers vertalen de algemene JDBC-opdrachten naar productafhankelijke opdrachten. Deze JDBC driver wordt meestal ter beschikking gesteld door de producent van de gegevensbank, maar soms ook door een onafhankelijke firma.

8.2.1 Verschillende types JDBC-drivers

Er bestaan vier soorten JDBC-drivers om een applicatie te verbinden met een gegevensbankserver.

JDBC/ODBC-brug *ODBC* (*Open Database Connectivity*) is de Microsoft's algemene API voor gegevensbanken. De ODBC API voorziet een aantal methodes om toegang te krijgen tot een gegevensbank. Deze methodes worden op zich geïmplementeerd door een ODBC-driver. De JDBC/ODBC-brug zorgt voor de vertaling van JDBC-opdrachten naar ODBC-opdrachten die op hun beurt door de ODBC-laag vertaald worden naar gegevensbankopdrachten.

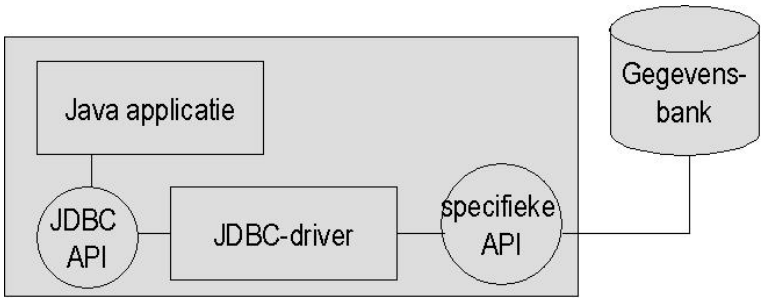
Deze werkwijze is nogal inefficiënt en dus niet bruikbaar voor applicaties die een zeer performante gegevensbanktoegang nodig hebben. Bovendien wordt de functionaliteit van de JDBC API beperkt tot de functionaliteit van de ODBC-driver.



Figuur 8.1: Structuur XML-document

De JDBC/ODBC-brug maakt deel uit van de JDK1.1 (Solaris of Windows-versie). De client-computer moet echter ook de ODBC-driver installeren en configureren.

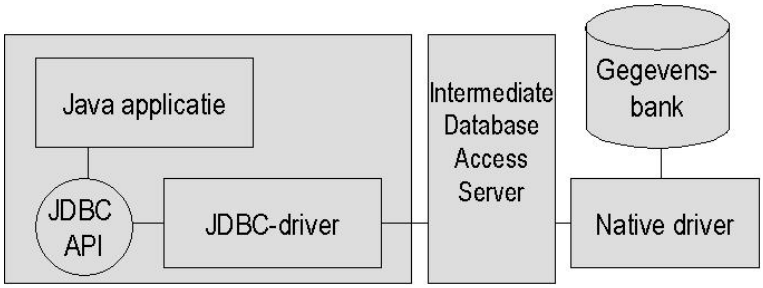
Deels Java, deels productafhankelijk De JDBC-drivers die onder deze categorie vallen, vertalen de methoden uit de JDBC-API naar methoden van de productafhankelijke API. Deze driver is efficiënter dan de JDBC/ODBC-brug en de applicatie kan gebruik maken van de volledige functionaliteit van de API voorzien door de producent. Ook hier moet een stuk binaire code specifiek voor het gegevensbanksysteem geïnstalleerd worden op de client-computer.



Figuur 8.2: Structuur XML-document

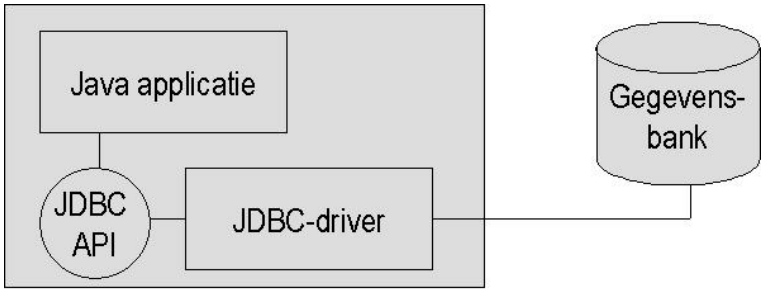
JDBC-Net JDBC-drivers van dit type maken gebruik van een tussenliggende gegevensbankserver. Java-clients kunnen nu met verschillende gegevensbanken connecteren via deze tussenliggende server. De tussenliggende server kan gebruik maken van verschillende productafhankelijke protocollen om verbindingen te maken met de verschillende gegevensbanken. De JDBC-opdracht van de Java-applicatie wordt nu via de JDBC-driver doorgestuurd naar de tussenliggende gegevensbankserver, die de aanvraag dan verder afhandelt.

Dit type driver is uitermate geschikt wanneer een applicatie gebruik moet maken van verschillende gegevensbanken.



Figuur 8.3: Structuur XML-document

Pure Java Deze drivers zijn volledig in Java geschreven en maken directe netwerkverbindingen met de gegevensbank (met behulp van de *Socket* API). Hierbij wordt gebruik gemaakt van product-afhankelijke netwerkprotocols. Deze werkwijze is de meest efficiënte wat betreft performantie, tijd om de applicatie te ontwikkelen en installatie.



Figuur 8.4: Structuur XML-document

8.2.2 Laden van de driver

Om gebruik te kunnen maken van de JDBC-drivers die horen bij de gegevensbanken die een applicatie nodig heeft, moeten deze drivers worden ingeladen. Dit verloopt als volgt:

```
import java.sql.*;
...
private final static String PAR_DRIVER = "org.postgresql.Driver";
...
    try {
        Class.forName (PAR_DRIVER);
    } catch (ClassNotFoundException e) {
        ... // driver niet gevonden
    }
```

De constante *PAR_DRIVER* stelt de volledige klassenaam van de JDBC-driver voor. In het voorbeeld wordt een PostgreSQL-gegevensbank gebruikt. Merk op dat in een echte applicatie de driverklasse niet hardgecodeerd wordt, maar wordt opgenomen in een configuratiebestand. De driverklasse wordt geleverd door de gegevensbankproducent en maakt uiteraard geen deel uit van de standaard Java API. De naam van de klasse vind je terug in de driverdocumentatie. Om de klassen van het driverpakket te kunnen gebruiken moet het jar-bestand dat de driverklasse bevat zich in het 'classpath' bevinden. Elke driver implementeert de *java.sql.Driver*-interface.

De methode *forName* laadt de driverklasse in de virtuele machine. Tijdens het laden van de driverklasse moet die een instantie van zichzelf aanmaken en registeren bij de *DriverManager*. De klasse *java.sql.DriverManager* vormt een gemeenschappelijke toegangslaag tot de verschillende JDBCdrivers van een applicatie. Merk op dat het perfect mogelijk is om tegelijkertijd verschillende drivers actief te hebben. Je toepassing kan op die manier bijvoorbeeld tegelijkertijd gebruik maken van verschillende gegevensbanken van verschillende leveranciers.

8.2.3 Verschillende versies van de JDBC API

Niet elke versie van de JDBC API wordt ondersteund door alle JDBC-drivers. De driverdocumentatie specifieert de JDBC-versie. In de online JDBC API wordt bij elke klasse en bij elke interface vermeld tot welke JDBC-versie ze behoort. Dit maakt het mogelijk om enkel die klassen en interfaces te gebruiken die ondersteund worden door de gekozen driver.

8.3 Verbindingen met een gegevensbank

Eenmaal de gepaste JDBC-driver is ingeladen, kan je een verbinding met een gegevensbank opzetten. Een dergelijke verbinding wordt voorgesteld door een object van het type *Connection*. *Connection* is een interface die behoort tot het pakket *java.sql* (net zoals de andere JDBC-interfaces en -klassen die we hier bespreken). De eigenlijke implementatie van een verbinding is afhankelijk van de gebruikte driver.

Het volgende voorbeeld illustreert hoe een verbinding met de gegevensbank geopend en gesloten kan worden.

```
private final static String
    PAR_JDBC_URL = "jdbc:postgresql:voorbeelden";
private final static String PAR_LOGIN = "iii4";
private final static String PAR_PASWOORD = "iii4pwd";
...
    try {
    Connection conn =
            DriverManager.getConnection
                (PAR_JDBC_URL, PAR_LOGIN, PAR_PASWOORD);
    try {
        ... // gegevens ophalen, toevoegen, veranderen, ...
    } finally {
        conn.close();
    }
    } catch (SQLException e) {
    ... // gegevensbankproblemen
```

```
    }
```

Om de verbinding aan te maken, gebruik je de methode *getConnection* van *DriverManager*. Deze methode heeft drie argumenten: een JDBC-URL, een gebruikersnaam en een wachtwoord. De gebruikersnaam en het wachtwoord zijn de gebruikersnaam en het wachtwoord van het DBMS (*Database Managment System*). De JDBC-URL bestaat uit drie stukken.

het protocol In een JDBC-URL is het protocol altijd `jdbc:`

het subprotocol Het subprotocol identificeert de gegevensbankdriver en wordt bepaald door de gegevensbankproducent. In het voorbeeld is het subprotocol `postgresql:`, bij een JDBC/ODBC-brug is dit bijvoorbeeld `odbc:`.

de 'subname' Dit is het laatste gedeelte van de JDBC-URL en het identificeert de gebruikte gegevensbank op een manier die kan verschillen van driver tot driver. De subname kan bijvoorbeeld de naam van de gegevensbank, de server en de poort waarop de gegevensbank draait bevatten. Bij een PostgreSQL-gegevensbank kan dit één van de volgende vormen aannemen:

```
naamdatabank
//host/naamdatabank
//host:poort/naamdatabank
```

Hierbij is `naamdatabank` de naam van de gegevensbank, `host` de naam van de server waarop de gegevensbank draait en `poort` de (netwerk)poort waarop de gegevensbank luistert naar aanvragen.

De verbinding zelf wordt gelegd met de methode *getConnection* die automatisch de juiste driver bepaalt aan de hand van de JDBC-URL. De methode *close* (van *Connection*) sluit de verbinding af.

Merk op dat de meeste methoden uit de JDBC API uitzonderingen kunnen genereren van het type *SQLException*. In bovenstaand fragment worden die opgevangen door de buitenste **try/catch**-blok. De **try/finally**-blok binnenin zorgt ervoor dat de verbinding met de databank zeker wordt afgesloten, zelfs wanneer er een fout optreedt. We hebben hier twee vernestelde **try**-opdrachten nodig omdat ook de *close* een uitzondering kan veroorzaken.

Verder merken we op dat de JDBC-URL, de loginnaam en het paswoord normaal niet hardgecodeerd worden, maar bv. worden ingelezen uit een configuratiebestand.

8.4 SQL-opdrachten

Wanneer de verbinding is gemaakt, kan je de gegevensbank vanuit een Java-programma een aantal SQL-opdrachten laten uitvoeren. Om dit te realiseren heb je een object van het type *Statement* nodig. Dit object stuurt SQL-opdrachten naar de gegevensbank en haalt de resultaten ervan op. *Statement* is een interface, de eigenlijke implementatie wordt bepaald door de JDBC-driver. Een *Statement*-object wordt op de volgende wijze aangemaakt en terug afgesloten:

```
try {
    Statement stmt = conn.createStatement ();
    try {
        ... // SQL-opdrachten uitvoeren
    } finally {
        stmt.close();
    }
} catch (SQLException e) {
    ...
}
```

Het object *conn* is van het type *Connection* en stelt een verbinding met de gegevensbank voor. De methode *close* zorgt ervoor dat de gereserveerde geheugenruimte in de gegevensbank en in de virtuele machine terug wordt vrijgegeven.

Met behulp van een *Statement*-object kan elke SQL-opdracht die de gegevensbank ondersteunt uitgevoerd worden. Als de applicatie met verschillende gegevensbanksystemen moet kunnen werken, dan moet de SQL-opdracht door al die systemen ondersteund worden. De gegevensbankonafhankelijkheid van de Java-applicatie wordt dus sterk bepaald door de gekozen SQL-opdrachten.

Bij het gebruik van het *Statement*-object wordt een onderscheid gemaakt tussen SQL-opdrachten die rijen of records van de databank als resultaat hebben (zoekopdrachten) en andere opdrachten.

8.4.1 DDL-opdrachten, insert, delete en update

Niet-zoekopdrachten zijn opdrachten die deel uitmaken van de *Data Definition Language* (DDL) en opdrachten die rijen in de gegevensbank toevoegen, verwijderen of aanpassen. De eerste reeks opdrachten wordt gebruikt om tabellen, gebruikers, en dergelijke, aan te maken en te verwijderen. De tweede reeks gebruikt de SQL-opdrachten `INSERT`, `DELETE` en `UPDATE` uit de *Data Manipulation Language* (DML). In beide gevallen wordt de methode *executeUpdate* van *Statement* opgeroepen om de opdracht uit te voeren. De signatuur van deze methode is de volgende.

```
public int executeUpdate (String sqlOpdracht)
    throws SQLException;
```

Het argument van deze methode is een *String* die de SQL-opdracht voorstelt. De SQL-opdracht bevat geen afsluitteken omdat dat verschillend is voor verschillende gegevensbanksystemen. De waarde is het aantal aangepaste rijen. In het geval van een DDL-opdracht is dit uiteraard 0.

In het volgende voorbeeld wordt een tabel `temp` aangemaakt die uit één kolom bestaat. De naam van deze kolom is `nr` en bevat een getal. Vervolgens wordt een rij aan deze tabel toegevoegd die bestaat uit het cijfer 1. Daarna wordt in alle rijen de waarde 1 vervangen door de waarde 2. Tot slot worden alle rijen met waarde 2 verwijderd.

```
// SQL-opdrachten (DDL)
String maakTabel   = "create table temp (nr numeric)";
// SQL-opdrachten (DML)
String voegRijToe  = "insert into temp values (1)";
String pasRijenAan = "update temp set nr=2 where nr=1";
String verwijderRijen = "delete from temp where nr=2";
```

```
// uitvoeren SQL-opdrachten
stmt.executeUpdate (maakTabel);
stmt.executeUpdate (voegRijToe);
int updateCnt = stmt.executeUpdate (pasRijenAan);
stmt.executeUpdate(verwijderRijen);
```

De waarde van de variabele *updateCnt* in dit voorbeeld is 1.

8.4.2 Zoekopdrachten

ResultSet bepalen

Zoekopdrachten zijn `SELECT`-opdrachten die gegevens selecteren uit één of meerdere tabellen in de gegevensbank en hun resultaat teruggeven als een aantal rijen met gegevens in één of meerdere kolommen. Deze opdrachten worden uitgevoerd met behulp van de methode *executeQuery* van *Statement*.

Het resultaat van de zoekactie is een object van het type *ResultSet*. Je kan dan later de inhoud van dit object (dat eigenlijk een soort tabel is met rijen en kolommen) opvragen met behulp van gepaste methoden.

```
public ResultSet executeQuery (String sqlOpdracht)
    throws SQLException;
```

De *StringsqlOpdracht* stelt de SELECT-opdracht voor.

Het *ResultSet*-object overlopen

Elk *ResultSet*-object bevat een wijzer (ook *cursor* genoemd). Deze wijzer staat na het aanmaken van het object voor de eerste rij. Met deze wijzer wordt het *ResultSet*-object rij na rij overlopen.

Om de cursor één rij naar beneden te schuiven, gebruik je de methode *next* (zonder parameters). Deze functie geeft **true** terug als er nog rijen zijn en **false** in het andere geval. Ze wordt vaak gebruikt in een lus van de volgende vorm:

```
    ResultSet res = stmt.executeQuery (...);
    while (res.next ()) {
       ... // haal de gegevens op van de huidige rij
    }
```

Data ophalen

Uit de rij waar de cursor naar wijst, kunnen gegevens opgevraagd worden. De interface *ResultSet* biedt verschillende *getter*-methoden om gegevens op te vragen. Bijvoorbeeld,

```
public boolean getBoolean (int columnIndex)
    throws SQLException;
```

```
public double  getDouble  (int columnIndex)
    throws SQLException;
public float   getFloat   (int columnIndex)
    throws SQLException;
public int     getInt     (int columnIndex)
    throws SQLException;
public String  getString  (int columnIndex)
    throws SQLException;
```

De methode *getXxx* geeft de waarde van de kolom met volgnummer *columnIndex* terug als een object of variabele van het type *xxx*. (Merk op dat de eerste kolom volgnummer 1 heeft.) Men raadt aan om de gegevens van een bepaalde rij zoveel mogelijk op te vragen in stijgende volgorde van kolomnummers.

Daarnaast bestaan er ook gelijknamige (iets minder efficiënte) methoden waarbij je de gewenste kolom mag aanduiden met behulp van zijn naam. Bijvoorbeeld,

```
public boolean getBoolean (String columnName)
    throws SQLException;
public double  getDouble  (String columnName)
    throws SQLException;
public float   getFloat   (String columnName)
    throws SQLException;
public int     getInt     (String columnName)
    throws SQLException;
public String  getString  (String columnName)
    throws SQLException;
```

In beide gevallen probeert de JDBC-driver de onderliggende data te converteren naar het Java-type gespecifieerd door de *getter*-methode. Deze methode geeft dan de overeenkomstige Java-waarde weer. Een overzicht van welke SQL-types in welke Java-types omgezet kunnen worden en omgekeerd vind je in de JDBC specificaties.

De implementatie van *ResultSet* kan verschillen van driver tot driver: in sommige gevallen wordt de inhoud van dit object meteen opgevuld wanneer *executeQuery* wordt opgeroepen, in andere gevallen wordt de gegevensbank slechts gecontacteerd op het moment dat de gegevens met de *getXxx*-methoden wordt opgevraagd. Dit verklaart waarom elk van deze methoden een *SQLException* kan veroorzaken.

Voorbeeld

In het volgende fragment maken we een lijst aan van alle boeken die zich in een gegevensbank bevinden. Een boek wordt gekenmerkt door een uniek ISBN-nummer, een titel en een prijs. In de gegevensbank worden de boeken bewaard in een tabel `boeken` met kolommen `isbn`, `titel` en `prijs`. In het geheugen stellen we de boekenlijst voor als een *ArrayList* van *Boek*-objecten.

```
static final String QUERY_BEPAAL_BOEKEN
    = "select * from boeken";
static final String PARAM_ISBN = "isbn";
static final String PARAM_TITEL = "titel";
static final String PARAM_PRIJS = "prijs";
...
    List boeken = new ArrayList ();
    try {
    Statement stmt = conn.createStatement ();
    try {
```

```
        ResultSet rs = stmt.executeQuery (
            QUERY_BEPAAL_BOEKEN);
        while (rs.next ()) {
            Boek boek = new Boek (rs.getString (PARAM_ISBN),
                rs.getString (PARAM_TITEL),
                rs.getDouble (PARAM_PRIJS) );
        boeken.add (boek);
        }
    } finally {
        stmt.close ();
    }
    } catch (SQLException e) {
    ...
    }
```

Hierbij stelt *conn* een verbinding met de gegevensbank voor die in een ander gedeelte van het programma wordt geopend en gesloten. De constructor van *Boek* heeft drie parameters: het unieke ISBN-nummer (een string), de titel (een string) en de prijs (een reëel getal) van het boek.

Sluiten van een *ResultSet*-object

Een *ResultSet*-object wordt automatisch afgesloten door het *Statement*-object dat het genereerde wanneer dit *Statement*-object gesloten wordt of opnieuw uitgevoerd wordt. Het kan eventueel expliciet gesloten worden met de methode *close*.

8.5 Prepared Statements

Als een SQL-opdracht meermaals uitgevoerd moet worden of verschillende keren uitgevoerd moet worden met andere parameters, dan is het efficiënter om een *PreparedStatement*-object te gebruiken. *PreparedStatement* is een interface afgeleid van de interface *Statement*. Tijdens het aanmaken van een prepared statement wordt reeds een SQL-opdracht meegegeven. Wanneer de driver dit ondersteunt wordt deze opdracht onmiddellijk naar de gegevensbank doorgestuurd en daar gecompileerd. Dit zorgt ervoor dat de SQL-opdracht sneller zal worden uitgevoerd.

8.5.1 Aanmaken van een *prepared statement*

Om een prepared statement aan te maken wordt de methode *prepareStatement* van een *Connection*-object opgeroepen. Het argument van deze methode is de SQL-opdracht, waarin eventuele parameters nog niet zijn ingevuld. Deze parameters worden door vraagtekens voorgesteld.

In het volgend voorbeeld wordt een prepared statement gecreëerd waarmee een rij toegevoegd kan worden aan een tabel *bestelling*. Deze tabel heeft drie kolommen: *klant*, *boek* en *aantal*. Op deze manier wordt bijgehouden hoeveel exemplaren een klant van één of meerdere boeken bestelt. Het object *conn* is een *Connection*-object dat op een andere plaats in de applicatie aangemaakt en afgesloten wordt.

```
static final String BEWAAR_BESTELLING_OPDRACHT
    = "insert into bestelling (klant, boek, aantal)"
      + " values (?,?,?)";
...
try {
    PreparedStatement bewaarBestellingStmt =
        conn.prepareStatement (BEWAAR_BESTELLING_OPDRACHT);
    try {
        ... // bestelling bewaren (zie verder)
    } finally {
        bewaarBestellingStmt.close();
    }
} catch (SQLException e) {
    ...
}
```

In dit voorbeeld werd een *PreparedStatement*-object aangemaakt dat zal gebruikt worden om rijen toe te voegen aan een tabel `bestelling`. De vraagtekens in de SQL-opdracht van *bewaarBestellingStmt* stellen parameters voor die later een waarde krijgen. Hetzelfde *PreparedStatement*-object kan opnieuw gebruikt worden met verschillende waarden voor elke parameter.

8.5.2 Parameters toekennen

Vooraleer je het aangemaakte *prepared statement* kan uitvoeren moet je nog waarden toekennen aan elke parameter. We illustreren dit eerst met een voorbeeld.

De volgende lijnen Java-code werken het bovenstaande voorbeeld verder uit. Voor een klant, gekenmerkt door de variabele *klantcode*, worden de bestelde boeken bewaard. Er wordt ook bijgehouden hoeveel exemplaren de klant van elke boek wil. Hiervoor wordt een lijst *artikels* overlopen. De variabele *artikels* is van het type *java.util.List*. Deze lijst bevat een reeks *Artikel*-objecten. Een *Artikel* bestaat uit een boek en het aantal aangekochte exemplaren van dit boek. Deze gegevens verkrijg je via de methoden *getBoek* en *getHoeveelheid*. De klasse *Boek* heeft de methode *getISBN* om het ISBN-nummer van het boek op te vragen.

```
// bestelling bewaren
bewaarBestellingStmt.setString (1, klantcode);
Iterator loperArtikels = artikels.iterator ();
while (loperArtikels.hasNext ()) {
    Artikel artikel = (Artikel) loperArtikels.next();
    bewaarBestellingStmt.setString(2, artikel.getBoek().getISBN());
    bewaarBestellingStmt.setInt(3, artikel.getHoeveelheid());
    bewaarBestellingStmt.executeUpdate();
}
```

De parameters van een prepared statement worden ingevuld met methoden van de volgende vorm:

```
public void setXxx (int indexParameter, xxx waardeParameter)
    throws SQLException;
```

Hierbij is *xxx* het Java-type van de parameter. Het eerste argument van de methode bepaalt het volgnummer van de parameter die de meegegeven waarde *waardeParameter* moet krijgen (parameters worden genummerd vanaf 1). Alle parameters moeten op deze manier een waarde krijgen vooraleer

de SQL-opdracht uitgevoerd kan worden. De methode *setXxx* converteert de waarde van de parameter naar een waarde van een gegevensbanktype. Zo wordt een *String* bijvoorbeeld omgezet in een *VARCHAR*.

De methoden *executeUpdate* en *executeQuery* voeren dan uiteindelijk de opdracht uit. Net zoals bij *Statement* gebruik je voor zoekopdrachten de methode *executeQuery* en in het andere geval de methode *executeUpdate*. Merk op dat deze methoden hier geen argument hebben omdat de SQL-opdracht reeds werd opgegeven bij constructie van het *prepared statement*.

8.5.3 Gegevensconversie

Hieronder illustreren we nog een andere reden om *PreparedStatement* boven *Statement* te verkiezen als de SQL-opdracht parameters bevat. Veronderstel dat we in een tabel met studentengegevens alle studenten met een gegeven familienaam wensen op te zoeken. Deze familienaam wordt gegeven in de vorm van een variabele *naam*. We zouden dit op de volgende manier kunnen doen:

```
Statement stmt = conn.createStatement ();
ResultSet rs  = stmt.executeQuery
    ("select * from studenten where name = \'" + naam + "\'");
...
```

of anders, met behulp van een prepared statement:

```
PreparedStatement pstmt = conn.createPreparedStatement
    ("select * from studenten where name = ?");
pstmt.setString (1, naam);
ResultSet rs = pstmt.executeQuery ();
...
```

Het nut van een prepared statement lijkt bij dit voorbeeld op het eerste zicht niet groot. We merken echter op dat wanneer *naam* een aanhalingsteken bevat, enkel het tweede fragment zal werken. Stel dat de variabele *naam* de waarde *D'haese* heeft, dan wordt de SQL-opdracht in het eerste fragment

```
select * from studenten where name = 'D'haese'
```

Aangezien dit geen correcte SQL-opdracht is, treedt er een fout op. Het tweede fragment werkt wel omdat hier een onderscheid gemaakt wordt tussen de SQL-opdracht en zijn parameters. Bij aanmaak van het *prepared statement* wordt bij de meeste drivers de SQL-opdracht gecompileerd in de gegevensbank. De parameters voor deze opdracht worden later doorgestuurd als parameters voor de reeds gecompileerde SQL-opdracht. Bovendien zorgt de methode *setXxx* waarmee de parameters ingesteld worden voor de conversie van de parameterwaarden. Speciale tekens worden dan omgezet.

8.5.4 SQL-injectie

SQL-injectie kan voorkomen als een gebruiker bv. via een webapplicatie parameters kan ingegeven die gebruikt worden in een SQL-opdracht. Een malafide gebruiker kan dan samen met die parameters een aantal willekeurige SQL-opdrachten meegegeven om voor hem onbeschikbare gegevens te zien,

om gegevens te verwijderen, ... Met andere woorden hij 'injecteert' een aantal SQL-opdrachten samen met de parameter die hij ingeeft.

We illustreren dit met een voorbeeld. Veronderstel dat we uit een tabel met studentengegevens de gegevens van een student met een opgegeven studentennummer willen zien. Dit nummer wordt bv. ingegeven via een HTML-formulier. De variabele *studentNummer* bevat de ingegeven waarde. We zouden de studentgegevens kunnen opvragen met het volgende stukje code. *conn* is een *Connection*-object.

```
Statement stmt = conn.createStatement ();
ResultSet rs  = stmt.executeQuery
    ("select * from studenten where studnr = "
    + studentNummer);
...
```

Indien *studentNummer* de waarde *200200234* dan wordt bij het uitvoeren van de methode *executeQuery* de volgende SQL-opdracht uitgevoerd.

```
select * from studenten where studnr = 200200234
```

Een gebruiker met slechte bedoelingen zou echter ipv *200200234* bijvoorbeeld *200200234 OR 1=1* kunnen invoeren. De uitgevoerde SQL-opdracht is dan

```
select * from studenten where studnr = 200200234 OR 1=1
```

en de gegevens van alle studenten worden getoond. Nog slechter zou de invoer *200200234; DROP TABLE studenten* zijn. Dit zou betekenen dat er twee SQL-opdrachten worden uitgevoerd.

```
select * from studenten where studnr = 0
```

en

```
DROP TABLE studenten
```

De eerste opdracht zal geen resultaat geven, maar de tweede verwijdert mogelijks de tabel *studenten*.

Een ander voorbeeld van niet-gewenste SQL-injectie is bv. de invoer *0 UNION SELECT username, password FROM users*. Dit kan eventueel de gebruikersnamen en wachtwoorden vrijgeven.

Het gebruik van *prepared statements* voorkomt de bovenstaande SQL-injecties. We kunnen de bovenstaande code als volgt vervangen.

```
PreparedStatement pstmt = conn.createPreparedStatement
    ("select * from studenten where studnr = ?");
pstmt.setString (1, studentNummer);
ResultSet rs = pstmt.executeQuery ();
...
```

Bij aanmaak van het *PreparedStatement*-object wordt de SQL-opdracht

```
select * from studenten where studnr = ?
```

gecompileerd in de gegevensbank. De methode *setString* converteert de parameterinvoer naar een *VARCHAR* die als argument aan de gecompileerd SQL-opdracht wordt gegeven. Deze *String* wordt niet meer gecompileerd.

Aangezien een klein aantal drivers *prepared statements* niet voorcompileren is het nodig om dit expliciet uit te testen.

8.6 Callable Statements

Callable Statements dienen om *stored procedures* van een gegevensbanksysteem uit te voeren. Een *stored procedure* is een functie of procedure die bewaard wordt in een gegevensbank en die door clientapplicaties kan opgeroepen worden. Deze functie of procedure haalt informatie op uit de gegevensbank of manipuleert zijn data.

Er is geen standaardsyntax of -taal om *stored procedures* aan te maken of op te roepen. Bovendien wordt deze functionaliteit niet aangeboden door alle gegevensbanksystemen. Ondanks het gebrek aan standardisering biedt de JDBC API een uniforme manier om *stored procedures* op te roepen vanuit een Java-applicatie.

8.6.1 Aanmaken *callable statement*

Om een *stored procedure* te kunnen oproepen moet je zijn naam en eventuele parameters kennen. In eerste instantie bekijken we hoe we een *stored procedure* zonder parameters kunnen uitvoeren.

Veronderstel dat we een *stored procedureSELECTEER_BOEKEN* willen uitvoeren die ons een tabel met de gegevens van alle beschikbare boeken teruggeeft. Dan moeten we eerst een *CallableStatement*-object aanmaken dat een oproep voor de procedure *SELECTEER_BOEKEN* bevat. Merk op dat dit object niet de procedure zelf bevat, maar enkel welke procedure het moet oproepen. De interface *CallableStatement* is een afgeleide interface van *PreparedStatement* en dus ook van *Statement*. Net zoals bij een *prepared statement* of een *statement* wordt een *CallableStatement*-object gecreëerd met behulp van een *Connection*-object (de variabele *conn* in het voorbeeld). In dit geval wordt de methode *prepareCall* gebruikt.

```
static final String SELECTEER_BOEKEN_OPDRACHT
    = "{call SELECTEER_BOEKEN}";
...
try {
    CallableStatement selecteerBoekenStmt =
        conn.prepareCall (SELECTEER_BOEKEN_OPDRACHT);
    try {
        ResultSet rs = selecteerBoekenStmt.executeQuery();
        ... // boeken ophalen uit de ResultSet
    } finally {
        selecteerBoekenStmt.close();
    }
} catch (SQLException e) {
```

```
    ...
}
```

Merk op dat de opdracht die de procedure *SELECTEER_BOEKEN* oproept tussen accolades staat. Dit noemen ze de *escape*-syntax. De *escape*-syntax in JDBC wordt gebruikt voor opdrachten die niet in de standaard SQL-syntax kunnen geformuleerd worden. Deze opdrachten worden geformuleerd in de *escape*-syntax en tussen accolades geplaatst. Op deze manier is het toch mogelijk om op een gestandaardiseerde wijze *stored procedures* uit te voeren. De driver moet deze opdrachten dan vertalen naar opdrachten in de SQL-syntax van het gegevensbanksysteem. In het voorbeeld wordt de opdracht *call SELECTEER_BOEKEN* vertaald naar de specifieke syntax van de gegevensbank om de *stored procedureSELECTEER_BOEKEN* op te roepen.

Aangezien de procedure *SELECTEER_BOEKEN* een zoekopdracht is, wordt de methode *executeQuery* gebruikt om de procedure uit te voeren. In het andere geval wordt uiteraard de methode *executeUpdate* opgeroepen. Indien de procedure meerdere opdrachten bevat en er dus eventueel meerdere resultaten kunnen zijn, moet de methode *execute* gebruikt worden. Meer informatie vind je in de Java API.

8.6.2 Invoerparameters

Aangezien de interface *CallableStatement* afgeleid is van de interface *PreparedStatement* is het gebruik van invoerparameters volledig analoog. De parameters zijn in dit geval de argumenten van de procedure en worden voorgesteld door vraagtekens. Met de methodes

```
public void setXxx (int indexParameter, xxx waardeParameter)
    throws SQLException;
```

krijgen de parameters een waarde, vooraleer de procedure uitgevoerd wordt. We illustreren dit met een voorbeeld.

Voor een klant, gekenmerkt door de variabele *klantcode*, worden de bestelde boeken bewaard. Er wordt ook bijgehouden hoeveel exemplaren de klant van elke boek wil. De procedure *BEWAAR_BESTELLING* bewaart deze gegevens in de gegevensbank. Deze procedure heeft drie argumenten: de unieke code van de klant, het ISBN-nummer van het boek en het aantal bestelde exemplaren. De bestelling van de klant wordt in het programma bijgehouden in een lijst *artikels*. De variabele *klantcode* bevat de unieke identificatie van de klant. De variabele *artikels* is van het type *java.util.List*. Deze lijst bevat een reeks *Artikel*-objecten. Een *Artikel* bestaat uit een boek en het aantal aangekochte exemplaren van dit boek. Deze gegevens verkrijg je via de methoden *getBoek* en *getHoeveelheid*. De klasse *Boek* heeft de methode *getISBN* om het ISBN-nummer van het boek op te vragen.

```
static final String BEWAAR_BESTELLING_OPDRACHT
    = "{call BEWAAR_BESTELLING(?,?,?)}";
...
try {
    CallableStatement bewaarBestellingStmt =
        conn.prepareCall(BEWAAR_BESTELLING_OPDRACHT);
    try {
        ...
        bewaarBestellingStmt.setString (1, klantcode);
        Iterator loperArtikels = artikels.iterator ();
```

```
        while (loperArtikels.hasNext ()) {
            Artikel artikel = (Artikel) loperArtikels.next();
            bewaarBestellingStmt.setString(2,
                artikel.getBoek().getISBN());
            bewaarBestellingStmt.setInt(3,
                 artikel.getHoeveelheid());
            bewaarBestellingStmt.executeUpdate();
        }
    } finally {
        bewaarBestellingStmt.close();
    }
} catch (SQLException e) {
    ...
}
```

8.6.3 Uitvoerparameters

Naast invoerparameters is het voor *callable statements* ook mogelijk om uitvoerparameters te specifieren. Deze parameters corresponderen dan met de uitvoerparameters van de procedure in de gegevensbank. Een uitvoerparameter is een parameter die een waarde krijgt of die verandert tijdens het uitvoeren van de procedure. Na het oproepen van de procedure kan de waarde van de parameter opgevraagd worden.

Uitvoerparameters worden net zoals invoerparameters aangeduid met een vraagteken in de opdracht in de *escape*-syntax. Vooraleer het *callable statement* uitgevoerd kan worden moet de uitvoerparameter geregistreerd worden met de methode *registerOutParameter*.

```
public void registerOutParameter(int parameterIndex,
    int sqlType) throws SQLException
```

Hierbij is *parameterIndex* het volgnummer van de parameter (beginnend bij 1), m.a.w. met welk vraagteken uit de opdracht de parameter correspondeert. De variabele *sqlType* staat voor het generieke SQL-type van de parameter. Deze types zijn de constanten van de klasse *Types*, die zoals alle klassen en interfaces in dit hoofdstuk behoort tot het pakket *java.sql*. Indien het SQL-type van de parameter *DECIMAL* of *NUMERIC* is moet ook het aantal cijfers na de komma opgegeven worden.

```
public void registerOutParameter(int parameterIndex, int sqlType,
    int schaal) throws SQLException
```

De derde parameter *schaal* stelt het aantal cijfers na de komma voor.

Als de procedure uitgevoerd is, kunnen de waarden van de uitvoerparameters opgehaald worden. De interface *CallableStatement* voorziet hiervoor een reeks *getter*-methodes van de volgende vorm.

```
public xxx getXxx(int parameterIndex) throws SQLException
```

Het type *xxx* is het Java-type dat correspondeert met het SQL-type van de parameter. (Meer informatie over de conversie tussen Java- en SQL-types in de JDBC specificaties). *parameterIndex* stelt het volgnummer van de parameter voor.

In het volgend voorbeeld wordt een *stored procedureSELECTEER_BOEKEN* uitgevoerd. Deze procedure haalt niet alleen alle beschikbare boeken op maar heeft bovendien nog een uitvoerparameter die aangeeft hoeveel boeken er beschikbaar zijn. De uitvoerparameter kan het resultaat van een functie zijn of een argument van een procedure. De *String* die de opdracht voorstelt is dus

```
final static String SELECT_BOEKEN_OPDR =
    "{ ? = call SELECTEER_BOEKEN}";
```

of

```
final static String SELECT_BOEKEN_OPDR =
    "{ call SELECTEER_BOEKEN(?)}";
```

afhankelijk van hoe de procedure gedeclareerd is in de gegevensbank. Het aanmaken van het *CallableStatement*-object, het registereren van de uitvoerparameter, het uitvoeren van de procedure en het ophalen van de boeken en het aantal beschikbare boeken verloopt als volgt. De variabele *conn* is een *Connection*-object.

```
CallableStatement cstmt = conn.prepareCall(SELECT_BOEKEN_OPDR);
try {
    cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
    ResultSet rs = cstmt.executeQuery();
    ... // boeken uit de ResultSet halen
    int aantalBoeken = cstmt.getInt(1);
    ...
} catch (SQLException e) {
    ...
} finally {
    cstmt.close();
```

Het is aan te raden om de uitvoerparameters pas te bepalen als alle gegevens uit de *ResultSet* zijn opgehaald. Dit omwille van de beperkingen van sommige gegevensbanksystemen.

Het volgend voorbeeld illustreert het samen voorkomen van invoer- en uitvoerparameters. De procedure in de gegevensbank selecteert de boeken die tot een bepaalde categorie (de variabele *categorieCode*) behoren en bepaalt ook het aantal geselecteerde boeken.

```
final static String SELECT_BOEKEN_VR_CAT_OPDR =
    "{ ? = call SELECTEER_BOEKEN_VR_CAT(?)}";
...
CallableStatement cstmt = conn.prepareCall(
    SELECT_BOEKEN_VR_CAT_OPDR);
try {
    cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
    cstmt.setString(2,categorieCode);
    ResultSet rs = cstmt.executeQuery();
    ... // boeken uit de ResultSet halen
    int aantalBoeken = cstmt.getInt(1);
    ...
} catch (SQLException e) {
    ...
} finally {
    cstmt.close();
```

Merk op dat uit de opdracht in de *escape*-syntax niet blijkt welke parameter een invoerparameter is en welke een uitvoerparameter.

8.6.4 Invoer- en uitvoerparameters

Sommige parameters zijn zowel invoer- als uitvoerparameters. Dat betekent dat de parameter een waarde heeft die meegegeven wordt aan de *stored procedure* en dat de waarde van de parameter verandert tijdens het uitvoeren van de procedure. Om dit te realiseren krijgt de parameter een waarde via een *setter*-methode en wordt geregistreerd met de methode *registerOutParameter*. Na het oproepen van de procedure en eventueel het ophalen van de *Resultset* kan dan de waarde van de parameter opgevraagd worden.

Veronderstel een procedure *BEWAAR_BESTELLING* in een gegevensbank. Deze procedure bewaart voor een bepaalde klant hoeveel hij van een welbepaald boek bestelt. Bovendien past deze procedure het totaal aantal bestelde boeken van deze klant aan. De procedure krijgt mee hoeveel boeken de klant tot nu toe reeds bestelde. Het totaal aantal parameters van deze procedure is dus vier: de klantcode, het ISBN-nummer van het boek, het aantal bestelde exemplaren van dit boek en het totaal aan bestelde boeken van de klant. De eerste drie parameters zijn invoerparameters. De laatste parameter is zowel een invoer- als een uitvoerparameter. Het stukje Java-code die voor één klant en één boek de bestelling bewaart ziet er dan zo uit.

```
final static String BEWAAR_BESTELLING_OPDR =
    "{call BEWAAR_BESTELLING(?,?,?,?)}";
...
CallableStatement cstmt = conn.prepareCall(
    BEWAAR_BESTELLING_OPDR);
try {
    ...
    // parameterwaarden toekennen
    cstmt.setString(1,klantCode);
    cstmt.setString(2,ISBNNummer);
    cstmt.setInt(3,aantalBoeken);
    cstmt.setInt(4,totaalAantalBoeken);
    // uitvoerparameter registreren
    cstmt.registerOutParameter(4, java.sql.Types.INTEGER);
    // procedure oproepen
    cstmt.executeUpdate();
    // uitvoerparameter ophalen
    int totaalAantalBoeken = cstmt.getInt(4);
    ...
} catch (SQLException e) {
    ...
} finally {
    cstmt.close();
```

8.7 Transacties

Sommige applicaties vereisen dat een reeks SQL-opdrachten ofwel allemaal succesvol zijn ofwel allemaal niet uitgevoerd worden. Het is niet toegelaten dat een deel van de opdrachten wel en een ander deel niet uitgevoerd worden. Als alle opdrachten uitgevoerd konden worden, dan mogen ze permanent gemaakt worden in de databank (*commit* in het Engels), in het andere geval moeten alle reeds uitgevoerde opdrachten ongedaan gemaakt worden (*rollback* in het Engels). Een transactie is een reeks opdrachten die samen uitgevoerd moeten worden.

De interface *Connection* voorziet de volgende methoden om transacties te implementeren.

```
public void setAutoCommit(boolean autoCommit)
    throws SQLException;
public void commit() throws SQLException;
public void rollback() throws SQLException;
```

Een *Connection*-object wordt standaard aangemaakt in de mode 'auto-commit'. Dit betekent dat de geslaagde uitvoering van een SQL-opdracht onmiddellijk permanent is. Als de methode *executeQuery* of *executeUpdate* van een *Statement*-object succesvol is, dan is zijn resultaat permanent.

Om meerdere opdrachten samen te kunnen uitvoeren moet de auto-commit-mode uitgeschakeld worden. De methode *setAutoCommit* schakelt de auto-commit-mode aan of uit afhankelijk van het argument (respectievelijk **true** of **false**). Indien de auto-commit-mode uitgeschakeld is, wordt het resultaat van één of meerdere SQL-opdrachten uitgevoerd over de verbinding pas permanent na het oproepen van de methode *commit*.

Om het resultaat van een aantal SQL-opdrachten ongedaan te maken roep je de methode *rollback* aan. Alle opdrachten sinds de laatste *commit* worden dan verwijderd.

In het voorbeeld uit §8.5 kunnen we eisen dat de bestelling van één klant volledig ingevoerd moet worden of als dat niet kan niets ingevoerd mag worden. De code wordt dan als volgt aangepast.

```
try {
    // bestelling bewaren
    bewaarBestellingStmt.setString(1,klantcode);
    Iterator loperArtikels = artikels.iterator();
    conn.setAutoCommit (false);
    while (loperArtikels.hasNext()) {
        Artikel artikel = (Artikel) loper.next ();
        bewaarBestellingStmt.setString(2,
            artikel.getBoek().getId());
        bewaarBestellingStmt.setInt(3,
            artikel.getHoeveelheid());
        bewaarBestellingStmt.executeUpdate ();
    }
    conn.commit ();
} catch (SQLException e) {
    conn.rollback ();
} finally {
    conn.setAutoCommit (true);
}
```

In het bovenstaande wordt de methode *executeUpdate* van *bewaarBestelling* nul of meerdere keren opgeroepen afhankelijk van het aantal verschillende boeken dat de klant bestelde. Indien er iets misloopt bij de uitvoering van één van de methodes *executeUpdate* dan worden de resultaten van alle uitvoeringen van deze methode voor de huidige klant ongedaan gemaakt. Het oproepen van de methode *rollback* in het *catch*-blok zorgt daarvoor. Konden alle methodes *executeUpdate* wel goed uitgevoerd worden dan worden de wijzigingen permanent gemaakt met de methode *commit*. In beide gevallen moet nadien de verbinding weer in de mode 'auto-commit' geplaatst worden. Zo kan het *Connection*-object opnieuw correct gebruikt worden op een andere plaats.

Een ander effect van het gebruik van transacties is dat andere gebruikers van de gegevensbank het resultaat van de aanpassingen pas merken als de methode *commit* is uitgevoerd.

Merk op dat bij gelijktijdige transacties elke transactie een ander *Connection*-object moet hebben.

Verschillende transactie kunnen uiteraard niet gelijktijdig over hetzelfde *Connection*-object verlopen.

8.8 Programmatorisch aanpassen van de gegevensbank

Sinds JDBC 2.0 is het ook mogelijk om gegevens in een *ResultSet* aan te passen (met blijvend effect in de gegevensbank). In plaats van SQL-opdrachten naar de gegevensbank te sturen, kan je nu Java-methodes oproepen om gegevens aan te passen, toe te voegen of te verwijderen.

In deze sectie zullen we de volgende eenvoudige voorbeeldapplicatie gebruiken om de werking van aanpasbare *ResultSet*-objecten te illustreren. Veronderstel een GUI-applicatie waarin we de administratie van een aantal cursussen beheren. Elke cursus heeft een unieke identificatie, een titel en een lesgever, die gekenmerkt wordt door zijn of haar code. De cursussen worden bewaard in een tabel *cursussen* met kolommen *cursusid*, *titel* en *lesgeverscode*. De gebruiker in de GUI-applicatie kan de titel of de lesgeverscode van de cursus veranderen. Ook kan hij of zij een cursus toevoegen of verwijderen.

De functionaliteit in de voorbeeldapplicatie zouden we kunnen realiseren door in het begin een zoekopdracht uit te voeren om te cursussen op te halen die in de GUI-getoond moeten worden. Vervolgens zouden we bij elke aanpassing, toevoeging of verwijdering een SQL-opdracht naar de gegevensbank kunnen sturen met bv. een *prepared statement*. In deze sectie illustreren we een andere wijze om dit te realiseren.

8.8.1 *ScrollableResultSet*

Sinds JDBC 2.0 is het ook mogelijk om de cursor in andere richtingen te verplaatsen dan van boven naar beneden, om de cursor naar een welbepaalde rij te verplaatsen of om de positie van de cursor te bepalen. De *ResultSet* heet dan *scrollable*. Deze nieuwe functionaliteit kan zinvol zijn om bv. een grafische applicatie te maken om *ResultSet*-objecten te bekijken of om een *ResultSet*-object aan te passen (§8.8)

Om een *scrollableResultSet*-object te krijgen moeten er een aantal extra opties meegegeven worden bij het aanmaken van het *Statement*-object. De volgende methodes van de interface *Connection* kunnen hiervoor gebruikt worden. De gebruikte methode hangt af van het type *statement* dat gewenst is.

```
public Statement createStatement(int resultSetType,
    int resultSetConcurrency) throws SQLException;
public PreparedStatement prepareStatement(String sql,
    int resultSetType, int resultSetConcurrency)
    throws SQLException;
public CallableStatement prepareCall(String sql,
    int resultSetType,
    int resultSetConcurrency) throws SQLException;
```

De parameter *resultSetType* bepaalt het type van het *ResultSet*-object na het uitvoeren van het *Statement*-object. Er zijn drie mogelijke types bepaald door de constanten *TYPE_FORWARD_ONLY*, *TYPE_SCROLL_INSENSITIVE* of *TYPE_SCROLL_SENSITIVE* van de interface *ResultSet*.

TYPE_FORWARD_ONLY De cursor kan enkel van boven naar beneden bewegen in het *ResultSet*-object. De enige mogelijke verplaatsing is één rij naar beneden.

TYPE_SCROLL_INSENSITIVE De cursor kan op een willekeurige wijze verplaatst worden. Aanpassingen aan de gegevens van het *ResultSet*-object in de gegevensbank zijn niet zichtbaar in het *ResultSet*-object.

TYPE_SCROLL_SENSITIVE De cursor kan op een willekeurige wijze verplaatst worden. Aanpassingen aan de gegevens van het *ResultSet*-object in de gegevensbank worden weerspiegeld in het *ResultSet*-object.

Of een *ResultSet*-object aanpasbaar is wordt bepaald door de parameter *resultSetConcurrency*. Deze parameter kan twee waarden aannemen: *ResultSet.CONCUR_READ_ONLY* of *ResultSet.CONCUR_UPDATABLE*. In het eerste geval kan het *ResultSet*-object niet aangepast worden, in het tweede wel (zie ook §8.8).

Een *scrollable resulset* kan dus op de volgende manier aangemaakt worden. De variabele *SQLZoekopdracht* is een *String* die een SELECT-opdracht in SQL voorstelt. De variabele *conn* is van het type *Connection*.

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery(SQLZoekopdracht);
```

Deze *ResultSet* kan van onder naar boven doorlopen worden met behulp van de methodes *afterLast* en *previous*. De methode *afterLast* plaatst de cursor na de laatste rij. De cursor wordt één rij naar boven geschoven met de methode *previous*. Deze methode geeft *true* als de cursor naar een rij wijst, in het andere geval is het resultaat *false*.

```
rs.afterLast();
while (rs.previous()) {
    ... // iets doen met de gegevens uit de huidige rij
}
```

In de Java API vind je meerdere methodes om de cursor op een relative of absolute manier te verplaatsen. Er bestaan ook nog methodes om de positie van de cursor te verifiëren bv. *isLast*, die aangeeft of de cursor naar de laatste rij wijst. Tot slot vermelden we nog de methode *getRow* die het rijnummer van de cursor weergeeft.

Merk op dat de functionaliteit die in deze paragraaf besproken werd, beperkt kan worden door de mogelijkheden van de driver of van het gegevensbanksysteem.

8.8.2 Aanmaken van een aanpasbaar *ResultSet*-object

In de vorige paragraaf zagen we hoe we een *scrollable* en aanpasbaar *ResultSet*-object kunnen aanmaken. In ons voorbeeld kan dit bijvoorbeeld zo. De variabele *conn* is een open *Connection*-object.

```
String SQLZoekopdracht =
    "select cursusid, titel, lesgeverscode from cursussen;
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet uprs  = stmt.executeQuery(SQLZoekopdracht);
```

Als type van het *ResultSet*-object kozen we *TYPE_SCROLL_SENSITIVE* omdat we de cursor dan willekeurig heen en weer kunnen bewegen. Bovendien is het nu ook mogelijk om veranderingen aan de gegevens van het *ResultSet*-object achteraf op te halen.

8.8.3 Een rij aanpassen

Als in de GUI de titel of lesgeverscode aangepast wordt, kunnen we dit doorgeven aan de gegevensbank door gebruik te maken van prepared statements (zie §8.5).

```
String pasTitelAan =
    "update cursussen set titel = ? where cursusid = ?";
String pasLesgeverAan =
    "update cursussen set lesgeverscode = ? where cursusid = ?";
PreparedStatement pasTitelAanStmt =
    conn.prepareStatement(pasTitelAan);
PreparedStatement pasLesgeverAanStmt =
    conn.prepareStatement(pasLesgeverAan);
```

Het aanpassen van de cursusgegevens in de gegevensbank verloopt dan als volgt. Hierbij stellen de variabelen *cursusid*, *titel* en *lesgeverscode* respectievelijk de unieke identificatie, de nieuwe titel en de nieuwe lesgeverscode van de aan te passen cursus voor.

```
pasTitelAanStmt.setString(1,titel);
pasTitelAanStmt.setString(2,cursusid);
pasTitelAanStmt.executeUpdate();
```

of

```
pasLesgeverAanStmt.setString(1,lesgeverscode);
pasLesgeverAanStmt.setString(2,cursusid);
pasLesgeverAanStmt.executeUpdate();
```

Gebruik makend van de JDBC 2.0 API kan dit ook op de volgende manier. De variabele *rijNr* stelt het rijnummer van de aan te passen cursus in het *ResultSet*-object voor. De variabelen *titel* en *lesgeverscode* bevatten de nieuwe titel en de nieuwe lesgeverscode. *uprs* is de hierboven aangemaakte *result set*. De constanten *TITEL* en *LESG_CODE* werden elders in de applicatie gedeclareerd en stellen de namen van de corresponderende kolommen voor.

```
static final String TITEL = "titel";
static final String LESG_CODE = "lesgeverscode";
...
uprs.absolute(rijNr);
uprs.updateString(TITEL,titel);
uprs.updateString(LESG_CODE,lesgeverscode);
uprs.updateRow();
```

De methode *absolute* verplaatst de cursor naar de rij in het *ResultSet*-object bepaald door zijn argument. De eerste rij heeft nummer 1, de tweede 2, enz. Als het argument van deze methode negatief is dan wordt er teruggeteld vanaf het einde van het *ResultSet*-object. Het oproepen van de methode *absolute(-1)* verplaatst de cursor naar de laatste rij, *absolute(-2)* naar de voorlaatste enz.

De rij waar de cursor naar wijst kan aangepast worden met methodes van de vorm.

```
public void updateXxx(String kolomnaam, xxx waarde)
    throws SQLException;
```

of

```
public void updateXxx(int kolomnr, xxx waarde)
    throws SQLException;
```

Het eerste argument van deze methodes stelt de kolom voor (bepaald door zijn naam of volgnummer in het *ResultSet*-object), het tweede de nieuwe waarde voor de kolom in de huidige rij. Het type *xxx* is het Java-type van de kolom en de waarde van de kolom wordt door de *updateXxx*-methode geconverteerd naar het corresponderende SQL-type. De volgnummers van de kolommen beginnen bij 1. De methode *updateXxx* verandert de waarde van de kolom in het *ResultSet*-object, maar niet in de onderliggende gegevensbank.

Om de veranderingen ook door te voeren in de gegevensbank moet de methode *updateRow* uitgevoerd worden. Indien de cursor verplaatst wordt vooraleer deze methode werd opgeroepen dan zijn alle veranderingen verdwenen.

Om alle veranderingen aan de huidige rij ongedaan te maken, roep je de methode *cancelRowUpdates*. Deze methode moet wel opgeroepen worden voordat de methode *updateRow* uitgevoerd is. Later oproepen van deze methode heeft geen effect meer.

8.8.4 Een rij toevoegen

Als er een nieuwe cursus aangemaakt wordt in de GUI-applicatie dan moet die ook worden toegevoegd in de gegevensbank. Met prepared statements (zie §8.5) gaat dit als volgt. Bij het starten van de applicatie wordt een *PreparedStatement*-object aangemaakt.

```
String voegCursusToe = "insert into cursussen values(?,?,?)";
PreparedStatement voegCursusToeStmt =
    conn.prepareStatement(voegCursusToe);
```

Bij het aanmaken van een cursus wordt het *PreparedStatement*-object uitgevoerd. De variabelen *cursusid*, *titel* en *lesgeverscode* hebben dezelfde functie als voorheen.

```
voegCursusToe.setString(1,cursusid);
voegCursusToe.setString(2,titel);
voegCursusToe.setString(3,lesgeverscode);
voegCursusToe.executeUpdate();
```

Zonder SQL-opdrachten, gebruik makend van de JDBC 2.0 API, wordt dit.

```
uprs.moveToInsertRow();
uprs.updateString(CURSUS_ID,cursusid);
uprs.updateString(TITEL,titel);
uprs.updateString(LESG_CODE,lesgeverscode);
uprs.insertRow();
```

Hierbij maken we gebruik van het *ResultSet*-object *uprs* uit §8.8.2. De constanten *CURSUS_ID*, *TITEL* en *LESG_CODE* werden elders in de applicatie gedeclareerd en stellen de kolomnamen in de *result set* voor. Een alternatief gebruik makend van de kolomnummers is.

```
uprs.moveToInsertRow();
uprs.updateString(1, cursusid);
uprs.updateString(2, titel);
uprs.updateString(3, lesgeverscode);
uprs.insertRow();
```

Om een nieuwe rij aan maken, beweeg je de cursor naar de invoegrij (in het Engels *insert row*). Deze rij kan je beschouwen als een soort buffer geassocieerd met het *ResultSet*-object waarin een nieuwe rij opgebouwd kan worden. Deze buffer maakt geen deel uit van de *result set*. De cursor verplaats je naar de invoegrij met de methode *moveToInsertRow*.

Als de cursor naar de invoegrij wijst, kan je waarden geven aan elke kolom met de *updateXxx*-methodes uit de §8.8.3. De gegevens maken nu nog geen deel uit van het *ResultSet*-object of de gegevensbank. Om dit te realiseren roep je de methode *insertRow* op. Deze methode voegt de nieuwe rij toe aan het *ResultSet*-object en aan de onderliggende gegevensbank.

De cursor wijst nu nog steeds naar de invoerrij. Om hem terug te verplaatsen naar de laatste rij waar hij naar wees voor hij verzet werd naar de invoegrij roepen we de methode *moveToCurrentRow* op. Bij het verplaatsen van de cursor naar de invoegrij werd namelijk de huidige positie van de cursor onthouden.

Aangezien het *ResultSet*-object van het type *ResultSet.TYPE_SCROLL_SENSITIVE* is kunnen we de aangebrachte veranderingen terug opvragen (bv. om te tonen in de GUI) met de getter-methodes van de interface *ResultSet* (Zie §8.4.2).

8.8.5 Een rij verwijderen

De derde en laatste manier om een *ResultSet*-object te veranderen is het verwijderen van een rij. Dit gaat als volgt: verplaats de cursor naar de te verwijderen rij en verwijder vervolgens de rij met de methode *deleteRow* van de interface *ResultSet*.

```
uprs.absolute(rijNr);
uprs.deleteRow();
```

rijNr is het volgnummer in het *ResultSet*-object van de te verwijderen rij. De methode *deleteRow* verwijdert de rij uit het *ResultSet*-object en uit de gegevensbank.

8.8.6 Welke zoekopdrachten?

Niet alle zoekopdrachten produceren een aanpasbare *result set*, ook al is het type van de *result set* goed ingesteld. Bijvoorbeeld als de zoekopdracht de primaire sleutel niet selecteert is het mogelijk dat de verkregen *result set* niet aanpasbaar is. Er zijn een aantal criteria die normaal gezien voor een aanpasbare *result set* moeten zorgen.

1. De zoekopdracht verwijst maar naar één tabel.
2. De zoekopdracht bevat geen 'join'- of *GROUP BY*-clausule.
3. De zoekopdracht selecteert de primaire sleutel van de tabel.
4. De gebruiker heeft lees- en schrijfpermissies op de tabel in de gegevensbank.

Bij het toevoegen van een rij moeten bovendien ook de volgende voorwaarden vervuld zijn.

1. De zoekopdracht selecteert alle kolommen die niet leeg mogen zijn.
2. De zoekopdracht selecteert alle kolommen zonder standaardwaarde.

8.8.7 Opmerking

Merk op dat in het voorbeeld het *ResultSet*-object tijdens de volledige applicatie open moet blijven. M.a.w. ook het *Connection*- en *Statement*-object zijn de hele tijd open en reserveren geheugenruimte in de gegevensbank. In een multi-user-omgeving is dit vaak te belastend en is het dus niet wenselijk om gebruik te maken van een aanpasbaar *ResultSet*-object.

8.9 Batch

Sinds JDBC 2.0 is het mogelijk om een aantal SQL-opdrachten (GEEN zoekopdrachten) als een éénheid uit te voeren. Dit maakt de uitvoering van die opdrachten soms efficiënter. Merk op dat dit een optionele functionaliteit is en dat dus niet alle drivers die ondersteunen. Bovendien is het niet zeker dat de driver deze functionaliteit zo implementeert dat ze performanter is.

In JDBC 1.0 is elke opdracht een apart proces, zelfs als meerdere opdrachten tot één transactie (zie §8.7) behoren. De *Statement*-objecten (en dus ook de *PreparedStatement*- en *CallableStatement*-objecten) in JDBC 2.0 kunnen een lijst van opdrachten bijhouden. Deze lijst is in eerste instantie leeg en kan als één batch doorgestuurd worden. Met de volgende methodes van de interface *Statement* kan deze lijst gemanipuleerd worden.

```
public void addBatch(String sql) throws SQLException;
public void clearBatch() throws SQLException;
public int[] executeBatch() throws SQLException;
```

De methode *addBatch* voegt de SQL-opdracht *sql* toe aan de lijst van opdrachten van het *Statement*-object. De lijst wordt leeggemaakt met de methode *clearBatch* en uitgevoerd met de methode *executeBatch*. Deze methode voert de opdrachten in volgorde van toevoegen uit. De teruggeefwaarde van deze methode is een tabel van *int*-waarden die aangeven hoeveel rijen elke opdracht veranderde.

In het volgend voorbeeld voegen we drie rijen toe aan een tabel met cursussen. Deze tabel heeft drie kolommen die respectievelijk de unieke identificatie, de titel en de lesgever, gekenmerkt door zijn of haar code, van de cursus bevatten. De variabele *conn* is een open *Connection*-object.

```
String SQLOpdrachten =
    {"insert into cursussen values ('INF401','Webservices','TDSM')",
    "insert into cursussen values ('INF301',"
     + "'Applicaties met JDBC','ELEE')",
    "insert into cursussen values ('INF402','Webtechnologie',"
    + "'DDRI')"};
Statement stmt = conn.createStatement();
try {
    conn.setAutoCommit(false);
    try {
        for (int i = 0; i < SQLOpdrachten.length; i++)
            stmt.addBatch(SQLOpdrachten[i]);
        stmt.executeBatch();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
    } finally {
        conn.setAutoCommit(true);
    }
} finally {
    stmt.close();
}
```

Om een batch correct uit te voeren moet hij doorgestuurd worden als een transactie. (zie §8.7).

Voor *PreparedStatement*-objecten - en dus ook voor *CallableStatement*-objecten - is er een andere *addBatch*-methode.

```
public void addBatch() throws SQLException;
```

Deze methode voegt de SQL-opdracht van het *PreparedStatement*-object met de huidige parameters toe aan de lijst met opdrachten. Ook hier mogen de corresponderende SQL-opdrachten geen resultaten hebben. Zoekopdrachten en procedures met uitvoerparameters zijn niet toegelaten. Het gebruik van deze methode wordt geïllustreerd in het volgend voorbeeld, een aanpassing van het voorbeeld in §8.5.

```
static final String BEWAAR_BESTELLING_OPDRACHT
    = "insert into bestelling (klant, boek, aantal)"
      + " values (?,?,?)";
...
try {
    PreparedStatement bewaarBestellingStmt =
        conn.prepareStatement (BEWAAR_BESTELLING_OPDRACHT);
    try {
        bewaarBestellingStmt.setString(1,klantcode);
        Iterator loperArtikels = artikels.iterator();
        conn.setAutoCommit (false);
        try {
            while (loperArtikels.hasNext()) {
                Artikel artikel = (Artikel) loper.next ();
                bewaarBestellingStmt.setString(2,
```

```
                    artikel.getBoek().getId());
                bewaarBestellingStmt.setInt(3,
                    artikel.getHoeveelheid());
                bewaarBestellingStmt.addBatch ();
            }
            bewaarBestellingStmt.executeBatch();
            conn.commit ();
        } catch (SQLException e) {
            conn.rollback ();
        } finally {
            conn.setAutoCommit (true);
        }
    } finally {
        bewaarBestellingStmt.close();
    }
} catch (SQLException e) {
    ...
}
```

De methode *executeBatch* kan niet alleen een *SQLException* gooien, maar ook een *BatchUpdateException*. De *SQLException* komt voor als er een gegevensbankfout optreedt of als de driver de batch-functionaliteit niet ondersteunt. Als één van de opdrachten van de batch uitvoer genereert of niet goed uitgevoerd kan worden dan wordt een *BatchUpdateException* gegooid. De klasse *BatchUpdateException* is afgeleid van de klasse *SQLException*. De enige extra methode die deze klasse voorziet is de methode

```
public int[] getUpdateCounts();
```

De teruggeefwaarde van deze methode is een tabel met *int*-waarden die het aantal aangepaste rijen bepalen van de methodes uitgevoerd voordat de fout optrad.

8.10 Metadata

In de voorgaande paragrafen werd meermaals opgemerkt dat bepaalde functionaliteit niet door alle drivers of gegevensbanken ondersteund wordt. Deze informatie kan uiteraard teruggevonden worden in de documentatie van de gegevensbank en de driver. In sommige gevallen is het echter ook nuttig om die informatie in de Java-applicatie zelf te kunnen opvragen. Daarvoor worden in de JDBC API een aantal metadata-interfaces voorzien.

Implementatie van die interfaces kunnen gebruikt worden om informatie over de gegevensbank, de driver, de opgehaalde gegevens, ... te verkrijgen. Dit is niet direct zinvol voor kleine applicaties waar de ontwikkelaar de eigenschappen en mogelijkheden van de gegevensbank en de JDBC-driver kent. Voor complexere applicaties die meerdere gegevensbanken bevatten kan het wel nuttig zijn om programmatorisch informatie op te vragen over de mogelijkheden van de gegevensbanken , de drivers, ...

De drie metadata-interfaces die we in deze sectie bespreken zijn *DatabaseMetaData*, *ResultSetMetaData* en *ParameterMetaData*. Ze behoren allen tot het pakket *java.sql*. De interface *ParameterMetaData* maakt deel uit van de JDBC API sinds versie 3.0.

8.10.1 DatabaseMetaData

Deze interface voorziet in methodes die informatie geven over de mogelijkheden van de gegevensbank en de gebruikte driver. De informatie die opgevraagd kan worden is onder andere de volgende

- de naam van de gegevensbank, de naam van de driver, het maximum aantal verbindingen, de ondersteunde JDBC-versies, ...
- welke SQL-syntax de gegevensbank gebruikt
- of de gegevensbank *stored procedures* ondersteund en zo ja, welke er beschikbaar zijn
- informatie over de tabellen in de gegevensbank
- of *batch*-aanpassingen ondersteund zijn
- welke type *result sets* ondersteund zijn
- of transacties ondersteund worden
- ...

Om deze informatie op te halen moet een *DatabaseMetaData* aangemaakt worden. Op dit object worden dan de gewenste methodes opgeroepen. Dit wordt geïllustreerd in het volgend stukje code. De variabele *conn* is een open *Connection*-object.

```
DatabaseMetaData metaData = conn.getMetaData();
boolean batchOndersteund = metaData.supportsBatchUpdates();
boolean storedProcOndersteund =
     metaData.supportsStoredProcedures();
boolean scrollRSOndersteund = metaData.supportsResultSetType(
    ResultSet.TYPE_SCROLL_INSENSITIVE);
...
```

8.10.2 ResultSetMetaData

De *ResultSetMetaData*-interface verschaft, zoals de naam aangeeft, informatie over het *ResultSet*-object. Deze informatie is voornamelijk informatie over de kolommen van het *ResultSet*-object: aantal kolommen, naam van de kolommen, type van de kolommen, ...

Een *ResultSetMetaData*-object wordt aangemaakt door de methode *getMetaData* op te roepen op een open *ResultSet*-object. Het voorbeeld illustreert dit. *rs* is een open *ResultSet*-object.

```
ResultSetMetaData RSMetaData = rs.getMetaData();
int aantalKol = RSMetaData.getColumnCount();
for (int i = 1; i <= aantalKol; i++) {
    System.out.println("Naam kolom " + i + ": "
        + RSMetaData.getColumnName(i));
}
```

De bovenstaande code drukt de namen van de kolommen van het *ResultSet*-object af. Merk op dat de kolomnummering begint met 1.

8.10.3 ParameterMetaData

De interface *ParameterMetaData* is nieuw in JDBC3.0 en wordt gebruikt om eigenschappen van parameters van een *prepared statement* (zie §8.5) te bepalen. Het volgend stukje code illustreert het gebruik van deze interface. Het gegevensbanktype van elke parameter van een *PreparedStatement*-object *pstmt* wordt uitgeschreven. Merk op dat de parameters genummerd worden vanaf 1.

```
ParameterMetaData paramInfo = pstmt.getParameterMetaData();
int aantalParam = paramInfo.getParameterCount();
for (int i = 1; i <= aantalParam; i++) {
    String dbType = paramInfo.getParameterTypeName(i);
    System.out.println("Parameter " + i " is van het type "
         + dbType);
}
```

8.11 SQL3-gegevenstypes

De SQL3- of SQL99-gegevenstypes breiden de types van de kolommen in een relationele gegevensbank uit. De extra functionaliteiten die deze gegevenstypes bieden zijn: de mogelijkheid om grote hoeveelheden binaire of tekstuele data te bewaren, om eigen gegevenstypes te declareren en om *arrays* te gebruiken.

Vanaf JDBC 2.0 worden SQL3-gegevenstypes ondersteund. SQL3-gegevenstypes kunnen nu geconverteerd worden naar Javatypes en omgekeerd. Het is met andere woorden mogelijk om deze gegevenstypes te gebruiken in *result sets*, *prepared statements*, ... De volgende tabel geeft een overzicht van de SQL3-types en hun corresponderende Javatype. De Javatypes maken deel uit van het pakket *java.sql*.

SQL3-type	**Javatype**	**Omschrijving**
BLOB	Blob	*Binary Large Object*
CLOB	Clob	*Character Large Object*
ARRAY	Array	Een tabel van waarden
gestructureerd SQL-type	Struct	Een zelfgemaakt gestructureerd SQL-type
REF	Ref	Een verwijzing naar een instantiatie van een zelfgemaakt gestructureerd SQL-type
DISTINCT	Javatype van het type waarvan het DISTINCT-type is afgeleid	Een nieuw type afgeleid van een basistype. Bijvoorbeeld een type *taalcode* of *ISBN-nummer* afgeleid van het type *VARCHAR*

8.11.1 SQL3-gegevenstypes gebruiken

Net zoals bij de gegevenstypes uit JDBC 1.0 kan je methodes *getXxx*, *setXxx* of *updateXxx* van de interfaces *ResultSet*, *PreparedStatement* of *CallableStatement* gebruiken om waarden op te halen, toe te kennen of aan te passen. De volgende tabel geeft een overzicht van de te gebruiken methodes.

SQL3-type	***getXxx*-methode**	***setXxx*-methode**	***updateXxx*-methode**
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
ARRAY	getArray	setArray	updateArray
gestructureerd SQL-type	getObject	setObject	updateObject
REF	getRef	setRef	updateRef
DISTINCT	De corresponderende methode van het type waarvan het DISTINCT-type is afgeleid		

We illustreren een aantal van deze types en methodes met een voorbeeld. Veronderstel dat de hoeveelheid neerslag voor een bepaalde plaats per dag bewaard wordt in een tabel *NEERSLAG_PER_DAG*. De kolom *METINGEN* van deze tabel is van het type *ARRAY* en bevat de 24 metingen die per dag uitgevoerd worden. Daarnaast heeft de tabel *NEERSLAG_PER_DAG* ook nog een kolom *PLAATS*. Op basis van deze gegevens wordt een tabel *JAAROVERZICHT* aangemaakt. Deze tabel bevat per plaats een figuur en een tekstje, die afhankelijk zijn van de totale hoeveelheid neerslag over het hele jaar. De figuren en teksten worden bewaard in een aparte tabel.

In het volgend codefragment wordt geïllustreerd hoe een SQL *ARRAY* opgehaald wordt.

```
PreparedStatement pstmt = conn.prepareStatement(
    "SELECT METINGEN FROM NEERSLAG_PER_DAG WHERE PLAATS = ?");
... // verschillende plaatsen overlopen,
    // plaatsparameter instellen, ...
ResultSet rs = pstmt.executeQuery();
while (rs.next()) {
    Array meting = rs.getArray("METINGEN");
    ... // gegevens verwerken
}
```

De variabele *meting* is een verwijzing naar SQL *ARRAY*-object in de gegevensbank en bevat dus niet de eigenlijke waarde.

Op basis van de hoeveelheid neerslag kan nu voor elke plaats de juiste figuur en tekst geselecteerd worden. Dit zijn grote hoeveelheden binaire en tekstuele data, m.a.w. van het SQL-type *BLOB* en *CLOB*. In het volgend voorbeeld wordt getoond hoe deze gegevens uit het *ResultSet*-object opgehaald worden en vervolgens toegewezen worden als parameter van een *PreparedStatement*-object.

```
PreparedStatement pstmt = conn.prepareStatement(
    "INSERT INTO JAAROVERZICHT VALUES(?,?,?)");
... // plaatsparameter instellen, ...
... // juiste figuur en tekst bepalen
Clob tekst = rs.getClob("TEKST");
Blob figuur = rs.getClob("FIGUUR");
pstmt.setClob(2,tekst);
pstmt.setBlob(3,figuur);
...
```

Ook hier zijn de variabelen *tekst* en *figuur* verwijzingen naar SQL-objecten in de databank. Deze verwijzingen blijven bestaan zolang de transactie waarin ze aangemaakt werden niet afgelopen is. Het uitvoeren van het *prepared statement* zal de tekst en de binaire data van de figuur kopieren naar de tabel *JAAROVERZICHT*.

8.11.2 *Blob*, *Clob* en *Array* zijn referenties

Merk op dat de *Blob*, *Clob*, *Array*-objecten referenties zijn naar objecten in de gegevensbank. Deze objecten kunnen dus gemanipuleerd worden zonder dat hun gegevens naar de Java-applicatie (en dus de clientcomputer) overgebracht worden. Dit kan de performantie behoorlijk verbeteren.

Wil je de gegevens van deze objecten toch kunnen manipuleren in de Java-applicatie, dan moeten die binnengehaald worden. Wil je bv. een *Array*-object kunnen behandelen als een gewone Java *array* dan moet je het *Array*-object converteren naar een Java *array*.

De interfaces *Blob*, *Clob* en *Array* voorzien in methodes om de gegevens van de SQL-objecten om te zetten naar Java-objecten. De gegevens van een *Blob*-object kunnen als een *array* van bytes of als een *stream* binnen gehaald worden. Een *Clob*-object kan gematerialiseerd worden door een *stream* of een string. Het *Array*-object kan omgezet worden naar een Java-*array* of een *ResultSet*-object. Hier volgt een overzicht van een aantal van de beschikbare methodes.

Blob

```
public InputStream getBinaryStream()
     throws SQLException;
public byte[] getBytes(long pos, int length)
    throws SQLException;
```

```
public InputStream getAsciiStream()
    throws SQLException;
public Reader getCharacterStream()
    throws SQLException;
public String getSubString(long pos, int length)
    throws SQLException;
```

Clob

Array

```
public Object getArray() throws SQLException;
public ResultSet getResultSet() throws SQLException;
```

In het bovenstaande voorbeeld zou het volgende stukje code gebruikt kunnen worden om de meetgegevens op te halen.

```
Array meting = rs.getArray("METINGEN");
double[] meetwaarden = (double[]) metingen.getArray();
```

8.11.3 Gestructureerd SQL-types

In SQL3 kan men nieuwe gestructureerde gegevenstypes declareren. Deze types bestaan uit één of meerdere leden (ook wel attributen genoemd) die van een bepaald type zijn. De types van deze attributen kunnen bestaande types zijn of zelf gemaakte types. De volgende SQL-opdracht creëert een nieuw SQL-type *ADRESTYPE* met vijf attributen.

```
CREATE TYPE ADRESTYPE  (
    STRAAT VARCHAR(40),
```

```
    NR INTEGER,
    POSTCODE CHAR(4),
    GEMEENTE VARCHAR(40),
    LAND VARCHAR(40)
)
```

Het resultaat van een zoekopdracht die één of meerdere rijen van het type *ADRESTYPE* bevat kan dan als volgt opgehaald worden. Hierbij is *ADRES* de naam van een kolom in de *result set*.

```
while (rs.next()) {
    Struct adres = (Struct)rs.getObject("ADRES");
    Object[] attr = adres.getAttributes();
    ...
}
```

Het resultaat van de methode *getObject* is een *Object* en moet naar een *Struct* geconverteerd worden om de attributen op te vragen. Met de methode *getAttributes* worden de attributen opgehaald. Merk op dat het *Struct*-object geen verwijzing is naar een gegevensbankobject en dus wel de waarden van zijn attributen bevat.

Het is ook mogelijk om een gestructureerd SQL-type naar een zelfgemaakte Java-klasse om te zetten. Daarvoor moet er een relatie gelegd worden tussen het SQL-type met zijn attributen en een Java-klasse met zijn leden. Voor meer informatie zie The JDBC Tutorial: Chapter 3 - Advanced Tutorial of JDBC 3.0 Getting Started - Custom Mapping

8.11.4 Het *DISTINCT*-type

Een andere manier om nieuwe types te maken in SQL3 is het maken van nieuwe types op basis van bestaande types. Zo zouden we in het voorbeeld uit §8.11.3 een type *POSTCODETYPE* kunnen declareren om te gebruiken bij de definitie van het type *ADRESTYPE*.

```
CREATE TYPE POSTCODETYPE AS CHAR(4)
```

De types op basis waarvan een *DISTINCT*-type gemaakt wordt, mogen geen zelfgedefinieerde types zijn zoals in §8.11.3.

Aangezien het type *POSTCODETYPE* afgeleid is van het type *CHAR* kan je de methodes *getString*, *setString* en *updateString* van de interfaces *ResultSet* of *PreparedStatement* gebruiken om een parameter van het SQL-type *POSTCODETYPE* in te stellen of op te halen.

```
    String postcode = rs.getString("POSTCODE");
    pstmt.setString(1,postcode);
```

De variabele *rs* is een *ResultSet*-object met een kolom *POSTCODE* van het type *POSTCODETYPE*. De postcode die in de eerste opdracht bepaald wordt, wordt dan gebruikt in een *prepared statementpstmt* met een parameter van het SQL-type *POSTCODETYPE*.

8.11.5 SQL-referenties naar gestructureerd SQL-objecten.

Indien in een gegevensbank een bepaald gestructureerd object frequent voorkomt, dan kan het zinvol zijn om niet telkens het object zelf op te nemen in de cel van een tabel, maar om een verwijzing (ook wel referentie genoemd) te maken naar het object. Het object is dan maar één keer aanwezig in de gegevensbank. Telkens wordt er dan naar dit object verwezen. Deze verwijzingen zijn van het SQL-type *REF*. Het corresponderende Java-type is *Ref*.

De methodes om parameters van het type *Ref* op te halen, aan te passen of in te stellen zijn *setRef*, *getRef* en *updateRef*. Aangezien het SQL-type *REF* een referentie is, is het Java-type *Ref* ook een verwijzing en wordt er dus geen object binnengehaald in de Java-applicatie.

8.12 Het gebruik van *DataSource*

Tot nu toe maakten we een *Connection*-object aan m.b.v. een *DriverManager* (zie §8.3). Voor meer professionele applicaties is er een beter alternatief, namelijk het gebruik van een *DataSource*-object. Eén van de voordelen van deze aanpak is dat de naam van de driverklasse niet in de code opgenomen moet worden. Een tweede voordeel is dat *connection pooling* (zie §8.12.3) en gedistribueerde transacties (zie §8.12.4) mogelijk zijn.

Aangezien het gebruik van een *DataSource* het gebruik van de JNDI API (zie §8.12.1) impliceert bespreken we eerst kort deze API. Vervolgens bekijken we hoe een *DataSource*-object aangemaakt wordt (zie §8.12.2). Tot slot komen *connection pooling* (zie §8.12.3) en gedistribueerde transacties (zie §8.12.4) aan bod.

Het aanmaken van een *DataSource*-object maakt gebruik van de JNDI-API. Die wordt dus eerst kort toegelicht.

8.12.1 De JNDI API

De JNDI API voorziet in een aantal interfaces en klassen om in een Java-applicatie gebruik te maken van *naming* en *directory services*. Deze API is zo opgebouwd dat hij onafhankelijk is van een specifieke *service*. Op deze manier wordt het aanspreken van bestaande *naming* en *directory services* in een Javaprogramma mogelijk. Bovendien kan van deze API gebruik gemaakt worden om Java-objecten die gedeeld worden door verschillende applicaties, te bewaren en op te zoeken.

De basispakketten van de JNDI API zijn *javax.naming* en *javax.naming.directory*. In deze paragraaf bespreken we geen klassen of interfaces van de JNDI API. Hun functionaliteit wordt besproken in §8.12.2.

Naming service

Het verbinden van verstaanbare namen met computerobjecten is de taak van een *naming service*. Het bestandssysteem en het *Domain Name System* zijn voor de hand liggende voorbeelden van een *naming service*. Bestandsnamen zijn verbonden met *handlers* die toegang verschaffen tot het eigenlijke bestand. DNS-namen zijn gelinked aan IP-adressen van computers. De *naming service* bewaart niet alleen de verbindingen tussen de namen en de objecten, maar biedt bovendien de mogelijkheid om een object op te zoeken of terug te vinden op basis van zijn naam.

De *naming service* bepaalt de syntax waaraan de gebruikte namen moeten voldoen. Dit wordt ook de naamconventie van de *naming service* genoemd.

Een *naming service* bestaat uit een aantal contexten met dezelfde naamconventie. Een *context* bestaat uit een aantal naam/object-verbindingen en heeft een eigen naamconventie. Een context heeft een opzoekactie om object terug te vinden aan de hand van zijn naam. Een naam kan verbonden zijn met een andere context met dezelfde naamconventie. Deze context noemen we dan de subcontext van de huidige context. Op deze manier kan een *naming service* opgebouwd worden uit verschillende contexten. In een bestandssysteem zijn de mappen (directory's) de contexten.

Directory service

Een *naming service* kan uitgebreid worden met een *directory service*. Die associeert attributen met de objecten in de *naming service*. Een attribuut bestaat uit een naam en mogelijke waarden.

Een *directory service* maakt het mogelijk om objecten te zoeken op basis van hun attributen ipv op naam. Uiteraard voorziet deze service ook in acties om de attributen te veranderen, aan te maken en te verwijderen.

De hiërarchische structuur van de *directory service* correspondeert met de contexten in de *naming service*.

8.12.2 Een *DataSource*-object aanmaken

Een *DataSource*-object is een fabriek (*factory*) voor *Connection*-objecten. De gegevensbron waarmee de verbindingen gelegd worden, kan zowel een gegevensbankserver zijn als bijvoorbeeld een bestand. De interface *DataSource* behoort tot het pakket *javax.sql* en maakt deel uit van de JDBC 2.0 API.

Een *DataSource*-object dat we willen gebruiken om een verbinding te maken met een gegevensbank moet geregistreerd zijn bij een *naming service*. Met behulp van de JNDI API kan dit object dan opgevraagd worden.

Dit betekent dat het Java-framework (ook wel *application server* genoemd) waarin jouw applicatie draait een *naming service* moet hebben. Bovendien moet er een *DataSource*-object geregistreerd zijn bij die *naming service*. Dat is de taak van de systeembeheerder van de *application server*.

Deze configuratie bestaat uit de volgende stappen:

***DataSource*-klasse instellen** Je hebt een JDBC 2.0 driver nodig die *data sources* ondersteund. De klassenaam van de *data source* van de gekozen driver moet opgegeven worden zodat een object van deze klasse aangemaakt kan worden door een tool van de *application server*.

Eigenschappen *DataSource* instellen De eigenschappen (*properties*) van het *DataSource*-object moet ingesteld worden. Dit kan onder andere het volgende zijn: de naam van de gegevensbank, de naam van de gegevensbankserver, een beschrijving, ... Al deze eigenschappen corresponderen normaal met *get/set*-methodes van de *DataSource*-klasse van de driver.

***DataSource* registreren** Tot slot moet het aangemaakte *DataSource*-object geregistreerd worden bij de *naming service* gebruikt door de *application server*. De naam die de systeembeheerder hier opgeeft kan dan in de Java-applicatie gebruikt worden om het *DataSource*-object op te halen.

Na het uitvoeren van de bovenstaande stappen is het *DataSource*-object beschikbaar voor de applicatie. Het opzoeken van dit object en het opvragen van een verbinding met de gegevensbank verloopt nu als volgt:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/mijnDB");
Connection conn = ds.getConnection("login","wachtwoord");
```

De interface *Context* en de klasse *InitialContext* maken deel uit van JNDI API en behoren tot het pakket *javax.naming*. De interface *Context* stelt een context van de *naming service* voor (zie §8.12.1). De klasse *InitialContext* stelt de context voor waarin de opdrachten van de *naming service* zoals opzoeken, registreren, ... starten. Deze context wordt bepaald omgevingsvariabelen of configuratiebestanden.

De string *"jdbc/mijnDB"* is de naam waarmee de systeembeheerder het *DataSource*-object registreerde. Met de methode *getConnection* van de interface *javax.sql.DataSource* kan dan een verbinding met de gegevensbron opgevraagd worden. Er zijn twee versies van deze methode: één zonder argumenten en één waarmee de loginnaam en het wachtwoord van de gegevensbankgebruiker meegegeven worden.

```
public Connection getConnection() throws SQLException;
public Connection getConnection(String loginnaam,
    String wachtwoord) throws SQLException;
```

Het voordeel van deze werkwijze op het gebruik van de *DriverManager* is dat er veel eenvoudiger van *datasource*, van JDBC-driver, van gegevensbank, ... veranderd kan worden. Deze veranderingen hebben namelijk geen invloed op de applicatiecode. Enkel de configuratie moet aangepast worden.

Een tweede voordeel is dat op deze manier *connection pooling* en gedistribueerde transacties mogelijk zijn. Producenten van JDBC-drivers hebben keuze uit drie types implementaties van de interface *DataSource*.

basis De verbindingen met de gegevensbank ondersteunen geen *connection pooling* en geen gedistribueerde transacties. Deze verbindingen zijn gelijkaardig als die verkregen via een *DriverManager*.

ondersteunt *connection pooling* De verbindingen met de gegevensbank maken automatisch gebruik van *connection pooling*.

ondersteunt gedistribueerde transacties De verbindingen kunnen gebruik maken van gedistribueerde transacties en meestal maken ze ook gebruik van *connection pooling*.

8.12.3 *Connection Pooling*

Het herhaaldelijk openen en sluiten van verbindingen met de gegevensbank kan in grote applicaties een overlast zijn en dus performantieverlies betekenen. Een oplossing hiervoor is *connection pooling*. Dit betekent dat er een aantal verbindingen met de gegevensbank beschikbaar zijn en dat als de applicatie een verbinding nodig heeft ze één van de beschikbare verbindingen kan gebruiken. Als de opdrachten in de gegevensbank uitgevoerd zijn wordt de verbinding weer ter beschikking gesteld.

Voor de ontwikkelaar van de applicatie verandert er echter nauwelijk iets. Het volgende stukje code illustreert dit.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/mijnDB");
try {
    Connection conn = ds.getConnection("login","wachtwoord");
    try {
        ...
    } catch (...) {
        ...
    } finally {
         if (conn != null)
              conn.close();
    }
} catch (SQLException e) {
    ...
}
```

De methode *getConnection* maakt in het geval dat het *DataSource*-object *connection pooling* ondersteund, geen nieuwe verbinding aan met de gegevensbank maar geeft een referentie naar één van de beschikbare verbindingen uit de *pool*. Het oproepen van de methode *close* sluit de verbinding niet af, maar stelt ze weer ter beschikking.

Opdat *connection pooling* mogelijk zou zijn, moet de configuratie zoals beschreven in §8.12.2 uitgebreid worden. Niet alleen moeten de klassenaam van de *data source*, zijn eigenschappen en zijn naam in de *naming service* opgegeven worden, de configuratie moet ook een *ConnectionPoolDataSource*-object beschrijven en registreren. Dit object maakt de verbindingen van de *connection pool* aan. Meer informatie vind je in [Fis99].

8.12.4 *Gedistribueerde transacties*

Gedistribueerde transacties zijn transacties die data manipuleren in verschillende gegevensbronnen. De acties op die verschillende gegevensbronnen vormen een eenheid die ofwel volledig uitgevoerd moet worden of wel niet uitgevoerd. In de JDBC 2.0 API is het mogelijk om *Connection*-objecten aan te maken die deel zijn van een gedistribueerde transactie. Dit kan enkel als er gebruik gemaakt wordt van een *DataSource*-object.

Applicaties die gebruik maken van gedistribueerde transacties scheiden de code die de acties op de gegevensbank voorstellen van de code die de transactie maakt, beheert en uitvoert. Dit betekent dat in de JDBC-code geen spoor van transacties te vinden zal zijn. Verschillende methodes met JDBC-code zullen samen genomen worden in een transactie.

Hoofdstuk 9

C# in een notedop

In de volgende hoofdstukken zullen we een aantal aspecten van het .NET-platform behandelen. We maken hierbij gebruik van de object-georiënteerde programmeertaal C# (C sharp uitgesproken). Dit hoofdstuk bevat een beknopt overzicht van C# en de gelijkenissen en verschillen met Java.

9.1 Gegevenstypes

De declaratie en toekenning van een variabele heeft in C# dezelfde syntax als in Java of C++.

```
type naamVariabele;
naamVariabele = waarde;
```

of verkort

```
type naamVariabele = waarde;
```

C# heeft twee soorten gegevenstypes: *waarde-types* en *referentie-types*. Het verschil tussen beide types is dat variabelen van het eerste type hun data bevatten en variabelen van het tweede type een verwijzing naar een object.

9.1.1 Waarde-types

Voorgedefinieerde waarde-types zijn onder andere: *int*, *long*, *float*, *double*, *bool*, *char*, ... De volgende tabel geeft een beknopt overzicht van deze types.

int	Geheel getal (32 bits)
long	Geheel getal (64 bits)
float	Vlottendekommagetallen met enkelvoudige precisie
double	Vlottendekommagetallen met dubbele precisie
bool	Logisch type kan waarde *true* of *false* aannemen
char	16 bits unicode letterteken

Met een *struct* (zie §9.6)- of *enum* (zie §9.4)-declaratie is het mogelijk om nieuwe waarde-types te definiëren.

Voorbeelden

```
int geheleWaarde = 257;
long langeWaarde = 2200000;
float fWaarde = 2.15 ;
double dWaarde = 4.5789 ;
bool gelijk = false;
char letter = 'k';
```

9.1.2 Referentie-types

In C# zijn er twee voorgedefinieerde referentie-types: *object* en *string*. Het type *object* is het basistype voor alle andere types, ook voor de waarde-types. Om unicode-strings voor te stellen wordt het type *string* gebruikt. Variabelen van het type *string* zijn *immutable*. Dit betekent dat de waarde van een *string*-variabele niet veranderd kan worden. Om een string aan te passen wordt een nieuw object aangemaakt.

Nieuwe referentie-types worden gecreëerd met behulp van *klasse* (zie §9.5)-, *interface* (zie §9.8)- en *delegate*-declaraties.

Voorbeelden

```
string naam = "Pieter-Jan";
object iets = null;
```

Merk op dat het sleutelwoord *null* gebruikt wordt voor een lege verwijzing.

Tabellen

Een één-dimensionale tabel wordt als volgt gedeclareerd

```
type[] tabel;
```

waarbij *type* het type voorstelt van de elementen in de tabel. De volgende pseudocode illustreert het aanmaken van een tabel van lengte *N*. *N* is een positieve gehele waarde en *type* is het type van de elementen in de tabel.

```
type[] tabel = new type[N];
```

Enkele voorbeelden:

```
int[] gehelen = new int[3];
gehelen[0] = 15;
gehelen[1] = -231;
gehelen[2] = 3;

int[] evenCijfers = new int[5]{0,2,4,6,8};

int[] onevenCijfers = {1,3,5,7,9};
```

De opdracht *int[] onevenCijfers* = {*1,3,5,7,9*}*;* is een verkorte notatie voor *int[] onevenCijfers* = *new int[5]*{*1,3,5,7,9*}*;*.

De volgende voorbeelden illustreren het gebruik van meerdimensionale tabellen.

```
int[][] getallen = new int[3][];
getallen[0] = new int[1]{15};
getallen[1] = new int[4]{-231,48,97,-2};
getallen[2] = new int[2]{3,56};

char[][][] letters = new char[2][][];
letters[0] = new char[3][];
letters[0][0] = new char[3]{'d','i','t'};
letters[0][1] = new char[2]{'i','s'};
letters[0][2] = new char[5]{'g','r','o','e','n'};
letters[1] = new char[1][];
letters[1][0] = new char[3];
letters[1][0][0] = 'a';
letters[1][0][1] = 'x';
letters[1][0][2] = 'p';
```

De syntax voor rechthoekige tabellen is iets eenvoudiger.

```
double [,] coord2D = {{1.3,-5.2},{1.0,4.5},{0.284,-1.003}};
```

```
double [,,] coord3D = new double[4,3,2];
for (int i = 0; i < 4; i++)
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 2; k++)
            coord3D[i,j,k] = 1;
```

9.1.3 Opmerkingen

Elk voorgedefinieerd type is een verkorte notatie van een *struct* (zie §9.6) uit de basisbibliotheek. De sleutelwoorden *int*, respectievelijk *string*, zijn bijvoorbeeld een verkorte vorm van *System.Int32*, respectievelijk *System.String*. De standaard programmeerstijl voor C# verkiest het gebruik van de verkorte notatie.

Elk type is afgeleid van het type *object*. Dit betekent dat de methodes van *object* methodes zijn voor alle andere types, ook voor waarde-types. De volgende uitdrukkingen zijn dus correct.

```
int geheleWaarde = 893;
string stringWaarde = geheleWaarde.ToString();
stringWaarde = 893.ToString();
```

9.1.4 Conversies

Voor de voorgedefinieerde waarde-types van C# bestaan conversies die een variabele van het ene type omzetten in een variabele van het andere type. Indien deze conversie steeds op een veilige manier kan gebeuren (bv. van *int* naar *long*) dan noemen we dit een *impliciete* conversie, in het andere geval spreken we van een *expliciete* conversie. In de syntax van C# hoef je een impliciete conversie niet aan te duiden, een expliciete conversie daarentegen duid je aan met een cast-uitdrukking.

Het volgende voorbeeld illustreert het gebruik van een impliciete en een expliciete conversie tussen een variabele van het type *int* en een variabele van het type *long*.

```
int geheleWaarde = 257;
long langeWaarde = geheleWaarde; // impliciete conversie

langeWaarde = 2200000;
geheleWaarde = (int) langeWaarde; // expliciete conversie
```

Het converteren van een variabele van een waarde-type naar een variabele van het type *object* en omgekeerd wordt respectievelijk *boxing* en *unboxing* genoemd.

```
int geheleWaarde = 257;
object objWaarde = geheleWaarde; // boxing
int geheel = (int) objWaarde; // unboxing
```

Merk op dat boxing een veilige conversie is en dus geen cast-uitdrukking nodig heeft.

Bij boxing wordt de waarde van de variabele gekopieerd in een nieuw gecreëerd object. Veranderingen aan dit object impliceren dus *geen* veranderingen aan de inhoud van de oorspronkelijke variabele.

```
int geheleWaarde = 257;
object objWaarde = geheleWaarde;
objWaarde = 342;
```

Na het uitvoeren van bovenstaande code is de waarde van *geheleWaarde* nog steeds *257*.

Boxing en unboxing maakt het mogelijk om methodes waarbij het argument of de waarde een object zijn, te gebruiken voor waardetypes. Een methode *Methode* met de volgende signatuur

```
public object Methode(object o)
```

zou bijvoorbeeld als volgt gebruikt kunnen worden:

```
int geheleWaarde, geheel;
geheleWaarde = -3654;
geheel = (int) Methode(geheleWaarde);
```

De laatste uitdrukking maakt dus zowel gebruik van boxing (om *geheleWaarde* om te vormen naar een object) als unboxing (om het resultaat van *Methode* te converteren naar een *int*-waarde).

9.2 Uitdrukkingen

De operatoren die je kan gebruiken om uitdrukkingen te maken zijn gelijkaardig aan die in C++ of Java. De tabel geeft een overzicht van een aantal bewerkingen. De operatoren zijn gerangschikt volgens prioriteit. Diegene die eerst voorkomen in de tabel hebben de hoogste prioriteit.

!	negatie
$*$	vermenigvuldiging
$/$	deling
$\%$	rest bij deling
$+$	optelling
$-$	aftrekking
$<$	kleiner dan
$>$	groter dan
$<=$	kleiner dan of gelijk aan
$>=$	groter dan of gelijk aan
$==$	gelijk aan
$!=$	niet gelijk aan
$\&\&$	en
$\|\|$	of
$=$	toekenning
$*=$	vermeningvuldig en toekenning
$/=$	deling en toekenning
$\%=$	rest bij deling en toekenning
$+=$	optelling en toekenning
$-=$	verschil en toekenning

Het resultaat van deze operatoren hangt af van het type van de gebruikte argumenten. Zo zal het resultaat van *4/5* de waarde *0* zijn, terwijl het resultaat van *4.0/5.0* de waarde *0.8* is.

9.2.1 Vergelijken van strings

In C# worden twee *strings* als gelijk beschouwd indien ze evenlang zijn en op elke positie hetzelfde karakter hebben staan, of als ze beide *null* zijn. Twee variabelen van het type *object* zijn gelijk als ze naar hetzelfde object verwijzen.

```
string woord1 = "Test";
string woord2 = string.Copy(woord1);
bool gelijkString = woord1==woord2; // true
bool gelijkObject = (object)woord1==(object)woord2; // false
```

In de bovenstaande code heeft de logische variabele *gelijkString* de waarde *true* en de logische variabele *gelijkObject* de waarde *false*, omdat de twee variabelen hetzelfde woord voorstellen, maar niet naar hetzelfde object verwijzen.

9.3 Programmastructuren

In deze sectie geven we een kort overzicht in pseudocode van een aantal programmastructuren in C#. De meeste van deze structuren zijn hetzelfde als in C++ of Java.

9.3.1 Selectie

De *if-else*-opdracht realiseert een selectie op basis van een logische uitdrukking.

```
if (voorwaarde) {
    ... // uit te voeren opdrachten als 'voorwaarde' waar is
} else {
    ... // uit te voeren opdrachten als 'voorwaarde' onwaar is
}
```

Hierbij is *voorwaarde* een logische uitdrukking van het type *bool*. De *else*-clausule mag weggelaten worden. Indien de uit te voeren opdrachten maar één opdracht omvat mogen de accolades weggelaten worden. Indien het *else*-blok enkel uit een *if* - of een *if-else*-opdracht bestaat gebruiken we de volgende verkorte notatie.

```
if (voorwaarde1) {
    ... // uit te voeren opdrachten als 'voorwaarde1' waar is
} else if (voorwaarde2) {
    ... // uit te voeren opdrachten als 'voorwaarde2' waar is
} else {
    ... /* uit te voeren opdrachten als 'voorwaarde1' en
         'voorwaarde2' onwaar zijn */
}
```

De *switch*-opdracht maakt een selectie op basis van de waarde van een uitdrukking mogelijk.

```
switch (uitdrukking) {
    case waarde1:
        ... /* uit te voeren opdrachten
                als de waarde van 'uitdrukking' 'waarde1' is */
       break;
    case waarde2:
        ... /* uit te voeren opdrachten
                als de waarde van 'uitdrukking' 'waarde2' is */
        break;

    ... // mogelijke andere case-structuren

    default:
        ... /* uit te voeren opdrachten als de waarde van
                'uitdrukking' geen enkele van de
           bovenstaande waarden ('waarde1', 'waarde2', ...) is */
        break;
  }
```

Het type van *uitdrukking* moet impliciet converteerbaar zijn naar een beperkt aantal voorgedefinieerde types (o.a. *int*, *long*, *char* en *string*) of naar een *enum* (zie §9.4)-type. De waarden *waarde1*, *waarde2*,

... zijn constante uitdrukkingen van een type dat impliciet converteerbaar moet zijn naar het type van *uitdrukking*. De *default*-optie mag weggelaten worden.

9.3.2 Herhaling

Herhaling afhankelijk van de waarde van een logische uitdrukking (*voorwaarde* in de pseudocode) wordt gerealiseerd door een *while*- opdracht.

```
while (voorwaarde) {
    ... // uit te voeren opdrachten zolang 'voorwaarde' waar is
}
```

Een herhaling die een welbepaald aantal keer uitgevoerd moet worden kan gebruik maken van een *for*-opdracht.

```
for (initialisatie; voorwaarde; iteratie) {
    ... // uit te voeren opdrachten zolang 'voorwaarde' waar is
}
```

In deze structuur zijn *initialisatie* en *iteratie* commando's en is *voorwaarde* een logische uitdrukking.

Om voor een aantal elementen van een verzameling een reeks opdrachten uit te voeren kan je gebruik maken van de *foreach*-opdracht.

```
foreach (type element in verzameling) {
    ... /* uit te voeren opdrachten voor elk element 'element'
             in 'verzameling' */
}
```

Hierbij is *type* het type van de variabele *element*. Het type van de variabele *verzameling* moet de interface *System.Collections.IEnumerable* implementeren. Bovendien mag de verwijzing (pointer) die de iteratie-variabele *element* bevat niet gewijzigd worden in de *foreach*-lus.

9.3.3 Uitzonderingen

Het gooien van een uitzondering verloopt als volgt:

```
throw uitzondering;
```

Het type van de variabele *uitzondering* moet *System.Exception* of een afgeleide klasse van *System.Exception* zijn.

Met behulp van een *try-catch-finally*-structuur kan je uitzonderingen opvangen.

```
try {
    ...
} catch (Exception1 e1) {
    ...
} catch (Exception2 e2) {
    ...

... // eventueel andere catch-blokken

} finally {
    ...
}
```

Exception1 en *Exception2* zijn de types van de variabelen *e1* en *e2*. Beide types zijn *System.Exception* of een hiervan afgeleid type. De *catch*-blokken worden in volgorde van voorkomen afgehandeld. Dit betekent dat de types van de uitzonderingen geen afgeleide klassen mogen zijn van die types die vroeger in een *catch*-blok voorkwamen. De *catch*-blokken en het *finally*-blok zijn beiden optioneel, slechts één van beiden moet voorkomen.

9.4 Enum

Het *enum*-type wordt gebruikt om een type te declareren dat bestaat uit verwante symbolische constanten. De declaratie van een *enum*-type heeft de volgende vorm:

```
enum NaamType {SymCst1, SymCst2, ..., SymCstn}
```

NaamType is de naam van het nieuwe type, *SymCst1*, *SymCst2*, ... de symbolische constanten waaruit het type bestaat. Het onderliggende type van zo'n *enum*-declaratie is *int*. Dit betekent dat elke symbolische constante een geassocieerde gehele waarde heeft. Met de uitdrukking *NaamType.SymCsti* kan je de waarde van de symbolische constante verkrijgen. Standaard krijgt de eerste symbolische constante de waarde 0, de tweede waarde 1, enzovoort. Het is ook mogelijk om dit zelf in te stellen.

Een aantal voorbeelden:

```
enum Vorm {Cirkel, Vierkant, Rechthoek, Driehoek}

enum Dag {Maandag, Dinsdag, Woensdag, Donderdag, Vrijdag,
    Zaterdag, Zondag}
```

```
enum Kleur {Geel, Rood, Blauw, Groen}
```

Het *enum*-type kan in combinatie met een *switch*-opdracht gebruikt worden, indien uit een vast aantal mogelijkheden gekozen moet worden.

```
Vorm mijnFiguur = Vorm.Driehoek;
switch (mijnFiguur) {
    case Vorm.Cirkel:
        ... // Teken cirkel
        break;
    case Vorm.Vierkant:
        ... // Teken vierkant
        break;
    case Vorm.Rechthoek:
        ... // Teken rechthoek
        break;
    case Vorm.Driehoek:
        ... // Teken driehoek
        break;
}
```

Ook in een *foreach*-opdracht kan een *enum*-type gebruikt worden. Het onderstaande stukje code illustreert ook dat het onderliggend type van een *enum*-type een *int* is.

```
Vorm[] verzameling = {Vorm.Vierkant, Vorm.Driehoek};
foreach (Vorm figuur in verzameling) {
    System.Console.WriteLine(figuur);
    System.Console.WriteLine((int) figuur);
}
```

De uitvoer van dit stukje code is

```
Vierkant
1
Driehoek
3
```

9.5 Klassen

Met behulp van klassedeclaraties kan je nieuwe referentietypes definiëren. Klassen kunnen afgeleid zijn van andere klassen (§9.12) of interfaces (zie §9.8) implementeren. Mogelijke leden van een klasse zijn

- constanten
- velden
- constructoren
- methodes (zie §9.7)
- eigenschappen (zie §9.9)
- indexen (zie §9.10)
- inwendige klassen
- gebeurtenissen
- operatoren
- destructoren

De bereikbaarheid van deze leden wordt bepaald door een sleutelwoord dat de toegang bepaalt. Mogelijke waarden zijn:

public	Leden van alle klassen hebben toegang.
protected	Enkel toegang voor leden van de klasse die het lid bevat en daarvan afgeleide klassen.
internal	Toegang beperkt tot de applicatie waartoe de klasse behoort.
protected internal	Combinatie van de twee voorgaande.
private	De toegang is beperkt tot de klasse die het lid bevat. Dit is de standaardtoegang als er geen toegang gespecifieerd is.

9.5.1 Declaratie van een klasse

Een klassedeclaratie heeft de volgende structuur. Hierbij kan *toegang* onder andere de waarden *private*, *protected*, *internal* en *public* aannemen.

```
[toegang] class NaamKlasse {
  ... // declaratie van de leden
}
```

9.5.2 Declaratie van een constante

De syntax van een constantedeclaratie is de volgende:

```
[toegang] const type naamConstante;
```

type is het type van de constante, *naamConstante* de naam van de constante.

9.5.3 Declaratie van een veld

De syntax van een velddeclaratie is de volgende:

```
[toegang] [static] [readonly] type naamVeld;
```

type is het type van het veld, *naamVeld* de naam van het veld. Het sleutelwoord *static* geeft aan of het al dan niet om een statisch veld gaat. Statische leden zijn leden van de klasse en niet van de instantiaties van de klassen (objecten). In C# kan je de statische leden van een klassen niet oproepen op een instantiatie van de klasse. Velden die *readonly* gedeclareerd zijn moeten geïntialiseerd worden bij declaratie of in de constructor. Nadien kunnen ze niet meer gewijzigd worden.

9.5.4 Declaratie van een constructor

Een constructor definieert alle opdrachten die nodig zijn om een object van het type van de klasse aan te maken en te initialiseren.

```
[toegang] NaamKlasse(argumentenlijst);
```

De constructor heeft dezelfde naam als de klasse. Een constructor kan een argumentenlijst hebben. Een argumentenlijst heeft de volgende syntax:

```
typeParameter1 param1, typeParameter2 param2, ... ,
    typeParameterN paramN
```

Hierbij is *typeParameteri* het type van de i-de parameter en *parami* de naam.

9.6 Struct

Met behulp van *struct*s kan je in C# nieuwe waardetypes declareren. Structs kunnen dezelfde leden hebben als klassen en kunnen interfaces implementeren. Ze kunnen echter niet afgeleid worden of afgeleid zijn van een andere struct. De syntax van een *struct*:

```
struct NaamStruct {
   ... // declaratie leden
}
```

Een struct kan geen constructor zonder argumentenlijst hebben. Deze is reeds voorzien en geeft alle leden hun standaardstatus. Bijvoorbeeld 0 voor een *int*.

9.7 Methodes

De syntax van een methodedeclaratie is

```
[toegang] [static] ReturnType NaamMethode(parameterLijst) {
    .. // uit te voeren opdrachten
}
```

Net zoals bij statische velden, zijn statische methodes, methodes van de klasse en niet van de instantiaties (objecten). *ReturnType* is het type van het resultaat van de methode of *void* als de methode een procedure is. De parameterLijst is optioneel en heeft de volgende vorm:

```
typeParameter1 param1, typeParameter2 param2, ... ,
    typeParameterN paramN
```

Hierbij is *typeParameteri* het type van de i-de parameter en *parami* de naam.

9.7.1 Referentie- en uitvoerparameters

Bij parameters van het type referentietype wordt steeds de referentie naar het object meegegeven aan de methode. De referentie kan niet gewijzigd worden door de methode, maar het object waarnaar de referentie verwijst kan dat wel. Bij parameters van het type waardetype wordt de waarde van de oorspronkelijke variabele gekopieerd en meegegeven aan de methode. De methode kan de oorspronkelijke waarde dus niet veranderen. Om dit toch te kunnen verwezenlijken moet je gebruik maken van referentie- of uitvoerparameters.

Een referentieparameter wordt gekenmerkt door het sleutelwoord *ref* voor het type in de parameterlijst en voor de naam van de variabele in de methode-oproep. We illustreren dit aan de hand van een voorbeeld. De methode *reset* stelt de meegegeven gehele parameter gelijk aan 0.

```
void reset(ref int getal) {
    getal = 0;
}
```

Na het uitvoeren van de volgende opdrachten zal de variabele *getal* de waarde 0 bevatten.

```
int getal = 10;
reset(ref getal);
```

Een uitvoerparameter is een speciale referentieparameter en wordt gekenmerkt door het sleutelwoord *out*. Hij hoeft niet geïnitialiseerd te worden voor de methode-oproep. Dit wordt geïllustreerd in het volgend voorbeeld. De methode met een uitvoerparameter:

```
void init(out int waarde) {
    waarde = 0;
}
```

Dit stukje code toont het gebruik van de bovenstaande methode:

```
int waarde;
init(out waarde);
```

9.7.2 Tabel van parameters

In C# is het mogelijk om een methode te definiëren die nul of meer argumenten van hetzelfde type heeft. Dit kan op de volgende manier:

```
  type1 naamMethode (params type2[] naamParam) {
  }
```

Hierbij is *type1* het gegevenstype van de waarde van de methode, *naamMethode* de naam van de methode, *type2* het gegevenstype van de argumenten en *params* een sleutelwoord van C#.

We kunnen nu een functie *Som* maken die de som van een willekeurig aantal reële getallen berekent.

```
double Som(params double[] elementen) {
    double som = 0.0;
    for (int i = 0; i < elementen.Length; i++)
        som += elementen[i];
    return som;
}
```

Deze functie kan je nu op de volgende manier gebruiken:

```
double resultaat;
resultaat = Som();
resultaat = Som(1.0);
resultaat = Som(2.3,2.6);
resultaat = Som(new double[] {1,3.0,4.2});
```

Merk op dat je zowel meerdere argumenten kan hebben als één argument dat een tabel is.

Een methode kan maar één tabel van parameters hebben. Indien de methode nog andere argumenten heeft, dan moet de parameter-tabel het meest rechtste argument zijn.

9.8 Interfaces

Een interface is een soort contract. Dit betekent dat een klasse of struct die een interface implementeert, alle leden van de interface moet uitwerken.

Een interface is een manier om naar een object te kijken. We hoeven enkel die interfaces van het object te kennen die we nodig hebben en niet het eigenlijke type (klasse) van het object.

De syntax van een interface-declaratie is de volgende:

```
[toegang] interface INaamInterface {
    ... // leden interface
}
```

De standaard programmeerstijl voor C# adviseert om de naam van een interface altijd te beginnen met een I (in het voorbeeld *INaamInterface*). Een interface kan methodes (zie §9.7), eigenschappen (zie §9.9), indexers (zie §9.10) en gebeurtenissen bevatten. De interface bevat hiervan enkel de signatuur, de klassen die de interface implementeren bevatten de eigenlijke uitwerking.

In de klasse-, struct- of interfacedeclaratie moet aangegeven worden welke interfaces de klasse implementeert. In het volgend voorbeeld implementeert de klasse *MijnKlasse* de twee interfaces *IInterface1* en *IInterface2*.

```
class MijnKlasse : IInterface1, Interface2 {
    ... // leden, implementeren leden interfaces
}
```

9.9 Eigenschappen

Eigenschappen zijn uitbreidingen van velden en beschrijven kenmerken van een object of klasse. Integenstelling tot een veld hoeft een eigenschap niet verbonden te zijn met een geheugenlocatie. Een

eigenschap heeft een *get*- en een *set*-methode die de opdrachten bevatten die uitgevoerd moeten worden bij het lezen of veranderen van een eigenschap. De syntax om de waarde van een eigenschap op te halen of te veranderen is dezelfde als bij velden. Maar in plaats van de inhoud van een geheugenlocatie te lezen of aan te passen, wordt de respectievelijke *get*- of *set*-methode uitgevoerd.

De declaratie-syntax van een eigenschap is de volgende:

```
[toegang] type NaamEigenschap {
    get {
        ... /* opdrachten uit te voeren bij het
            lezen van de eigenschap*/
    }
    set {
        ... /* opdrachten uit te voeren bij het
            veranderen van de eigenschap*/
    }
}
```

type is het gegevenstype van de eigenschap, *NaamEigenschap* de naam. Indien de *set*-declaratie weggelaten wordt dan is de eigenschap *read-only*. De *get*-declaratie moet een *return*-opdracht bevatten die de waarde van de eigenschap weergeeft.

De waarde die in de *set*-methode aan de eigenschap moet toegekend worden, wordt weergegeven door een impliciete variabele *value*. Dit wordt geïllustreerd in het volgende voorbeeld. De onderstaande klasse *Persoon* heeft een publieke eigenschap *Leeftijd*.

```
public class Persoon {
    private int leeftijd;

    public Persoon(int leeftijd) {
        this.leeftijd = leeftijd;
    }

    public int Leeftijd {
        get {
            return this.leeftijd;
        }

        set {
            if (value >= 0 && value <= 150)
                this.leeftijd = value;
            else
                 throw new Exception("Onmogelijke leeftijd");
        }
    }
}
```

We kunnen nu de eigenschap *Leeftijd* gebruiken alsof het een veld was.

```
// aanmaken van een student
Persoon student = new Persoon(20);
// leeftijd uitschrijven
System.Console.WriteLine(student.Leeftijd);
// leeftijd aanpassen en opnieuw uitschrijven
student.Leeftijd = 21;
System.Console.WriteLine(student.Leeftijd);
```

Een eigenschap hoeft niet verbonden te zijn met een privaat veld. Om dit te illustreren voegen we een *read-only*-eigenschap *Volwassen* toe aan de bovenstaande klasse *Persoon*.

```
public bool Volwassen {
    get {
        if (this.leeftijd < 18)
            return false;
        else
            return true;
    }
}
```

9.10 Indexers

In C# is het mogelijk om klassen te creëren die zich gedragen als virtuele tabellen of hashtabellen. Dit wordt gerealiseerd met behulp van *indexers*. Je kan deze klassen gebruiken alsof het tabellen zijn. Dit betekent dat als je een klasse MijnCol hebt met een indexer van het type *object* en een default-constructor, je de volgende opdrachten kan uitvoeren.

```
MijnCol objMetInd = new MijnCol();
object obj1, obj2 = new object();
obj1 = objMetInd[1];
objMetInd[5] = obj2;
```

Merk op dat *objMetInd* helemaal geen tabel is. Bovendien is het ook mogelijk om als index (i.e. het argument van de operator []) niet enkel natuurlijke getallen te gebruiken, maar ook *strings*.

De syntax van een indexer-declaratie bestaat uit een *get*- en een *set*-deel. Het *get*-gedeelte bestaat uit de opdrachten die uitgevoerd worden als men de waarde van *objMetInd[index]* moet bepalen. Dit stuk bevat bijgevolg dus een *return*-opdracht. De opdrachten die *objMetInd[index]* veranderen worden opgenomen in het *set*-blok. De waarde die toegekend moet worden wordt voorgesteld door de impliciete variabele *value*. De syntax in pseudocode:

```
[toegang] type this[int index] {
    get {
```

```
        ...
    }

    set {
        ...
    }
}

[toegang] type this[string sleutel] {
    get {
        ...
    }

    set {
        ...
    }
}
```

Hierbij stelt *type* het gegevenstype voor. Een klasse kan beide type indexers hebben of maar één van de twee. In deze cursus zullen we zelf niet veel klassen met een indexer aanmaken, maar vooral gebruik maken van klassen uit de Klassenbibliotheek van het .NET-Framework (.NET Framework Class Library). In het volgend voorbeeld gebruiken we de klasse *NameValueCollection* uit de bibliotheek *System.Collections.Specialized*. In de methode *VulOp* worden er drie *strings* toegevoegd aan een *NameValueCollection*. Deze strings worden geassocieerd met de sleutelwoorden 'naam', 'voornaam' en 'onderwerp'.

```
private static void VulOp(NameValueCollection geg,
    string naam,
    string voornaam, string onderwerp) {
geg["naam"] = naam;
geg["voornaam"] = voornaam;
geg["onderwerp"] = onderwerp;
}
```

In de methode *Schrijf* worden alle objecten van een *NameValueCollection* uitgeschreven en vervolgens ook de objecten geassocieerd met de sleutelwoorden 'naam', 'voornaam' en 'onderwerp'.

```
private static void Schrijf(NameValueCollection geg) {
    for (int i = 0; i < geg.Count; i++)
        System.Console.WriteLine(geg[i]);
    System.Console.WriteLine();
    System.Console.WriteLine(geg["naam"]);
    System.Console.WriteLine(geg["voornaam"]);
    System.Console.WriteLine(geg["onderwerp"]);
    System.Console.WriteLine();
}
```

De uitvoer van het volgend programma

```
public static void Main() {
    NameValueCollection geg = new NameValueCollection();

    VulOp(geg,"De Ridder","Jan","Muziek");
    Schrijf(geg);

    VulOp(geg,"Peeters","Piet","Sport");
    Schrijf(geg);
}
```

is dan

```
De Ridder
Jan
Muziek

De Ridder
Jan
Muziek

Peeters
Piet
Sport

Peeters
Piet
Sport
```

9.11 Namespaces

Net zoals packages in Java, worden *namespaces* in C# gebruikt om stukken code (klassen, interfaces, enums, structs, ...) te structureren. Samenhangende klassen, interfaces, enums, structs, ... behoren tot dezelfde *namespace*.

In het volgend voorbeeld wordt een klasse *Test* aangemaakt die behoort tot de namespace *Be.Hogent.Iii*. Dit kan op twee manieren.

```
namespace Be {
    namespace Hogent {
        namespace Iii {
            public class Test { ... }
        }
    }
}
```

of

```
namespace Be.Hogent.Iii {
    public class Test { ... }
}
```

Merk op dat de klasse *Test* niet tot een directorystructuur "Be/Hogent/Iii"moet behoren.

Om de klassen, interfaces, ... van de namespace *Be.Hogent.Iii* te gebruiken zonder expliciet de volledige naam te gebruiken, voeg je boven de klasse-definitie de volgende opdracht toe.

```
using Be.Hogent.Iii;
```

Je kan ook een *alias* maken voor een klasse die je vaak gebruikt. Bijvoorbeeld,

```
using Test = Be.Hogent.Iii.Test;
```

Nu kan je in een programma *Test* gebruiken in plaats van *Be.Hogent.Iii.Test*.

9.12 Afgeleide klassen en abstracte klassen

9.12.1 Afgeleide klassen

C# ondersteunt enkelvoudige overerving. De klasse *object* is de klasse waarvan alle andere klassen afgeleid zijn. In het volgend voorbeeld is *MijnKlasse* een afgeleide klasse van de klasse *BasisKlasse* met één methode *MethodeBK*. *MijnKlasse* neemt de methode *MethodeBK* over van *BasisKlasse* en heeft een eigen methode *MethodeMK*.

```
public class BasisKlasse {
    public void MethodeBK() {
        System.Console.WriteLine("BasisKlasse.MethodeBK");
    }
}

    public class MijnKlasse:BasisKlasse {
        public void MethodeMK() {
            System.Console.WriteLine("MijnKlasse.MethodeMK");
        }
    }
```

Illustratie gebruik

De uitvoer van de volgende opdrachten

```
MijnKlasse mijnObject = new MijnKlasse();
mijnObject.MethodeBK();
mijnObject.MethodeMK();

BasisKlasse basisObject = mijnObject;
basisObject.MethodeBK();

basisObject = new BasisKlasse();
basisObject.MethodeBK();
```

is dan

```
BasisKlasse.MethodeBK
MijnKlasse.MethodeMK
BasisKlasse.MethodeBK
BasisKlasse.MethodeBK
```

Net zoals bij het implementeren van interfaces, wordt het afleiden van een klasse aangeduid met een dubbel punt. Indien een klasse afgeleid is van een andere klasse en één of meerdere interfaces implementeert, dan moeten na de dubbelpunt eerst de klasse en vervolgens de interfaces vermeld worden.

```
public class MijnKlasse : BasisKlasse, IInterface1, IInterface2 {
      ...
}
```

In het bovenstaande voorbeeld kan *MijnKlasse* de methode *MethodeBK* niet overschrijven. Opdat dit mogelijk zou zijn moet de methode *MethodeBK* van *BasisKlasse* de prefix *virtual* hebben. Bovendien moet je in de klasse *MijnKlasse* expliciet vermelden dat je de *MethodeBK* overschrijft. Dit gebeurt met de prefix *override*. Merk op dat in de code het sleutelwoord *base* gebruikt wordt om de klasse aan te duiden waarvan afgeleid wordt. Een afgeleide klasse kan een 'virtuele' methode overschrijven, maar hoeft dit niet te doen.

```
public class BasisKlasse {
    public virtual void MethodeBK() {
        System.Console.WriteLine("BasisKlasse.MethodeBK");
    }
}

public class MijnKlasse:BasisKlasse {
```

```
    public override void MethodeBK() {
         base.MethodeBK();
         System.Console.WriteLine("MijnKlasse.MethodeBK");
    }

    public void MethodeMK() {
        System.Console.WriteLine("MijnKlasse.MethodeMK");
    }
}
```

De uitvoer van de bovenstaande opdrachten (zie §9.12.1) is nu

```
BasisKlasse.MethodeBK
MijnKlasse.MethodeBK
MijnKlasse.MethodeMK
BasisKlasse.MethodeBK
MijnKlasse.MethodeBK
BasisKlasse.MethodeBK
```

Op een analoge manier kunnen eigenschappen van klassen in afgeleide klassen overschreven worden.

De constructor van een afgeleide klasse moet steeds eerst de constructor van basis- of bovenliggende klasse oproepen. Het volgend voorbeeld illustreert hoe dit geïmplementeerd wordt.

```
public class BasisKlasse {
    public BasisKlasse() {
        System.Console.WriteLine("Constructor BasisKlasse");
    }
}

public class MijnKlasse:BasisKlasse {
    public MijnKlasse() : base() {
        System.Console.WriteLine("Constructor MijnKlasse");
    }
}
```

In de hoofding van de constructor van de afgeleide klasse *MijnKlasse* wordt de constructor van de basisklasse *BasisKlasse* aangegeven die opgeroepen moet worden voor het uitvoeren van de constructor van de afgeleide klasse. De constructor van de basisklasse kan uiteraard ook argumenten hebben.

9.12.2 Abstracte klassen

Het volgend voorbeeld illustreert de syntax voor abstracte klassen.

```
public abstract class AbstracteKlasse {
    public abstract void Methode();
}

public class MijnKlasse:AbstracteKlasse {
    public override void Methode() {
        System.Console.WriteLine("MijnKlasse.Methode");
    }
}
```

De uitvoer van de opdrachten

```
MijnKlasse mijnObject = new MijnKlasse();
mijnObject.Methode();
AbstracteKlasse absObject = mijnObject;
absObject.Methode();
```

is

```
MijnKlasse.Methode
MijnKlasse.Methode
```

9.13 Een eerste programma

9.13.1 De methode Main

Een applicatie moet tenminste één klasse hebben met de methode *Main*. Deze methode is het startpunt van de applicatie. Er zijn verschillende mogelijke signaturen voor de methode *Main*, onder andere de twee volgende.

```
public static void Main() {
    ... // uit te voeren opdrachten
}

public static void Main(string[] args) {
    ... // uit te voeren opdrachten
}
```

9.13.2 Compileren

Bestanden met C#-code hebben de extensie *cs*. Om een uitvoerbaar programma te maken heb je een bestand met C#-code nodig met één of meerdere klassen, waarvan juist één klasse met een *Main*-

methode. De standaard online-compiler wordt opgeroepen met het commando *csc*. Als *MijnBestand.cs* de code voor een C#-programma bevat, dan wordt om de volgende manier het uitvoerbaar programma *MijnBestand.exe* aangemaakt.

```
csc MijnBestand.cs
```

De naam van het uitvoerbaar programma is afgeleid van de naam van het cs-bestand en niet van de klasse met de *Main*-methode.

Verschillende bestanden compileren tot één programma gaat als volgt:

```
csc /out:Programma.exe Bestand1.cs Bestand2.cs Bestand3.cs
```

De optie *out* biedt de mogelijkheid om de naam van het uitvoerbaar programma te specifiëren.

9.13.3 Bibliotheek (Assembly)

Het is ook mogelijk om verschillende bestanden met klassen, ... te compileren tot een bibliotheek. De klassen uit deze bestanden hoeven niet tot dezelfde namespace te behoren. Het bestand dat de gecompileerde bibliotheek bevat heeft de extensie *dll*.

```
csc /target:library /out:Bib.dll Bestand1.cs Bestand2.cs
    Bestand3.cs
```

Wil je klassen, ... uit de aangemaakte bibliotheek gebruiken in een programma, dan moet je dit expliciet opgeven bij het compileren.

```
csc /reference:Bib.dll Programma.cs
```

9.13.4 .NET klasse-bibliotheken

Het .NET-platform bevat verschillende bibliotheken die bruikbaar zijn voor meerdere programmeertalen waaronder C#. In dit hoofdstuk maakten we onder andere gebruik van de namespaces *System* en *System.Collection*

Hoofdstuk 10

Delegates en events in C#

Delegates en Events (C#)

Veerle Ongenae

Inhoud

- Delegates
  - Instructieobjecten
- Events
  - Eventmechanisme
  - Oproepen en afhandelen van gebeurtenissen
  - Luisteren naar gebeurtenissen

Voorbeeldapplicatie

- Keuken met koelkasten en dieren
- Een aantal spelregels
  - Elke koelkast kan maar één dier bevatten
  - Een dier kan maar in de koelkast als de deur openstaat
  - De deur kan maar open als ze dicht is
  - De deur kan maar dicht als ze openstaat

Beginsituatie – applicatie

- Eerst is de keuken leeg
- Een koelkast en een dier worden gekenmerkt door een (unieke) naam
- Een koelkast en een dier verschijnen als hun naam genoemd wordt

Doel applicatie

- Scenario's bouwen
- Een scenario bestaat uit nul of meerdere opdrachten
- Mogelijke acties
  - Scenario uitvoeren (commando: start)
  - Scenario verwijderen (commando: reset)
  - Inhoud van de keuken tonen (commando: toon)
  - Programma afsluiten (commando: exit)
  - Opdracht toevoegen aan scenario

Mogelijke opdrachten

- Deur van een koelkast openen
  naamKoelkast deur open
- Deur van een koelkast sluiten
  naamKoelkast deur dicht
- Dier in koelkast steken
  naamKoelkast naamDier in
- Dier verwijderen uit koelkast
  naamKoelkast naamDier uit

Demonstratie

- Keuken.exe

Delegate object

- Scenario
  - Lijst van objecten met een uit te voeren methode
  - Elke methode heeft dezelfde parameterlijst en dezelfde teruggeefwaarde (return-type)
  - Elke methode wordt opgeroepen met dezelfde argumentenlijst
  - Concreet
    - Lege parameterlijst
    - void als teruggeefwaarde

Dier toevoegen of verwijderen

- Werkwijze
  - Dier selecteren (actief maken)
  - Koelkast dier laten toevoegen of verwijderen
  - Dier inactief maken

Een aantal klassen

- Dier.cs
  - Naam?
  - Geselecteerd?
- Keuken.cs
  - Dieren
  - Koelkasten
  - Is dier in keuken (en niet in koelkast)?
  - Haal geselecteerd dier
- Koelkast.cs
  - Dier? Geselecteerd dier in/uit
  - Deur open/dicht

Delegate

- Delegate-object
- Delegate-klasse
- Bewerkingen op delegates
- Uitvoeren van een delegate

Delegate

- Delegate-object
  - Lijst
    - Object + methode van dit object
    - Statische methode
  - Elke methode heeft dezelfde parameterlijst en teruggeefwaarde
- Delegate-klasse
  - Bepaalt type van de delegate-objecten
    - Parameterlijst
    - Teruggeefwaarde

Delegate

- Delegate-object
- Delegate-klasse
- Bewerkingen op delegates
- Uitvoeren van een delegate

Delegate-klasse

- Declaratie
  - Voorbeeld: (nieuw type/klasse!)

    public delegate void ScenarioOpdracht();
  - Syntax

    [toegang] delegate returnType NaamDelegateKlasse (parameterLijst);
- Afgeleide klasse van System.Delegate
  - compiler

Delegate-object

- Voorbeeld

private ScenarioOpdracht opdrachten = null;
- Constructie
  - Voorbeeld

    new ScenarioOpdracht(koelkast.OpenDeur)
  - Algemeen : new DelegateType(argument)
    - argument
      - object.Methode
      - Klasse.Methode
      - Delegate-object
- Delegates zijn immutable

Delegate

- Delegate-object
- Delegate-klasse
- Bewerkingen op delegates
- Uitvoeren van een delegate

Bewerkingen op delegates

- + en +=
- - en -=

Bewerkingen op delegates

- + en +=
  - Nieuwe delegate
  - De lijst van methodes is de combinatie van de lijst van methodes van het eerste argument en de lijst van methodes van het tweede argument

Voorbeeld + en +=

```
delegate void D (int x);
class K {
    public static void M1 (int i) {…}
    public static void M2 (int i) {…}
}
public class Test {
    public static void Main() {
        D d1, d2, d3;
        d1 = new D(K.M1);
        d2 = new D(K.M2);
        d3 = d1 + d2 + d2 + d1; // M1 + M2 + M2 + M1
    }
}
```


Voorbeeld + en += in Keuken

```
private ScenarioOpdracht opdrachten = null;

opdrachten += new ScenarioOpdracht(koelkast.SluitDeur);
opdrachten += new ScenarioOpdracht(koelkast.OpenDeur);
opdrachten += new ScenarioOpdracht(dier.MaakActief);
opdrachten += new ScenarioOpdracht(koelkast.LaatBinnen);
opdrachten += new ScenarioOpdracht(dier.MaakInactief);
```


Bewerkingen op delegates

- - en -=

d3 = d1 – d2; //d1, d2 en d3 delegates van type D

- Mogelijke situaties
  - De lijst van methodes van d2 is een aaneensluitende deellijst van de lijst van methodes van d1 → De lijst van methodes van d3 is de lijst van methodes van d1 waaruit die deellijst verwijderd is
    - Deellijst komt meermaals voor → meest “rechtse” wordt verwijderd.
  - Anders → d3 is d1

Voorbeeld - en -=

```
delegate void D (int x);
class K {
    public static void M1 (int i) { }
    public static void M2 (int i) { }
}
```


Voorbeeld +, +=, - en -=

```
D d1, d2, d3;
d1 = new D(K.M1);
d2 = new D(K.M2);
d3 = d1 + d2 + d2 + d1; // M1 + M2 + M2 + M1
d3 -= d1; // M1 + M2 + M2
d3 = d1 + d2 + d2 + d1; // M1 + M2 + M2 + M1
d3 -= d1 + d2; // M2 + M1
d3 = d1 + d2 + d2 + d1; // M1 + M2 + M2 + M1
d3 -= d2 + d1; // M1 + M2
d3 = d1 + d2 + d2 + d1; // M1 + M2 + M2 + M1
d3 -= d1 + d1; // M1 + M2 + M2 + M1
d3 = d1 + d2 + d2 + d1; // M1 + M2 + M2 + M1
d3 -= d2 + d2; // M1 + M1
```


De klasse Scenario

- Scenario.cs
  - Main
  - Verwerk
  - VerwerkScenarioOpdrachtOfLuisteraar
  - Reset

Delegate

- Delegate-object
- Delegate-klasse
- Bewerkingen op delegates
- Uitvoeren van een delegate

Uitvoeren van een delegate

- Eén voor één uitvoeren van de methodes waaruit de delegate bestaat
- Het argument voor elke methode is het argument van de delegate

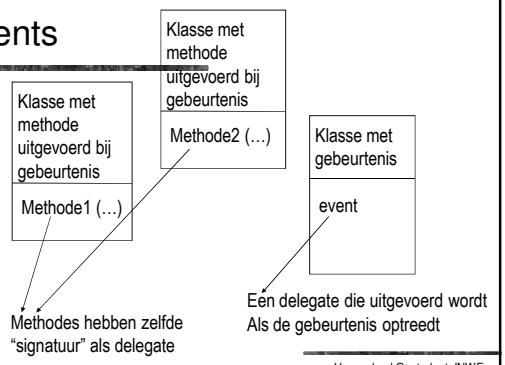
Voorbeeld uitvoeren delegate

```
delegate void D (int x);
class K {
    public static void M1 (int i) {…}
    public static void M2 (int i) {…}
    public void M3 (int i) {…}
}
public class Test {
    public static void Main() {
        D d1;
        K k1 = new K();
        d1 = new D(K.M1);
        d1 += new D(K.M2);
        d1 += new D(k1.M3);
        d1(36);
    }}
```

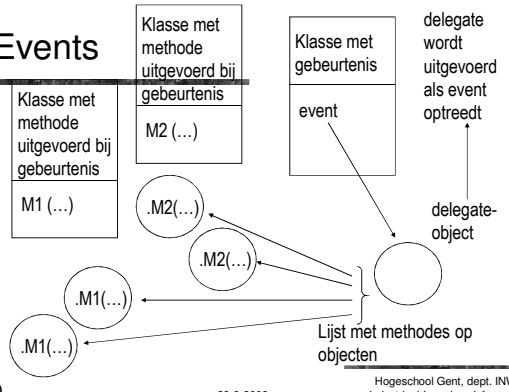

Uitvoeren delegate in Keuken

- Scenario.cs
  - VoerOpdrachtenUit

Events



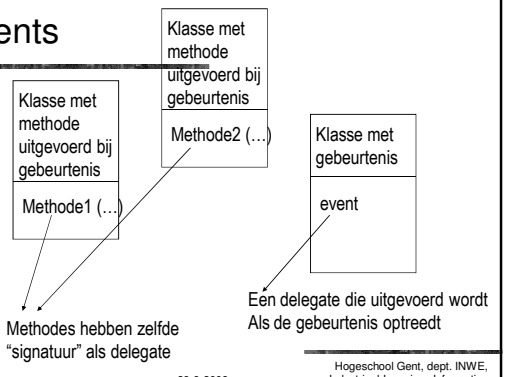
Events



Voorbeeldapplicatie

- Uit Koelkast worden alle mededelingen naar het scherm verwijderd
- Het is mogelijk om “luisteraars” toe te voegen aan een koelkast
  - Lamp, bel, bieper, algemene luisteraar
  - Opdracht
  naamKoelkast naamLuisteraar [erbij|weg]
- KeukenMetEvents

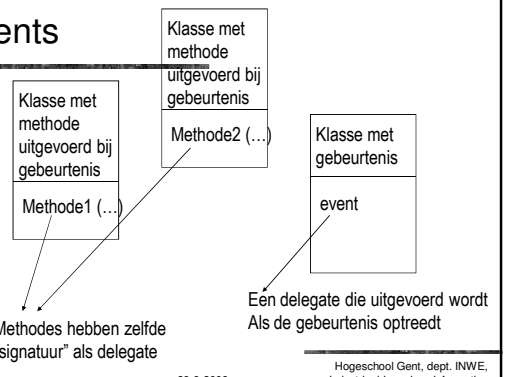
Events



Aantal extra klassen

- KoelkastEventArgs.cs
  - Delegate KoelkastEventHandler
  - Opsomming Actie
  - Klasse KoelkastEventArgs
    - Dier (betrokken bij event)
    - Actie

Events



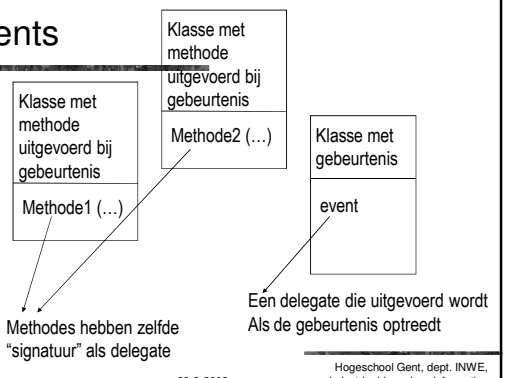
Event declareren

```
public class Koelkast {
    …
    public event KoelkastEventHandler Executing;
    public event KoelkastEventHandler Executed;
    …
}
```

- Syntax

[toegang] event *TypeEvent NaamEvent*;

Events



Methodes uit te voeren bij gebeurtenis

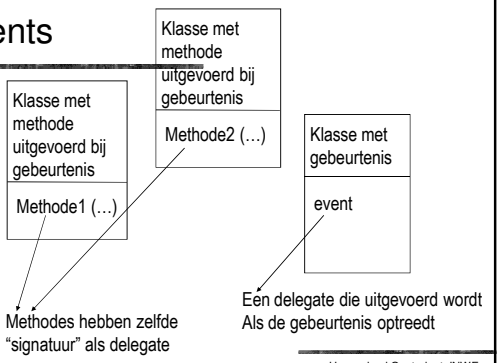
- Elke luisteraar (Lamp, Bieper, Bel, StandaardKoelkastLuisteraar) implementeert de interface KoelkastLuisteraar

```
namespace Be.Hogent.III.Keuken {
   public interface KoelkastLuisteraar {
    void Koelkast_Executing(object sender,
        KoelkastEventArgs e);
    void Koelkast_Executed(object sender,
        KoelkastEventArgs e);
   }
}
```

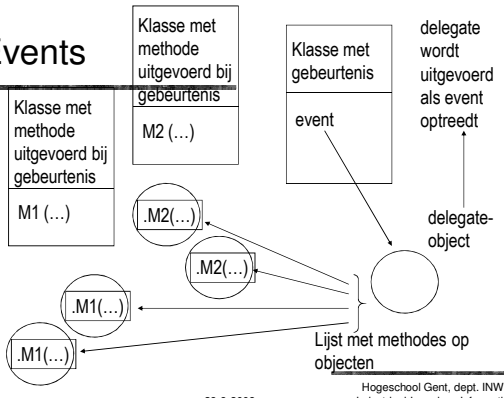

De luisteraars

- Algemeen
  - Luisteren naar acties in de keuken
  - Afhankelijk van actie iets ondernemen
- Beschikbare luisteraars
  - Lamp
  - Bel
  - Bieper
  - StandaardKoelkastLuisteraar

Events



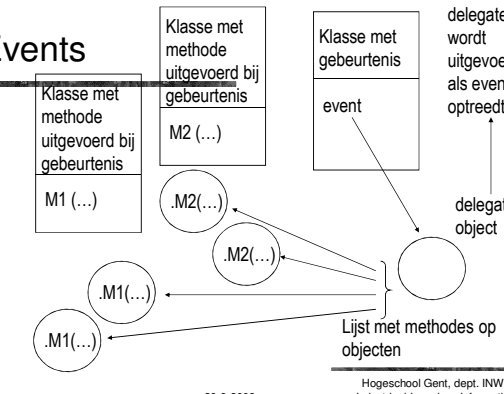
Events



Delegate toevoegen aan eventlijst

- Scenario aanpassen
  - Methode GetLuisteraar
    - Luisteraar ophalen, indien nodig aanmaken
  - Methode VerwerkScenarioOpdrachtOfLuisteraar
    - Lijnen van de vorm *naamKoelkast naamLuisteraar* [erbij|weg]
    - Ook verwerken

Events



Koelkast aanpassen

- Toevoegen methodes die opgeroepen moeten worden als de gebeurtenis optreedt

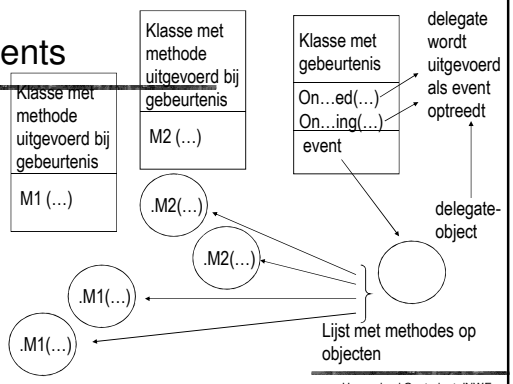
```
protected virtual void OnExecuting(KoelkastEventArgs e) {
    if (Executing != null)
        Executing(this,e);
}
protected virtual void OnExecuted(KoelkastEventArgs e) {
    if (Executed != null)
        Executed(this,e);
}
```


Koelkast aanpassen

- Oproepen methodes
  - OnExecuting bij start gebeurtenis
  - OnExecuted bij einde gebeurtenis

```
public void OpenDeur() {
    OnExecuting(new KoelkastEventArgs(Actie.DeurOpen,null));
    if (deuropen) {
        Error("deur reeds open");
    } else {
        deuropen=true;
    }
    OnExecuted(new KoelkastEventArgs(Actie.DeurOpen,null));
}
```


Events



Conventies

- Delegate-klasse eindigt op EventHandler
- De EventHandler heeft twee argumenten
  - object sender
  - Afgeleide klasse van EventArgs, naam eindigend op EventArgs
- Delegate/event uitgevoerd voor gebeurtenis eindigt op ing
- Delegate/event uitgevoerd na gebeurtenis eindigt op ed

Conventies

- De delegates/events worden opgeroepen vanuit een methode On…ed en On…ing
  - Deze methode kan overschreven worden door afgeleide klassen
- Methodes die de acties implementeren die uitgevoerd worden als de gebeurtenis optreedt zijn van de vorm …ing en … ed

Hoofdstuk 11

ADO.NET: een inleiding

11.1 Wat is ADO.NET?

ADO.NET is de technologie van het .NET-platform die het mogelijk maakt om vanuit een .NET-applicatie toegang te verkrijgen tot data. Het is de opvolger van ADO, wat staat voor Activex Data Objects. Met ADO.NET is het mogelijk om data van verschillende types databronnen op te halen, te manipuleren en aan te passen. Je kan niet enkel data van gegevensbanken manipuleren, maar ook data van andere bronnen zoals XML-files of XML-streams.

Eén van de principes van ADO.NET is dat het een losse koppeling voorziet tussen de applicatie en de databron. In tegenstelling tot wat gebruikelijk is bij een standaard Client-Server-architectuur is er geen constante verbinding met de gegevensbank en kan een deel van de data lokaal in de applicatie bewaard en gemanipuleerd worden. ADO.NET ondersteunt dus probleemloos het bouwen van applicaties volgens de meerlagenarchitectuur.

De ADO.NET architectuur bestaat uit twee grote pijlers: de *DataSet* en .NET Data Providers. De *DataSet* maakt het mogelijk om gegevens lokaal in een programma te bewaren en te manipuleren, los van een gegevensbron. De .NET Data Providers voorzien in toegang tot databronnen buiten de applicatie. De ADO.NET klassen vormen de namespace System.Data in de klassenbibliotheek van het .NET-platform.

11.1.1 *DataSet*

Een *DataSet* is een verzameling van *DataTable*-objecten en hun onderliggende relaties en beperkingen, zoals primaire sleutels, verwijssleutels, ... Een *DataSet* kan opgevuld worden met gegevens uit een database of een xml-bestand, maar heeft geen informatie over de onderliggende gegevensbron. Het is dus een alleenstaande component met informatie in het geheugen. Een *DataTable* is een collectie rijen van een bepaalde tabel.

Tussen *DataSets* en XML is er een hechte band. Een *DataSet* kan gegevens inlezen uit een XML-

bestand of een XML-stream en kan de data die het bevat uitschrijven in XML-formaat, ook als de gegevens oorspronkelijk uit een gegevensbank kwamen. Het gebruik van *DataSets* wordt uitgebreider besproken in §11.4.1.

11.1.2 .NET Data Provider

De toegang tot gegevensbanken in het .NET-platform verloopt via .NET Data Providers. Ze bieden de mogelijkheid om een verbinding te maken met de database, opdrachten uit te voeren in de database en gegevens op te halen uit de database. Om dit te verwezenlijken maken ze gebruik van vier sleutelobjecten: *Connection*, *Command*, *DataReader* en *DataAdapter*. *Connection* maakt een verbinding met de gegevensbron, *Command* voert opdrachten uit in de gegevensbron, *DataReader* leest gegevens uit een databron en een *DataAdapter* kan een *DataSet* opvullen en een databron aanpassen op basis van de gegevens in de *DataSet*.

De namespace *System.Data* bevat de interfaces van de sleutelobjecten die de verschillende providers moeten implementeren, namelijk *IDbConnection*, *IDbCommand*, *IDataReader* en *IDbDataAdapter*. Bij het bouwen van programma's waarbij de gegevensbron niet vast ligt, kan men ook van deze interfaces gebruik maken. Applicaties die vaste databaseservers hebben, gebruiken de elementen van de namespace van de bijhorende providers. De namespace horende bij een provider is een subnamespace van *System.Data*.

Het .NET-platform bevat standaard twee .NET Data Providers: SQL Server .NET Data Provider en de OLE DB .NET Data Provider (OLE DB=Object Linking and Embedding for Databases). Bij deze providers hoort respectievelijk de namespace *System.Data.SqlClient* en *System.Data.OleDb*.

De SQL Server .NET Data Provider is ontworpen voor SQL Server 7.0 en latere versies. Deze provider communiceert direct via het data-transfer-protocol van SQL Server en is daardoor zeer performant en efficient. Voor alle OLE DB gegevensbanken kan je gebruik maken van de OLE DB .NET Data Provider. Deze provider is iets minder efficient dan de vorige omdat hij via een OLE DB Service Component en een OLE DB Provider communiceert met de database. Het voordeel van deze provider is dat hij een uniforme toegang biedt tot verschillende databronnen.

Verschillende producenten van databases voorzien hun eigen provider om op een efficiëntere manier gegevens uit te wisselen met hun gegevensbank. In de rest van dit hoofdstuk zullen het gebruik van .NET Data Providers illustreren aan de hand van de SQL Server .NET Data Provider en de OLE DB .NET Data Provider.

11.2 Een verbinding maken

Om een verbinding te maken met een gegevensbank maak je gebruik van *SqlConnection* respectievelijk *OleDbConnection*. Beiden implementeren uiteraard de interface *IDbConnection*. De volgende stukken code illustreren hoe beide klassen gebruikt kunnen worden om een verbinding te maken met een gegevensbank.

```
using System.Data.SqlClient;
...
String connString = "server=(local)\\NetSDK;database=pubs;
    Trusted_Connection=yes";
SqlConnection conn = new SqlConnection(connString);
conn.Open();
... // gegevens uitwisselen met de gegevensbank
conn.Close();
```

```
using System.Data.OleDb;
...
String connString = "Provider=Microsoft.Jet.OLEDB.4.0;" +
    "User ID=Admin;" +
    "Data Source=C:\\vongenae\\gegevensbanken\\vbn.mdb";
OleDbConnection conn = new OleDbConnection(connString);
conn.Open();
... // gegevens uitwisselen met de gegevensbank
conn.Close();
```

Beide klassen *SqlConnection* en *OleDbConnection* hebben een constructor die een *string* (*connectiestring* genoemd) als parameter heeft. Met behulp van deze string kunnen een aantal eigenschappen van de verbinding met de gegevensbank geïnitialiseerd worden. De connectiestring bestaat uit naam/waarde-paren gescheiden door een punt-komma's. De volgende tabel geeft een overzicht van een aantal in te stellen eigenschappen. In het geval van een *OleDbConnection* is de *Provider*-eigenschap vereist.

Eigenschap	**Stand.**	**Omschrijving**
Provider		De .NET Data Provider die gebruikt wordt om een verbinding te maken met de gegevensbank. Dit attribuut is niet nodig voor de SQL Server .NET Data Provider.
Connect Timeout of *Connection Timeout*	15	Het aantal seconden dat er gewacht wordt op een verbinding met de server. Daarna wordt de poging gestaakt en wordt een fout gegenereerd.
Data Source of *Server*		De naam of het adres van de gegevensbankserver of gegevensbron.
Initial Catalog of *Database*		De naam van de gegevensbank.
Integrated Security of *Trusted_Connection*	*false*	Indien de waarde van deze eigenschap *false* is moeten de eigenschappen *User Id* en *Password* gespecifieerd worden. De waarde *true* impliceert dat de huidige Windowsgebruiker gebruikt wordt voor authenticatie.
Password of *Pwd*		Paswoord om in te loggen op de gegevensbank.
User ID		De gebruiker waarmee ingelogd wordt op de gegevensbank.

Een alternatief voor het toekennen van de connectiestring is gebruik maken van de eigenschap *ConnectionString*. Deze eigenschap kan enkel ingesteld worden als de verbinding met de gegevensbank

niet geopend is. De bovenstaande code wordt dan

```
using System.Data.SqlClient;
...
String connString = "server=(local)\\NetSDK;database=pubs;
    Trusted_Connection=yes";
SqlConnection conn = new SqlConnection();
conn.ConnectionString = connString;
conn.Open();
... // gegevens uitwisselen met de gegevensbank
conn.Close();
```

```
using System.Data.OleDb;
...
String connString = "Provider=Microsoft.Jet.OLEDB.4.0;"
    + "User ID=Admin;" +
    "Data Source=C:\\vongenae\\gegevensbanken\\vbn.mdb";
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = connString;
conn.Open();
... // gegevens uitwisselen met de gegevensbank
conn.Close();
```

De methoden *Open* en *Close* zorgen voor het openen en sluiten van de verbinding met de gegevensbank.

11.3 Opdrachten uitvoeren

11.3.1 Het commando-object

Om opdrachten te kunnen uitvoeren op de gegevensbank moet men na het aanmaken van een verbinding (zie §11.2) een commando-object aanmaken. Dit object implementeert de interface *IDbCommand*. Afhankelijk van de gebruikte provider is dit respectievelijk een *SqlCommand*- of een *OleDbCommand*-object.

Een commando-object kan aangemaakt worden met behulp van een constructor. De argumenten van de constructor kunnen respectievelijk een *string* die een SQL-opdracht voorstelt, een *string* en een connectie-object of een *string*, een connectie-object en een transactie-object zijn. Er is ook een constructor zonder argumenten. Het is ook mogelijk om deze objecten in te stellen, te bekijken of te wijzigen met behulp van de eigenschappen *CommandText*, *Connection* en *Transaction*. De volgende code illustreert het aanmaken van een commando-object aan de hand van een connectie-object en een string die een SQL-query voorstelt.

```
using System.Data.SqlClient;
...
String connString = "server=(local)\\NetSDK;database=pubs;
    Trusted_Connection=yes";
SqlConnection conn = new SqlConnection(connString);
string query = "select voornaam, achternaam from personen";
SqlCommand bepPers = new SqlCommand(query,conn);
...
```

```
using System.Data.OleDb;
...
String connString = "Provider=Microsoft.Jet.OLEDB.4.0;"
    + "User ID=Admin;"+
    "Data Source=C:\\vongenae\\gegevensbanken\\vbn.mdb";
OleDbConnection conn = new OleDbConnection(connString);
string query = "select voornaam, achternaam from personen";
OleDbCommand bepPers = new OleDbCommand(query,conn);
...
```

Een alternatieve manier om een commando-object te verkrijgen is het oproepen van de methode *CreateCommand* die deel uitmaakt van de *IDbConnection*-interface zoals geïllustreerd in de volgende voorbeelden.

```
using System.Data.SqlClient;
...
String connString = "server=(local)\\NetSDK;database=pubs;
    Trusted_Connection=yes";
SqlConnection conn = new SqlConnection(connString);
string query = "select voornaam, achternaam from personen";
SqlCommand bepPers = conn.CreateCommand();
bepPers.CommandText = query;
...
```

```
using System.Data.OleDb;
...
String connString = "Provider=Microsoft.Jet.OLEDB.4.0;"
    + "User ID=Admin;" +
    "Data Source=C:\\vongenae\\gegevensbanken\\vbn.mdb";
OleDbConnection conn = new OleDbConnection(connString);
string query = "select voornaam, achternaam from personen";
OleDbCommand bepPers = conn.CreateCommand();
bepPers.CommandText = query;
...
```

Merk op dat in dit geval de SQL-opdracht ingesteld wordt met de eigenschap *CommandText*. Deze methode biedt de mogelijkheid om het commando-object aan te maken onafhankelijk van de gebruikte provider.

De *IDbCommand*-interface bevat een aantal methodes om opdrachten uit te voeren, o.a. *ExecuteReader* en *ExecuteNonQuery*. De methode *ExecuteReader* dient om zoekopdrachten uit te voeren (zie §11.3.2) en met de methode *ExecuteNonQuery* kunnen gegevens toegevoegd, gewijzigd of verwijderd worden (zie §11.3.3).

Om SQL-opdrachten met parameters aan te maken wordt de methode *CreateParameter* gebruikt (zie §11.3.4).

11.3.2 Zoekopdrachten

Om het resultaat van zoekopdrachten op te halen kan je gebruik maken van een object dat de *IDataReader*-interface implementeert. In het geval van de SQL Server .Net Data Provider, respectievelijk de OLE DB .Net Data Provider, zijn dat objecten van het type *SqlDataReader* en *OleDbDataReader*. Met een *DataReader* kan je de gegevensstroom van de databank éénmaal lezen van het begin naar het einde.

Een *DataReader* wordt verkregen als het resultaat van het oproepen van de methode *ExecuteReader* van een commando-object (zie §11.3.1). De methode *Read* verplaatst de positie van de *DataReader* naar de volgende rij van het resultaat van de uitgevoerde zoekopdracht. Voor de eerste oproep van de methode *Read* is de positie van de *DataReader* voor de eerste rij van het verkregen resultaat. Het resultaat van de methode *Read* is een logische grootheid (*bool*) die aangeeft of er nog rijen in het verkregen resultaat zijn of dat de positie van de *DataReader* na de laatste rij is.

De interface *IDataReader* heeft een indexer die zowel een string als een geheel getal als argument aanvaardt. Je kan deze indexer gebruiken om de gegevens van de verschillende kolommen van de huidige rij in het verkregen resultaat op te halen. Als argument van de indexer gebruik je dan de naam van de kolom of het volgnummer van de kolom (0, 1, 2, ...). Een alternatief zijn de methodes *GetXxx* van de klassen *SqlDataReader* en *OleDbDataReader*. Hierbij stelt *Xxx* het type van de kolomgegevens voor. Deze methodes hebben het volgnummer van de kolom als argument. De laatste methode om gegevens uit één rij op te halen is performanter onder andere omdat er dan minder conversies gebeuren.

De volgende voorbeelden illustreren hoe de gegevens van een zoekopdracht op het scherm getoond kunnen worden. De variabele *bepPers* is van het type *SqlCommand* respectievelijk *OleDbCommand* (zie ook §11.3.1).

```
using System;
using System.Data.SqlClient;
...
SqlDataReader reader = bepPers.ExecuteReader();
while (reader.Read()) {
    Console.WriteLine(reader.GetString(0) + ", " +
         reader.GetString(1));
}
reader.Close();
...
```

```
using System;
using System.Data.OleDb;
...
OleDbDataReader reader = bepPers.ExecuteReader();
while (reader.Read()) {
    Console.WriteLine(reader.GetString(0) + ", " +
        reader.GetString(1));
}
reader.Close();
...
```

Na het ophalen van het resultaat van een zoekopdracht moet de *DataReader* afgesloten worden met de methode *Close* om andere opdrachten te kunnen uitvoeren over dezelfde verbinding met de databank.

11.3.3 Gegevens wijzigen, toevoegen of verwijderen

Om gegevens te wijzigen, toe te voegen of te verwijderen wordt opnieuw een commando-object (zie §11.3.1) aangemaakt. Het enige verschil met het voorbeeld in §11.3.1 is dat de eigenschap *CommandText* nu een UPDATE, INSERT of DELETE SQL-opdracht voorstelt. Vervolgens wordt de methode *ExecuteNonQuery* van de interface *IDbCommand* opgeroepen. Het resultaat van deze methode is het aantal aangepaste rijen.

Het volgende voorbeeld illustreert de uitvoering van een INSERT SQL-opdracht. De variabele *conn* is een *OleDbConnection*.

```
using System.Data.OleDb;
...
string opdracht = "insert into personen(achternaam,voornaam)
    values ('De Ridder','Jan')";

conn.Open();
try {
    OleDbCommand voegPersToe = new OleDbCommand(opdracht,conn);
    voegPersToe.ExecuteNonQuery();
} finally {
    conn.Close();
}
...
```

Op een analoge manier is het mogelijk om DDL-opdrachten (Data Definition Language) of stored procedures uit te voeren.

11.3.4 Parameters

Parameters zijn handig om herhaaldelijk een zelfde SQL-opdracht uit te voeren met andere waarden. Het commando-object (zie §11.3.1) heeft een eigenschap *Parameters* die de parameters bijhoudt. De eigenschap *Parameters* is van het type *IDataParameterCollection* (respectievelijk *SqlParameterCollection* of *OleDbParameterCollection*). De parameter zelf implementeert de *IDbDataParameter*-interface die zelf een afgeleide interface is van *IDataParameter*. De klassen die parameters voor stellen bij de SQL Server .Net Data Provider en de OLE DB .Net Data Provider zijn respectievelijk *SqlParameter* en *OleDbParameter*.

Om een parameter aan te maken moeten er twee dingen gebeuren: de plaats van de parameter aanduiden in de string die de SQL-opdracht voorstelt en de parameter toevoegen aan het commando-object.

De plaats van de parameter in de SQL-opdracht wordt aangegeven door een string van de vorm @*parameter* waarbij *parameter* de naam van de parameter voorstelt. Het volgend voorbeeld toont een INSERT SQL-opdracht waarbij twee parameters gebruikt worden, namelijk @*naam* en @*voornaam*.

```
string opdracht = "insert into personen(naam,voornaam)
    values (@naam,@voornaam)";
```

In de constructor van *SqlParameter*, respectievelijk *OleDbParameter* kunnen onder andere de volgende argumenten worden meegegeven:

- geen argumenten
- de naam en de waarde van de parameter
- de naam en het type van de parameter

In het volgend voorbeeld worden twee parameters aangemaakt van het type *string* horende bij de bovenstaande parameters @*naam* en @*voornaam*.

```
OleDbParameter paramNaam =
    new OleDbParameter("@naam",DbType.String);
OleDbParameter paramVNaam =
    new OleDbParameter("@voornaam",DbType.String);
```

De naam, de waarde en het type van de parameter kunnen ook ingesteld, bekeken en gewijzigd worden met de eigenschappen *ParameterName*, *Value* en *SqlDbType* (respectievelijk *OleDbType*). Het volgende stukje code illustreert dat het ook mogelijk is om een parameter aan te maken onafhankelijk van het type provider. De variabele *voegPersToe* is van het type *IDbCommand*.

```
IDataParameter paramNaam = voegPersToe.CreateParameter();
paramNaam.ParameterName = "@achternaam";
paramNaam.DbType = DbType.String;
```

Het type van een parameter is afhankelijk van de gebruikte provider. De verschillende types bij de SQL Server .Net Data Provider en de OLE DB .Net Data Provider worden bepaald door de opsommingen *SqlDbType* en *OleDbType*. De opsomming *DbType* biedt ook de mogelijkheid om op een meer generieke wijze types instellen.

De methode *Add* voegt de parameters dan toe aan de parameterverzameling horende bij het commando-object. Het volgend voorbeeld illustreert het toevoegen van de parameters aan het commando-object *voegPersToe*, het toekennen van een waarde aan de parameters en het uitvoeren van de opdracht. In tegenstelling tot wat men zou verwachten moeten de parameters in de volgorde van voorkomen in de SQL-opdracht toegevoegd worden aan de parameterverzameling.

```
voegPersToe.Parameters.Add(paramNaam);
voegPersToe.Parameters.Add(paramVNaam);

voegPersToe.Parameters["@achternaam"].Value = "Peeters";
voegPersToe.Parameters["@voornaam"].Value = "Piet";

conn.Open();
try {
    voegPersToe.ExecuteNonQuery();
} finally {
    conn.Close();
}
```

Merk op dat de *IDataParameterCollection*, de *SqlParameterCollection* en de *OleDbParameterCollection* een indexer met een *string*-argument hebben die respectievelijk een *object*, een *SqlParameter* of een *OleDbParameter* teruggeven. Dit betekent dat als in het bovenstaande voorbeeld *voegPersToe* van het type *IDbCommand* is i.p.v. *OleDbCommand* de code aangepast moet worden.

```
((IDataParameter)voegPersToe.Parameters["@achternaam"]).Value
    = "Peeters";
((IDataParameter)voegPersToe.Parameters["@voornaam"]).Value
    = "Piet";
```

11.4 DataSet en DataAdapter

In §11.3.1 werd besproken hoe opdrachten uitgevoerd kunnen worden op de gegevensbank vanuit een c#-programma. Om de gegevens op te halen uit de gegevensbank werd gebruik gemaakt van een *DataReader*. Met behulp van deze *DataReader* kan het resultaat van een query rij voor rij overlopen en opgehaald worden. In deze sectie bekijken we een alternatieve manier om gegevens op te halen uit de gegevensbank. De opgehaalde gegevens worden in het geheugen bewaard in een *DataSet*-object en kunnen in het geheugen gemanipuleerd worden. Achteraf kunnen deze wijzigingen dan teruggestuurd worden naar de gegevensbank.

11.4.1 DataSet

Een *DataSet* is een voorstelling van gegevens in het geheugen zoals in een relationele databank. Deze gegevensvoorstelling is niet verbonden met een bestaande gegevensbron en kan de volgende elementen bevatten: gegevens, tabellen, beperkingen (bv. *primary keys*) en relaties tussen tabellen zoals bv. verwijssleutels (*foreign keys*). De gegevens in een *DataSet* kunnen data aangemaakt door de huidige applicatie zijn of data van één of meerdere externe databronnen. De communicatie met de bestaande databronnen wordt bepaald door een *DataAdapter*.

Het volgende voorbeeld toont hoe je een *DataSet*-object kan aanmaken. Elk *DataSet*-object heeft een naam, die o.a. gebruikt wordt om dit object om te zetten naar XML. Indien er geen naam gespecifieerd wordt dan krijgt de *DataSet* de standaardnaam *NewDataSet*. Dit is het geval in het eerste voorbeeld. In het tweede voorbeeld is de naam van de *DataSet* wel opgegeven en is die *CustomerOrders*.

```
DataSet gegevens = new DataSet();
```

```
DataSet custDS = new DataSet("CustomerOrders");
```

Een *DataSet* kan één of meerdere *DataTable*-objecten bevatten. Bovendien kan je op bepaalde kolommen van deze *DataTable*-objecten beperkingen plaatsen zoals *primary keys* en *unique*. In het volgende voorbeeld wordt een *DataSet* aangemaakt met één *DataTable*-object. De tabel heeft drie kolommen *OrderID*, *OrderQuantity* en *CompanyName*. De kolom *OrderID* is de primaire sleutel van de tabel.

```
DataSet custDS = new DataSet("CustomerOrders");

DataTable ordersTable = custDS.Tables.Add("Orders");

DataColumn pkCol = ordersTable.Columns.Add("OrderID",
    typeof(Int32));
ordersTable.Columns.Add("OrderQuantity", typeof(Int32));
ordersTable.Columns.Add("CompanyName", typeof(string));

ordersTable.PrimaryKey = new DataColumn[] {pkCol};
```

Uit de bovenstaande code leren we dat een *DataSet* de eigenschap *Tables* heeft van het type *DataTableCollection*. De methode *Add* van *DataTableCollection* voegt een tabel toe aan de *DataSet*. Het argument van deze methode is de naam van de tabel, het resultaat is een verwijzing naar de aangemaakte tabel. De klasse *DataTableCollection* heeft ook een indexer met als argument een *string* of een *int*. Het resultaat van de indexer is een *DataTable*-object met de opgegeven naam of het opgegeven volgnummer.

Een *DataTable* heeft onder andere de eigenschappen *Columns* van het type *DataColumnCollection* en *PrimaryKey* (een tabel van *DataColumn*). In het voorbeeld wordt de methode *Add* van *DataColumnCollection* gebruikt om een kolom toe te voegen aan de tabel. De argumenten van deze methode

kunnen onder andere de naam en het type van de kolom zijn. Het resultaat is een verwijzing naar de aangemaakte kolom.

Om het type van een klasse te bepalen kan het sleutelwoord *typeof* gebruikt worden. Een alternatief voor *Type t = typeof(string);* is bijvoorbeeld de volgende uitdrukking.

```
using System;
...
string mijnString = "Een String";
Type t = mijnString.GetType();
```

Het volgend voorbeeld illustreert hoe een rij toegevoegd kan worden aan de bovenstaande tabel.

```
// een rij aanmaken
DataRow ordersRow = ordersTable.NewRow();
ordersRow[0] = 3145;
ordersRow[1] = 3;
ordersRow["CompanyName"] = "Mijn Bedrijf";

// een rij toevoegen aan de tabel
ordersTable.Rows.Add(ordersRow);
```

De methode *NewRow* van *DataTable* maakt een nieuwe rij aan op basis van de structuur van het *DataTable*-object. De gegevens van deze rij kunnen ingevuld worden met de indexer van *DataRow*. Deze indexer is van het type *object*. Het argument van de indexer kan onder andere een *string*, een *int*, een *DataColumn*, ... zijn die de kolom aanduidt.

Om de rij toe te voegen aan de tabel wordt gebruik gemaakt van de methode *Add* van de klasse *DataRowCollection*. Het *DataRowCollection*-object wordt verkregen met de eigenschap *Rows* van *DataTable*.

Met behulp van de methode *Find* van het *DataRowCollection*-object kunnen rijen gevonden worden aan de hand van hun primaire sleutel. In het volgend voorbeeld wordt de hierboven toegevoegde rij geselecteerd.

```
DataRow selectRow = ordersTable.Rows.Find(3145);
```

11.4.2 DataAdapter

Om een *DataSet* op te vullen met gegevens uit een databank wordt gebruik gemaakt van een *DataAdapter*. Een *DataAdapter* vormt de brug tussen de *DataSet* en de onderliggende databronnen. De *DataAdapter* wordt achteraf ook gebruikt om de wijzigingen door de voeren op de gegevensbronnen.

Een *DataAdapter* implementeert de interface *IDbDataAdapter*. De klassen die *DataAdapters* voorstellen bij de SQL Server .Net Data Provider en de OLE DB .Net Data Provider zijn respectievelijk *SqlDataAdapter* en *OleDbDataAdapter*.

Een *DataAdapter* kan vier commando-objecten hebben die respectievelijk zorgen voor het ophalen van de gegevens uit de gegevensbank, het toevoegen van gegevens aan de gegevensbank, het wijzigen van gegevens in de gegevensbank en het verwijderen van gegevens uit de gegevensbank. Deze vier commando-objecten worden ingesteld, gewijzigd en geselecteerd met de eigenschappen *SelectCommand*, *InsertCommand*, *UpdateCommand* en *DeleteCommand*. Verder beschikt de *DataAdapter* ook over een connectie-object (zie §11.2) via zijn commando-objecten.

Om een *DataSet* op te vullen wordt de *Fill*-methode van de *DataAdapter* gebruikt. Deze methode heeft twee argumenten: de op te vullen *DataSet* en de tabel die het resultaat van de query, voorgesteld door de eigenschap *SelectCommand*, zal bevatten. Het argument dat naar de tabel verwijst kan een *DataTable*-object zijn of een *string* die de naam van de tabel is. In het onderstaande voorbeeld wordt een *DataSet* opgevuld met een tabel die het resultaat van een SQL-zoekopdracht *query* bevat. De tabel krijgt de naam *Personen*.

```
string connString = "...";
string query = "select personenID, voornaam, achternaam
    from personen";

OleDbConnection conn = new OleDbConnection(connString);
OleDbDataAdapter adapter = new OleDbDataAdapter(query, conn);
DataSet ds = new DataSet();
adapter.Fill(ds, "Personen");
```

De SQL-zoekopdracht en de connectiestring worden hier met de constructor meegegeven. Een alternatief zou het volgende zijn

```
OleDbDataAdapter adapter = new OleDbDataAdapter();
adapter.SelectCommand = new OleDbCommand(query,conn);
```

Merk op dat de verbinding met de gegevensbank niet expliciet geopend en gesloten wordt. Dit gebeurt impliciet in de *Fill*-methode. Als de verbinding al geopend was bij het oproepen van de *Fill*-methode dan wordt de verbinding niet geopend en gesloten tijdens het uitvoeren van de methode.

Bij het opvullen van de *DataSet* worden geen primaire sleutels aangemaakt tenzij dit expliciet wordt aangegeven op de volgende manier.

```
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

Hierbij is de eerste *MissingSchemaAction* een eigenschap van *adapter*, en dus ook een eigenschap van een *IDataAdapter*-interface en de tweede *MissingSchemaAction* is een opsomming uit de bibliotheek *System.Data*.

Nu we de *DataSet* opgevuld hebben met de tabel *Personen* kunnen we hieraan gegevens toevoegen, wijzigen en verwijderen. Het volgend stukje code geeft een voorbeeld van mogelijke wijzigingen.

```
DataRow rij = ds.Tables["Personen"].NewRow();
rij[2] = "Dierick";
rij[1] = "Dirk";
ds.Tables["Personen"].Rows.Add(rij);

rij = ds.Tables["Personen"].Rows.Find(12);
rij[2] = "Danneels";

rij = ds.Tables["Personen"].Rows.Find(17);
rij.Delete();
```

De methode *Delete* van *DataRow* markeert de rij als te verwijderen. Deze actie kan nog ongedaan gemaakt worden door het oproepen van de methode *RejectChanges* van *DataRow*. De bovenstaande code voert wijzigingen uit aan de *DataSet ds*. Om deze wijzigingen ook door te voeren aan de gegevensbank moeten de eigenschappen *InsertCommand*, *UpdateCommand* en *DeleteCommand* van de bijhorende *DataAdapter* ingevuld worden en moet de methode *Update* van die adapter worden opgeroepen. In het volgend stuk code worden deze commando-objecten aangemaakt en toegekend aan de adapter.

```
// commando's aanmaken
// insert
adapter.InsertCommand = new OleDbCommand(insert,conn);
OleDbParameter paramNaam = adapter.InsertCommand.Parameters.Add(
    "@achternaam",DbType.String);
paramNaam.SourceColumn = "achternaam";
OleDbParameter paramVNaam = adapter.InsertCommand.Parameters.Add(
    "@voornaam",DbType.String);
paramVNaam.SourceColumn = "voornaam";

// update
adapter.UpdateCommand = new OleDbCommand(update,conn);
paramNaam = adapter.UpdateCommand.Parameters.Add("@achternaam",
    DbType.String);
paramNaam.SourceColumn = "achternaam";
paramNaam.SourceVersion = DataRowVersion.Current;
OleDbParameter paramID = adapter.UpdateCommand.Parameters.Add(
    "@personenID",DbType.Int64);
paramID.SourceColumn = "personenID";
paramID.SourceVersion = DataRowVersion.Original;

// delete
adapter.DeleteCommand = new OleDbCommand(delete,conn);
paramID = adapter.DeleteCommand.Parameters.Add("@personenID",
    DbType.Int64);
paramID.SourceColumn = "personenID";
```

```
paramID.SourceVersion = DataRowVersion.Original;

// wijzigingen doorvoeren op gegevensbank
adapter.Update(ds,"Personen");
```

In het bovenstaande voorbeeld wordt een alternatieve methode (anders dan in §11.3.4) gebruikt om een parameter aan te maken. Deze methode kan je enkel gebruiken indien je de provider kent omdat de methode *Add* met meerdere parameters niet bestaat voor een *IDataParameterCollection*, maar wel voor een *OleDbParameterCollection*.

Een ander verschil met §11.3.4 is dat de parameter geen waarde wordt toegekend. Hier wordt met de eigenschap *SourceColumn* de kolom aangeduid die de waarde voor de parameter bevat. Als je een rij verandert dan worden zowel de oorspronkelijk als de huidige inhoud van de rij bijgehouden. Bovendien wordt ook de status (toegevoegd, veranderd, verwijderd, ...) van de rij aangepast. De eigenschap *SourceVersion* van *IDataParameter* geeft aan welke versie van de kolom van een rij gebruikt moet worden bij het uitvoeren van de opdracht. De opsomming *DataRowVersion* bevat de mogelijke versies van een rij. De mogelijke statussen van een rij worden bepaald door de opsomming *DataRowState*.

Bij het uitvoeren van de methode *Update* wordt nagegaan welke rijen toegevoegd, veranderd of verwijderd zijn. Vervolgens wordt het corresponderende commando voor deze rij uitgevoerd. Dit betekent dat als er een rij toegevoegd is, het *InsertCommand* wordt uitgevoerd, dat als er een rij veranderd is het *UpdateCommand* wordt uitgevoerd en dat als er een rij verwijderd is, het *DeleteCommand* wordt uitgevoerd.

De waarde van de parameters in het commando-object zijn dan de waarden voor de huidige rij. Welke versie van de kolom er gebruikt wordt, hangt af van de eigenschap *SourceVersion* voor die parameter. Als in het bovenstaande voorbeeld voor een bepaalde rij zowel het *personenID* als de *achternaam* veranderd zijn, dan wordt bij het uitvoeren van het *UpdateCommand* de nieuwe *achternaam*, maar het oude *personenID* gebruikt omdat de waarde van de eigenschap *SourceVersion* respectievelijk *DataRowVersion.Current* en *DataRowVersion.Original* zijn.

De keuze tussen het gebruik van een *DataReader* en een *DataAdapter* in combinatie met een *DataSet* wordt bepaald door een aantal factoren. Het gebruik van een *DataReader* is performanter omdat er veel minder gegevens in het geheugen bewaard worden. Anderszijds is een *DataSet* handig in combinatie met ASP.NET-pagina's en biedt het de mogelijkheid om data te manipuleren los van de gegevensbank en pas later wijzigingen toe te brengen.

11.5 Transacties

Een transactie is een groep van gegevensbankopdrachten die samen uitgevoerd moeten worden. Als er een fout optreedt bij het uitvoeren van de opdrachten dan worden alle opdrachten terug ongedaan gemaakt. Het uitvoeren van een transactie in ADO.NET bestaat uit een aantal stappen:

1. Om een transactie te starten wordt de methode *BeginTransaction* van *IDbConnection* uitge-

voerd. Het resultaat van deze methode is een transactie-object van het type *IDbTransaction*.

2. Vervolgens wordt dit transactie-object toegekend aan alle commando-objecten die tot de transactie behoren. Hierbij wordt gebruik gemaakt van de eigenschap *Transaction* van *IDbCommand*.

3. Daarna worden alle opdrachten van de transactie uitgevoerd.

4. Tot slot wordt de methode *Commit* van het transactie-object opgeroepen als alle opdrachten gelukt zijn, in het andere geval wordt de methode *Rollback* uitgevoerd.

Het bovenstaande principe wordt geïllustreerd in het volgend voorbeeld.

```
string connString = "Data Source=localhost;"
    + "Initial Catalog=Northwind;"
    + "Integrated Security=SSPI;";
SqlConnection myConnection = new SqlConnection(connString);
myConnection.Open();

// Start a local transaction.
SqlTransaction myTrans = myConnection.BeginTransaction();

// Enlist the command in the current transaction.
SqlCommand myCommand = new SqlCommand();
myCommand.Connection = myConnection;
myCommand.Transaction = myTrans;

try {
    myCommand.CommandText = "Insert into Region (RegionID, " +
        + "RegionDescription) "
        + "VALUES (100, 'Description')";
    myCommand.ExecuteNonQuery();
    myCommand.CommandText = "Insert into Region (RegionID, " +
       "RegionDescription) " +
        "VALUES (101, 'Description')";
    myCommand.ExecuteNonQuery();
    myTrans.Commit();
    Console.WriteLine("Both records are written to database.");
} catch(Exception e) {
    myTrans.Rollback();
    Console.WriteLine(e.ToString());
    Console.WriteLine("Neither record was written to database.");
} finally {
    myConnection.Close();
}
```

.Net Data Providers

- Verschaffen toegang tot gegevensbanken vanuit een .NET-applicatie
- Standaardacties
  - Verbinding maken met een gegevensbank
  - Opdrachten uitvoeren
  - Gegevens ophalen
- Koppeling maken tussen gegevensbanken en DataSet
  - DataSet opvullen
  - Veranderingen aan DataSet doorvoeren in gegevensbank

Sleutelobjecten

- Mogelijke acties → sleutelobjecten
  - Connection (verbinding maken)
  - Command (opdrachten uitvoeren)
  - DataReader (gegevens ophalen)
  - DataAdapter (koppeling DataSet en gegevensbank)

Interfaces sleutelobjecten

- Elke Provider implementeert bijhorende interfaces
  - System.Data.IDbConnection
  - System.Data.IDbCommand
  - System.Data.IDataReader
  - System.Data.IDbDataAdapter
- Klassen behoren tot een subnamespace van System.Data

Abstracte implementatie sleutelobjecten

- Abstracte klassen
  - System.Data.Common.DbConnection (sinds 2.0!)
  - System.Data.Common.DbCommand (sinds 2.0!)
  - System.Data.Common.DataReader (sinds 2.0!)
  - System.Data.Common.DbDataAdapter
- Implementeren corresponderende interfaces
- Klassen typisch voor een .Net Provider zijn hiervan afgeleid
- Kunnen gebruikt worden om Provider onafhankelijk te werken

.Net Data Providers

- Standaard beschikbaar
  - .NET Framework Data Provider for SQL Server
  - .NET Framework Data Provider for OLE DB
  - .NET Framework Data Provider for ODBC
  - .NET Framework Data Provider for Oracle

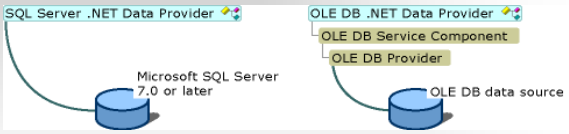
SQL Server

- .NET Framework Data Provider for SQL Server
  - SQL Server 7.0 of later
  - Communiceert direct met data-transfer-protocol van SQL Server
  - System.Data.SqlClient

OleDB

- .NET Framework Data Provider for OLE DB
  - Object Linking and Embedding for Databases
  - OLE DB gegevensbanken
    - SQL Server
    - Oracle
    - Microsoft Jet
  - Iets minder efficiënt
  - Uniforme toegang tot verschillende databronnen
  - Gebruikt OLE DB-laag om te communiceren
  - System.Data.OleDb

.Net Data Providers



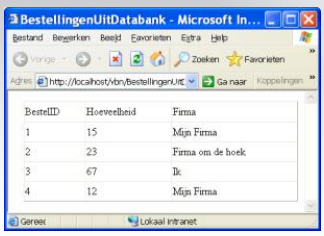
ODBC

- .NET Framework Data Provider for ODBC
  - Open Database Connectivity
  - ODBC-drivers
    - SQL Server
    - Microsoft ODBC for Oracle
    - Microsoft Access Driver (*.mdb)
  - Iets minder efficiënt
  - Uniforme toegang tot verschillende databronnen
  - Gebruikt ODBC-laag om te communiceren
  - System.Data.Odbc

Oracle

- .NET Framework Data Provider for Oracle
  - Oracle Client Software 8.1.7 of hoger
  - Performanter
  - System.Data.OracleClient

Voorbeeld gegevensbank



- Gegevens uit de gevensbank inlezen in DataSet

DataSet vullen met gegevens uit databank

- BestellingBeheer.cs
  - Constructor
    - Configuratie
    - Factory
      - Aanmaken verbinding-objecten , commando-objecten, …
  - Methode GetConnection
  - Methode GetBestellingenMetDataReader
    - DataSet opvullen
- App.config
  - Connectiestring
  - SQL-opdrachten
- Program.cs
  - Uittesten

Verschillende stappen

- Configuratiegegevens inlezen
- Factory voor de provider aanmaken
- Connectie-object aanmaken
- Commando-object aanmaken
- Commando-object uitvoeren → eventueel DataReader overlopen

Configuratiegegevens inlezen

- Configuratiebestand
  - Applicaties
    - App.config
  - Webapplicaties
    - Web.config
- Klasse ConfiguratieManager
  - Ophalen configuratie

Klasse ConfiguratieManager

- Eigenschap ConnectionStrings
  - Indexer
    - Argument: waarde name-attribuut
    - Resultaat: ConnectionStringSettings-object
      - Provider-naam
      - Connectiestring

```
ConnectionStringSettings connStringSet =
   ConfigurationManager.ConnectionStrings["bestellingDB"];

<connectionStrings>
   <add name="bestellingDB"
   providerName="System.Data.OleDb" connectionString="…"/>
 </connectionStrings>
```

Bepaalt Provider

Klasse ConnectionStringSettings

- Eigenschappen van ConnectionStringSettings
  - ProviderName
    - Identificeert de Provider
  - ConnectionString
    - connectiestring

Klasse ConfiguratieManager

- Eigenschap AppSettings
  - Indexer
    - Argument: waarde key-attribuut
    - Resultaat: string, waarde value-attribuut

```
String opdracht =
   ConfigurationManager.AppSettings["SELECT_BESTELLINGE
   N"];

<appSettings>
   <add key="SELECT_BESTELLINGEN" value="select * from
   Bestellingen"/>
</appSettings>
```


Verschillende stappen

- Configuratiegegevens inlezen
- Factory voor de provider aanmaken
- Connectie-object aanmaken
- Commando-object aanmaken
- Commando-object uitvoeren → eventueel DataReader overlopen

Factory voor de provider aanmaken

- Maakt ADO.NET-objecten voor een bepaalde provider
  - Connecties, commando's, parameters, data-adapters, …
- Statische methode GetFactory van DbProviderFactories
  - aanmaak DbProviderFactory-object op basis van naam provider (kan namespace zijn, …)

```
DbProviderFactory factory  =
   DbProviderFactories.GetFactory(connStringSet.ProviderName)
   ;
```


Verschillende stappen

- Configuratiegegevens inlezen
- Factory voor de provider aanmaken
- Connectie-object aanmaken
- Commando-object aanmaken
- Commando-object uitvoeren → eventueel DataReader overlopen

Connectie-object aanmaken

- Connectie-objecten aanmaken met DbProviderFactory-object

```
DbConnection connection = factory.CreateConnection();
```

- Connectiestring instellen

```
connection.ConnectionString =
   connStringSet.ConnectionString;
```

ConnectionStringSettings-object

Connectiestring

- Naam/waarde-paren gescheiden door ';'
- Namen:
  - Server of Data Source
  - Database
  - Password of Pwd
  - User ID
  - Trusted_Connection (false → User ID & Password)
  - Provider (verplicht bij OleDbConnection)

Alternatief aanmaken connectie-object

- Expliciet provider-klassen gebruiken
  - SqlConnection, OleDbConnection, …
  - Connectiestring
    - Meegeven met constructor
    - Toekennen via eigenschap ConnectionString

```
string connString =
   "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data
   Source=C:\\Inetpub\\wwwroot\\vbn\\klantorders.mdb";
 OleDbConnection conn =
   new OleDbConnection(connString);
```


Verschillende stappen

- Configuratiegegevens inlezen
- Factory voor de provider aanmaken
- Connectie-object aanmaken
- Commando-object aanmaken
- Commando-object uitvoeren → eventueel DataReader overlopen

Commando-object aanmaken

- Commando-objecten aanmaken met DbConnection-object

```
DbConnection conn = …
DbCommand command = conn.CreateCommand();
```

- SQL-opdracht instellen

```
command.CommandText =
    ConfigurationManager.AppSettings["SELECT_BESTELLINGE
    N"];
```


Alternatief aanmaken commando-object

- Expliciet provider-klassen gebruiken
  - SqlCommand
  - OleDbCommand
  - …

```
OleDbConnection conn = …;
String opdracht = …;
OleDbCommand opdracht = new
    OleDbCommand(opdracht,conn);
```


Command

- Nodig
  - Verbinding: object of string
  - Opdracht
  - Eventueel transactie
- Via constructor
- Of via de eigenschappen
  - CommandText
  - Connection
  - Transaction

Verschillende stappen

- Configuratiegegevens inlezen
- Factory voor de provider aanmaken
- Connectie-object aanmaken
- Commando-object aanmaken
- Commando-object uitvoeren → eventueel DataReader overlopen

Commando-object uitvoeren

- Connectie openen
- Opdracht uitvoeren
- Connectie sluiten

```
DbConnection conn = …
conn.Open();
try {
    … // commando-object uitvoeren
} finally {
    conn.Close();
}
```


Commando-object uitvoeren

- insert, update, delete SQL-opdrachten en DDL-opdrachten
  - Methode ExecuteNonQuery
    - Resultaat: aantal aangepaste rijen
- Zoekopdrachten
  - Methode ExecuteReader
    - Resultaat: DataReader

Zoekopdrachten: DataReader

- Leest gegevens van een databron
- Eén keer lezen van begin tot einde
- Interface IDataReader, abstracte klasse DbDataReader
- Afhankelijk van .Net Data Provider (implementeren IDataReader)
  - SqlDataReader, OleDbDataReader, …
- Teruggeefwaarde van de methode ExecuteReader van het Command-object

DbDataReader: voorbeeld

- Voorbeeld

```
conn.Open();
try {
    DbDataReader reader = command.ExecuteReader();
    while (reader.Read()) {
        DataRow rij = tabelBestel.NewRow();
        rij[0] = reader["BestelID"];
        rij[1] = reader[1];
        rij[2] = reader.GetString(2);
        tabelBestel.Rows.Add(rij);
    }
} finally {
    conn.Close();
}
```


DataReader

- Methodes
  - Read(): verplaatst de cursor één rij naar beneden
  - GetXxx(int volgnummer)
    - Xxx is een type vb. string, …
    - Haalt de gegevens van de kolom bepaald door volgnummer
  - Close(): sluit reader
- Indexer
  - Argument: naam of volgnummer kolom

```
Console.WriteLine(reader[0] + ", " +
    reader["Hoeveelheid"]);
```

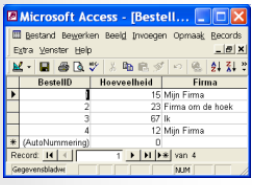

Verschillende stappen

- Configuratiegegevens inlezen
- Factory voor de provider aanmaken
- Connectie-object aanmaken
- Commando-object aanmaken
- Commando-object uitvoeren → eventueel DataReader overlopen

Sleutelobjecten

- Mogelijke acties → sleutelobjecten
  - Connection (verbinding maken)
  - Command (opdrachten uitvoeren)
  - DataReader (gegevens ophalen)
  - DataAdapter (koppeling DataSet en gegevensbank)

Voorbeeld gegevensbank



Gegevens uit de gevensbank inlezen in DataSet

DataSet vullen met gegevens uit databank herbekijken

- BestellingBeheer.cs
  - Methode GetBestellingenMetDataAdapter
- App.config
  - Connectiestring
  - SQL-opdrachten

DataAdapter

- Alternatieve wijze om om te gaan met gegevensbanken in een applicatie → lossere koppeling
  - Ophalen nodige gegevens en bewaren in DataSet
  - Manipuleren DataSet
  - Wijzigingen doorvoeren op gegevensbank
- DataAdapter: brug tussen DataSet en gegevensbank
  - Opvullen DataSet
  - Wijzigen onderliggende databron

DataAdapter-object aanmaken

- DataAdapter-objecten aanmaken met DbProviderFactory-object

```
DbDataAdapter adapter = factory.CreateDataAdapter();
```

- Zoekopdracht die de adapter gebruikt om gegevens op te halen instellen

```
DbCommand command = …
adapter.SelectCommand = command;
```


Klasse DataAdapter

- Methode Fill
  - Argumenten: DataSet en naam van de tabel in de DataSet
  - Functionaliteit
    - Openen verbinding gegevensbank
    - Query uitvoeren en resultaat bewaren als een tabel van de DataSet (met opgegeven naam)
    - Sluiten verbinding gegevensbank

```
DbDataAdapter adapter = …
DataSet klantDS = new DataSet(DS_ORDERS);
adapter.Fill(klantDS,TABEL_BESTELLING);
```


Alternatief aanmaken DataAdapter-object

- Gebruik maken van de provider specifieke klassen
  - SqlDataAdapter
  - OleDbDataAdapter
  - …
- Argumenten constructor
  - Query + connectie-object

Voorbeeld alternatief

```
OleDbConnection conn = …
string opdracht = ….
OleDbDataAdapter adapter =
   new OleDbDataAdapter(opdracht,conn);
```

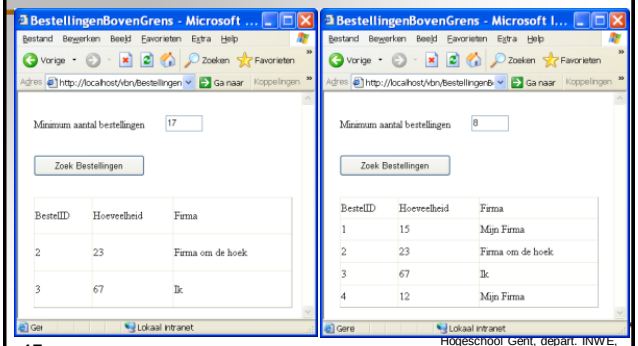
Opmerking

- De volgende objecten kunnen ook visueel aangemaakt worden
  - OleDbConnection
  - OleDbDataAdapter
  - DataSet
- Toolbox → Data

DataReader ↔ DataSet en DataAdapter

- DataReader
  - Enkel resultaten tonen
  - Performanter
  - Wijzigingen worden onmiddellijk doorgevoerd in de gegevensbank
- DataSet
  - Lokaal data bijhouden en bewerken
  - Handig in gebruik (bv. ASP.NET-pagina's)
  - Wijzigingen worden pas later doorgevoerd in de gegevensbank

Voorbeeld opdracht met parameter



Voorbeeld opdracht met parameter

- BestellingBeheer.cs
  - Methode GetBestellingen(int minAantal)
- App.config
  - Connectiestring
  - SQL-opdrachten

Zoekopdracht met parameter

- Werkwijze
  - SQL-opdracht met parameter
  - Command-object aanmaken
  - Parameter aanmaken
  - Parameter toevoegen aan Command-object
  - Parameter een waarde geven

SQL-opdracht met parameter

- Plaats parameter in SQL-opdracht aangeven.
  - Met *@naamParameter*
    - !afhankelijk van provider! (soms : …)
  - ? kan soms ook

```
string query = "select * from Bestellingen where
   Hoeveelheid > @min";
```

Parameter aanmaken en toevoegen aan command

- Parameter-object aanmaken met DbProviderFactory
- Naam en type instellen
- Toevoegen aan verzameling parameters van het commando-object

```
DbParameter parameter = factory.CreateParameter();
parameter.ParameterName = MIN;
parameter.DbType = DbType.Int32;
opdracht.Parameters.Add(parameter);
```


Parameter een waarde geven

- De eigenschap Parameters (type IDataParameterCollection) van IDbCommand
  - Houdt parameters bij
- Selecteer parameter op naam via indexer
- Geef de eigenschap Value van DbParameter een waarde

```
opdracht.Parameters[MIN].Value = minimumAantal;
```

Geheel getal

Parameter

- SQL-opdracht met parameter uitvoeren
- Herhaaldelijk uitvoer van dezelfde opdracht met andere waarden
- Gegevensconversi
- Voorkomen SQL-injectie
- Alternatief
  - Provider-klassen gebruiken: SqlParameter, OleDbParameter, …

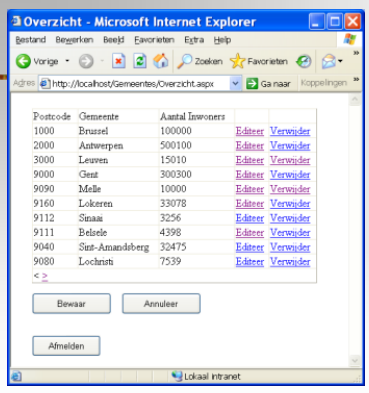
```
OleDbCommand opdracht = …
OleDbParameter param =
    opdracht.Parameters.Add("@min",DbType.Int32);
SqlParameter paramNaam = new
    SqlParameter("@naam",DbType.String);
```


Parameter

- Constructie
  - Geen argumenten
  - Naam en type
  - Naam en waarde
- Eigenschappen
  - ParameterName
  - Value
  - DbType
    - Opsommingen: DbType, SqlDbType, OleDbType,…

IDataParameterCollection

- Methode Add
- Indexer
  - Argument: string
  - Teruggeefwaarde afhankelijk van het type: object, SqlParameter, OleDbParameter



- Editeerbare tabel
- Achterliggende DataSet wordt veranderd
- Indien veranderingen OK → doorsturen naar gegevensbank

Principe

- Tabel toont gegevens van een DataSet
- DataSet wordt veranderd in de loop van het programma
  - Deze veranderingen zijn niet definitief
    - Methode RejectChanges van DataRow
    - Oorspronkelijke rij wordt bijgehouden
  - De rij wordt gemarkeerd
    - Verwijderd
    - Aangepast
    - …

    → Opsomming verschillende statussen: DataRowState

Principe

- DataAdapter
  - Opvullen DataSet
  - Aanpassingen DataSet doorvoeren in gegevensbank
    - Voor elke gemarkeerde rij wordt een opdracht uitgevoerd
  - Heeft 4 Command-objecten
    - SelectCommand, InsertCommand, UpdateCommand en DeleteCommand (Eigenschappen!!)
    - Bepalen hoe wijzigingen aan de DataSet vertaald worden in de gegevensbank

Configuratie: Web.config

- ConnectieString
- SQL-opdrachten
  - Select
  - Update
  - Delete
- Syntax

```
<appSettings>
<add key="" value=""/>
</appSettings>
```

- Ophalen configuratie

```
String connString =
    ConfigurationManager.AppSettings["connString"];
```

Waarde attribuut value

Waarde attribuut key

Aanmaak DataAdapter

- In Global.asax.cs
- Maakt gebruik van de specifieke DataAdapter OleDbAdapter
- Methode MaakAdapter

Aanmaak DataAdapter

- SelectCommand
  - Ingesteld bij constructie data-adapter
  - Primaire sleutels ook ophalen

```
adapter.MissingSchemaAction =
    MissingSchemaAction.AddWithKey;
```

- UpdateCommand
  - Hoe gegevens van een aangepaste rij doorsturen naar de gegevensbank
- DeleteCommand
  - Hoe gegevens van een verwijderde rij verwijderen in de gegevensbank

Aanmaak Command-object

- Constructor
  - SQL-opdracht (String)
  - OleDbConnection-object

```
String connString =
    ConfigurationManager.AppSettings["connString“];
OleDbConnection conn = new OleDbConnection(connString);
String update = ConfigurationManager.AppSettings["update“];
OleDbCommand command = new OleDbCommand(update,
    conn);
```

Aanmaak Command-object

- Instellen parameters
  - Aanmaak parameter
    - Naam
    - Type

```
param = command.Parameters.Add("@naam",
   OleDbType.VarChar);
```

  - Verbinden met een kolom in de DataSet

```
param.SourceColumn = "naam";
```

  - Aangeven welke versie gebruikt moet worden bij uitvoeren opdracht
    - Oude ↔ nieuwe waarde

```
param.SourceVersion = DataRowVersion.Current;
```


OleDbParameter

- Eigenschap SourceColumn
  - Welke kolom bevat de waarde van de parameter
- Eigenschap SourceVersion
  - bepaalt welke versie van de kolom gebruikt wordt bij het uitvoeren van de opdracht

  → opsomming DataRowVersion
    - Current, Original, …

Wijzigingen doorvoeren

- Markeren van elke rij die toegevoegd, gewijzigd of verwijderd wordt
- Ongedaan maken wijzigingen
  - Methode RejectChanges van DataRow
- De methode Update van de DataAdapter past alle gemarkeerde rijen uit de DataSet aan in de gegevensbank

```
adapter.Update(gemeenteDS);
```


Zoeken van een rij in een DataSet

- De sleutels moeten ook opgehaald worden
  - Instellen in DataAdapter
- DataRowCollection
  - Methode Find
    - Argument: object of tabel van objecten die de primaire sleutel voorstellen
    - Resulaat: DataRow horende bij de opgegeven primaire sleutel

```
DataRow dr =
  gemeenteDS.Tables[0].Rows.Find(postcode);
```


Transaction

- Verschillende gegevensbankopdrachten moeten samen uitgevoerd worden
- Analoog verschillende .Net Data Providers
- Verschillende stappen
  - Starten transactie → transactie-object
  - Transactie-object toekennen aan Command-objecten
  - Uitvoeren opdrachten
  - Commit - Rollback

Transaction

```
// verbinding met gegevensbank
SqlConnection conn;
...
conn.Open();

// start transactie
SqlTransaction trans = conn.BeginTransaction();

// commando aanmaken en toevoegen aan transactie
SqlCommand opdracht = new SqlCommand();
opdracht.Connection = conn;
opdracht.Transaction = trans;
```

Transaction

```
// opdrachten uitvoeren
try {
    opdracht.CommandText = "Insert into Region (RegionID,
      RegionDescription) VALUES (100, 'Description')";
    opdracht.ExecuteNonQuery();
    opdracht.CommandText = "Insert into Region (RegionID,
      RegionDescription) VALUES (101, 'Description')";
    opdracht.ExecuteNonQuery();
    trans.Commit();
} catch(Exception e) {
    trans.Rollback();
} finally {
    conn.Close();
}
```


Informatie over ADO.NET

- Accessing Data with ADO.NET (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaccessingdatawithadonet.asp)
- ADO.NET Overview (http://samples.gotdotnet.com/quickstart/aspplus/doc/adoplusoverview.aspx)
- .Net Data Access Architecture Guide (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp)

Deel III

Webtechnologie

Hoofdstuk 12

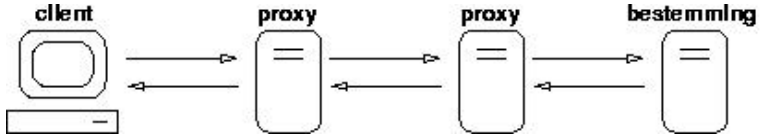
Het HTTP-protocol

HTTP (*Hypertext Transfer Protocol*) is het protocol dat wordt gebruikt om informatie uit te wisselen op het wereldwijde web. Sinds haar ontstaan hebben we reeds drie verschillende 'officiële' versies van dit protocol gekend: versies 0.9, 1.0 en 1.1 (gepubliceerd in 1999). HTTP is een standaard van het *World Wide Web Consortium* ([W3C]), een internationaal consortium van commerciële en academische partners die webstandaarden ontwikkelen.

In deze bladzijden beperken we ons hoofdzakelijk tot versie 1.1. Deze versie staat beschreven in RFC 2616 ([FIG+99]), een document van 176 bladzijden waarvan we hier slechts enkele belangrijke punten overlopen. Voor een aantal andere interessante aspecten die we hier niet behandelen, zoals de werking van een proxyserver en de verschillende cachetechnieken, verwijzen we naar deze RFC.

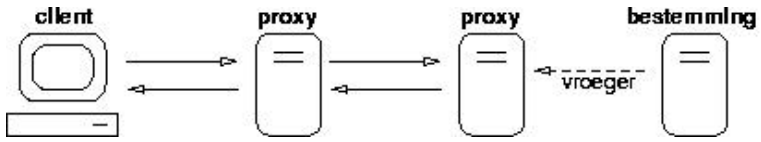
Het HTTP-protocol is een client/server-protocol. Het maakt meestal gebruik van TCP (poort 80) hoewel de specificatie elk betrouwbaar transportprotocol toelaat. HTTP is een aanvraag/antwoord-protocol. Met andere woorden, de client en de server sturen elkaar om beurt berichten. Aanvragen van de client worden afgewisseld met antwoorden van de server.

In vele gevallen stuurt de client zijn HTTP-aanvraag rechtstreeks naar de server die de bewuste informatie bevat, maar dit is niet noodzakelijk. Aanvragen en antwoorden kunnen ook via tussenliggende HTTP-servers aan de bestemming worden doorgegeven. Dergelijke servers noemt men *proxy*servers.



Figuur 12.1: Proxyserver

Een aanvraag hoeft niet noodzakelijk de oorspronkelijke bestemming te bereiken. In sommige gevallen kan een tussenliggende server zelf een antwoord op de aanvraag terugsturen. Dit is bijvoorbeeld het geval wanneer berichten onderweg worden ge*cached*.



Figuur 12.2: Caching

De berichten die worden verstuurd hebben een formaat dat veel weg heeft van wat er bij het elektronisch mailprotocol SMTP wordt gebruikt. Hieronder tonen we een voorbeeld van een antwoordbericht dat werd verstuurd van server naar client, nadat de client de webpagina met adres *http://localhost/student* had aangevraagd:

```
HTTP/1.1 301 Moved Permanently
Date: Sat, 16 Feb 2002 19:20:54 GMT
Server: Apache/1.3.20 (Unix)  (Red-Hat/Linux)
Location: http://watson.rug.ac.be/student/
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved
<A HREF="http://watson.rug.ac.be/student/">here</A>.
</BODY></HTML>
```

Elk HTTP-bericht begint met een lijst van *header*lijnen, afgesloten met een lege lijn. Het einde van elke lijn wordt aangegeven met een CRLF-combinatie: een *carriage return* gevolgd door een *line feed*. In bovenstaand voorbeeld begint de eerste headerlijn dus met `HTTP/1.1` en de laatste met `Content-Type`.

Na deze hoofding volgt (eventueel) een corpus (*message body*). Van dit corpus wordt eigenlijk geen interne structuur verondersteld. HTTP behandelt het als een gewone opeenvolging van bytes. Vaak gebruikt men echter ook MIME om de inhoud van dit bericht te coderen. Bij dit voorbeeld bevat het corpus een foutbericht, opgesteld in HTML (met MIME-type `text/html`), dat aangeeft dat de naam van de server niet *localhost* is maar *watson.rug.ac.be*.

De eerste headerlijn van een bericht heeft een bijzonder formaat en ziet er anders uit bij aanvragen dan bij antwoorden. De andere headers hebben dezelfde algemene vorm: eerst komt de *naam* van de header, daarna een dubbele punt en een spatie en tenslotte de *waarde* van de header, afgesloten met CRLF.

12.1 HTTP-aanvragen en HTTP-antwoorden

HTTP-aanvragen (Engels: *HTTP-request*) beginnen steeds met een lijn van de volgende vorm:

```
GET http://gonzo.hogent.be/intranet HTTP/1.1
HEAD /intranet HTTP/1.1
POST /bin/gastenboek.pl HTTP/1.1
```

Een aanvraag bestaat uit drie delen, onderling van elkaar gescheiden door één enkele spatie. Het eerste deel is de zogenaamde *aanvraagmethode* (*request method*) in hoofdletters. De drie methoden GET, HEAD en POST worden het meest gebruikt, maar er zijn ook andere (zie §12.4).

Het tweede gedeelte van de HTTP-aanvraag is een *URL* (*Uniform Resource Locator*) die de *bron* aangeeft waarover de aanvraag handelt (zie ook §13.1). De RFC gebruikt hier eigenlijk de term *URI* (*Uniform Resource Identifier*) wat nogal eens tot spraakverwarring leidt met termen die elders gangbaar zijn.

In het eerste voorbeeld specifieert de URL zowel protocol, computernaam als padnaam. Deze vorm wordt in principe enkel toegepast bij aanvragen aan proxyservers, maar werd ook heel courant gebruikt in versie 1.0 van het protocol. In het gewone geval gebruikt HTTP een URL zonder protocolindicatie en zonder servernaam. Dit noemt men een *absoluut pad* (Engels: *absolute path*). De naam van de server wordt in dat geval opgegeven in een afzonderlijke `Host`-header (zie §12.3.1).

Het laatste onderdeel van de eerste headerlijn bevat de protocolversie (let op de notatie). De meeste HTTP-servers zijn ook in staat om berichten van oudere protocolversies te verwerken.

Een HTTP-antwoordbericht begint met een zogenaamde *statuslijn*.

```
HTTP/1.1 200 OK
HTTP/1.1 301 Moved Permanently
```

Een statuslijn bestaat uit een protocolversie, een spatie, een decimale foutcode van 3 cijfers, opnieuw een spatie en een kort tekstbericht. Dit bericht is een leesbare 'vertaling' van de foutcode en is verder voor het protocol niet van belang. Het eerste cijfer van de code geeft de 'ernst' van de fout aan, zoals opgegeven in onderstaande tabel.

`1??`	Informatief	Aanvraag ontvangen, verdere berichten volgen
`2??`	Succes	De aanvraag werd met succes afgehandeld.
`3??`	Omleiding	De aanvraag zal moeten opnieuw gebeuren.
`4??`	Fout bij client	De aanvraag is verkeerd.
`5??`	Fout bij server	De aanvraag was geldig, maar de server kan ze niet afhandelen

Een aantal statuscodes komt regelmatig terug:

200 OK Dit betekent dat de aanvraag is gelukt. Het corpus van het antwoord bevat dan de gegevens die werden aangevraagd, bijvoorbeeld de HTML-tekst die bij een bepaalde URL hoort.

301 Moved Permanently, 302 Found, 303 See Other De aanvraag was wel gelukt, maar moet opnieuw gebeuren om dat de gevraagde entiteit zich niet meer op de oorspronkelijke plaats bevindt (al dan niet permanent of tijdelijk). Deze statuscodes worden vaak gebruikt als antwoord op een GET-methode (zie §12.4).

In elk van deze gevallen wordt er een nieuwe lokatie bij het antwoord gevoegd in een `Location`-header van de volgende vorm:

```
Location: http://watson.rug.ac.be/student/
```

Het is de bedoeling dat de HTTP-client onmiddellijk deze nieuwe lokatie aanvraagt en het resultaat hiervan op het scherm toont, liefst zonder eerst de gebruiker te verwittigen, want deze statuscodes wijzen niet op een echte fout.

401 Unauthorized Dit antwoord wordt door de server gegeven wanneer de gebruiker geen toegang krijgt tot de gevraagde bron. Als resultaat moet de client een wachtwoord vragen aan de gebruiker en de aanvraag opnieuw proberen (zie §12.3.2).

403 Forbidden Dit antwoord betekent dat de bron niet toegankelijk is voor de client, zelfs niet met het juiste wachtwoord.

404 Not Found De gevraagde bron bestaat niet (meer). Je krijgt dit foutbericht bijvoorbeeld te zien wanneer je in je browser een onbestaande URL hebt ingetikt.

501 Not Implemented Dit type aanvraag wordt door deze serversoftware (nog) niet ondersteund. Dit antwoord krijg je bijvoorbeeld wanneer je in de eerste lijn van de aanvraag een methode opgeeft die de server niet herkent.

12.2 Het corpus

Zoals reeds gezegd, is het corpus van een bericht niets anders dan een stroom bytes die door de server of client zelf niet hoeft te worden geïnterpreteerd. Het begin van het corpus wordt aangegeven door de blanco lijn die ook het einde van de headers aangeeft. Omdat het corpus echter zelf een willekeurige opeenvolging van bytes kan bevatten, kan men echter het einde van het corpus niet met een dergelijke markering aanduiden.

In de plaats gebruikt het protocol hiervoor één van twee mogelijke technieken. Ten eerste kan men de lengte (in bytes) van het corpus aangeven in een afzonderlijke `Content-Length`-header.

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Content-Type: text/html
Content-Length: 410
```

```
<html>
...
</html>
```

Om de ontwikkeling van serverprogrammatuur eenvoudiger te maken, voorziet men echter sinds versie 1.1 nog een tweede manier om dit corpus te versturen, genaamd *chunked transfer coding* (dit wordt aangegeven door een speciale headerlijn). Hierbij is het corpus in verschillende 'brokken' (*chunks*) verdeeld.

```
HTTP/1.1 301 Moved Permanently
Date: Sat, 16 Feb 2002 20:18:00 GMT
Server: Apache/1.3.20 (Unix)  (Red-Hat/Linux)
Location: http://watson.rug.ac.be/student/
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

f2
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved
<A HREF="http://watson.rug.ac.be/student/">here</A>.
</BODY></HTML>

0
```

Elk brok begint met een reeks hexadecimale cijfers die de totale lengte aangeeft (in bytes) van het brok dat volgt. Er kunnen meerdere brokken na elkaar volgen (die dan bijvoorbeeld met tussenpozen over de netwerkverbinding worden verstuurd). De lijst wordt afgesloten met een brok van lengte nul en een lege lijn. In bovenstaand voorbeeld hebben we eerst een brok van 242 bytes, hexadeximaal f2, en tot slot een brok van 0 bytes.

Net zoals SMTP maakt HTTP gebruik van MIME om onderscheid te maken tussen de verschillende soorten gegevens die in het corpus worden overgestuurd. Omdat HTTP een 8-bit protocol is, is er geen nood aan een `Content-Transfer-Encoding`-header. Een `Content-Type`-header (met dezelfde betekenis als in MIME) is echter verplicht.

12.3 Headers

De RFC beschrijft 47 verschillende headerlijnen en deze lijst is bovendien niet beperkend: bepaalde toepassingen voegen nog hun eigen headers aan deze lijst toe (bijvoorbeeld cookies, zie §13.4). We geven hieronder een kort overzicht van de meest gebruikte headers en hun betekenis.

Een aantal headers hebben we reeds ontmoet: De `Location`-header waarmee de client naar een andere URL wordt doorverwezen en de `Content-Length`-, `Transfer-Encoding`- en `Content-Type`-headers die de vorm en betekenis van het corpus vastleggen. Daarnaast vermelden we ook nog de `Date`-header die het tijdstip aangeeft waarop het bericht werd opgesteld en de `Expires`-header die aangeeft hoelang het bericht geldig blijft (en dus een belangrijke rol speelt bij *caching*).

Trouwens, de gemakkelijkste manier om er zeker van te zijn dat je gegevens onderweg niet zullen worden gecached, is het toevoegen van de volgende header aan je aanvraag:

```
Cache-Control: no-cache
```

Deze header wordt echter niet begrepen door oudere clients of servers. Bij het HTTP/1.0-protocol moet je in de plaats het volgende schrijven:

```
Pragma: no-cache
```

Om beide versies van het protocol te ondersteunen, kan je best beide headerlijnen tegelijk opgeven.

12.3.1 Virtual hosts

Dikwijls zal dezelfde webserver dienst doen als HTTP-server voor verschillende instellingen, denk maar aan de webserver van een Internetprovider die vele honderden websites van kleine bedrijven bevat. Meestal heeft elke instelling haar eigen domeinnaam, met als gevolg dat ook de webserver onder verschillende namen moet gekend zijn. Men zegt dat één fysische server de rol speelt van verschillende *virtuele* servers (Engels: *virtual hosts*). Dit zorgt voor een aantal bijkomende complicaties.

Zo stellen de namen *qed.rug.ac.be* en *twizz.rug.ac.be* bijvoorbeeld dezelfde machine voor. Je krijgt echter een verschillend resultaat te zien, wanneer je de URLs *http://qed.rug.ac.be/* of *http://twizz.rug.ac.be/* intikt.

Aangezien de eerste lijn van een HTTP-aanvraag enkel het absolute pad bevat, en niet de volledige URL, begint de HTTP-aanvraag voor beide URLs op dezelfde manier. De machinenaam wordt dan in een afzonderlijke `Host`-header doorgegeven:

```
GET / HTTP/1.1
Host: qed.rug.ac.be
```

Deze header is sinds HTTP/1.1 verplicht. De enige mogelijkheid om virtual hosts te implementeren in oudere versies van HTTP, was door elke virtuele servernaam met een afzonderlijke IP-adres te verbinden op dezelfde fysische machine, wat niet altijd door het bedrijfssysteem werd ondersteund.

12.3.2 Authorisatie en authenticatie

Niet alle webpagina's op een server zijn zomaar altijd voor iedereen toegankelijk. In sommige gevallen vraagt de browser hiervoor een identificatie (*user ID*) en wachtwoord.

HTTP voorziet twee verschillende wachtwoordschema's die in deze context gekend staan als '*basic*' en '*digest*'. Hoewel het basic schema helemaal niet zo veilig is (omdat het wachtwoorden in ongecodeerde vorm over het netwerk verstuurt) is het voorlopig het enige dat door bijna alle browsers en servers wordt ondersteund. Meer informatie over het digest schema vind je in RFC 2617.

Het HTTP-protocol gebruikt wachtwoorden op de volgende manier: wanneer een client de eerste keer een URL aanvraagt die niet publiek toegankelijk is, antwoord de server met een 401-status (`Unauthorized`). In dit antwoordbericht bevindt zich ook een `WWW-Authenticate`-header. Deze header heeft de volgende vorm:

```
WWW-Authenticate: Basic realm=TripleEyeIntranet
```

De eerste parameter bepaalt het gebruikte wachtwoordschema. De tweede parameter duidt het zogenaamde *rijk* aan waarover deze authenticatie moet gelden (Engels: *realm*). Een dergelijk rijk duidt meestal een bepaalde toepassing aan of een bepaalde groep bestanden op de server.

Het is de bedoeling dat de gebruiker zich slechts één keer in een bepaald rijk hoeft aan te melden en dat hem daarna alle informatie over dat rijk ter beschikking staat. Het alternatief zou betekenen dat de gebruiker een wachtwoord moet geven bij elke URL uit het betreffende rijk. Dus niet alleen bij elke nieuwe pagina, maar ook voor elke afbeelding en elk ikoon dat op die pagina staat. Wat een bepaald rijk precies behelst, wordt geconfigureerd op de server (en maakt geen deel uit van het HTTP-protocol).

Wanneer de client een bericht terugkrijgt met een `WWW-Authenticate`-header, vraagt hij aan de gebruiker een identificatie en wachtwoord voor het opgegeven rijk (tenzij dit vroeger binnen dezelfde sessie reeds is gebeurd). De client doet daarna een nieuwe aanvraag van dezelfde URL, maar voegt de wachtwoordgegevens toe in een `Authorization`-header. Bijvoorbeeld:

```
GET /intranet HTTP/1.1
Host: gonzo.hogent.be
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
...
```

Identificatie en wachtwoord worden eerst geconcateneerd met een scheidende dubbele punt (in bovenstaand voorbeeld werd dit '`Aladdin:open sesame`') en daarna omgezet in base64 formaat, hetzelfde formaat dat ook in MIME wordt gebruikt.

Volgt er op dit nieuwe bericht opnieuw een 401-statusbericht, dan betekent dit dat de doorgespeelde authenticatie door de server niet aanvaard wordt (bijvoorbeeld omdat het wachtwoord verkeerd is).

Het staat de client vrij om in een volgende aanvraag aan de server meteen al de `Authorization`-header door te sturen om een antwoord met een 401-status te vermijden. Meestal zal de client dit automatisch doen voor URLs waarvan de padnaam de oorspronkelijke padnaam als prefix bevat (m.a.w., URLs die er uitzien als subdirectories van de eerste directory waarvoor een authenticatie werd gevraagd).

Tot slot is het ook mogelijk authenticatie uit te voeren *buiten* het protocol. Zo kan de webprogrammeur deze authenticatie zelf programmeren aan de hand van inlogformulieren als onderdeel van de webtoepassing. Dit noemt men *form based authentication* en wordt door de meeste moderne webcontainers ondersteund.

Anderzijds kan je ook gebruik maken van het HTTPS-protocol, de versie van HTTP die gebruik maakt van de *Secure Socket Layer* (SSL). SSL biedt verschillende mechanismen waarmee je kan verifiëren dat een gebruiker wel degelijk is wie hij beweert te zijn.

12.3.3 Persistente connecties

In versies 0.9 en 1.0 van HTTP werd er telkens een nieuwe verbinding opgezet voor elke URL die door de client werd opgevraagd. Een typische gegevensoverdracht verliep dus op de volgende manier:

- De verbinding tussen client en server wordt geopend.
- De client stuurt een aanvraag met daarin de URL die hij wil raadplegen.
- De server stuurt een antwoord terug met de gevraagde gegevens in het corpus, of een foutbericht met meer informatie.
- De verbinding wordt afgesloten.

Dit betekent dat er vaak achtereenvolgens verschillende verbindingen met dezelfde server moeten worden geopend en opnieuw gesloten, bijvoorbeeld wanneer men een webpagina opvraagt met daarin een aantal afbeeldingen.

Omdat deze techniek een overlast betekent voor het netwerk, voorziet versie 1.1 van HTTP nu de mogelijkheid om de verbinding open te houden en tegelijkertijd meerdere aanvragen te doen over dezelfde verbinding.

Bij elke aanvraag hoort nog steeds een afzonderlijk antwoord, maar nu hoeft de verbinding niet telkens opnieuw afgesloten en daarna opnieuw opengemaakt te worden. De client mag zelfs de verschillende aanvragen vlak na elkaar sturen, zonder telkens eerst op een antwoord van de server te wachten (dit principe heet *pipelining*).

Om dergelijke *persistente verbindingen* mogelijk te maken, werd de `Connection`-header geïntroduceerd. De volgende headerlijn:

```
Connection: close
```

geeft aan dat ofwel de client, ofwel de server, de verbinding wil afsluiten na het volgende antwoordbericht.

Meestal staat deze header dus in de laatste van een reeks aanvragen van de client. Servers of clients die om één of andere reden geen persistente verbinding wensen te gebruiken (bijvoorbeeld omdat dit concept nog niet is geïmplementeerd) nemen deze header op in elk bericht dat ze versturen.

Merk op dat de meeste servers ook een bepaalde time-outwaarde in acht nemen. Wanneer dit tijdsinterval overschreden is, sluiten ze de verbinding vanzelf af, ook zonder dat ze een `Connection`-header hebben te zien gekregen. Dit gedrag wordt door HTTP toegelaten, clients en servers moeten dus steeds met deze mogelijkheid rekening houden.

12.4 Aanvraagmethoden

Van de verschillende aanvraagmethoden (*request methods*) die door HTTP worden ondersteund, bespreken we hier slechts de voornaamste. We herinneren je eraan dat het type van de aanvraag het eerste woord is op de eerste lijn van een aanvraagbericht.

GET Hiermee geeft de client te kennen dat hij een bepaalde bron wil raadplegen waarvan de inhoud door de server moet worden teruggestuurd. De aanvraag bevat enkel headerlijnen maar geen corpus.

Vaak correspondeert de gevraagde bron met een bestand op de server, maar bij echte webtoepassingen worden GET-methoden ook voor meer complexe aanvragen gebruikt, zoals raadplegingen in een gegevensbank. Een GET mag in principe echter nooit een verandering op de server als zij-effect hebben. (Dit betekent onder andere dat GET-aanvragen ook mogen beantwoord worden door proxyservers die de corresponderende gegevens hebben gecached.)

Wanneer je een URL opgeeft aan een browser of wanneer je op een link in een HTML-pagina klikt, wordt er een GET-aanvraag doorgestuurd. De GET-methode kan ook gebruikt worden bij HTML-formulieren (zie ook §13.3).

HEAD HEAD vraagt informatie over dezelfde soort bronnen als GET, maar nu bestaat het antwoordbericht van de server enkel uit headerlijnen en bevat het geen corpus. De headerlijnen zijn dezelfde als bij een GET met dezelfde URL, inclusief de `Content-Length`-header.

Deze methode wordt door een browser voornamelijk gebruikt om op voorhand de grootte van een bepaalde bron te weten te komen, of om te weten of een bepaald document is gewijzigd sinds de laatste keer dat het werd opgevraagd. In dit laatste geval kan dan bijvoorbeeld in de plaats het document worden gebruikt dat zich nog in de cache bevindt.

PUT Dit is in zekere zin het omgekeerde van GET. Hiermee kan de client een bestand (of een andere bron) integraal naar de server overbrengen met de bedoeling dat deze als nieuwe bron op de server wordt aangemaakt om dan door latere GETs van andere gebruikers te worden geraadpleegd.

Bij deze methode heeft de aanvraag ook een corpus. Sommige webpagina-editors bieden de mogelijkheid om je eigen pagina's op die manier naar een server te uploaden (in plaats van met het traditionele FTP).

Hoewel de oorspronkelijke ontwerpers van het web deze functionaliteit zeer belangrijk vonden, zijn er echter slechts weinig server-implementaties die deze methode ondersteunen.

POST Deze methode laat toe om bepaalde gegevens op de servermachine aan te passen, zonder dat daarom meteen volledige bestanden moeten worden doorgestuurd of dat er als resultaat een nieuwe bron wordt gecreëerd zoals bij PUT. Met POST kan je bijvoorbeeld gegevens toevoegen aan een databank of elektronische betalingen verrichten.

Deze methode wordt meestal toegepast in combinatie met HTML-formulieren die door de gebruiker worden ingevuld (§13.3). De inhoud van dit formulier wordt dan naar de server verstuurd als corpus van de aanvraag.

In tegenstelling tot GET, wordt hier dus wel een zij-effect op de server verwacht en daarom worden POST-aanvragen doorgaans onderweg niet gecached.

12.5 WebDAV

Hoewel de eerste prototypebrowser van Tim Berners-Lee (de vader van het WereldWijde Web) reeds zowel lees- als schrijfcapaciteiten bezat (GET en PUT), blijft het web tot nog toe voornamelijk een instrument om informatie te raadplegen en niet zozeer om informatie met anderen uit te wisselen. Een nieuwe Internetwerkgroep (gesteund door belangrijke softwarehuizen zoals Microsoft, IBM en Netscape) wil daarin verandering brengen met de introductie van een nieuwe versie van het HTTP-protocol genaamd *WebDAV*, afkorting van *World Wide Web Distributed Authoring and Versioning* ([WEBb], [GMW+99], [Whi98]).

Met behulp van een aantal bijkomende HTTP-aanvraagmethoden breidt WebDAV het HTTP-protocol uit met zes nieuwe *features*. De eerste drie worden ondertussen reeds door verschillende webservers ondersteund en hun specificaties zijn min of meer definitief.

- HTTP laat in principe toe om niet alleen documenten van het net te downloaden, maar ook om nieuwe versies van die documenten op het web te plaatsen. Dit opent de weg om documenten door verschillende auteurs tegelijkertijd te laten bewerken. WebDAV introduceert hierbij de mogelijkheid om documenten (tijdelijk) te *vergrendelen* (Engels: *to lock*) zodat ze niet per ongeluk door een andere auteur kunnen worden gewijzigd terwijl iemand reeds veranderingen aan het aanbrengen is.

- Behalve de inhoud van het document zal je met WebDAV ook *eigenschappen* (Engels: *properties*) van een document op het web kunnen opslaan en opvragen: de auteur, het onderwerp, een aantal sleutelwoorden, enz. Bij HTML-pagina's kan je dergelijke meta-informatie nu reeds opslaan als onderdeel van de hoofding, maar bij WebDAV is dit onafhankelijk van het documenttype, dus werkt dit ook bij PDF-bestanden, afbeeldingen, geluidsfragmenten, enz.

- Met WebDAV zal de URL-hiërarchie zich nog meer gedragen als een bestandshiërarchie: je kan documenten van naam veranderen (m.a.w., zorgen dat hetzelfde document een andere naam krijgt), documenten kopiëren of verplaatsen en overzichten opvragen van alle URLs die thuishoren in een bepaalde abstracte 'directory' (die in deze context echter de naam *collection* draagt).

- Daarnaast zal WebDAV ook voorzien in *versiebeheer*: je zal niet alleen het versienummer van een document kunnen opvragen, maar ook oudere versies van hetzelfde document kunnen nog toegankelijk zijn.
- In de toekomst voorziet men ook meer ingewikkelde vormen van collecties. Men denkt aan het gebruik van *soft links* en de introductie van geordende collecties.
- Tot slot komt er ook een standaard voor toegangscontrole: niet iedereen zal zomaar elk bestand kunnen aanpassen of overschrijven. Deze toegangscontrole zou gebaseerd worden op de digest authenticatie (zie §12.3.2 die reeds in HTTP/1.1 is verweven.)

Hoofdstuk 13

HTTP in de praktijk

Hoewel RFC 2616 ([FIG+99]) hierover zwijgt, moet je in de praktijk bij het ontwerp en de implementatie van webtoepassingen ook nog van een aantal andere conventies op de hoogte zijn. Deze vormen het onderwerp van dit hoofdstuk.

13.1 Structuur van een URL

Een URL is een string waarmee je op eenduidige wijze een bepaalde *bron* (Engels: *resource*) aanduidt. In deze paragraaf geven we een kort overzicht van de verschillende onderdelen waaruit een dergelijke URL bestaat. Meer (en meer exacte) informatie vind je in RFC 2396 ([BLMF+98]).

Elke URL begint met een zogenaamde *schema*-component (Engels: *scheme*), afgesloten met een dubbele punt. Veelgebruikte schema's zijn bijvoorbeeld `http`, `https`, `ftp`, `mailto`. Het is dit schema dat uiteindelijk bepaalt hoe de URL er verder uitziet. We beperken ons hier tot het schema `http`.

Onderstaande figuur geeft aan wat de verschillende delen zijn waaruit een HTTP-URL bestaat.

Figuur 13.1: Structuur URL

Servernaam (Engels: *host name*) en *poort*nummer (Engels: *port*) spreken voor zich. Wanneer dubbele punt en poort worden weggelaten, veronderstelt men dat poortnummer 80 wordt gebruikt.

Het *pad* (Engels: *path*) is hiërarchisch gestructureerd zoals bij een bestandsnaam (onder de UNIX conventies). In vele gevallen correspondeert de URL ook werkelijk met de naam van een bestand op

de server maar in principe staat het de server vrij om dit pad op een andere manier te interpreteren (wat bij J2EE ook meestal gebeurt).

Na het pad volgt een optionele *querystring* voorafgegaan door een vraagteken. In de praktijk komt dit gedeelte voor bij URLs die niet naar een statische webpagina verwijzen, maar naar een webcomponent (of programma) op de server en zorgt het in dat geval voor extra uitvoeringsparameters. De webapplicatie kan dan aan de hand van deze informatiestring bijvoorbeeld bepaalde gegevens opzoeken in een databank. Vandaar de naam *query*string.

In de meeste webtoepassingen wordt een querystring nog verder gestructureerd als een lijst van naam/waarde-paren (zie §13.2).

Tot slot kan de URL ook nog een *fragment*identifier bevatten (voorafgegaan door een kruis). Deze identifier specifieert een bepaald onderdeel van de bron, bijvoorbeeld een bepaalde lijn in een HTML-document. Officieel hoort dit gedeelte niet meer tot de URL en wordt er ook door de webserver of webcontainer geen rekening mee gehouden.

De fragmentidentifier is er enkel voor de browser en heeft pas effect nadat de bron door de client reeds is opgehaald: de client haalt het volledige document op, maar beeldt bijvoorbeeld slechts een fragment af.

Wat we hierboven kort hebben beschreven zijn zogenaamde *absolute* URLs, die rechtstreeks kunnen worden geïnterpreteerd. Daarnaast zijn er ook *relatieve* URLs, waarbij bepaalde delen van de string zijn weggelaten (bijvoorbeeld schema, servernaam en protocol). De ontbrekende delen worden dan afgeleid uit de context waarin de URL zich bevindt (bijvoorbeeld het adres van de webpagina waarin de URL als link voorkomt).

Het is meestal de webclient die relatieve URLs omzet naar absolute URLs. Webservers krijgen in principe geen relatieve URLs te verwerken.

13.2 Naam/waarde-paren

Wanneer je vanuit een webclient informatie wenst door te geven aan een webtoepassing, zijn er een aantal mogelijkheden:

- Je gebruikt de GET-aanvraagmethode en geeft eventueel extra parameters mee aan de webtoepassing via de querystring van de URL.
- Je gebruikt de POST-aanvraagmethode en geeft extra parameters mee via het corpus van de HTTP-aanvraag. Voor het MIME-type van het bericht zijn er twee mogeljkheden: ofwel `application/x-www-form-urlencoded` (de default), ofwel `multipart/form-data`. Dit laatste type wordt gebruikt wanneer de aanvraag ook bestanden bevat in bijlage (als *attachment*).

  Merk op dat je daarnaast ook nog steeds een querystring in de URL kan gebruiken, maar dit is veelal verwarrend.

- Je geeft extra informatie door via het pad. De server kan bijvoorbeeld zo zijn geconfigureerd dat het eerste gedeelte van het pad de webcomponent aanduidt, zodat je de rest van het pad voor eigen doeleinden kan benutten. Dit wordt in de praktijk echter weinig toegepast, tenzij het bijvoorbeeld belangrijk is dat de URL er uitziet als de URL van een statische webpagina. We gaan hier verder niet dieper op in.

Het voordeel van de eerste methode is dat alle gegevens voor de webtoepassing zich in de URL zelf bevinden. Je kan een dergelijke URL dus gewoon intikken in een browser of gebruiken als webadres voor een link in een statische webpagina. Met de tweede methode kan je echter meer volumineuze gegevens naar de webserver versturen. Dit kan nu niet meer rechtstreeks, maar wel bijvoorbeeld via een webformulier in een browser (zie §13.3). Webformulieren ondersteunen trouwens ook de eerste methode.

In beide gevallen worden gegevens doorgestuurd als een lijst van *naam/waarde-paren*. Je kan je dit voorstellen als namen van parameters voor de webtoepassing samen met de actuele waarden voor deze parameters. (Het is echter toegelaten om meerdere keren dezelfde naam in dezelfde lijst te gebruiken.)

Wanneer je een querystring gebruikt, of een POST-methode met het standaard MIME-type dan wordt deze lijst voorgesteld als een string van de volgende vorm:

```
naam1=waarde1&naam2=waarde2&...&naamN=waardeN
```

Bij een POST met MIME-type `multipart/form-data` worden echter de MIME-conventies gevolgd voor berichten die uit meerdere delen bestaan. Het HTTP-bericht heeft dan een corpus dat er zo uitziet:

```
-----------------------------168169277784693088618042893 83
Content-Disposition: form-data; name="naam1"

waarde1
-----------------------------168169277784693088618042893 83
Content-Disposition: form-data; name="naam2"

waarde2
...
-----------------------------168169277784693088618042893 83
Content-Disposition: form-data; name="naamN"

waardeN
-----------------------------168169277784693088618042893 83--
```

We merken nogmaals op dat je in principe GET als methode gebruikt wanneer je enkel informatie wenst op te vragen zonder ze te wijzigen. Gebruik POST wanneer je met de HTTP-aanvraag gegevens op de server aanpast of toevoegt.

Wanneer naam/waarde-paren tekens bevatten die in een URL een bijzondere betekenis hebben (zoals vraagtekens, ampersands en spaties), moet je bijzondere voorzorgen nemen bij het opstellen van de querystring. Men gebruikt hiervoor de zogenaamde *URL-codering* (Engels: *URL encoding*):

- Speciale lettertekens worden voorgesteld als een procentteken gevolgd door twee hexadecimale cijfers die samen de ASCII-code van dit letterteken representeren.

- Spaties kunnen op bovenstaande manier worden voorgesteld, of anders als plustekens.

De meeste programmeertalen of -omgevingen bieden klassen of methoden aan waarmee je deze omzetting (in beide richtingen) kan uitvoeren. URL-codering wordt eveneens gebruikt bij de POST-methode met het standaard MIME-type (dat dus niet voor niets `application/x-www-form-urlencoded` heet) maar niet bij `multipart/form-data`.

Verwar het begrip URL-codering niet met de codering die in HTML-pagina's wordt gebruikt voor bijzondere lettertekens zoals < (genoteerd als `<`) en '&' (`&`). In het volgende HTML-fragment

```
Meer informatie bij
<a href="http://twizz.rug.ac.be/find?n=D%27hondt&amp;v=Jan">
de verantwoordelijke</a>
```

zorgt de notatie `&` ervoor dat de URL op die plaats een ampersand bevat die de twee naam/waarde-paren in de querystring van elkaar scheidt en is '%27' de URL-codering voor het afkappingsteken in de waarde van de eerste parameter. Zonder de `&` is dit eigenlijk geen geldige HTML (een entiteit met de naam `&v=Jan` bestaat niet) hoewel we toegeven dat de meeste browsers hier niet over zullen vallen.

13.3 HTML-formulieren

HTML-formulieren zijn de meest gangbare manier om gegevens van de browser naar een webapplicatie te sturen. Dergelijke formulieren bevatten typische GUI-elementen zoals drukknoppen, radioknoppen, tekstvelden en keuzelijsten, elk genoteerd als HTML-tags met bepaalde attributen. Zo heeft elk element bijvoorbeeld een `name`-attribuut dat zijn naam bepaalt. De corresponderende waarde wordt dan door de gebruiker ingevuld of geselecteerd. Deze naam/waarde-paren worden in de regel pas naar de server doorgestuurd wanneer de gebruiker op een speciale *submitknop* drukt. Meer informatie over HTML vind je in de specificatie ([HTMb]) of in verschillende handleidingen ([HTMa])

Dezelfde HTML-pagina kan tegelijkertijd meerdere formulieren bevatten, elk met eigen componenten en een eigen URL waarnaar de gegevens worden gestuurd. Elk formulier bezit zijn eigen submitknop.

Een formulier wordt omsloten door `<form>` en `</form>`-tags. Daartussen worden de formuliercomponenten met passende HTML-tags aangeduid. Ook gewone HTML-elementen (hoofdingen, afbeeldingen, tabellen) mogen in een formulier worden gebruikt.

De `<form>`-tag bezit drie belangrijke attributen:

`action`	URL waarnaar de gegevens moeten worden verstuurd.
`method`	HTTP-aanvraagmethode: `"GET"` of `"POST"`
`enctype`	MIME-type in het geval van een POST-methode. Mag worden weggelaten wanneer het standaardtype `application/x-www-form-urlencoded` wordt bedoeld.

Een typisch formulier heeft bijvoorbeeld de volgende vorm:

```
<form action="http://www.hogent.be/cgi/findphone" method="GET">
...
</form>
```

De meeste interface-elementen van een formulier worden aangeduid met met behulp van de `<input>`-tag. Deze tag kent onder andere de volgende attributen:

`type`	Geeft het soort interface-element aan (knop, tekstveld, enz.)
`name`	De naam van de corresponderende parameter.
`value`	De initiële waarde van dit element.

Het `value`-attribuut is in vele gevallen zichtbaar en kan meestal door de gebruiker worden aangepast.

We geven een kort overzicht van de verschillende interface-elementen die met de `<input>`-tag worden aangeduid.

`type="SUBMIT"` Een submitknop. Wanneer je op deze knop drukt wordt de inhoud van het omsluitende formulier verstuurd naar de URL uit het `action`-attribuut van het formulier. Het `value`-attribuut verschijnt als opschrift op de knop. Wanneer het formulier slechts één submitknop bezit, hoef je geen `name`-attribuut op te geven (en wordt het `value`-attribuut niet naar de server doorgestuurd). Eenzelfde formulier mag echter ook meerdere submitknoppen bezitten waarop dan eventueel door de webtoepassing telkens anders kan worden gereageerd.

`type="RESET"` Een resetknop. Wanneer je op deze knop drukt worden alle onderdelen van een formulier teruggezet op hun initiële waarden. (De informatie in het formulier wordt niet naar de server doorgestuurd.) Het `value`-attribuut bepaalt het opschrift van de knop maar wordt nooit doorgestuurd naar de server als onderdeel van de formulierinhoud.

`type="TEXT"` Een tekstveld van één enkele lijn. Het `value`-attribuut van deze HTML-tag bevat de initiële inhoud van dit veld, maar het is de uiteindelijke inhoud van het tekstveld die als waarde naar de server wordt verstuurd.

Andere attributen zijn `size`, die de zichtbare breedte van het tekstveld bepaalt (in lettertekens) en `maxchars` die aangeeft hoeveel lettertekens er maximaal in dit veld mogen worden ingetikt. Is dit groter dan de breedte, dan zal de inhoud van het veld desnoods bij het intikken naar links of rechts worden verschoven.

type="PASSWORD" Heeft dezelfde attributen als een tekstveld en gedraagt zich ongeveer gelijkaardig. Het enige verschil is dat alle ingetikte lettertekens als sterretjes (of als bolletjes) op het scherm worden afgebeeld. Merk op dat de beveiling die je met dit soort velden bekomt slechts minimaal is. Wachtwoorden worden onversleuteld over het net verstuurd.

type="HIDDEN" Zoals de naam aangeeft, is dit formulierelement onzichtbaar. De opgegeven `name` en `value` worden echter wel met het formulier doorgestuurd. Een dergelijk element wordt enkel om programmatorische redenen gebruikt.

type="CHECKBOX" Een checkbox is een element dat kan aan- of uitgevinkt worden. Enkel de naam/waarde-paren van de checkboxen die zijn aangevinkt, worden naar de server doorgegeven. Het `value`-attribuut heeft als defaultwaarde `on` en wordt niet op het scherm getoond. Hoewel dit het schrijven van webtoepassingen moelijker maakt, zal men soms verschillende checkboxen gebruiken met dezelfde naam maar met een andere waarde. In andere gevallen is de inhoud van dit attribuut eigenlijk niet belangrijk.

Het attribuut `checked` (zonder gelijkheidsteken of waarde) geeft aan dat de checkbox reeds is aangevinkt wanneer het formulier voor het eerst op het scherm verschijnt.

```
<form method="POST" action="http://...">
<input type="CHECKBOX" name="veg">
    Vegetarisch <br>
<input type="CHECKBOX" name="nrok" checked>
    Niet-roker <br>
...
</form>
```

type="RADIO" Radioknoppen. Net zoals checkboxen kunnen radioknoppen door de gebruiker worden aan- of uitgevinkt. De browser zorgt er echter voor dat er van een volledige groep radioknoppen slechts één tegelijkertijd is geselecteerd. Radioknoppen behoren tot dezelfde groep als ze hetzelfde `name`-attribuut hebben. Enkel de gegevens van de aangevinkte knoppen worden doorgestuurd. Het attribuut `checked` heeft dezelfde functie als bij checkboxes.

```
<form method="POST" action="http://...">
<input type="RADIO" name="kamer" value="2" checked>
    2-persoonskamer <br>
<input type="RADIO" name="kamer" value="2bad">
    2-persoonskamer met bad<br>
<input type="RADIO" name="kamer" value="1">
    1-persoonskamer <br>
...
</form>
```

type="IMAGE" Soms past het uiterlijk van een knop niet echt bij de rest van de layout van het formulier. Met deze `<input>`-tag kan je een eigen afbeelding introduceren als alternatief voor een submitknop.

Behalve `name`- en `type`-attributen bevat deze tag ook nog het attribuut `src` met de URL van een afbeelding die moet worden getoond. Het `align`-attribuut bepaalt de relatieve positie van deze afbeelding ten opzichte van de omringende tekst (zoals bij gewone HTML-afbeeldingen).

Net zoals bij de submitknop wordt de volledige inhoud van een formulier doorgestuurd op het moment dat je op de afbeelding klikt. De informatie over de tag wordt echter doorgestuurd als *twee afzonderlijke* naam/waarde-paren. Is de naam van de `<input>`-tag `naam`, dan wordt een paar doorgestuurd met `naam.x` als naam en als waarde de X-coördinaat van de muispointer op het moment dat de muis werd ingedrukt, en een analoog paar `naam.y` en de Y-coördinaat.

Vroeger gebruikt men dit `IMAGE`-type namelijk om zogenaamde *image maps* te implementeren, grote afbeeldingen (meestal landkaarten of zelfgetekende menubalken) die de gebruiker doorverbinden naar een URL die afhangt van de plaats waar er op de afbeelding werd geklikt. Tegenwoordig gebruik je hiervoor beter de `<map>`-constructie uit HTML.

`type="FILE"` Met dit element kan de gebruiker één of meer bestanden (in bijlage) naar de webtoepassing versturen. Dit werkt alleen bij formulieren die gedeclareerd zijn met een `enctype`-attribuut van het type `multipart/form-data`.

Het element ziet eruit als een tekstveld en heeft gelijkaardige `size`- en `maxchars`-attributen. De gebruiker kan een naam van een bestand intikken in dit veld, maar in plaats van deze naam wordt de inhoud van het opgegeven bestand opgestuurd. (Het `value`-attribuut wordt niet gebruikt.) Naast het tekstveld is er ook een knop die een bestandskiezer oproept waarmee het bestand rechtstreeks kan worden geselecteerd.

`type="BUTTON"` Voor de volledigheid vermelden we ook nog dit element waarmee een willekeurige drukknop op het formulier kan worden geplaatst. Het `value`-attribuut bepaalt het opschrift van de knop. Opdat dit element enig nut zou hebben, moet het worden verbonden met een functie of methode in een clientscripttaal zoals JavaScript ([Cha01, Fla06]).

Voor twee formulierelementen wordt er in plaats van `input` een eigen tag gebruikt: voor een tekstgebied (een tekstveld van meerdere lijnen) en voor een lijst.

Je neemt een tekstgebied in je formulier op met behulp van de de `<textarea>`-tag:

```
<textarea name="adresinfo" rows="5" cols="40">
Naam Voornaam
Straatnaam 42
9000 GENT
</textarea>
```

Zoals gebruikelijk bevat het `name`-attribuut de naam van het element en van de bijhorende parameter. Wat de gebruiker in het veld intikt wordt teruggeven als corresponderende waarde. De tekst die tussen de begin- en eindtags van de `textarea` staat, geldt als defaultwaarde. (Deze tekst wordt ook in het tekstveld ingevuld wanneer het formulier op het scherm verschijnt.) Andere attributen zijn `rows`, het aantal lijnen van het tekstveld, en `cols`, de breedte van het tekstveld uitgedrukt in aantal lettertekens.

Voor een keuzelijst gebruik je `<select>` en `<option>`.

```
Kies een fruit<br>
<select name="fruit">
  <option value="A">Appel
```

```
  <option value="P">Peer
  <option value="S">Sinaasappel
  <option value="K">Kokosnoot
</select>
```

In bovenstaand voorbeeld wordt er een lijst afgebeeld van 4 fruitnamen waaruit de gebruiker er één kan kiezen. Als naam/waarde-paar wordt het `name`-attribuut van de omhullende `select` doorgestuurd (in ons geval `fruit`) en het `value`-attribuut van de gekozen optie (één van de letters A, P, S of K).

De `<select>`-tag ondersteunt een optie `multiple` (zonder waarde) die aangeeft dat er in de lijst ook tegelijkertijd meer dan één element mag worden aangeduid. In dit geval kunnen er dan meerdere naam/waarde-paren voorkomen met dezelfde naam, wat soms een extra programmeerinspanning vraagt aan de kant van de server.

Het `size`-attribuut voor `select` geeft het maximaal aantal elementen van de lijst aan dat tegelijkertijd wordt afgebeeld. Is het aantal opgegeven opties groter, dan worden scrollbars gebruikt. Is `size` gelijk aan één, dan zal de lijst er uitzien als een zogenaamde *drop down list*.

```
Kies een aantal fruitnamen<br>
<select name="fruit" size="1" multiple>
  <option value="A">Appel
  <option value="P">Peer
  <option value="S" selected>Sinaasappel
  <option value="K">Kokosnoot
</select>
```

Zoals je ziet kan de `<option>`-tag behalve het `value`-attribuut ook nog het attribuut `selected` dragen. Hiermee wordt de overeenkomstige optie per default geselecteerd.

13.4 Cookies

Een *cookie* is een klein stukje tekstuele informatie dat door een webserver naar een client wordt gestuurd, door de client wordt bewaard en bij elke aanvraag naar deze server wordt teruggestuurd. Dit gebeurt via de HTTP-headers. Cookies werden ingevoerd door Netscape maar zijn ondertussen gemeen goed geworden (RFC 2109, [KLT+97]).

De server stuurt een cookie naar de klant in een `Set-Cookie`-header van de volgende vorm:

```
SetCookie: NAAM=WAARDE; Domain=DOMEINNAAM; Path=PAD; Max-Age=SECONDEN
```

Elke cookie heeft een naam en daarmee verbonden waarde, een levensduur en een optionele domein- en padnaam die de URLs aangeven waarvoor de cookie geldig is. Bijvoorbeeld, wanneer de volgende cookie voorkomt in een HTTP-antwoord aan een browser

```
SetCookie: uid=kc; Max-Age=86400; Domain=".rug.ac.be"; Path="/"
```

dan zal de browser de volgende header toevoegen

```
Cookie: uid=kc
```

aan elke HTTP-aanvraag die hij verstuurt naar een URL met een server uit het domein *rug.ac.be* en met een pad dat begint met / (elk pad dus in dit voorbeeld) binnen de eerstvolgende 24 uur (86400 seconden).

HTTP is een protocol *zonder toestand* (Engels: *stateless protocol*): elke HTTP-aanvraag (en corresponderend antwoord) staat in principe los van de volgende. Hoewel het voor de server misschien mogelijk is te herkennen dat twee opeenvolgende aanvragen van dezelfde computer afkomstig zijn, is het niet zeker dat ze ook bij dezelfde client zijn ontstaan. Er zouden heel goed meerdere gebruikers tegelijkertijd bezig kunnen zijn met dezelfde webtoepassing, elk met een verschillende browser. Cookies bieden hier een uitweg: op deze manier kan de toestand door de client zelf worden bijgehouden.

We geven een aantal voorbeelden van typische toepassingen waarbij cookies worden ingeschakeld.

- Bij een E-commerce-toepassing identificeren cookies de gebruikers. Dergelijke toepassingen gebruiken vaak een 'winkelwagentje' waaraan de gebruiker producten kan toevoegen. Omdat de HTTP-verbinding wordt gesloten na elke handeling van de gebruiker, heb je een cookiemechanisme nodig om bij te houden dat de volgende actie van dezelfde gebruiker effect heeft op hetzelfde winkelwagentje. Persistente connecties (zie §12.3.3) zouden hier slechts een gedeeltelijke oplossing bieden omdat ze meestal slechts enkele minuten worden aangehouden.
- Bij toepassing waarbij een (formuliergebaseerde) login nodig is, kan een cookie ervoor zorgen dat een gebruiker niet meer hoeft in te loggen de volgende keer dat hij dezelfde toepassing oproept.
- Sommige webtoepassingen laten de gebruiker toe om bepaalde voorkeuren in te stellen. Met cookies kan je ervoor zorgen dat dit niet telkens opnieuw moet gebeuren wanneer de gebruiker de toepassing bezoekt. Ofwel worden de instellingen in de cookies zelf opgeslagen, of anders bevat het cookie een gebruikersidentificatie die als sleutel dient voor een voorkeurendatabank die op de server wordt bewaard.
- Een forum kan met behulp van een cookie bijhouden welke berichten de gebruiker reeds heeft gelezen en welke berichten voor de lezer nieuw zijn.

Hoewel cookies de bruikbaarheid van een webtoepassing sterk kunnen verhogen, zijn niet alle gebruikers even gelukkig met dit mechanisme. Vele browsers laten hun gebruikers daarom de keuze cookies al dan niet te accepteren, of minder ingrijpend, cookies enkel terug te sturen naar de server vanwaar ze afkomstig zijn. Belangrijk is dus te onthouden dat je webtoepassing dus ook zonder cookies nog moet kunnen werken.

Cookies zijn nochtans niet direct een veiligheidsprobleem: ze kunnen geen virussen overbrengen en ook de harde schijf van de client niet overbelasten, aangezien de meeste browsers serieuse beperkingen opleggen op de grootte van een cookie, het aantal cookies per webserver en het totaal aantal cookies dat wordt bewaard.

Anderzijds vormen cookies wel een potentiele bedreiging van de privacy: ze houden immers impliciet het surfgedrag van de gebruiker bij, en kunnen dit eventueel met andere servers uitwisselen. Merk op dat bepaalde browsers ook emailclients bevatten die cookies accepteren, en het is niet moeilijk om op die manier de browsergebruiker met zijn emailadres te associëren.

Hoofdstuk 14

Dynamische webapplicaties

Het Internet bestaat al lang niet meer uit enkel statische HTML-pagina's. Zowel client-side als server-side is er dynamiek toegevoegd.

Client-side scripting maakt het mogelijk om de layout van een webpagina te veranderen zonder dat er een HTTP-bericht heen en weer gestuurd wordt naar de webserver.

Met server-side scripting kan de functionaliteit van de webserver uitgebreid worden zodat hij niet enkel statische HTML-bestanden kan doorsturen naar de client, maar ook, afhankelijk van de aanvraag van de client, dynamisch een webpagina kan genereren.

14.1 Client-side scripting

Client-side scripts worden toegevoegd in de HTML-code van de webpagina en uitgevoerd door de browser. De belangrijkste taken van client-side scripts is het controleren van de invoer van de gebruiker en het genereren van dynamische webpagina's zonder dat hiervoor HTTP-berichten naar de server gestuurd moeten worden ([Seb08, WEBa]). Voorbeelden van het gebruik van client-scripts zijn

- Nagaan of alle gegevens van een HTML-formulier ingevuld zijn voordat het formulier wordt doorgestuurd naar de server. Dit voorkomt nodeloos netwerkverkeer indien niet alle gevraagde gegevens ingevuld zijn.
- Het realiseren van een dynamische layout: uitklapbare menu's, wisselende figuren, de stijl (kleur, ...) van HTML-elementen veranderen, ...

De meest gebruikte taal voor client-side scripts is JavaScript ([ISO, Cha01, Fla06]), maar ook VBScript en JScript kunnen gebruikt worden als scripttaal. JScript is een aangepaste versie van JavaScript die ontwikkeld werd door Microsoft. VBScript kan enkel gebruikt worden in combinatie met Internet Explorer.

Client-side scripts worden toegevoegd aan HTML-pagina's m.b.v. <script>-tags. Om de webpagina te kunnen manipuleren maken de scripts gebruik van het Document Object Model (DOM, [DOM]). De HTML-pagina wordt voorgesteld als een *Document* uit de DOM-API.

De webpagina in het volgend voorbeeld bestaat uit een HTML-formulier met een tekstveld en een submit-knop. De gegevens van het formulier worden doorgestuurd naar de server als het tekstveld ingevuld is. In het andere geval wordt er een popup-venster met een waarschuwing getoond, maar wordt het formulier niet doorgestuurd.

```
<html>
    <head>
        <script type="text/javascript">
            function controleer_invoer() {
                var verder = false;
                if (document.invoerGegevens.naam.value == "") {
                    window.alert("Naam invoeren!");
                } else {
                    verder = true;
                }
                return verder;
            }
        </script>
    </head>
    <body>
        <form name="invoerGegevens" action="..." onsubmit="return controleer_invoer();">
            Naam: <input type="text" name="naam" size="20"><br>
            <input type="submit" value="Log in" >
        </form>
    </body>
</html>
```

Listing 14.1: voorbeeld javascript controle invoer

De impliciete variabele *document* stelt de webpagina voor. De verschillende componenten van de webpagina vormen een volgens het DOM-model een boomstructuur waarvan de knopen op naam opgevraagd kunnen worden, bv. *document.invoerGegevens.naam*.

De HTML 4 standaard ([HTMb]) ondersteunt een aantal gebeurtenissen (*events*) die afgehandeld kunnen worden door Javascript. Voor elke gebeurtenis heeft de HTML-component een bijhorend attribuut (bv. *onsubmit*). Dit attribuut wordt gebruikt om te verwijzen naar de opdrachten die uitgevoerd moeten worden als de gebeurtenis optreedt.

Het volgend stukje code illustreert hoe Javascript in combinatie met CSS ([CSSa]) en DOM gebruikt kan worden om de layout van een webpagina te veranderen. In dit geval wordt de kleur van een titel veranderd bij het selecteren van een keuzerondje (*radio button*).

```
<html>
    <head>
        <script type="text/javascript">
            function veranderKleur(kleur) {
                document.getElementById('titel').style.color = kleur;
            }
        </script>
    </head>
    <body>
        <h1 id="titel" style="color:yellow">
            Titel in kleur
        </h1>
        <form action="...">
```

```
            Kies kleur:
            <input type="radio" name="kleur" onclick="veranderKleur('red')"> rood
            <input type="radio" name="kleur" onclick="veranderKleur('blue')"> blauw
            <input type="radio" name="kleur" onclick="veranderKleur('green')"> groen
        </form>
    </body>
</html>
```

Listing 14.2: voorbeeld javascript layout

Merk op dat de methode *getElementById* behoort tot de DOM API.

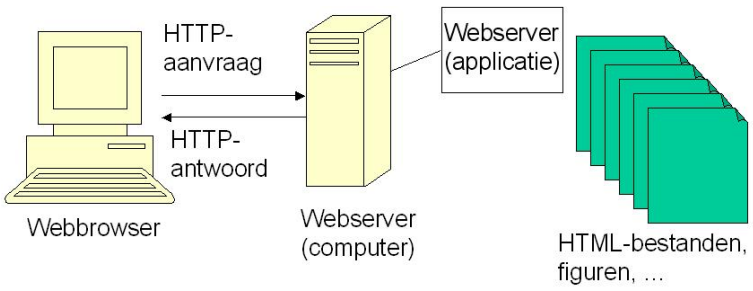
DHTML, Adobe Flash

De combinatie van HTML, CSS en Javascript om dynamische, interactieve en geanimeerde webpagina's te maken wordt vaak DHTML, Dynamic HTML, ([DYNa, DYNb]) genoemd.

Om webpagina's op te fleuren met animaties, filmpjes en spelletjes kan je Adobe Flash gebruiken.

14.2 Server-side scripting

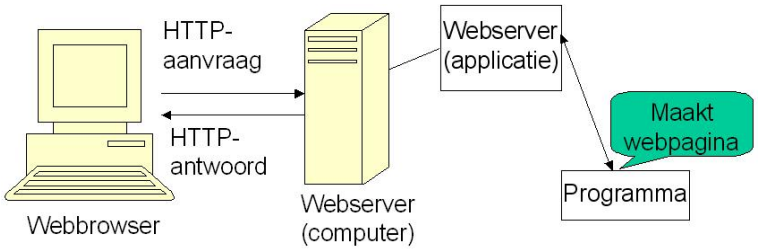
Webserverapplicaties zoals Apache ([APA]) of Microsofts Internet Information Services ([IIS]), voorzien een directory structuur voor elke webapplicatie die ze huisvesten. In die mappen bevinden zich HTML-bestanden, PDF-bestanden, figuren, geluidsbestanden, ... Deze bestanden kunnen opgevraagd worden door de webclients (bv. browsers) door het opgeven van de juiste URL. De taak van de webserverapplicatie is om aan de hand van de URL het juiste bestand te vinden en het door te sturen naar de client verpakt in een HTTP-antwoordbericht (zie figuur 14.1). Merk op de we het woord webserver zowel gebruiken voor de webserverapplicatie als voor de computer waarop deze applicatie draait.



Figuur 14.1: Statische webapplicatie

Indien webpagina's dynamische elementen bevatten, waarvan de waarde niet op voorhand kan opgenomen worden in de HTML-code omdat de inhoud variabel kan zijn (bv. de huidige datum), dan moet een deel van de HTML-pagina bepaald worden op het ogenblik van de aanvraag. Hiervoor doet de webserver beroep op een andere applicatie of programma, zonder dat de client hiervan iets merkt (zie

figuur 14.2). De webserver geeft de parameters van de HTTP-aanvraag door aan het hulpprogramma. Dit programma gebruikt de parameters om dynamisch de HTML-pagina te genereren die dan door de webserver verpakt wordt in een HTTP-antwoord en doorgestuurd wordt naar de client. De parameters van de HTTP-aanvraag kunnen door het hulpprogramma ook gebruikt worden om gegevens te bewaren op de server, bv. in een gegevensbank.



Figuur 14.2: Dynamische webapplicatie

Er bestaat een waaier aan mogelijkheden om deze hulpprogramma's te realiseren. Ze worden ontwikkeld in zowel geïnterpreteerde programmeertalen (bv. ASP, PHP, ...) als in gecompileerde (bv. CGI, servlets, ASP.NET, ...).

14.2.1 De Common Gateway Interface

Er bestaan verschillende manieren om de functionaliteit van bestaande webserversoftware naar eigen wensen uit te breiden. Zo kan je de meeste webservers zodanig configureren dat ze kleine programma's die je zelf hebt geschreven (vaak *scripts* genoemd) automatisch oproepen wanneer een gepaste URL wordt opgevraagd. Deze scripts verwerken dan de clientgegevens en zorgen op deze manier voor dynamische HTML-pagina's. Het schrijven van een dergelijk script is vanzelfsprekend een stuk eenvoudiger dan zelf het volledige HTTP-protocol te implementeren.

Opdat deze scripts goed zouden samenwerken met de webserver, moeten ze een aantal conventies volgen die de server begrijpt, en dan nog liefst conventies die onafhankelijk zijn van de gekozen serversoftware. Gelukkig zijn de softwareproducenten het hier al vrij vroeg in de geschiedenis van het web over een standaard eens geworden: de *Common Gateway Interface* (*CGI*). Programma's die deze conventies volgen, noemt men *CGI-scripts*.

Omdat veel CGI-scripts op het Internet in de programmeertaal PERL zijn gecodeerd, denken vele programmeurs dat dit noodzakelijk is. Je kan CGI-scripts echter even goed in een andere programmeertaal schrijven of zelfs rechtstreeks als UNIX shell-script implementeren.

In principe kan een CGI-script dus ook in Java zijn geschreven. In de praktijk zal men in een Java-omgeving echter streven naar een meer rechtstreekse communicatie met de webserver (of meer precies, met een web*container*) en gaat de voorkeur uit naar servlets (zie hoofdstuk 17).

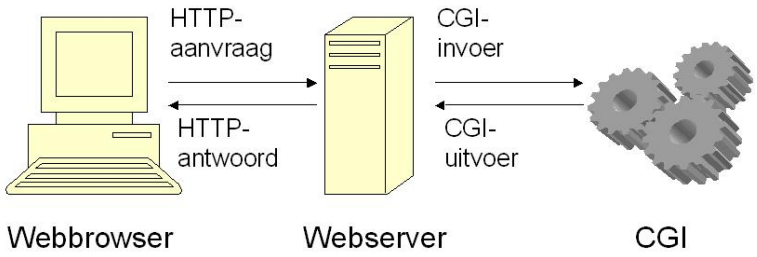
Hoewel het concept van een CGI-script betrekkelijk eenvoudig is, duiken er bij grotere toepassingen

al snel een aantal complicaties op. CGI is eigenlijk enkel geschikt voor haastklussen en prototypetoepassingen en niet voor professioneel werk.

Basiswerking

Een typische CGI-sessie verloopt op de volgende manier:

1. De gebruiker geeft aan zijn browser een URL op, klikt op een link of drukt op de submitknop van een HTML-formulier.
2. Als gevolg hiervan lanceert de HTTP-client een HTTP-aanvraag met methode GET of POST en de gegeven URL. De server verwerkt deze aanvraag en houdt een kopie bij van het corpus en van een aantal gegevens uit de headerlijnen.
3. De server bepaalt welk CGI-script er met de gevraagde URL overeenkomt en start het op in een nieuw proces. De gegevens uit de headerlijnen en het corpus worden aan het script doorgegeven.
4. Het script verwerkt deze gegevens en geeft een resultaat terug aan de server.
5. De server stuurt de gegevens die hij van het script heeft gekregen terug naar de client in de vorm van een HTTP-antwoord.
6. De browser toont dit resultaat op het scherm.



Figuur 14.3: CGI

Hoewel in de praktijk de meeste URLs die met CGI-scripts worden geassocieerd een naam hebben waarin ergens de letters `cgi` voorkomen, is dit absoluut niet noodzakelijk. Je kan aan een URL in principe niet zien of er een CGI-script achter steekt. Het is uiteindelijk de webserversoftware die bepaalt of er een script wordt opgeroepen of niet.

Het CGI-*protocol* bepaalt welke gegevens er naar het CGI-script worden doorgegeven en in welke vorm, en omgekeerd, hoe het CGI-script aan de server duidelijk maakt wat hij naar de client moet terugsturen.

Hoofdstuk 15

ASP.NET

15.1 Wat is ASP.NET?

ASP.NET is de opvolger van ASP (Active Server Pages). De functionaliteit van ASP.NET is echter veel groter en uitgebreider dan zijn voorganger. ASP.NET biedt een professioneel ontwikkelingsplatform voor het bouwen van veilige, schaalbare en stabiele webapplicaties. ASP.NET-applicaties kunnen in elke taal beschikbaar voor het .NET-platform geschreven worden, dus ook in VB.NET, C# of JScript.NET. In het kader van deze cursus zal gebruikt gemaakt worden van de programmeertaal C#. In tegenstelling tot ASP's zijn ASP.NET-pagina's gecompileerd i.p.v. geïnterpreteerd, wat de performantie en de snelheid verhoogt.

De ASP.NET-technologie bestaat uit twee grote stukken: webformulieren en webservices.

Het doel van webformulieren is het aanmaken van webpagina's die gebruik maken van HTML-formulieren. Deze webformulieren worden componentsgewijs opgebouwd. Dit betekent dat elke webformulier bestaat uit verschillende herbruikbare componenten of bouwstenen. Deze componenten (servercomponenten of *server controls* genoemd) creëren een grafische interface en voeren een aantal taken uit.

Webservices bieden de mogelijkheid om serverfunctionaliteit of diensten aan te bieden. Applicaties op andere servers of computers kunnen dan gebruik maken van deze diensten. Webservices zijn onder andere een interface die programma's kunnen aanroepen om logische functionaliteit en gegevens op een andere computer te benutten. Webservices maken gebruik van XML en HTTP om te kunnen communiceren.

De rest van dit hoofdstuk bestaat uit de bespreking van de verschillende facetten van webformulieren. Webservices komen hier niet aan bod, maar wel in het vak Capita Selecta, partim XML.

15.2 Een eerste webformulier

Het volgend voorbeeld bevat de code van een webformulier dat een webpagina genereert die acht keer het woordje "Welkom!" toont in een steeds groter wordend lettertype. Het voorbeeld toont de inhoud van het bestand *welkom.aspx*.

```
<%@ Page Language="C#"%>
<html>
  <head>
    <title>Welkom</title>
  </head>
  <body><center>
      <p>
        <% for (int i = 0; i < 8; i++) { %>
          <font size="<%=i%>"> Welkom!</font>
          <br>
        <% }%>
      </p>
  </center></body>
</html>
```

In het voorbeeld worden stukken script (tussen $<\%$ en $\%>$) gecombineerd met stukken HTML. De gebruikte taal in de scriptstukken is C#.

De stukken script kunnen de volgende structuur hebben:

- Richtlijnen
- Stukken code
- Uitdrukkingen
- Commentaar

Een webformulier is een .aspx-bestand. Dit bestand wordt bij een eerste aanvraag door de gebruiker, gecompileerd naar een afgeleide klasse van de klasse *Page* uit de bibliotheek *System.Web.UI.Page* en de gegenereerde dll wordt in het geheugen geladen. De volgende aanvragen worden dan door deze dll afgehandeld.

15.2.1 Richtlijnen

Richtlijnen (*directives*) bevatten instellingen die nodig zijn voor en bij het compileren van het .aspx-bestand naar een klasse. De syntax van een richtlijn is de volgende:

```
<%@ NaamRichtlijn LijstAttributen%>
```

NaamRichtlijn is de naam van de richtlijn, *LijstAttributen* is een lijst van attributen die een reeks instellingen bevat. Mogelijke richtlijnen zijn onder andere *Page* en *Import*.

```
<%@ Page Language="C#" %>
```

De richtlijn *Page* bevat de configuratie voor het parsen en compileren van het .aspx-bestand. Mogelijke attributen zijn onder andere *ErrorPage* en *Language*. Het attribuut *ErrorPage* is een URL waarnaar doorverwezen wordt als er een fout optreedt bij het uitvoeren van de aanmaak van de webpagina. De gebruikte programmeertaal in de stukken script wordt vastgelegd door het attribuut *Language*.

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.OleDb" %>
```

De *Import*-richtlijn kan maar één attribuut hebben. Dit attribuut heeft de naam *namespace* en heeft als waarde een namespace (bv. *System.Collections*). Om meerdere namespaces te importeren worden meerdere *Import*-richtlijnen gebruikt. De *Import*-richtlijn geeft aan welke namespaces er gebruikt worden in de stukken code.

15.2.2 Code, uitdrukkingen en commentaar

Stukken code die uitgevoerd moeten worden tijdens het aanmaken van de webpagina staan tussen $<\%$ en $\%>$. Deze code bestaat enkel uit opdrachten en kan *geen* functie- of proceduredeclaraties bevatten.

Uitdrukkingen waarvan het resultaat aan de HTML-stroom toegevoegd moet worden staan tussen $<\%=$ en $\%>$. Het resultaat van de uitdrukking wordt geconverteerd naar een *string*. Deze notatie is een verkorte vorm voor het oproepen van de methode *Write* van *HttpResponse* (zie §15.3.3).

De syntax voor *server side* commentaar is de volgende:

```
<%-- commentaar --%>
```

Hierbij is *commentaar* de commentaar-tekst. Deze commentaar wordt niet toegevoegd aan de gegenereerde webpagina.

15.2.3 Tussenvoegen van bestanden

Om de inhoud van een bestand of een url tussen te voegen in een webformulier wordt gebruik gemaakt van de volgende syntax.

```
<!-- #include file="filename" -->
```

of

```
<!-- #include virtual="pad" -->
```

Bij het gebruik van het attribuut *file* moet de waarde *filename* een bestandsnaam zijn uit de huidige map of uit een onderliggende map van de huidige map. Het attribuut *virtual* specifieert een virtueel pad, al dan niet relatief.

15.3 De klasse *Page*

Zoals in de vorige sectie (zie §15.2) vermeld wordt elk webformulier gecompileerd naar een afgeleide klasse van de klasse *System.Web.UI.Page*. Deze paragraaf illustreert hoe gebruik kan gemaakt worden van eigenschappen, methodes, ... van deze klasse in het webformulier.

15.3.1 Een voorbeeld

In dit voorbeeld wordt een webformulier getoond dat een webpagina genereert waarop een gebruiker een naam en een categorie kan ingeven. De actie van het formulier op de webpagina is terug hetzelfde webformulier als datgene dat de webpagina genereerde. Na het indienen van het formulier wordt de ingegeven naam en categorie getoond en een mededeling die beide gegevens bevat.

```
<%@ Page Language="C#" %>
<html>
  <head><title>Keuze</title></head>
  <body><center>

      <%-- Start HTML-formulier --%>
      <form action="keuze.aspx">

        <h3>
          <%-- Ingeven en weergeven naam --%>
          Naam: <input name="Naam" type="text"
            value="<%=Request.QueryString["Naam"]%>">

          <%-- Ingeven en weergeven categorie --%>
          Categorie:
          <select name="Categorie" size=1>
            <%
              String [] values = { "muziek", "sport",
                  "actualiteit" };
```

```
            for (int i=0; i<values.Length; i++) {
          %>
          <option
            <% if (Request.QueryString["Categorie"]
                == values[i]) {
                Response.Write("selected");
            } %>>
            <%=values[i]%>
          </option>
    <%
            }
          %>
  </select>
        </h3>

        <%-- knop --%>
        <input type="submit" name="ZoekOp" value="Zoek Op">

        <%-- Weergeven mededeling --%>
        <p>
          <% if (Request.QueryString["ZoekOp"] != null) { %>
            Hallo <%=Request.QueryString["Naam"] %>,
            je koos: <%=Request.QueryString["Categorie"] %>
          <% } %>
      </form>
  </center></body>
</html>
```

Het HTML-formulier dat door deze code aangemaakt wordt bevat drie gegevens namelijk *Naam*, *Categorie* en *ZoekOp*. Als het formulier ingediend wordt kunnen de waarden van deze gegevens opgehaald worden. Hiervoor wordt gebruik gemaakt van de publieke eigenschap *Request* van de klasse *Page*. Aangezien het webformulier een afgeleide klasse wordt van de klasse *Page* kunnen deze eigenschappen opgeroepen worden met behulp van hun naam (zoals bv. *Request*, *Response*, *Application*, *Session*, ...). Deze eigenschappen zijn respectievelijk van de volgende types *System.Web.HttpRequest*, *System.Web.HttpResponse*, *System.Web.HttpApplicationState* en *System.Web.SessionState.HttpSessionState*.

De klasse *HttpSessionState* maakt het mogelijk om sessieinformatie te beheren, te bewaren en op te halen. Om informatie te delen tussen verschillende sessies en HTTP-aanvragen kan gebruik gemaakt worden van de klasse *HttpApplicationState*.

15.3.2 System.Web.HttpRequest

Een object van de klasse *HttpRequest* stelt de HTTP-aanvraag van de client voor. Dit object kan gebruikt worden om de gegevens uit de HTTP-headers en uit het HTTP-corpus op te vragen, bv. de gebruikte browser, de cookies, de gebruikte karakterset, ...

Het aanvraagtype (GET of POST) van de HTTP-aanvraag wordt bepaald door de eigenschap *RequestType*. De eigenschap *QueryString* geeft de querystring van de HTTP-aanvraag weer. De naam/waardeparen die in het corpus van een HTTP-aanvraag staan worden weergegeven door de eigenschap *Forms*.

De eigenschap *Params* is de combinatie van de naam/waarde-paren bepaald door de eigenschappen *QueryString*, *Params*, *Cookies* en *ServerVariables*. Al deze eigenschappen zijn van het type *NameValueCollection*.

De klasse *NameValueCollection* stelt een gesorteerde verzameling van naam/waarde-paren voor waarbij zowel de naam (ook wel sleutel genoemd) als de waarde een *string* zijn. De waarde kan een lijst van *strings* zijn die gescheiden worden door komma's. De klasse *NameValueCollection* heeft een indexer die zowel een *int* als een *string* als argument aanvaardt. In beide gevallen wordt de waarde horende bij de opgegeven index of naam teruggegeven. Hieronder volgen een aantal eigenschappen en methodes van de klasse *NameValueCollection*.

public string[AllKeys] Alle sleutels (namen) van het *NameValueCollection*-object.

public int Count Het aantal naam/waarde-paren dat het *NameValueCollection*-object bevat.

public void Add(string sleutel, string waarde) Voegt een naam/waarde-paar toe aan het *NameValueCollection*-object.

public void Clear() Verwijdert alle naam/waarde-paren uit het *NameValueCollection*-object.

public string[GetValues(string sleutel)] Geeft een tabel van strings terug van alle waarden horende bij de opgegeven sleutel (naam).

public void Remove(string sleutel) Verwijdert het naam/waarde-paar met de opgegeven naam (sleutel) uit het *NameValueCollection*-object.

15.3.3 System.Web.HttpResponse

Een object van de klasse *HttpResponse* stelt het HTTP-antwoord voor dat teruggestuurd wordt naar de client en biedt de mogelijkheid om bv. het contenttype in te stellen, om HTTP-headers toe te voegen, om te schrijven naar het corpus van het HTTP-antwoord, ... Hieronder volgen een aantal eigenschappen en methodes van deze klasse.

public string ContentType Bepaalt het contenttype van het HTTP-antwoord. De standaardwaarde is "text/html".

public void AppendHeader(string naam, string waarde) Voegt een HTTP-header toe met naam *naam* en waarde *waarde*.

public void Clear() Verwijdert alle gegevens uit de huidige buffer.

public void Flush() Stuurt alle uitvoer in de buffer door naar de client.

public void Redirect(string url) Stuurt de client door naar een nieuwe URL bepaald door *url*.

public void Write(object obj) Schrijft het object *obj* naar HTTP-uitvoerstroom. Deze methode bestaat ook met als argument een *string* of een *char*.

15.4 Servercomponenten

In het voorbeeld (zie §15.3.1) uit §15.3 wordt in het webformulier nogal veel gebruikt gemaakt van scripting. Dit maakt webformulieren echter moeilijk onderhoudbaar. Het ASP.NET-platform biedt een veel elegantere manier om een webformulier met dezelfde functionaliteit aan te maken. Op die manier wordt een webformulier opgebouwd uit verschillende servercomponenten. Elke servercomponent genereert zijn eigen HTML-presentatie en zorgt voor een stuk functionaliteit (bv. het bewaren van de informatie die ingetikt werd). In deze sectie bekijken we de verschillende mogelijke servercomponenten. Er zijn vier verschillende types servercomponenten.

- HTML-servercomponenten (zie §15.4.2)
- Webservercomponenten (zie §15.4.3)
- Controleservercomponenten (zie §15.4.5)
- Gebruikersservercomponenten (zie §15.4.6)

15.4.1 Voorbeeld

Het voorbeeld (zie §15.3.1) uit §15.3 herschreven met servercomponenten ziet er als volgt uit.

```
<html>
  <head><title>Keuze</title></head>
  <script language="C#" runat="server">
      void SubmitBtn_Click(Object sender, EventArgs e) {
        Bericht.Text = "Hallo " + Naam.Text + ", je koos: "
          + Categorie.SelectedItem;
      }
  </script>

  <body><center>

      <form method="post" runat="server">

        <h3>
          Naam: <asp:textbox id="Naam" runat="server"/>

          Categorie:
          <asp:dropdownlist id="Categorie" runat="server">
            <asp:listitem>muziek</asp:listitem>
            <asp:listitem>sport</asp:listitem>
            <asp:listitem>actualiteit</asp:listitem>
          </asp:dropdownlist>
        </h3>

        <asp:button text="Zoek Op" OnClick="SubmitBtn_Click"
```

```
            runat="server"/>
        <p>
          <asp:label id="Bericht" runat="server"/>
        </p>
      </form>

  </center></body>
</html>
```

Merk op dat in het bovenstaande webformulier een aantal stukken voorkomen die geen HTML en geen script (tussen $<\%$ en $\%>$) zijn, namelijk

- HTML-elementen met een attribuut *runat="server"*, HTML-servercomponenten (zie §15.4.2) genoemd
- elementen waarvan de naam begint met *asp*, webservercomponenten (zie §15.4.3) genoemd
- $<$ *script*$>$ -elementen

In §15.2.2 werd reeds vermeld dat in scriptstukken (tussen $<\%$ en $\%>$) geen velden, methodes, eigenschappen, ... van de te genereren klasse kunnen beschreven worden, maar enkel code uit te voeren bij het aanmaken van de webpagina. Om dit toch te kunnen doen moet gebruik gemaakt worden van het $<$ *script*$>$ -element. In §15.5 zullen we zien dat een beter alternatief voor het $<$ *script*$>$ -element het gebruik van *code behind files* is. De syntax van het $<$ *script*$>$ -element is

```
<script runat="server" language="codetaal" src="padnaam">
  ... // code
</script>
```

Hierbij is *codetaal* de programmeer- of scripttaal die gebruikt wordt in het $<$ *script*$>$ -element. Als het attribuut *src* wordt opgegeven dan mag het $<$ *script*$>$ -element geen code bevatten. De code staat dan in het bestand met de naam *padnaam*.

In het bovenstaande voorbeeld wordt het $<$ *script*$>$ -element gebruikt om een methode *SubmitBtn_Click* te declareren. Het klikken op de knop (gegenereerd door $<$ *asp:button text="Zoek Op" OnClick="SubmitBtn_Click" runat="server"/*$>$) zorgt ervoor dat het formulier op de webpagina ingediend wordt. Op de server wordt dan de methode *SubmitBtn_Click* uitgevoerd en de webpagina wordt weer aangemaakt. De servercomponenten zorgen ervoor dat de ingevulde gegevens bewaard blijven.

15.4.2 HTML-servercomponenten

In een .aspx-bestand worden HTML-elementen beschouwd als tekst. Dit betekent dat ze niet toegankelijk zijn vanuit de code horende bij het .aspx-bestand (bv. in stukken script, in het $<$ *script*$>$

-element of in *code behind files*). Om dit toch mogelijk te maken moet het HTML-element een attribuut *runat="server"* krijgen. Deze HTML-elementen noemen we HTML-servercomponenten.

De waarde van het unieke *id*-attribuut is de naam van de variabele waarmee je in de code een referentie hebt naar de component. Bovendien moeten alle HTML-servercomponenten voldoen aan de XML-syntax (afgesloten worden en deel uitmaken van een boomstructuur).

HTML-servercomponenten bieden het voordeel dat als het formulier teruggestuurd wordt naar de server en als de pagina terug aangemaakt wordt, de component nog steeds de ingegeven waarde bevat zonder dat dit expliciet geprogrammeerd is (zoals in het voorbeeld uit §15.3.1).

Alle servercomponenten moeten behoren tot $<$ *form*$>$ -element met een attribuut *runat="server"*. Dit zorgt ervoor dat bij gebruikersacties zoals klikken op een knop het formulier teruggestuurd wordt naar de server en de pagina terug wordt aangemaakt. Op de server wordt dan de actie die bij het klikken horen uitgevoerd. De componenten die niet gewijzigd worden door deze acties behouden dan hun waarde. Dit impliceert dat elke .aspx-pagina slechts één HTML-formulier heeft.

In het bovenstaande voorbeeld (zie §15.4.1) wordt bij het klikken op de knop het formulier terug naar de server gestuurd, wordt de methode *SubmitBtn_Click* uitgevoerd en wordt de pagina terug aangemaakt. De inhoud van het tekstveld en de keuzelijst bevatten de door de gebruiker ingegeven waarden. De inhoud van het label werd aangepast.

De standaardwaarde van het attribuut *method* is POST. Het *action*-attribuut is altijd de URL van de pagina. Dit betekent dat een formulier op een webformulier telkens doorgestuurd wordt naar het webformulier zelf.

Deze constructie is gebaseerd op het idee dat een webapplicatie opgebouwd, ontwikkeld en gebruikt moet worden zoals een windowsapplicatie: je hebt één venster met daarop en daarin verschillende componenten die met elkaar kunnen interageren. In een windowsapplicatie kunnen componenten ook echt met elkaar communiceren, in een webpagina niet. Hetzelfde effect wordt echter bereikt door bij elke handeling, waarbij er interactie moet zijn tussen componenten, de webpagina terug te sturen naar de server. Op de server worden dan objecten gecreëerd die de componenten van de webpagina voorstellen en daar kunnen ze dan wel communiceren. Vervolgens wordt een webpagina die de nieuwe status van de applicatie voorstelt aangemaakt en doorgestuurd naar de webbrowser. De extra informatie die bewaard moet worden om dit te kunnen uitvoeren, zit in verborgen velden.

Alle HTML-servercomponenten worden voorgesteld door klassen die behoren tot de bibliotheek (*namespace*) *System.Web.UI.HtmlControls*.

15.4.3 Webservercomponenten

Webservercomponenten hebben dezelfde functie als HTML-servercomponenten, maar bieden meer mogelijkheden en zijn geen loutere vertaling van bestaande HTML-elementen naar servercomponenten. Net zoals HTML-servercomponenten zijn ze componenten waaruit een webpagina wordt opgebouwd, maar ze corresponderen niet met HTML-tags. Dit maakt dat ze een complexere layout en functionaliteit kunnen bieden zoals bv. een kalender en verschillende soorten lijsten.

Syntax

De syntax van een webservercomponent is de volgende

```
<asp:naamComponent runat="server">
</asp:naamComponent>
```

of

```
<asp:naamComponent runat="server" />
```

Behalve het attribuut *runat="server"* kan het element uiteraard nog andere attributen bevatten. De bibliotheek waartoe deze componenten behoren is *System.Web.UI.WebControls*.

In het bovenstaande voorbeeld (zie §15.4.1) worden de webservercomponenten *textbox*, *dropdownlist*, *label* en *button* gebruikt.

Tussen de start- en eindtag van een < *asp:dropdownlist*> kunnen één of meerdere < *asp:listitem*> -elementen staan. Deze elementen stellen de verschillende strings voor waaruit de keuzelijst bestaat. Met behulp van de eigenschap *Value* van *ListItem* (of het attribuut *value* van het element < *asp:listitem*>) kunnen de corresponderende waarden ingesteld worden. Indien dit niet gebeurt dan is de waarde van het item dezelfde als de getoonde *string*.

Een aantal webservercomponenten kennen ook gebeurtenissen. Zo heeft de knop in het bovenstaande voorbeeld (zie §15.4.1) een *Click*-event, m.a.w. het klikken op de knop genereert een event. Hoe op deze gebeurtenis gereageerd moet worden, wordt beschreven in het *onClick*-attribuut. Dit attribuut bevat de naam van de uit te voeren methode.

Andere webservercomponenten zijn onder andere de *DataGrid* en *DataList*. Beide componenten stellen gegevens in tabelvorm voor, maar de *DataList*-component biedt meer mogelijkheden. In §15.4.4 wordt het gebruik van deze componenten geïllustreerd.

15.4.4 Gegevens binden

Het principe van "gegevens binden" is dat men een servercomponent kan verbinden met een gegevensbron, maar dat deze component de gegevens uit die bron pas bevat als de methode *DataBind* van die component wordt opgeroepen. De verbinding geeft dus aan waar de informatie voor een bepaalde servercomponent zich bevindt. De methode *DataBind* zorgt ervoor dat de gegevens voor deze component opgehaald en toegekend worden zodat hij deze kan tonen.

Het is zinvol om de methode *DataBind* uit te voeren de eerste keer dat het webformulier opgeroepen wordt en telkens als de bijhorende data veranderen. Het oproepen van deze methode op een bepaalde servercomponent impliceert het oproepen van de corresponderende methode op al zijn onderliggende

(kind-) componenten. Bij het uitvoeren van de methode *DataBind* op een *Page*-object worden ook alle componenten die dit object bevat gebonden.

Er is een verschil tussen het binden van gegevens die uit één waarde bestaan en gegevens die uit verschillende waarden bestaan (bijvoorbeeld een tabel, een lijst, ...). Bij het binden van één enkele waarde is het voldoende om voor de component die gebonden wordt te specifieren waar hij zijn waarde moet halen. In het tweede geval moet de gegevensbron de *IEnumerable*-interface implementeren zodat de verschillende items in de gegevensbron bij het binden overlopen kunnen worden. De eigenschap *DataSource* van de servercomponent is dan een verwijzing naar de gegevensbron. Voorbeelden van structuren die gebruikt kunnen worden als bindbare gegevensbron zijn:

- *DataSet*
- *ArrayList*
- *Collections*
- *DataReader*
- *DataView*
- *DataRow*
- *DataTable*

Tijdens het binden van een gegevensbron die meerdere waarden bevat, wordt elke item in de bron overlopen. Wat er met dit item gebeurt hangt af van de servercomponent waaraan de gegevens gebonden worden. Zo kan een *DataGrid* een volledige tabel tonen, maar moet je voor een *DropDownList* specifieren welke kolom uit de tabel getoond moet worden.

Het volgend voorbeeld toont hoe je een *DataView* kunt binden aan een *DataGrid*. Een *DataView* is een *view* op een *DataTable*-object die gebruikt kan worden om te binden, te sorteren, te editeren, te zoeken, ... Het standaard *DataView*-object van een *DataTable*-object wordt verkregen met de eigenschap *DefaultView*.

Voorbeeld

```
 ...
<script language="C#" runat=server>
private void SubmitBtn_Click(Object sender, EventArgs e) {
     ...
    // gegevens ophalen uit gegevensbank
    String connString = "...;
    String query = "select voornaam, achternaam from personen";
    OleDbConnection conn = new OleDbConnection(connString);
    OleDbDataAdapter adapter = new OleDbDataAdapter(query, conn);
```

```
    DataSet ds = new DataSet();
    adapter.Fill(ds, "Personen");

     // gegevens verbinden met de DataGrid Lijst
     Lijst.DataSource = ds.Tables["Personen"].DefaultView;
     Lijst.DataBind();
}
</script>

<body>
<form method="post" runat="server">
  ...
<asp:button text="Zoek Op" OnClick="SubmitBtn_Click"
    runat="server"/>
  ...
<asp:datagrid id="Lijst" HeaderStyle-BackColor="#aaaadd"
  BackColor="#ccccff" runat="server"/>
</form></body></html>
```

Om in het .aspx-bestand gegevens te binden buiten de *script*-tag wordt gebruik gemaakt van de volgende syntax. Hierbij kenmerkt *tagprefix:tagname* de servercomponent, verwijst het attribuut *eigenschap* naar de te binden eigenschap, is het resultaat van *uitdrukking* de gegevensbron en *tekst* is een stukje HTML-tekst. In de eerste syntaxregel is het resultaat van de *uitdrukking* de set gegevens die servercomponent moet tonen. In de tweede regel wordt het resultaat van de uitdrukking toegevoegd aan de HTML-stroom.

```
<tagprefix:tagname eigenschap='<%# uitdrukking %>'
    runat="server" />
```

of

```
tekst <%# uitdrukking %>
```

In het bovenstaande voorbeeld (zie §15.4.4) wordt de *DataGrid* vervangen door een *DataList*. Met een *DataList* kan de layout van de hoofding, de voet, een rij en de scheiding tussen rijen van een tabel ingesteld worden. Voor al deze delen van de tabel kan een template gespecifieerd worden. Bij het beschrijven van de layout van een rij in de tabel is het te tonen item niet gekend (bv. omdat dit het resultaat van een zoekopdracht in een gegevensbank is) en wordt er gebruikt gemaakt van de syntax om gegevens te binden. Merk op dat alleen het element $<$ *asp:datagrid*$>$

vervangen wordt.

```
<asp:datalist id="Lijst" borderwidth="0" runat="server">
<HeaderTemplate>
    <table cellspacing=0>
```

```
    <tr><td>
        <b><font face="Verdana" size=3>Personen</font></b>
    </td></tr>
    <tr><td height=5 bgcolor="000000"></td></tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td width=150 valign=top>
    <b>
    <%# DataBinder.Eval(Container.DataItem, "achternaam") %>
    </b>
    <%# DataBinder.Eval(Container.DataItem, "voornaam") %>
</td></tr>
</ItemTemplate>

<SeparatorTemplate>
    <tr><td height=1 bgcolor="000000"></td></tr>
</SeparatorTemplate>

<FooterTemplate>
    <tr><td height=5 bgcolor="000000"></td></tr>
    </table>
</FooterTemplate>
</asp:datalist>
```

Om naar de gegevens van een rij te kunnen verwijzen wordt in uitdrukkingen die gegevens binden gebruik gemaakt van de volgende objecten.

Container Het *Container*-object verwijst naar de oudercomponent.

Container.DataItem Dit object verwijst naar het huidige item bij het overlopen van de gegevensbron die gebonden wordt aan de oudercomponent.

De statische methode *Eval* van de klasse *DataBinder* zorgt voor automatische omzetting van types. Het resultaat van de methode wordt in uitdrukkingen die gegevens binden omgezet naar het type van het object waaraan het wordt toegekend. Bovendien vermijdt deze methode een expliciete *cast*. De uitdrukking

```
DataBinder.Eval(Container.DataItem, "achternaam")
```

is een andere vorm voor

```
((DataListItem)Container.DataItem)["achternaam"]
```

De methode *Eval* in het bovenstaande voorbeeld heeft twee argumenten: een object en een string. De string is een pad van de container naar de eigenschap die gebonden moet worden. Dit pad bestaat uit namen van eigenschappen, velden en indexen van indexers eventueel gescheiden door punten.

15.4.5 Controleservercomponenten

Controleservercomponenten worden gebruikt om de gebruikersinvoer op een webpagina na te kijken. Deze component wordt geassocieerd met een servercomponent waarvan hij de invoer moet controleren. Een controleservercomponent bevat dus logica om de test uit te voeren en een grafische voorstelling, die de gebruiker verwittigt bij verkeerde invoer. Mogelijke controles zijn onder andere: nagaan of een veld ingevuld is, of het voldoet aan een bepaald patroon, of de waarde in een bepaald interval ligt, ...

Controleservercomponenten worden op dezelfde manier toegevoegd aan een formulier als de andere servercomponenten (§15.4.3). Elke controleservercomponent verwijst naar een invoercomponent op de pagina, bepaald door het attribuut *ControlToValidate*. Als de gebruikersinvoer verwerkt wordt (bv. bij het indienen van het formulier) wordt de waarde van de invoer doorgegeven aan de juiste controlecomponent(en). Als alle controles uitgevoerd zijn en voldoen dan is de hele formulier geldig. Indien het formulier niet geldig is wordt het niet verwerkt en wordt het opnieuw getoond aan de gebruiker met de nodige foutmeldingen.

Bij het indienen van het formulier wordt de invoer van de gebruiker gecontroleerd, m.a.w. alle controlecomponenten worden gevalideerd. Om een bepaalde controle vroeger uit te voeren kan je de methode *Validate* van de corresponderende controlecomponent oproepen. Deze methode zorgt ervoor dat de controle uitgevoerd wordt en dat de eigenschap *IsValid* van de component een waarde krijgt (namelijk *true* of *false*).

Controleservercomponenten zijn normaal onzichtbaar voor de gebruiker. Ze worden enkel getoond als er een fout optreedt. Vaak wordt een foutboodschap getoond naast de invoercomponent, maar het is ook mogelijk om een overzicht van verschillende fouten (van verschillende componenten) te tonen.

Het uitvoeren van de controles kan zowel op de server als op de client gebeuren. Om op de client te kunnen valideren moet de browser van de gebruiker DHTML (Dynamic HTML) ondersteunen. Bij het indienen van het formulier wordt dan op de server nog eens gecontroleerd. Als validatie op client mogelijk is, zal de component dit automatisch doen. Dit betekent dat hij de nodige scripts en HTML-code genereert bij het aanmaken van de webpagina. Met de eigenschap *EnableClientScript* of met de eigenschap *ClientTarget* van de klasse *Page* is het mogelijk om validatie op de client uit te schakelen.

In de klassenbibliotheek van het .NET-platform zijn de volgende controlecomponenten beschikbaar. Ze behoren allen tot de namespace *System.Web.UI.WebControls*.

- *RequiredFieldValidator*: zorgt ervoor dat een gebruiker voor een bepaalde component iets invoert.
- *RegularExpressionValidator*: zorgt ervoor dat de invoer van de gebruiker voor de geassocieerde component voldoet aan een reguliere uitdrukking.
- *RangeValidator*: controleert of de ingegeven waarde tussen een onder- en bovengrens ligt.
- *CompareValidator*: vergelijkt de invoer van de gebruiker met een constante waarde of de waarde van een andere component of een waarde uit een gegevensbank.

- *CustomValidator*: maakt het mogelijk om eigen controles op te stellen, hierbij wordt gebruikt gemaakt van een zelfgeschreven logische functie.

Voorbeeld

Het voorbeeld uit §15.4.1 wordt uitgebreid met een aantal *RequiredFieldValidators*, die ervoor zorgen dat er foutboodschappen getoond worden als de gebruiker geen naam intikt of geen categorie kiest.

```
...
 <table>
    <tr><td>
      Naam: <asp:textbox id="Naam" runat="server"/>
    </td><td>
      <asp:RequiredFieldValidator ControlToValidate="Naam"
        Display="Dynamic" errormessage="Naam invullen aub!"
            runat=server/>
    </td></tr>
    <tr><td>
      Categorie:
      <asp:dropdownlist id="Categorie" runat="server">
        <asp:listitem><!--Kies Categorie--></asp:listitem>
        <asp:listitem>muziek</asp:listitem>
        <asp:listitem>sport</asp:listitem>
        <asp:listitem>actualiteit</asp:listitem>
      </asp:dropdownlist>
    </td><td>
      <asp:RequiredFieldValidator ControlToValidate="Categorie"
        Display="Dynamic" InitialValue="<!--Kies Categorie-->"
        errormessage="Categorie selecteren aub!" runat=server/>
    </td></tr>
  </table>
    ...
```

De foutboodschap (bepaald door het attribuut *errormessage*) wordt getoond als de waarde van de te controleren component (aangeduid door het attribuut *ControlToValidate*) de beginwaarde is. Standaard is de beginwaarde een lege string. Ze kan ook worden opgegeven met het attribuut *InitialValue*. In het voorbeeld betekent dit dat de foutboodschap getoond wordt als de lijst *Categorie* nog steeds de waarde $<$ *!–Kies Categorie–*$>$ heeft.

Het attribuut *Display* bepaalt of er op voorhand plaats wordt voorzien voor de foutboodschap of niet. Heeft dit attribuut de waarde *Dynamic* dan wordt er geen plaats voorzien, is de waarde *Static* dan is dit wel het geval.

15.4.6 Gebruikersservercomponenten

Gebruikersservercomponenten zijn zelfgemaakte, herbruikbare componenten die bestaan uit HTML-code en andere servercomponenten. Je kan het vergelijken met webformulieren die geen *html*, *body* of *form*-tags bevatten.

15.5 Webformulieren

Webformulieren worden gebruikt om webpagina's te genereren. Deze webpagina's vormen de gebruikersinterface van een webapplicatie. Eén van de basisconcepten in ASP.NET is dat men webapplicaties wil ontwikkelen zoals grafische (windows-)applicaties. Dit betekent dat men een grafische layout heeft met een onderliggende logica. De applicatie reageert op gebruikersacties zoals het intikken van gegevens, het klikken met de muis, ... Een verschil tussen een windowsapplicatie en een webapplicatie is dat bij elke gebruikersactie er een HTTP-bericht heen en terug gaat naar de server omdat een (gegenereerde) webpagina nauwelijks functionaliteit kan bevatten. Dit betekent onder andere dat een webformulier maar één HTML-formulier bevat en dat de actie van dit formulier verwerkt wordt door hetzelfde webformulier. De *form*-tag bevat dan ook geen *action*-attribuut als hij een attribuut *runat="server"* heeft. Een ander gevolg is dat het afhandelen van gebeurtenissen (*events*) niet gebeurt op de plaats waar de gebeurtenis optreedt. De gebeurtenis vindt plaats op de client, zorgt ervoor dat het formulier ingediend wordt en wordt afgehandeld op de server. Bovendien moet de inhoud van de verschillende componenten na elk HTTP-bericht opnieuw ingevuld worden. Dit betekent dat de gegevens van de verschillende componenten op het webformulier bijgehouden worden.

Een webformulier bestaat uit twee stukken: het grafische stuk en het logische stuk, dat de functionaliteit van de gebruikersinterface bevat.

Het visuele stuk bestaat uit een .aspx-bestand. Dit bestand kan HTML en servercomponenten (*server controls*) bevatten. Eigenlijk kan je het beschouwen als een container met verschillende componenten (statische tekst en servercomponenten) erin.

Het logische stuk is een .aspx.cs-bestand en wordt *code behind* file genoemd. Dit bestand bevat een klasse afgeleid van de klasse *Page*. Het visuele stuk is eigenlijk een afgeleide klasse van de klasse voorgesteld door de *code behind* file. Deze klassen worden gecompileerd bij de eerste aanvraag van een gebruiker van de webapplicatie. De klassen die horen bij de *code behind* files kunnen op voorhand gecompileerd worden tot een bibliotheek.

15.5.1 Voorbeeld

De webpagina uit het voorbeeld toont een tabel met de naam en voornaam van een reeks personen uit een gegevensbank. De eerste keer dat de webpagina opgeroepen wordt vanuit een bepaald browservenster worden de gegevens uit de gegevensbank gehaald. Als er op de knop van het formulier geklikt wordt, dan wordt dezelfde tabel getoond met daarboven een korte boodschap. De gegevens van de tabel worden niet opnieuw uit de gegevensbank gelezen. Het bestand *personenMetCodeBehind.aspx*

is het grafische stuk van het webformulier, het bestand *personenMetCodeBehind.aspx.cs* het logische.

De webpagina (bepaald door *personenMetCodeBehind.aspx*) bestaat uit een label, een datagrid en een knop.

```
<%@ Page Inherits="CodeOverzichtPersoon"
  Src="personenMetCodeBehind.aspx.cs" Language="c#" Debug="true"%>
<html>
<head><title>Overzicht personen</title></head>
<body>
<form method="post" runat="server">
<asp:label id="Bericht" runat="server"/>
<asp:datagrid id="Lijst" HeaderStyle-BackColor="#aaaadd"
  BackColor="#ccccff" runat="server"/>
<input type="submit" value="Dien In" runat="server">
</form>
</body>
</html>
```

In de eerste lijn geeft het attribuut *Inherits* aan van welke klasse de klasse, waarin het grafische stuk wordt omgezet, afgeleid is. In het voorbeeld is het webformulier een afgeleide klasse van *CodeOverzichtPersoon*. De locatie van de broncode die de klasse *CodeOverzichtPersoon* beschrijft, wordt bepaald door het attribuut *Src*.

Het bestand *personenMetCodeBehind.aspx.cs* beschrijft de klasse *CodeOverzichtPersoon* en dus de logica van het webformulier.

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Web.UI;
using System.Web.UI.WebControls;

public class CodeOverzichtPersoon : Page {

    public DataGrid Lijst;
    public Label Bericht;

    public void Page_Load(Object sender, EventArgs e) {

        if (!IsPostBack) {

            // gegevens ophalen uit gegevensbank
            String connString = "...";
            String query =
                "select voornaam, achternaam from personen";
            OleDbConnection conn =
                new OleDbConnection(connString);
            OleDbDataAdapter adapter =
```

```
                new OleDbDataAdapter(query, conn);

            DataSet ds = new DataSet();
            adapter.Fill(ds, "Personen");

            // gegevens verbinden met Lijst
            Lijst.DataSource =
                ds.Tables["Personen"].DefaultView;
            Lijst.DataBind();
        } else {
            Bericht.Text="Hallo, welkom terug!";
        }
    }
}
```

De klasse *CodeOverzichtPersoon* is een afgeleide klasse van *Page* en bevat twee publieke variabelen *Lijst* en *Bericht* die overeenstemmen met de servercomponenten *Lijst* en *Bericht* uit het bestand *personenMetCodeBehind.aspx*.

Het afhandelen van een HTTP-aanvraag voor een webformulier verloopt in verschillende stadia (zie §15.5.2). Eén van die stadia is het verwerken van de gebeurtenis *Page_Load*, het laden van de pagina. Het afhandelen van de gebeurtenis wordt beschreven in de methode *Page_Load*. Elke methode die gebeurtenissen afhandelt (en dus ook *Page_Load*) heeft twee argumenten: het object dat de gebeurtenis veroorzaakte en specifieke informatie over de gebeurtenis. In het voorbeeld zijn dit de variabelen *sender* en *e*.

IsPostBack is een logische publieke eigenschap van de klasse *Page* en dus ook van de klasse *CodeOverzichtPersoon*. Deze eigenschap geeft aan of het webformulier voor de eerste keer wordt geladen of dat het wordt geladen als reactie op het indienen van het formulier op de webpagina.

15.5.2 Levensloop webformulier

De levensloop van een webformulier bestaat uit verschillende fazes, tijdens al deze fazes kunnen gebeurtenissen optreden en afgehandeld worden. Hieronder volgt een beschrijving van de meest voorkomende fazes en de gebeurtenissen die ze veroorzaken.

Initialisatie van de pagina Tijdens deze faze wordt de inhoud van de pagina en zijn componenten weer opgebouwd en worden de ingediende gegevens opgehaald. De informatie van de verschillende componenten werd bewaard in een verborgen veld *ViewState* genoemd (zie §15.5.4). De bijhorende gebeurtenis is *Page_Init*.

Laden pagina Tijdens deze faze worden de zelfgeschreven initialisaties uitgevoerd door het afhandelen van de gebeurtenis *Page_Load*, m.a.w. de methode *Page_Load* van de ‘code behind file’ wordt uitgevoerd.

Validaties of controles De methode *Validate* van alle controleservercomponenten wordt uitgevoerd.

Gebeurtenissen afhandelen Als de pagina werd ingediend omdat er een gebeurtenis optrad, dan wordt deze afgehandeld.

Genereren pagina en afsluiten De uitvoer van het webformulier wordt aangemaakt, de bijhorende gegevens worden bewaard in verborgen velden (zie §15.5.4) en vervolgens wordt de gebeurtenis *Page_Unload* opgeroepen. Daarna wordt de informatie van de pagina uit het geheugen verwijderd.

15.5.3 Levensloop servercomponent

Het HTTP-protocol is een statusloos protocol. Dit betekent dat de twee aanvragen van een zelfde gebruiker volledig onafhankelijk zijn en niet met elkaar verbonden. Om de gebruiker toch het idee te geven van een samenhangende applicatie, is het nodig allerlei statusinformatie te bewaren (zie §15.5.4). Bij het ontwikkelen van servercomponenten moet hiermee rekening gehouden worden. Zo moet de levensloop van een component een aantal stadia doorlopen:

Initialisatie Initialisatiefaze van de component.

Ophalen statusinformatie Alle informatie i.v.m. deze component uit de verborgen velden halen en bewaren in de eigenschap *ViewState*.

Dataveranderingen verwerken Gegevens van het ingediende formulier met betrekking tot deze component verwerken.

Laden component Acties uitvoeren die gemeenschappelijk zijn voor alle aanvragen. Tijdens deze faze wordt ook de boom van servercomponenten op een pagina aangemaakt.

Veranderingen melden Gebeurtenissen lanceren om de veranderde status van de component te melden.

Gebeurtenissen afhandelen Gebeurtenissen veroorzaakt op de client afhandelen.

Aanpassingen uitvoeren Veranderingen aan de component uitvoeren.

Statusinformatie bewaren De status van de component bewaren in verborgen velden. Dit betekent dat de eigenschap *ViewState* omgezet wordt naar een string, die meegegeven kan worden aan een verborgen veld.

Uitvoer genereren Uitvoer genereren die deel uit maakt van de te genereren webpagina.

Opruimen Opruimen, externe bronnen afsluiten, ...

Verwijderen De component verwijderen uit het geheugen.

15.5.4 Statusinformatie bewaren

Zoals hiervoor reeds aangegeven is het HTTP-protocol een statusloos protocol en moet een webapplicatie er zelf voor zorgen dat statusinformatie tussen verschillende HTTP-aanvragen bewaard wordt. Hiervoor zijn verschillende technieken mogelijk en ze maken meestal gebruik van verborgen velden, URL-herschrijven of cookies. Eén van de meest gebruikte methodes voor het bewaren van pagina-specifieke informatie in het .NET-platform is gebruik maken van de *ViewState* van componenten in een webformulier. Daarnaast is het ook mogelijk om informatie van een sessie of informatie van de hele applicatie te bewaren.

ViewState

Elke component (merk op dat de klasse *Page* ook een component is) heeft een beschermde (*protected*) eigenschap *ViewState*. Deze eigenschap is van het type *StateBag*. Deze klasse is een mechanisme om naam/waarde-paren van een component te bevatten, te bewaren in een string en te lezen uit een string. De naam is een *string*, de waarde een *object*.

Tijdens de verwerking van een webformulier wordt de informatie van de eigenschap *ViewState* gelezen uit en geschreven naar een verborgen veld van het formulier.

Het is ook mogelijk om zelf informatie toe te voegen aan of te lezen van de *ViewState*-eigenschap van het *Page*-object. Dit wordt geïllustreerd in het volgend voorbeeld.

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;

public class VbViewState : Page {

    public Label Bericht;

    public void Page_Load(Object sender, EventArgs e) {
        int teller;
        if (!IsPostBack) {
            teller = 0;
        } else {
            teller = (int)ViewState["Teller"];
            teller++;
        }
        Bericht.Text="De waarde van de teller is " + teller;
        ViewState["Teller"] = teller;
    }
}
```

Merk op dat de klasse *StateBag* een indexer heeft om de naam/waarde-paren toe te voegen of op te halen.

Sessie-informatie

Sessie-informatie is informatie die gedeeld wordt tussen alle aanvragen van een browser op één machine voor webpagina's van één webapplicatie. Deze informatie wordt bewaard door een object van het type *HttpSessionState*. Elke sessie heeft zo'n object. De sessie-informatie wordt bewaard in sleutel/object-paren. Een referentie naar het sessie-object kan verkregen worden via de eigenschap *Session* van de klasse *HttpContext* of van de klasse *Page*.

Een sessie start als een gebruiker voor de eerste keer een pagina van de webapplicatie opvraagt. Op dat ogenblik wordt er een instantiatie gemaakt van de klasse *HttpSessionState*. Dit object heeft een unieke identiteit. Deze unieke identiteit wordt meegegeven bij elke communicatie tussen de gebruiker en de webapplicatie ofwel via een cookie ofwel via een herschreven URL. Standaard worden cookies gebruikt, maar dit kan aangepast worden in de configuratie van de webapplicatie (zie §15.5.5). Een sessie eindigt als de webapplicatie ze expliciet afsluit of als er een bepaald tijdsinterval verstreken is waarin de sessie niet actief was (*time out*).

Er zijn twee gebeurtenissen die optreden bij elke sessie: één bij het opstarten van de sessie (*Session_OnStart*) en één bij het afsluiten van de sessie (*Session_OnEnd*). De beschrijving van het afhandelen van deze gebeurtenissen staat in het bestand *global.asax* (zie §15.5.5).

In het volgend voorbeeld wordt geïllustreerd hoe een *Kaart*-object aangemaakt en bewaard wordt bij het opstarten van de sessie. Bij het laden van een welbepaalde pagina van de webapplicatie wordt een referentie naar dit object opgehaald. Aan het einde van het afhandelen van de *Page_Load*-gebeurtenis wordt de sessie expliciet afgesloten door de applicatie met de methode *Abandon*.

Inhoud van het bestand *global.asax*

```
void  Session_Start(Object sender, EventArgs E) {
    Session[Constanten.PARAM_KAART] = new Kaart();
}
```

In een webpagina van de applicatie:

```
public void Page_Load(Object sender, EventArgs e) {
    // kaart bepalen
    Kaart kaart = (Kaart)Session[Constanten.PARAM_KAART];
    ...
    Session.Abandon();
}
```

Applicatie-informatie

ASP.NET voorziet een mechanisme om informatie te bewaren die beschikbaar moet zijn voor alle pagina's van een webapplicatie. Het object dat deze informatie bewaart is van het type *HttpAppli-*

cationState. Dit object wordt meestal verkregen via de eigenschap *Application* van het *HttpContext*-object of via de eigenschap *Application* van de klasse *Page*. De klasse *HttpApplicationState* kan sleutel/object-paren bewaren, die dan beschikbaar zijn voor de hele applicatie. Een instantie van deze klasse wordt aangemaakt de eerste keer dat er een HTTP-aanvraag is voor een pagina van de applicatie.

In het volgend voorbeeld wordt geïllustreerd hoe een *Catalogus*-object wordt aangemaakt tijdens het opstarten van de applicatie, m.a.w. tijdens het afhandelen van de gebeurtenis *Application_Start*. Het afhandelen van deze gebeurtenis wordt beschreven in het bestand *global.asax* (zie §15.5.5). Bij het laden van een bepaalde pagina van de applicatie wordt een verwijzing naar dit object bepaald. Vervolgens wordt de lijst van boeken van de catalogus toegekend aan een datalist *Lijst*.

Inhoud van het bestand *global.asax*

```
void Application_Start (Object sender, EventArgs E) {
    Application[Constanten.PARAM_CATALOGUS] =
        new Catalogus();
}
```

In een webpagina van de applicatie:

```
void Page_Load(Object Sender, EventArgs E) {
    // gegevens verbinden met Lijst
    Lijst.DataSource = ((Catalogus)
        Application[Constanten.PARAM_CATALOGUS]).Boeken;
    Lijst.DataBind();
}
```

15.5.5 Configuratie: global.asax

De standaard home directory voor het publiceren van webpagina's met IIS Server is 'c:\Inetpub\wwwroot'. Voor een webapplicatie maken we aparte map die alle gegevens van de applicatie bevat. Om de applicatie te publiceren op de webserver moet die map een virtuele map van IIS zijn (zie handleiding IIS). Zo'n virtuele map heeft een alias die gebruikt wordt in de URL om toegang te krijgen tot de webpagina's van de applicatie. De bibliotheken die een applicatie nodig heeft staan in de *bin*-directory van de virtuele map.

Het bestand *global.asax* is een optioneel bestand dat code bevat om gebeurtenissen op het niveau van de applicatie of de sessie af te handelen. Dit bestand bevindt zich in de basismap van de applicatie. Dit bestand wordt omgevormd en gecompileerd naar een klasse afgeleid van de klasse *HttpApplication*. Bij veranderingen aan dit bestand zal het ASP.NET-framework alle huidige aanvragen afhandelen, de gebeurtenis *Application_OnEnd* lanceren en de applicatie opnieuw opstarten. Daarna wordt de file *global.asax* opnieuw omgevormd en gecompileerd en vindt de gebeurtenis *Application_OnStart* plaats.

Syntax global.asax

Een *global.asax*-bestand kan de volgende zaken bevatten: richtlijnen, code, tags om objecten te declareren en aan te maken, en richtlijnen om bestanden in te sluiten.

De richtlijnen kunnen onder andere gebruikt worden om namespaces te importeren, om te verwijzen naar bibliotheekbestanden die niet in de *bin*-map van de applicatie staan, ...

In de stukken code worden velden en methodes van de klasse waarnaar *global.asax* wordt omgevormd gedeclareerd. Dit kan o.a. gebruikt worden om methodes te beschrijven die gebeurtenissen afhandelen. De syntax van de stukken code is de volgende:

```
<script runat="server" language="c#" src="naambestand">
      .. // code
</script>
```

Indien het attribuut *src* gebruikt wordt, dan mag er geen code in de *script*-tag staan. Het bestand *naambestand* bevat dan de code. De volgende gebeurtenissen kunnen hier afgehandeld worden.

- Application_Start
- Application_End
- Session_Start
- Session_End
- Application_BeginRequest
- Application_EndRequest
- Application_Error

Voorbeeld global.asax

Het volgend stukje code bevat een voorbeeld van een *global.asax*-bestand. De namespace *VbnBo* bevat de klassen *Constanten*,*Catalogus* en *Kaart*.

```
<%@ Import Namespace="VbnBo" %>

<script language="C#" runat="server">

    void Application_Start (Object sender, EventArgs E) {
        Application[Constanten.PARAM_CATALOGUS] =
            new Catalogus();
```

```
    }


    void  Session_Start(Object sender, EventArgs E) {
        Session[Constanten.PARAM_KAART] = new Kaart();
    }

</script>
```

ASP.NET: vervolg

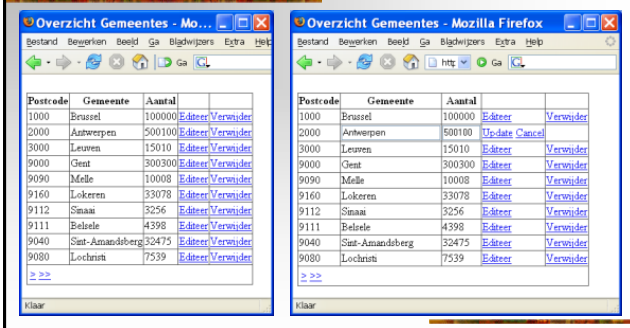
Veerle Ongenae

Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

Voorbeeld Editeerbare tabellen

- Webapplicatie Gemeente
  - overzicht.aspx



Editeerbare tabellen

- Sinds Visual Studio 2005
- Zonder code te schrijven
- Combinatie
  - GridView
  - DataSourceControl
- Moeilijkere scheiding verschillende lagen in applicatie

Aanmaak Editeerbare GridView

- GridView-component
  - Attributen/eigenschappen
    - AutoGenerateColumns
      - False: zelf kolommen en bijhorende eigenschappen bepalen
    - DataSourceID
      - Id van de databron (type: DataSourceControl)
    - DataKeyNames
      - Kolomnamen van de primaire sleutel(s)
  - Kindelement/eigenschap Columns
    - Beschrijving van de kolommen

Beschrijving kolommen

- Kindelement/eigenschap Columns
  - BoundField toevoegen
    - HeaderText: Tekst hoofding
    - DataField
      - Corresponderende kolom in de tabel
    - Read only?
  - CommandField toevoegen
    - Opschrift knop wijzigen
    - "Knop" toevoegen

DataSourceControl

- Component
  - Gebruikt om te binden aan servercomponenten
  - Toegang tot gegevensbron
- Drie types in ASP.NET
  - SqlDataSource
    - Beschikbaar voor gegevensbank waarvoor een .NET-provider bestaat
  - AccessDataSource
    - Access-bestanden
  - XmlDataSource
    - XML-gegevens

AccessDataSource

- Attributen/eigenschappen
  - DataFile
    - Pad naar bestand
  - SelectCommand
    - SQL-string om gegevens te selecteren
  - UpdateCommand
    - SQL-string om gegevens aan te passen
  - DeleteCommand
    - SQL-string om gegevens te verwijderen

Opmerking Access

- De ASP.NET gebruiker moet schrijf- en wijzig-rechten hebben op het access-bestand en op de map waarin het bestand staat

GridView: Paging

- Eigenschap/attribuut AllowPaging
  - True
- Eigenschap PagerSettings
  - Eigenschap Mode
    - Volgende-Vorige of Met nummers
  - Eigenschap NextPageText
    - >
  - Eigenschap PrevPageText
    - <

Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

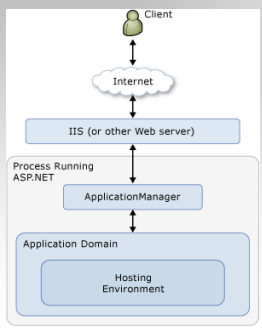
Levensloop

- Levensloop webapplicatie
- Levensloop ASP.NET Page (webformulier)
- Levensloop component

Levensloop webapplicatie

- Starten applicatie-omgeving
- Voor elke HTTP-aanvraag
  - Aanmaak HTTP-objecten
  - Toewijzen HttpApplication-object
  - Aanvraag afhandelen
- Afsluiten applicatie

Starten applicatie-omgeving

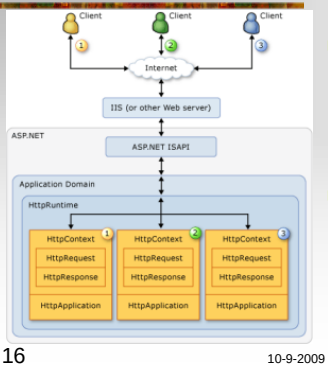


- ApplicationManager
  - Aanmaken en initialiseren webapplicaties
  - Beheer levensloop
    - Webapplicatie
    - Objecten in de webapplicatie
- Per webapplicatie één Application Domain
  - HostingEnvironment
    - Informatie over de webapplicatie
      - Bv. pad

Aanmaak HTTP-objecten

- Aanmaak objecten die de HTTP-aanvraag voorstellen
  - HttpContext
    - Alle HTTP-informatie van de huidige aanvraag
    - bevat
      - HttpRequest
      - HttpResponse

Toewijzen HttpApplication



- Aanmaak HttpApplication-object
  - Instantie van Global.asax, indien beschikbaar
- HttpApplication-object
  - Handelt één aanvraag af
  - Kan hergebruikt worden

Aanvraag afhandelen

- Een hele reeks gebeurtenissen worden opgeroepen en afgehandeld
- Gebeurtenissen die je zelf kan afhandelen
  - Maken deel uit van Global.asax
    - Application_Start
      - Wordt maar één keer uitgevoerd: bij het opstarten van de applicatie
    - Application_End
      - Wordt maar één keer uitgevoerd: bij het afsluiten van de applicatie
    - …

Levensloop

- Levensloop webapplicatie
- Levensloop ASP.NET Page (webformulier)
- Levensloop component

Levensloop Page-object (webformulier)

- Bij elke aanvraag wordt een Page-object aangemaakt
- Het Page-object bevat de verschillende componenten (controls)
- Na het doorsturen van het antwoord wordt het Page-object verwijderd
- Wat zijn de verschillende stappen in de levensloop van dit Page-object?

Levensloop webformulier

- Start
  - aanmaak Page-object, instellen eigenschap IsPostBack
- Initialisatie
  - Toevoegen componenten aan webformulier
- Laden
  - Herstellen eigenschappen componenten (uit verborgen formulierelementen en HTTP-parameters)
- Validatiefaze
  - Controles van de verschillende componenten uitvoeren (methode Validate)

Levensloop webformulier (vervolg)

- Afhandelen events
  - Oproepen eventhandlers van de verschillende componenten
- Renderen
  - Statusinformatie verschillende componenten bewaren
  - Uitvoer genereren
- Opruimen pagina

Bijhorende events (van Page)

- PreInit
- Init
- Load
  - Gebruik: instellen eigenschappen componenten
- Events van de componenten
- PreRender
- UnLoad

Koppelen methodes

- Hoe eventhandlers koppelen aan de events van het webformulier (Page-object)?
- Afhankelijk van het attribuut AutoEventWireup (@Page-richtlijn)
  - true (standaard)
    - Methode met naam Page_*Event* toevoegen aan Code Behind
  - false
    - Methode schrijven in Code Behind
    - Methode zelf toevoegen aan event

Levensloop

- Levensloop webapplicatie
- Levensloop ASP.NET Page (webformulier)
- Levensloop component

Levensloop component

- Initialisatie
- Ophalen statusinformatie
- Verwerken veranderingen data
- Laden
- Veranderingen melden
- Afhandelen gebeurtenissen
- Aanpassingen uitvoeren
- Statusinformatie bewaren
- Uitvoer genereren
- Opruimen
- Verwijderen

Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

Beveiliging

- Afschermen delen van een website
- Twee stappen
  - Authenticatie
    - Gebruiker "herkennen"
    - De gebruiker is wie hij zegt te zijn
    - Inloggen
  - Authorisatie
    - Bepaalde rechten (niet) toekennen aan bepaalde gebruikers
      - Leden kunnen meer pagina's bekijken dan niet-leden
      - Administrators kunnen nog andere pagina's raadplegen

Authenticatie

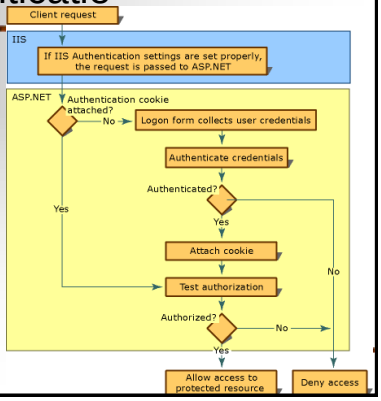
- Verkrijgen identificatiegegevens gebruiker
  - Bv. Login en wachtwoord
- Valideren van deze gegevens
  - Gegevensbank, XML-bestand, Windows

Authenticatie in ASP.NET

- Verschillende mogelijkheden
  - Windows
    - Windowsgebruiker
  - Forms
    - Met een loginformulier
  - Passport
    - Microsoft paspoort
  - None
    - Geen authenticatie
- Instellen in Web.config

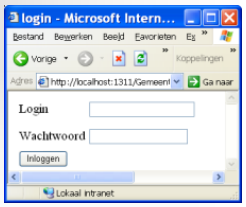
```
<authentication mode="Forms"/>
```


Forms Authenticatie



Voorbeeld inloggen

- Webapplicatie Gemeente
  - login.aspx
  - Web.config
- Loginformulier wordt getoond bij een aanvraag van een willekeurige pagina van de applicatie



Forms Authenticatie

- Niet geauthenticeerde aanvragen
  - Naar inlogformulier
- Inloggen gelukt
  - Cookie (al dan niet persistent)
  - Volgende aanvragen niet meer inloggen

Forms Authenticatie

- Web.config

```
<authentication mode="Forms">
   <forms loginUrl="login.aspx"/>
   </authentication>
```

- login.aspx
  - Controle OK

```
FormsAuthentication.RedirectFromLoginPage(L
  oginNaam.Text,False);
```

Geen persistente cookie

Ingebouwde Forms Authenticatie

- Web.config

```
<authentication mode="Forms">
   <forms loginUrl="LoginPagina.aspx"/>
</authentication>
```

- LoginPagina.aspx
  - Login-component
    - Loginnaam, wachtwoord, …
    - Configuratie via eigenschappen/attributen
  - ValidationSummary
    - Toon foutboodschappen

Aanmaken gebruikers

- Via Web Site Administration Tool
  - Onderliggende gegevensbank in SQL Server Express
  - DB-bestand automatisch aangemaakt in de map App_Data
- Maakt gebruik van de AspNetSqlMembershipProvider
  - Aanmaken en beheren gebruikers
  - …

Windows Authenticatie

- Gebruiker moet ook gebruiker van de servercomputer zijn
- Toegang wordt dan bepaald door de rechten die de gebruiker op de servercomputer heeft

Authorisatie

- Bepalen of een gebruiker al dan niet toegang heeft tot een bepaalde pagina
- Verschillende mogelijkheden
  - Bestandsniveau
    - Windows authenticatie nodig
  - URL-authorisatie
    - In Web.config
      - authorization-element
        - allow
        - deny

URL-authorisatie

- <[element] [users] [roles] [verbs]/>
  - Element
    - allow
    - deny
  - Users
    - Gebruikersnamen gescheiden door ,
    - * : alle gebruikers
    - ? : anonieme gebruikers
  - Roles
    - Gebruikersgroepen
  - Verbs
    - GET, HEAD, POST, … (soorten HTTP-aanvragen)

URL-authorisatie: Web.config

```
<authorization>
    <deny users="?" />
</authorization>
```

- Wie ingelogd is mag de pagina's bekijken
- De eerste regel die toegepast kan worden, wordt gebruikt.
  - Volgende regels worden dan genegeerd

Authorisatie

- ASP.NET
  - NTFS file permissions
  - XML-bestand
    - Gebruikers
    - Rollen
    - HTTP types
- IIS
  - Hostname/IP-adres

Authorisatie voor verschillende rollen

- Web.config bestanden in verschillende map
  - Neemt instellingen bovenliggende map over
  - Voegt eigen instellingen toe

Meer invloed op loginproces

- Events van de Login-component afhandelen
  - LoggingIn: voor controle door component
    - bv. nakijken of gebruikersnaam een email is
  - LoggedIn: na controle door component
  - LoginError: fout bij controle door component
    - Bv. extra links voor hulp tonen
  - Authenticate: zelf controle uitvoeren

Andere Security-componenten

- LoginStatus
  - Toont een link login (of logout) afhankelijk van de inlogstatus van de gebruiker
- LoginView
  - Biedt de mogelijkheid om ingelogde gebruikers andere informatie te tonen dan niet-ingelogde gebruikers
- PasswordRecovery
  - Gebruiker is wachtwoord vergeten
    - Stuurt email met (nieuw?) wachtwoord indien juist antwoord op wachtwoordvraag

Voorbeeld afhandelen Authenticate-event

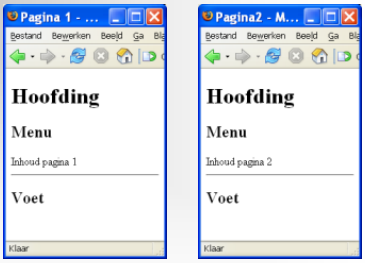
```
protected void
   LoginComponent_Authenticate(object
   sender, AuthenticateEventArgs e) {
       e.Authenticated = … // true of false
}
```


Overzicht

- Wat is ASP.NET?
- Gebeurtenissen
- Servercomponenten
- Statusinformatie bewaren
- Editeerbare tabellen in ASP-pagina's
- Levensloop
- Beveiliging
- Sjablonen en Navigatie

Sjablonen

- Uniforme layout voor alle pagina's van de webapplicatie
- Webapplicatie VbSjabloon
  - MasterPage.master
  - pagina1.aspx
  - pagina2.aspx



Master pages (sjabloon)

- Uniforme layout voor alle pagina's van de webapplicatie
- Layout is bepaald door een master page
  - .master-bestand
  - Ipv een page-richtlijn is er een master-richtlijn

```
<%@ Master Language="C#"
   CodeFile="MasterPage.master.cs"
   Inherits="MasterPage" %>
```


Inhoud master page

- HTML
- Servercomponenten
- Eén of meerdere ContentPlaceHolder-componenten
  - Definiëren een gebied waar variabele inhoud komt
  - Die "variabele inhoud" wordt beschreven in gewone ASP.NET-pagina's

ContentPlaceHolder

- In master page

<asp:contentplaceholder id="Inhoud" runat="server" />

- In ASP.NET-pagina's

<asp:Content ID="Content1" ContentPlaceHolderID="Inhoud" Runat="Server">Inhoud pagina 1 </asp:Content>

- De uitvoer van de master page en de ASP.NET-pagina worden gecombineerd tot één antwoord-pagina

Link master page en inhoud

- Per ASP.NET-pagina

<% @ Page Language="C#" MasterPageFile="~/Master.master" Title="Content Page 1" %>

- Globaal voor de hele applicatie
  - In Web.config

<pages masterPageFile="MySite.Master" />

  - Sjabloon wordt toegepast voor alle webformulieren met Content-componenten

Toegang tot leden van de master page

- Koppeling tussen webformulier en master

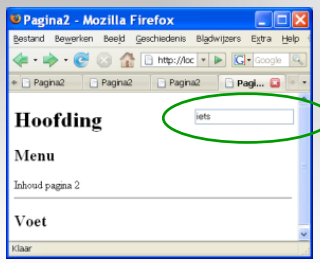
<%@ Page masterPageFile="~/MasterPage.master"%>

- Type van de master declareren

<%@ MasterType virtualPath="~/MasterPage.master"%>

- Eigenschap Master van Page
  - Referentie naar de master page
  - Oproepen leden en componenten van de master page is dan mogelijk

Voorbeeld

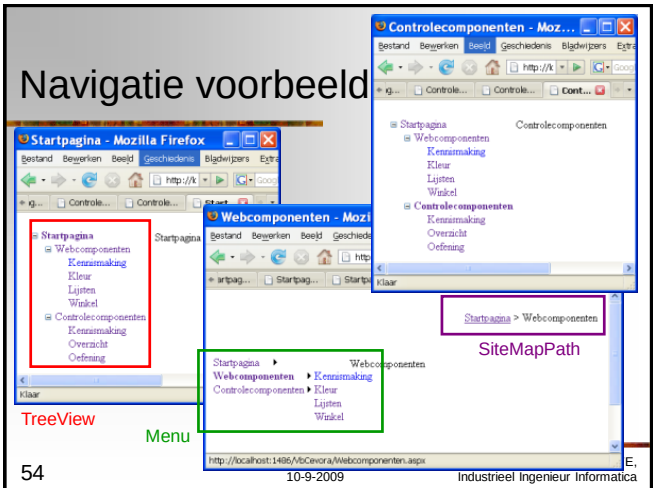


Ingevuld door pagina2.aspx
Deel masterpage

Navigatie

- Consequent en overzichtelijk
- Ingebouwd in ASP.NET
- Drie stappen
  - Structuur applicatie
    - Bestand Web.sitemap
  - Structuur inlezen in een data-object
    - SiteMapDataSource-object definiëren
  - Data-object tonen in een navigatiecomponent
    - TreeView
    - Menu

Navigatie voorbeeld



Structuur vastleggen

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
    <siteMapNode url="~/Start.aspx" title="Startpagina“
      description="Startpagina">
        <siteMapNode url="~/Webcomponenten.aspx" … >
            <siteMapNode url="Hallo.aspx" title="Kennismaking" … />
            <siteMapNode url="Kleur.aspx" title="Kleur" … />
            …
        </siteMapNode>
        <siteMapNode url="~/Controlecomponenten.aspx" … >
            <siteMapNode url="KeuzeMetControle.aspx" … />
            …
        </siteMapNode>
    </siteMapNode>
</siteMap>
```


Structuur vastleggen

- XML-bestand **Web.sitemap**
- Basiselement (root element) **siteMap**
  - Eén kindelement: **siteMapNode** – element
    - Kan opnieuw (meerdere) **siteMapNode**-elementen bevatten
    - Boomstructuur
    - Stellen een pagina van de webapplicatie voor

Structuur → data-object

- Door XmlSiteMapProvider
- Resultaat
  - SiteMapDataSource-object
  - Kan gebonden worden aan navigatiecomponenten
- Hoe?
  - Grafisch SiteMapDataSource-component (uit ToolBox → Data) op aspx-pagina plaatsen

Navigatiecomponenten

- Navigatiecomponenten
  - **TreeView**
  - **Menu**
  - **SiteMapPath**
    - Pad naar pagina
- ToolBox → Navigation
- Attribuut/Eigenschap **DataSourceID**
  - Verwijst naar het **ID**-attribuut van het **SiteMapDataSource**-object
- Geselecteerd menu in **TreeView** kan in vetjes
  - Aanpassen eigenschap **SelectedNodeStyle**

Opmaak **Menu**

- Zichtbare menu-items: statisch
- Onzichtbare “uitklapbare” menu-items: dynamisch
- Verschillende stijlen editeerbaar o.a. via properties-venster
  - **StaticMenuItemStyle, StaticMenuStyle, StaticSelectedStyle**
  - **DynamicMenuItemStyle, DynamicMenuStyle, DynamicSelectedStyle**

Installatie van een ASP.NET-applicatie

- Menu Website
  - Copy Website
- Applicatie aanmaken in IIS

Structuur webapplicatie

- aspx-bestanden, aspx.cs-bestanden, ascx-bestanden, ascx.cs-bestanden, asax-bestanden, Web.config
  - App_Code
    - Alle andere codebestanden
  - App_Data
    - Bv. Lokale gegevensbanken
  - …

Informatie over ASP.NET

- ASP.NET Web Applications in the .NET Framework (http://msdn.microsoft.com/en-us/library/655cec97.aspx)
- Microsoft ASP.NET QuickStarts Tutorial (http://quickstarts.asp.net/QuickStartv20/aspnet/Default.aspx)
- The Official Microsoft ASP.NET Site (http://www.asp.net/)

Informatie over ASP.NET

- ASP.NET syntax (http://msdn.microsoft.com/en-us/library/fy30at8h.aspx)
- MSDN Code Gallery (http://code.msdn.microsoft.com/)

Hoofdstuk 16

Webservices

Webservices

Veerle Ongenae

Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

Gegevens uitwisselen

- Data uitwisselen tussen programma's op verschillende computers
- Tot nu toe
  - EDI
  - CORBA, RMI, DCOM

EDI

- Electronic Data Interchange
- WAN voor een geografische regio, waar partners kunnen inpluggen
- Afspraak: boodschappen en protocol voor datatransfer
- Networkdetails overgelaten aan de implementatie
- Verschillende netwerken
  - Duur om in te stappen
  - Moeilijk en duur om andere netwerken te bereiken

CORBA, RMI, DCOM

- Afkortingen
  - Common Object Request Broker Architecture
  - Remote Method Invocation
  - Distributed Component Object Model
- Gedistribueerde object-infrastructuur over het Internet
- Transportprotocol bovenop TCP/IP (verschillend voor CORBA, RMI en DCOM) voor verschillende platformen
- Afspraken over communicatie tussen objecten

CORBA, RMI, DCOM

- Voordeel: de verschillende servers hoeven niet tot het zelfde netwerk te behoren
- Kunnen enkel met dezelfde infrastructuur praten, tenzij extra lagen op een reeds ingewikkelde structuur

SOAP

- SOAP zorgt voor verandering:
  - Mogelijkheid tot het uitwisselen van data overal op het web
  - Geen koppeling tussen data en transport
  - Niet binair

Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

SOAP

- Simple Object Acces Protocol
- Protocol gebaseerd op XML
- Uitwisseling van informatie in een gedecentraliseerde, gedistribueerde omgeving
- Transport via het web (HTTP, …)
- SOAP-bericht
  - Verstuurd door zender
  - Naar ontvanger

SOAP en HTTP

- De eerste lijn en de headerlijnen blijven bestaan
- Het corpus van een HTTP-aanvraag en HTTP-antwoord bevat XML ipv HTML, …
- Analoog via FTP of SMTP

SOAP

- XML-envelop
  - Met XML-inhoud
- Een aantal regels voor servers die een SOAP-boodschap ontvangen
- Maakt gebruik van bestaande transportprotocols (bv. HTTP)
- Oorspronkelijk Microsoft, ondertussen standaard van W3C

Wat is er nodig om data uit te wisselen via SOAP?

- Afgesproken DTD of XML Schema voor de uit te wisselen gegevens
- SOAP-server
  - Afhandelen aanvragen
- Client
  - Software om XML-documenten te genereren

Overzicht SOAP

- XML-tags die een SOAP-bericht beschrijven en die een framework geven voor de inhoud van het bericht
- Regels voor het uitwisselen van zelf gedefinieerd gegevenstypes (o.a. aanvaarden, verwerpen, uitzonderingen)
- Conventies voor het uitvoeren van methodes op een andere server en het voorstellen van het resultaat

SOAP Bericht: voorbeeld aanvraag

```
POST /ZwiftBooks HTTP/1.1
Host: www.zwiftbooks.com
Content-Type:text/xml
Content-Length: 134
SOAP Action: "Some-URI"

<Envelope xmlns="http://www.w3.org/2001/09/soap-envelope"
   encodingStyle="http://www.w3.org/2001/09/soap-encoding">
    <Body>
        <zwiftbooks:GetGuaranteedDeliveryTime
          xmlns:zwiftbooks="www.zwiftbooks.com">
            <zwiftbooks:isbn>0-13-18188-222-2</zwiftbooks:isbn>
            <zwiftbooks:zipcode>75240</zwiftbooks:zipcode>
        </zwiftbooks:GetGuaranteedDeliveryTime>
    </Body>
</Envelope>
```


SOAP Bericht: voorbeeld antwoord

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 122

<Envelope xmlns="http://www.w3.org/2001/09/soap-envelope">
    <Body>
        <zwiftbooks:GetBestDeliveryTimeResponse
          xmlns:zwiftbooks="www.zwiftbooks.com">
            <zwiftbooks:Time>2 hours</zwiftbooks:Time>
        </zwiftbooks:GetBestDeliveryTimeResponse>
    </Body>
</Envelope>
```


Structuur SOAP Bericht

- SOAP-envelop
- SOAP-hoofding
- SOAP-corpus

Structuur SOAP Bericht: envelop

- SOAP-envelop:
  - Envelope-element
    - Attributen
      - encodingStyle: hoe geserialiseerd wordt
  - Rootelement van het XML-document dat een SOAP-bericht voorstelt
  - Verplicht

Structuur SOAP Bericht: hoofding

- SOAP-hoofding:
  - Header-element
  - Extra richtlijnen en informatie voor SOAP-server
    - Transacties
    - Beveiliging
    - Filters
    - …
  - Optioneel

Voorbeeld: SOAP-hoofding

```
<Header>
    <n:alertcontrol
      xmlns:n="http://example.org/alertcontrol">
        <n:priority>1</n:priority>
        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
</Header>
```


Structuur SOAP Bericht: corpus

- SOAP-corpus:
  - Body-element
  - Bevat de uit te wisselen XML
    - Onafhankelijk van SOAP-structuur
    - Domein-specifieke structuur
  - Inhoud
    - Data/Resultaat
    - Aanroep van een methode
  - Verplicht

SOAP-berichtenpad

- Bestaat uit SOAP-knopen:
  - SOAP-zender
  - Nul of meerdere tussenliggende SOAP-servers
  - SOAP-ontvanger
- Tussenliggende SOAP-servers
  - Stuurt SOAP-bericht door
  - Behandelt de voor hem bedoelde SOAP-hoofdingen

SOAP-rollen

- Elke SOAP-knoop heeft één of meerdere rollen
- Mogelijke rollen
  - Gespecifieerd door SOAP:
    - none (niet bedoeld voor een specifieke knoop)
    - next (ontvanger en alle tussenliggende)
    - ultimateReceiver (ontvanger)
  - Gespecifieerd door de applicatie
    - Proxy
    - Beveiliging
    - Caching
    - …

SOAP-rollen

- Een SOAP-hoofding kan aangeven voor welke SOAP-knoop de hoofding bedoeld is door gebruik te maken van rollen
  - Attribuut role
  - Waarde attribuut
    - Rol gespecifieerd door SOAP
    - URI SOAP-knoop

Voorbeeld: SOAP-hoofding

```
<Header
  role="http://schemas.xmlsoap.org/soap/actor/next">
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol"
      mustUnderstand="true">
        <n:priority>1</n:priority>
        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
</Header>
```

Regels voor SOAP-server

- Indentificatie van de stukken van het SOAP-bericht die voor de huidige server bedoeld zijn
- Nagaan of deze server alle delen van de headers die bedoeld zijn voor deze server en die het attribuut mustUnderstand hebben ondersteunt. Indien niet fout genereren

Regels voor SOAP-server

- Uitvoeren van de stukken van de headers bedoeld voor deze server
- Indien niet de SOAP-ontvanger: alle headers bedoeld voor deze server verwijderen en bericht doorsturen

SOAP-fouten

- Wanneer
  - Niet begrijpen bericht
  - Fout bij behandelen van bericht
- fault-element
  - Kind van body-element
  - Kinderen
    - faultcode: foutcode
    - faultstring: leesbare verklaring
    - detail: informatie over het probleem

Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

Webservices

- Kader
  - Applicaties bouwen op basis van losse componenten
- Wat?
  - Diensten aanbieden voor applicaties
  - Vergelijking
    - Presentatielaag maakt gebruik van diensten van de applicatielogica
- Technologie
  - Verschillende protocollen

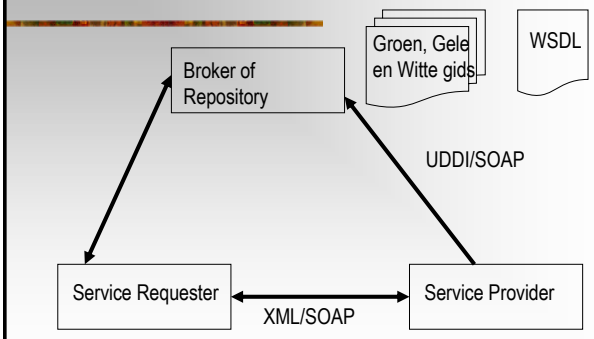
Verschillende facetten van webservices

- Beschrijving
  - Aangeboden diensten
- Publiceren
  - Bij een ‘repository’
  - Vergelijkbaar met telefoonboek: witte, gele en groene gids
- Uitvoeren
  - Een applicatie maakt gebruik van de aangeboden diensten
- Antwoorden
  - De dienst stuurt een resultaat terug naar de applicatie

Webservices: onderdelen

- Service provider
  - Biedt een interface
  - Voor software
  - Bepaalde set taken
- Service requester
  - Doet een aanvraag bij service provider
  - Krijgt antwoord van service provider
  - Maakt deel uit van een applicatie
- Broker
  - Publiceert beschikbare diensten

Overzicht webservices



Webservices: technologieën

- UDDI
  - Universal Description, Discovery and Integration
- WSDL
  - Web Services Definition Language
- SOAP

UDDI

- UDDI
  - XML-Protocol
  - Webservices beschrijven en registreren bij een broker
  - Communicatie tussen
    - Service provider en broker
    - Broker en service requester
  - Inhoud van een SOAP-envelop

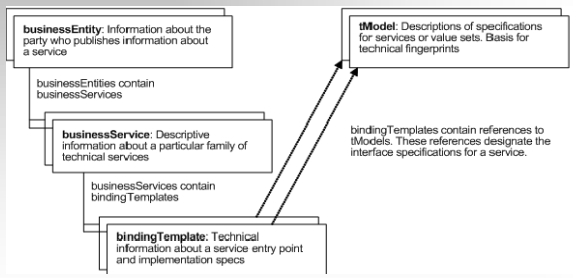
Informatie in een register

- Witte gids
  - Contactinformatie: naam, adres, webstek, …
  - Registratie van de provider
- Gele gids
  - Categorieën van diensten
  - Hoe vind een client een webservice
- Groene gids
  - Technische informatie
  - Hoe kan een client gebruik maken van een webservice

Elementen van UDDI

- businessEntity
  - Informatie over bepaalde provider
- businessService
  - Beschrijving van een bepaalde webservice bestaande uit één of meerdere diensten
- bindingTemplate
  - Technische informatie, informatie voor implementatie
- tModel
  - Technische informatie per dienst

Elementen van UDDI



WSDL

- WSDL
  - XML-standaard W3C
  - Beschrijven webservice
    - Interface
    - Implementatiedetails
  - Document kan meegestuurd worden met UDDI
  - Bepaalt hoe de clientapplicatie de aangeboden dienst moet aanspreken

WSDL

- Twee versies
  - WSDL 1.1 (http://www.w3.org/TR/2001/NOTE-wsdl-20010315)
  - WSDL 2.0 (http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/)

Voorbeeld WSDL 1.1

```
<definitions name="BoekAgent"
   targetNamespace="http://www.zwiftbooks.com/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://www.zwiftbooks.com/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" >

   <message name="GetDeliveryInfoRequest">
      <part name="zipcode" type="xsd:integer"/>
      <part name="isbn" type="xsd:string"/>
   </message>

   <message name="GetDeliveryInfoResponse">
      <part name="result" type="xsd:duration"/>
   </message>
```


Voorbeeld WSDL 1.1

```
<portType name="BoekAgent">
   <operation name="GetDeliveryInfo" parameterOrder="zipcode
     isbn">
      <input message="tns:GetDeliveryInfoRequest"
        name="GetDeliveryInfoRequest "/>
      <output message="tns:GetDeliveryInfoResponse "
        name="GetDeliveryInfoResponse"/>
   </operation>
   <!-- andere diensten -->
</portType>
```


Voorbeeld WSDL 1.1

```
<binding name="BoekAgentBinding" type="tns:BoekAgent">
   <soap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http" />
   <operation name="GetDeliveryInfo">
      <soap:operation
        soapAction="http://zbooks.org/action/ZBooks.Get-
        DeliveryInfo" />
      <input>
         <soap:body use="encoded"
           namespace="http://zbooks.org/message/"
           encodingStyle="http://schemas.xmlsoap.org/soap/en
           coding/" />
      </input>
```

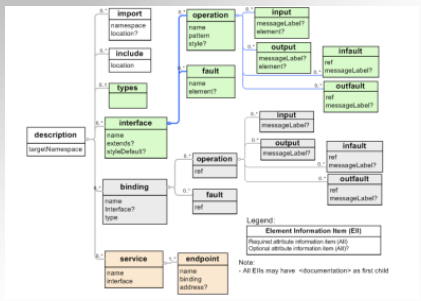
Voorbeeld WSDL 1.1

```
            <output>
                <soap:body use="encoded"
                  namespace="http://zbooks.org/message/"
                  encodingStyle="http://schemas.xmlsoap.org
                  /soap/encoding/" />
            </output>
        </operation>
    </binding>
</definitions>
```


WSDL 1.1 -structuur

- Messages
  - Berichten
- Port Types
  - Verzameling van alle diensten van een webservice
- Bindings
  - Beschrijving concreet formaat van de berichten en het gebruikte protocol
- Services
  - Beschrijving van een webservice bestaande uit verschillende “port types”

WSDL 2.0 -structuur



WSDL 2.0 -structuur

- types: beschrijving berichten in XML Schema
- interface: abstracte funcionaliteit
- binding: hoe toegang tot diensten
- service: waar toegang tot diensten

Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/ns/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc" . . . >
    . . .
</description>
```


Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description  . . . >
    ...
<types>
    <xs:schema ...>
     <xs:element name="checkAvailability"
    type="tCheckAvailability"/>
    <xs:complexType name="tCheckAvailability">
    <xs:sequence>
          <xs:element name="checkInDate" type="xs:date"/>
          <xs:element name="checkOutDate" type="xs:date"/>
          <xs:element name="roomType" type="xs:string"/>
    </xs:sequence> </xs:complexType>
          <xs:element
          name="checkAvailabilityResponse"
          type="xs:double"/>
          <xs:element name="invalidDataError"
          type="xs:string"/>
    </xs:schema>
    </types>
          . . .
</description>
```

Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description …>
    . . . <types> ... </types>
<interface name = "reservationInterface" >
<fault name = "invalidDataFault" element = "ghns:invalidDataError"/>
<operation name="opCheckAvailability" pattern="http://www.w3.org/ns/wsdl/in-out"
    style="http://www.w3.org/ns/wsdl/style/iri" wsdlx:safe = "true">
    <input messageLabel="In" element="ghns:checkAvailability" />
    <output messageLabel="Out" element="ghns:checkAvailabilityResponse" />
    <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
</operation> </interface> . . . </description>
```


Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description … > . . . <types> . . . </types>
<interface name = "reservationInterface" > ... </interface>
<binding name="reservationSOAPBinding" interface="tns:reservationInterface"
    type="http://www.w3.org/ns/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
    <operation ref="tns:opCheckAvailability"
    wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
    <fault ref="tns:invalidDataFault" wsoap:code="soap:Sender"/>
</binding>
. . .
</description>
```


Voorbeeld WSDL 2.0

```
<?xml version="1.0" encoding="utf-8" ?>
<description …>
. . . <types> . . . </types>
<interface name = "reservationInterface" > . . . </interface>
<binding name="reservationSOAPBinding" interface="tns:reservationInterface" . . . >
. . . </binding>
<service name="reservationService" interface="tns:reservationInterface">
    <endpoint name="reservationEndpoint" binding="tns:reservationSOAPBinding"
    address ="http://greath.example.com/2004/reservation"/>
</service>
</description>
```


Webservices: technologieën

- SOAP
  - Communicatie tussen broker, service provider en service requester
  - SOAP-corpus bevat UDDI

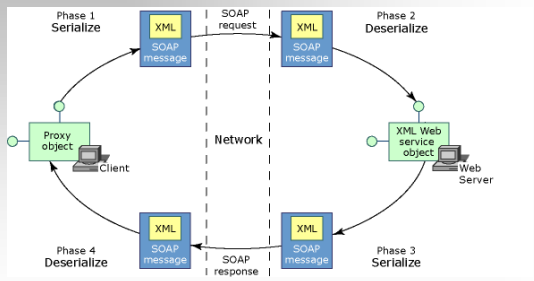
Overzicht

- Gegevens uitwisselen
- SOAP
- Webservices
  - Technologieën
  - Gebruik

Wat is een webservice?

- Uitvoeren van een methode op een andere server
- Een dienst
  - Eén of meerdere methodes
  - Een stuk programma met een bepaalde functionaliteit
  - Toegankelijk via XML (SOAP) en HTTP

Overzicht



Versturen SOAP-aanvraag

- Proxy-object: een lokaal object voor de clientapplicatie die webservice “voorstelt”
- Client-applicatie roept methodes van het proxy-object op
- Infrastructuur serialiseert → SOAP-bericht
- Bericht wordt verstuurd via het netwerk
  - Communicatie via HTTP en het SOAP-protocol

Verwerken SOAP-bericht

- Serverinfrastructuur
  - Deserialisatie van het SOAP-bericht
  - Al dan niet aanmaken object webservice
  - Methode webservice uitvoeren
  - Antwoord serialiseren → SOAP-bericht
  - SOAP-antwoord wordt verstuurd via het netwerk

SOAP-antwoord vertalen

- Clientinfrastructuur
  - Ontvangt SOAP-antwoord
  - Deserialiseert
  - Geeft resultaat door aan proxy-object
- Proxy-object genereert resultaat voor clientapplicatie

Voorbeeld

- Boek-object opvragen aan de hand van ISBN/ID
- In .NET
  - http://localhost/WebCatalogus/BoekenLijst.asmx
- In J2EE
  - http://localhost:8084/WebCatalogus/CatalogusService

SOAP

- Structureert HTTP-corpus
- Envelop-tag
  - Header-tag(s)
  - Body-tag
    - Bevat XML
    - Aan te roepen methode + argumenten
    - Resultaat van een methode

SOAP-aanvraag

```
POST /iit/webservices/Catalogus.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: ...
SOAPAction: "http://be.hogent.iii/webservices/GetBoek"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="..." xmlns:xsd="..."
  xmlns:soap="...">
    <soap:Body>
      <GetBoek xmlns="...">
        <ISBN>...</ISBN>
      </GetBoek>
    </soap:Body>
</soap:Envelope>
```


SOAP-antwoord

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: ...

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="..." xmlns:xsd="..."
  xmlns:soap="..."> <soap:Body>
    <GetBoekResponse xmlns="...">
      <GetBoekResult>
        <Isbn>...</Isbn>
        <Titel>...</Titel>
        <Prijs>...</Prijs>
      </GetBoekResult>
    </GetBoekResponse>
</soap:Body> </soap:Envelope>
```


Informatie voor proxy-object

- De structuur van de SOAP-berichten
  - Methodes
  - Argumenten
  - Teruggeefwaarden
- Locatie webservices
- Transportprotocol
- WSDL: Webservice Description Language

Structuur WSDL

- Types: Type-declaraties van argumenten, methodes, teruggeefwaardes, …
- Message: Beschrijving bericht (aanvraag/antwoord)
- PortType: Beschrijving mogelijke opdracht (naam, input, output)
- Binding: Verbinden portType met een locatie en een protocol (SOAP, HTTP GET, …)
- Service: Uit welke portTypes bestaat de webservice

Voorbeeld WSDL

- .NET
  - http://localhost/WebCatalogus/BoekenLijst.asmx?WSDL
- J2EE
  - http://localhost:8080/WebCatalogus/catalogusservice?WSDL

Werkwijze

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken
  - Webservice maken
  - WSDL genereren
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Voorbeeld in .NET

Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Eigenlijke functionaliteit

- Project Catalogus
  - Klasse Boek
    - Resultaat methode
    - Moet geserialiseerd worden
      - Default-constructor
      - set- en get-declaraties voor eigenschappen → anders NIET in XML
  - Klasse BoekenMagazijn
    - Indexer
      - Boek op basis van isbnnummer

Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Webservice

- Klasse afgeleid van System.Web.Services.WebService
- Klasse en methode
  - Extra attribuut
    - WebService
    - WebMethod
    - Tussen [en] voor klassedefinitie of methodedeclaratie
- WebCatalogus.BoekenLijst.cs

Webservice

- asmx-bestand
- Bestaat uit één richtlijn nl.

```
<%@ WebService Language="c#"
   Codebehind="BoekenLijst.asmx.cs"
   Class="WebCatalogus.BoekenLijst" %>
```

Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Proxy-klassen genereren

Wsdl /namespace:Be.Hogent.III.ClientWebservice http://localhost/WebCatalogus/BoekenLijst.asmx

- Wsdl.exe → BoekenLijst.cs
- BoekenLijst.cs
  - BoekenLijst -klasse
    - GetBoek-methode
  - Boek-klasse

Proxy-klassen genereren

- Alternatief: menu “Add Service References”
  - url webservice opgeven
- ClientWebCatalogus\Service References\ServiceWebCatalogus\Reference.cs
- Klassen bekijken in objectbrowser

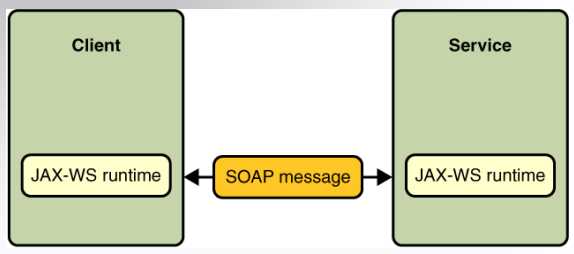
Werkwijze in .NET

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (Catalogus.dll)
  - Webservice maken (.asmx-bestand + codebehind)
  - WSDL genereren (automatisch)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (Wsdl.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Clientapplicatie

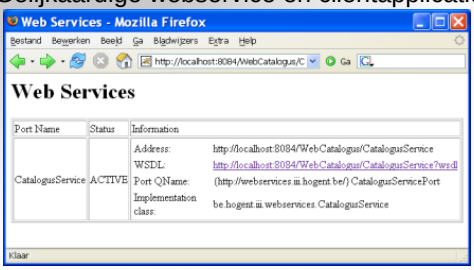
- Boeken met ID/ISBN van 1 tot 5 uitschrijven
- TestGetBoek.cs
- In objectbrowser beschikbare klassen bekijken

Webservices in J2EE



Webservices in J2EE

- Gelijkaardige webservice en clientapplicatie



Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Eigenlijke functionaliteit

- be.hogent.iii.catalogus.Boek
  - Default-constructor
  - get/set-methodes voor id, titel en prijs
- be.hogent.iii.catalogus.BoekenLijst
  - Interface
    - Methode geefBoek
- be.hogent.iii.catalogus.impl.BoekenLijstImpl
  - Implementatie interface BoekenLijst
- In jar catalogus.jar

Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Webservice

- Klasse met annotatie @Webservice()
- Methode met annotatie @WebMethod
  - Parameters met annotatie @WebParam(…)

```
@WebService()
public class CatalogusService {
   @WebMethod
   public Boek geefBoek(@WebParam(name = "isbn") String isbn) {
      try {
         BoekenLijst boeken = new BoekenLijstImpl();
         return boeken.geefBoek(isbn);
      } catch (Exception e) {
         return null;
      } } }
```


Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

WSDL genereren

- Automatisch bij build project in Netbeans
  - CatalogusServiceService.wsdl
  - CatalogusServiceService_schema1.xsd
    - beschrijft types
  - Wsdl af te halen via http://localhost:8084/WebCatalogus/CatalogusService
  - wsgen: tool om dit handmatig te doen

Webservice deployen

- War-bestand
  - Aantal extra XML-bestanden o.a.
    - sun-jaxws.xml
    - wsdl
    - web.xml bevat gegenereerd info
  - Doel
    - wrappen in een servlet
- Publiceren webservice

Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Genereren proxy-klassen

- Automatisch in Netbeans in een webproject
  - New → Web Service Client
    - WSDL doorgeven
- Klassen terug te vinden in
  - build\generated\
- In Projectsvenster
  - Map Web Service References
    - CatalogusServiceService

Werkwijze in J2EE

- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - WSDL genereren (wsgen.exe)
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Clientapplicatie

- Project GebruikWebservice
  - gebruikwebservice.Main.java
- Oproepen methode webservice
  - Rechts klikken in editeervenster
    - Web Service Client Resource
      - Call Web Service Operation
    - Genereert de nodige code om de proxyobjecten te gebruiken

Informatie over webservices

- W3C SOAP Tutorial (http://www.w3.org/TR/2007/REC-soap12-part0-20070427/)
- W3C SOAP 1.2 (http://www.w3.org/TR/soap12-part1/)
- UDDI (http://uddi.org/pubs/uddi_v3.htm)

Informatie over webservices

- XML Web Services Created Using ASP.NET and XML Web Service Clients

(http://msdn2.microsoft.com/en-us/library/7bkzywba.aspx)

- The JavaEE 5 Tutorial (http://java.sun.com/javaee/5/docs/tutorial/doc/)
- Java™ 2 Platform Enterprise Edition, v 5.0 API Specification (http://java.sun.com/javaee/5/docs/api/)
- Webservices in Netbeans (http://www.netbeans.org/kb/trails/web.html)

Hoofdstuk 17

Java Servlets

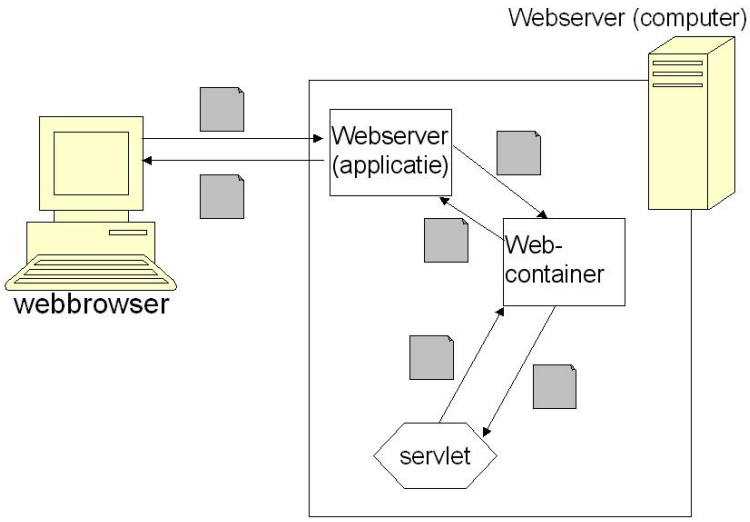
Hoewel in de geschiedenis van het WWW aanvankelijk dynamische webinhoud werd gerealiseerd met behulp van CGI-scripts (zie §14.2.1), bleek deze techniek al snel een aantal belangrijke gebreken te hebben (zoals reeds eerder aangehaald). Ze zijn onder andere platform-afhankelijk en moeilijk schaalbaar. Dit wil zeggen dat het niet eenvoudig is om met CGI-scripts een website te creëren die een groot aantal bezoekers kan verwerken.

De Java Servlettechnologie biedt een oplossing voor die gebreken. Een *servlet* is een Java-klasse waarmee de functionaliteit van een vraag/antwoord-georiënteerde server kan worden uitgebreid. Servlets worden in de praktijk meestal gebruikt voor webapplicaties. In dat geval is de server een webserver en spreken we van HTTP-servlets. In de rest van dit hoofdstuk zullen we uitsluitend HTTP-servlets behandelen.

De informatie in dit hoofdstuk en de volgende hoofdstukken over servlets en JSP's is grotendeels gebaseerd op het boek "Head First Servlets & JSP" ([BSB04]) en de webtutorial "The Java EE 5 Tutorial"([JBC+07]).

17.1 Webcontainer

Om de functionaliteit van een webserver te kunnen vergroten met behulp van servlets, moet de webserver uitgebreid worden met een *webcontainer* (in deze context ook *servletcontainer* genoemd). Een webcontainer is een runtime-omgeving die onder andere een implementatie van de servlet API voorziet. Deze webcontainer is verantwoordelijk voor het beheer van de servlets die hem zijn toevertrouwd: het inladen, initialiseren, oproepen en dergelijke meer. Een aantal webcontainers (zoals bijvoorbeeld Tomcat) luistert zelf naar de HTTP-poort. In een dergelijk geval heb je geen afzonderlijke webserver nodig.



Figuur 17.1: Webcontainer

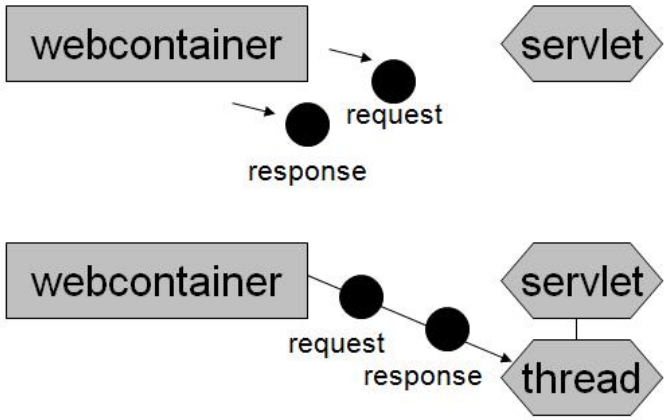
17.1.1 Een HTTP-aanvraag afhandelen

Het proces om een HTTP-aanvraag door een servlet te laten afhandelen kan als volgt verlopen. Merk op dat de eerste twee stappen overbodig zijn indien de webcontainer zelf naar de HTTP-poort luistert.

- De webserver stelt vast dat de HTTP-aanvraag correspondeert met een applicatie in de webcontainer.
- De webserver stuurt de aanvraag door naar de webcontainer. (zie figuur 17.2)

Figuur 17.2: HTTP-aanvraag wordt door gestuurd naar webcontainer

- De webcontainer gaat na of de aanvraag afgehandeld moet worden door een servlet of een statische pagina.
- Indien de aanvraag door een servlet afgewerkt wordt, zoekt de container een beschikbare instantie van de servletklasse of maakt er één aan.
- De container delegeert de aanvraag naar het servletobject. Hij maakt de objecten aan die de HTTP-aanvraag en het HTTP-antwoord voorstellen. Vervolgens start hij een thread waarin de *service*-methode van het servlet-object wordt uitgevoerd. (zie figuur 17.3)



Figuur 17.3: Webcontainer stuurt aanvraag door naar servlet

- Afhankelijk van de aanvraagmethode van de HTTP-aanvraag (bv. GET of POST), roept de *service*-methode de corresponderende methode van het servlet-object (bv. *doGet*, *doPost*, ...) op. Deze methode verwerkt de aanvraag en schrijft het antwoord op een *PrintWriter* of *ServletOutputStream* van het antwoordobject. (zie figuur 17.4)

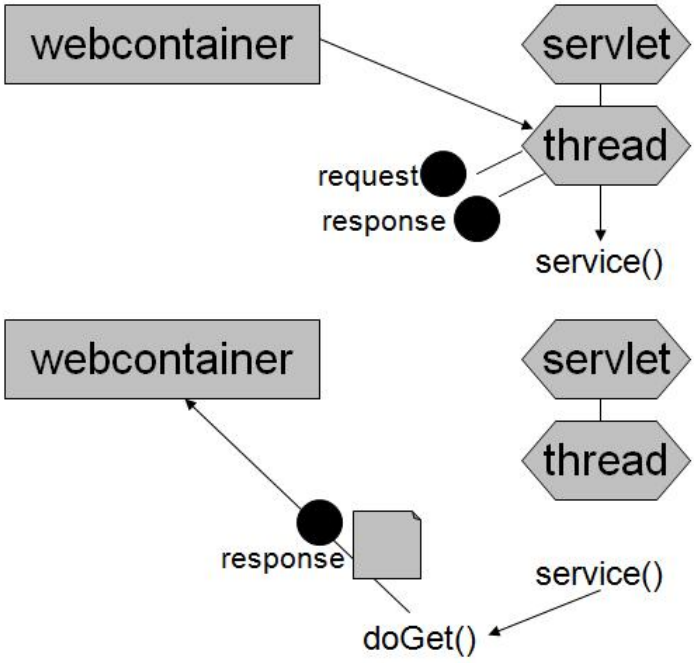
- Als de thread met de *service*-methode afgelopen is wordt het corresponderende HTTP-antwoord wordt doorgestuurd naar de client, eventueel via bemiddeling van de webserver. Het aanvraag- en antwoordobject worden vernietigd. (zie figuur 17.5)

17.1.2 De taken van een webcontainer

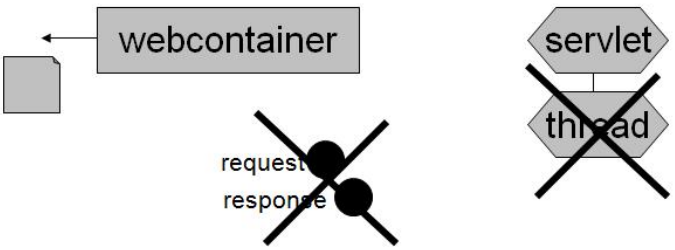
De webcontainer beheert servlets en roept methodes van de servlet-objecten op. Functionaliteit die voor elke webapplicatie hetzelfde is zoals bv. het opzetten van netwerkverbindingen, het aanmaken van sockets, het luisteren naar bepaalde poorten, ... wordt voorzien door de webcontainer. Zo moet bij het ontwikkelen van een webapplicatie enkel de specifieke functionaliteit voor een bepaalde applicatie uitgewerkt worden. De belangrijkste taken van een webcontainer zijn de volgende.

Communicatie met de webserver De webcontainer regelt de communicatie tussen de webserver en de servlets. Het is niet nodig om sockets aan te maken, te luisteren naar bepaalde poorten, *streams* te creëren, ... Het protocol tussen de webserver en de container is gekend voor de container en hoeft niet geïmplementeerd te worden in de servlet. De servlet bevat enkel logica en functionaliteit van de specifieke webapplicatie bv. een online boekenwinkel.

Beheer van servletobjecten De webcontainer beheert de servletobjecten en bepaalt wanneer ze aangemaakt of verwijderd worden. Hij zorgt voor het laden van de servletklassen, het aanmaken en initialiseren van de servletobjecten, het oproepen van de juiste servlet-methodes en geeft servletinstanties vrij voor *garbage collection*.



Figuur 17.4: Servlet verwerkt aanvraag



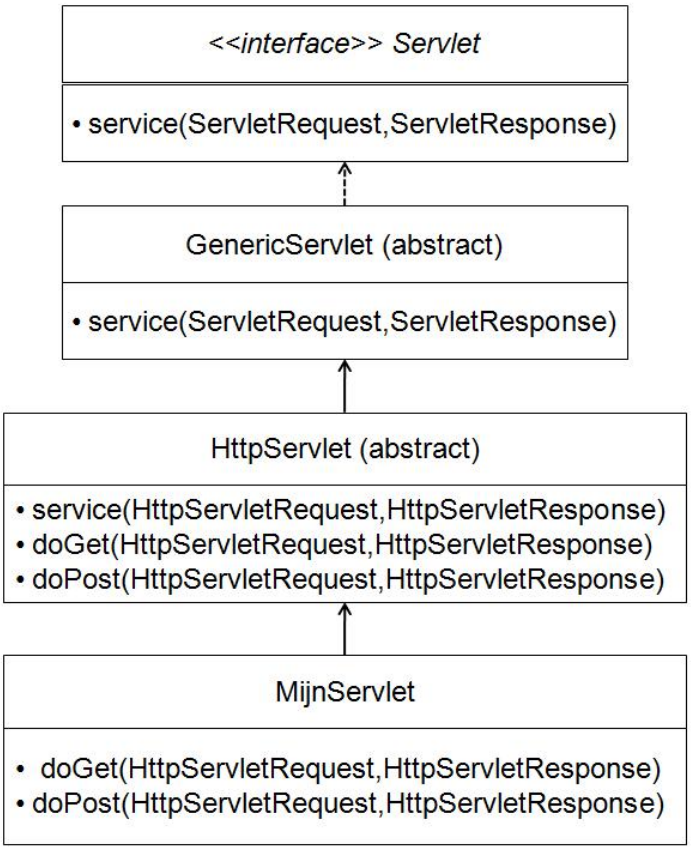
Figuur 17.5: Antwoord wordt door gestuurd naar client

Ondersteuning voor *multithreading* De webcontainer maakt automatisch een nieuwe Java *thread* aan voor elke aanvraag. Als de aanvraag afgehandeld is, wordt de *thread* afgesloten.

Beveiliging door configuratie De webcontainer voorziet beveiliging in de vorm van een XML-configuratiebestand. Het is dus niet nodig om beveiliging te programmeren in de servletcode.

17.2 Een eenvoudig servlet

Om een HTTP-servletklasse te schrijven, maak je een afgeleide klasse van de abstracte klasse *HttpServlet*, zelf een afgeleide klasse van de abstracte klasse *GenericServlet* die op haar beurt de *Servlet*-interface implementeert. (zie figuur 17.6)



Figuur 17.6: Servlet API

In deze klasse overschrijf je een aantal methoden die de verschillende types HTTP-aanvragen verwerken. In onderstaand voorbeeld wordt de methode *doGet* overschreven. Deze methode handelt een HTTP-aanvraag af met aanvraagmethode GET en maakt het HTTP-antwoord aan. Moet het servlet de aanvraagmethode POST kunnen verwerken, dan wordt de methode *doPost* overschreven.

In dit eenvoudige voorbeeld bestaat het antwoord uit een HTML-pagina waarop enkel de tekst `Hallo!` is te lezen. De titel van het venster is eveneens `Hallo!`. (zie figuur 17.7)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HalloServlet1
    extends HttpServlet
{
    public void doGet (HttpServletRequest request,
                HttpServletResponse response)
    throws ServletException, IOException
    {
        response.setContentType ("text/html");
        PrintWriter out = response.getWriter();

        out.println ("<html>");
        out.println ("<head><title>Hallo!</title></head>");
        out.println ("<body>");
        out.println ("Hallo!");
        out.println ("</body></html>");

        out.close();
    }
}
```

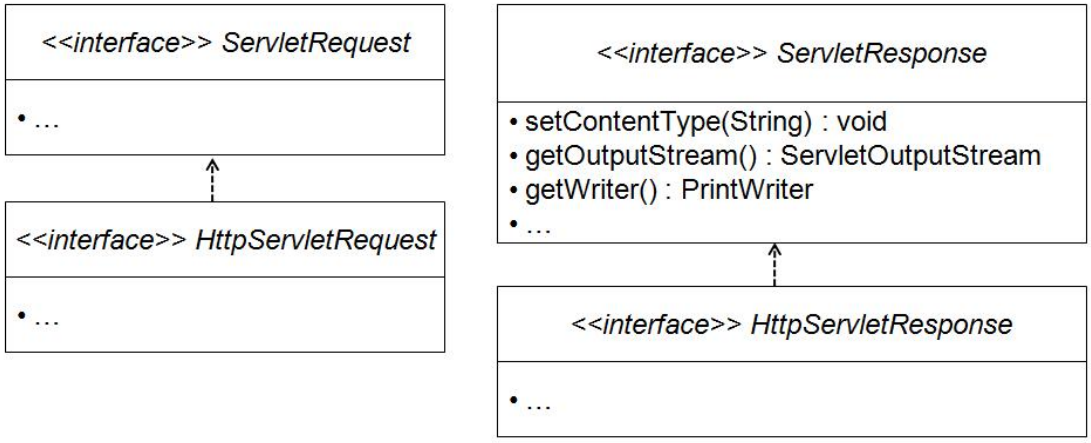
Figuur 17.7: HalloServlet

De pakketten *javax.servlet* en *javax.servlet.http* voorzien interfaces en klassen om servlets te ontwerpen. Vandaar dat deze twee pakketten geïmporteerd werden bij het schrijven van de klasse *HalloServlet1*. Elk servlet implementeert de *Servlet*-interface uit het pakket *javax.servlet*. Deze interface definieert alle methoden die een servlet moet implementeren. Die specifiëren het contract tussen de webcontainer en de servletontwikkelaar. Merk op dat een servlet geen *main*-methode heeft. De webcontainer roept de *service*-methode van het servletobject op, die op haar beurt de juiste methode bepaald door het type HTTP-aanvraag (bv. *doGet*) activeert. De methoden die de levensloop van een servlet bepalen (o.a. de *service*-methode) maken deel uit van de *Servlet*-interface. Meer uitleg over de levensloop van een servlet vind je in §17.6.

17.2.1 Aanvraag- en antwoordobject

De HTTP-aanvraag wordt voorgesteld door een *HttpServletRequest*-object en het HTTP-antwoord door een *HttpServletResponse*-object. De referenties naar deze twee objecten worden doorgegeven als argumenten van de methode die opgeroepen wordt om de HTTP-aanvraag af te handelen. In het voorbeeld is dat de methode *doGet*. De interfaces *HttpServletRequest* en *HttpServletResponse* zijn

afgeleide interfaces van respectievelijk de interfaces *ServletRequest* en *ServletResponse* (zie figuur 17.8).



Figuur 17.8: Aanvraag- en antwoordinterfaces

In het voorbeeld gebruiken we de volgende methoden van de interface *ServletResponse* :

```
public void setContentType (String type)
```

Met deze methode specifieer je het MIME-type van het HTTP-antwoord.

```
public PrintWriter getWriter ()
```

Hiermee verkrijg je een *PrintWriter*-object dat gebruikt wordt om het antwoord naar de aanvrager te sturen. Met behulp van deze *PrintWriter* kan men de HTML-code terugsturen.

Aangezien de methode *setContentType* de codering van de *stream* bepaalt, moet ze aangeroepen worden vóór de methode *getWriter*. Als het HTTP-antwoord uit binaire data bestaat (bv. een figuur), wordt er gebruik gemaakt van de methode *getOutputStream*. Deze methode geeft een *ServletOutputStream*, een afgeleide klasse van *OutputStream*, weer. Het afsluiten van het *PrintWriter*-object of het *ServletOutputStream*-object met de methode *close* zorgt ervoor dat de *stream* gesloten wordt en dat de webserver weet dat het HTTP-antwoord beëindigd is. De methoden *getWriter* en *getOutputStream* kunnen niet samen gebruikt worden.

17.3 HTTP-aanvragen met parameters

Zoals reeds eerder besproken (zie §13.2) kan de client in de HTTP-aanvraag informatie doorsturen naar de server in de vorm van naam/waarde-paren, parameters genoemd. Een servlet kan deze informatie opvragen met behulp van de volgende methoden van *ServletRequest*.

```
public String getParameter (String naam)
```

Geeft de waarde terug van het naam/waarde-paar (parameter) met de gegeven *naam*. Als er geen parameter bestaat met deze naam, dan is de waarde van deze methode **null**. Indien een parameter meerdere waarden heeft, geeft deze methode enkel de eerste waarde weer.

```
<form action="MijnServletUrl" method="POST">
    Kies Categorie:
    <select name="categorie">
        <option value="SPORT">Sport</option>
        <option value="MUZ">Muziek</option>
        <option value="ACTUA">Actualiteit</option>
    </select>
    <br>
    <input type="submit" value="Toon Keuze">
</form>
```

Listing 17.1: Voorbeeld formulier met parameter

Het bovenstaand formulier bevat één parameter met naam *categorie*. Als de gebruiker de optie "Muziek" selecteert, dan bevat het corpus van de HTTP-aanvraag de volgende informatie.

```
categorie=MUZ
```

In het servlet wordt de waarde van de parameter *categorie* opgevraagd. De string *categorieParam* heeft in dit voorbeeld de waarde *MUZ*.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    String categorieParam = request.getParameter("categorie");
    ...
}
```

Listing 17.2: Ophalen parameter in servlet

17.3.1 Parameter met meerdere waarden

```
public String[] getParameterValues (String naam)
```

Geeft *alle* waarden terug die bij de gegeven *naam* horen als een tabel van strings (en **null** als de parameter niet bestaat).

```
<form action="MijnServletUrl" method="POST">
    Selecteer je hobby's:
    <br>
    <input type="checkbox" name="hobby" value="zwemmen">Zwemmen <br>
    <input type="checkbox" name="hobby" value="fietsen">Fietsen <br>
    <input type="checkbox" name="hobby" value="surfen">Surfen <br>
    <input type="checkbox" name="hobby" value="lezen">Boeken lezen <br>
    <input type="checkbox" name="hobby" value="duiken">Diepzeeduiken <br>
    <input type="checkbox" name="hobby" value="computerspel">Computerspelletjes spelen <br>
    <input type="submit" value="Toon Keuze">
</form>
```

In het bovenstaande voorbeeld kan de gebruiker meerdere hobby's selecteren. Als hij "Zwemmen", "Surfen" en "Computerspelletjes spelen" kiest, dan ziet het corpus van de HTTP-aanvraag er als volgt uit.

```
hobby=zwemmen&hobby=surfen&hobby=computerspel
```

In het servlet die het bovenstaande formulier verwerkt, bevat de tabel *hobbysPersoon* nu de strings {*zwemmen,surfen,computerspel*}.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    String[] hobbysPersoon = request.getParameterValues("hobby");
    ...
}
```

Listing 17.3: Ophalen parameter met meerdere waarden

17.3.2 Opvragen parameternamen

```
public Enumeration getParameterNames ()
```

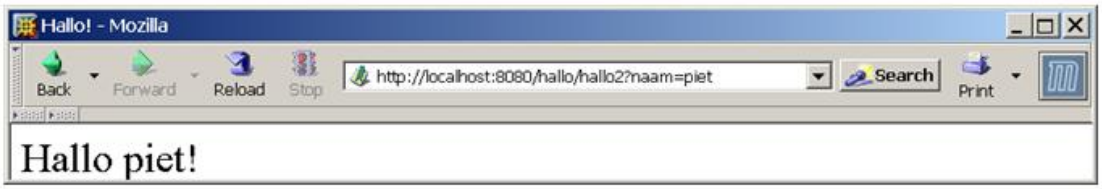
Met deze methode kan je de namen verkrijgen van alle opgegeven parameters. De waarde van deze methode is een *Enumeration*-object bestaande uit *String*-objecten.

17.3.3 Voorbeeld

Het volgend voorbeeld is een HTTP-servlet dat zowel een POST- als een GET-aanvraag kan verwerken en één parameter verwacht. De naam van de parameter is `naam`. Het HTTP-antwoord bevat een webpagina waarop het woord `Hallo` staat en de waarde van de parameter `naam` als er één is. (zie figuur 17.3.3)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HalloServlet2 extends HttpServlet
{
    private final static String PAR_NAAM = "naam";

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType ("text/html");
        PrintWriter out = response.getWriter();
                String naam = request.getParameter (PAR_NAAM);

        out.println ("<html>");
        out.println ("<head><title>Hallo!</title></head>");
        out.println ("<body>");
                out.print ("Hallo");
                if (naam != null) {
                out.print (" ");
                out.print (naam);
                }
                out.println ("!");
                out.println ("</body></html>");

        out.close ();
```

```
    }

    public void doPost (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
                doGet (request, response);
    }
}
```



17.4 De Deployment Descriptor

De webserver stuurt een aanvraag voor een servlet door naar de webcontainer. Die dan op zijn beurt de aanvraag doorspeelt naar de juiste servlet (zie §17.1). Om dit te realiseren moet de webcontainer de URL van de aanvraag analyseren. De URL bevat zowel de *context root* als een alias voor de servlet. Als er meerdere webapplicaties draaien in dezelfde webcontainer dan bepaalt de *context root* voor welke applicatie de aanvraag bedoeld is.

```
http://host:port/context-root/alias
```

In het voorbeeld uit de paragraaf 17.2 is de URL de volgende (zie figuur 17.7).

```
http://localhost:8080/hallo/hallo1
```

De webpagina wordt hier aangemaakt door een servlet met alias */hallo1*. De servlet behoort tot de webapplicatie die gekenmerkt wordt door de context */hallo*. De alias identificeert de servlet die de aanvraag moet afhandelen en begint steeds een schuine streep (/). Uit het voorbeeld blijkt dat de alias (*/hallo1*) kan verschillen van de klassenaam van de servlet (*HalloServlet1*).

De link tussen de alias en de klassenaam van de servlet wordt gelegd in het configuratiebestand van de webapplicatie. Dit bestand heeft de naam *web.xml* en wordt de *Deployment Descriptor* genoemd. De *Deployment Descriptor* is een xml-bestand dat alle configuratie voor één webapplicatie bevat. In dit bestand worden de drie namen van een servlet vastgelegd.

De alias is een publieke naam voor de servlet die een stuk is van de URL waarmee de servlet opgeroepen wordt. */hallo1* in het voorbeeld.

De interne naam is een geheime naam van de servlet die enkel gekend is voor de *deployer*, de persoon die de webapplicatie opzet in een echte operationele omgeving. De interne naam kan gelijk zijn aan de alias of de klassenaam, maar dat hoeft niet.

De volledige klassenaam is de combinatie van het *package* en de klassenaam van de servlet. Indien *HalloServlet1* tot de package *be.hogent.iii.servlets* behoort, dan is de volledige klassenaam *be.hogent.iii.servlets.HalloServlet1*.

Het gebruik van een alias voor een servlet heeft twee voordelen. Bij een reorganisatie van de applicatiecode waarbij de volledige klassenaam van het servlet verandert, hoef je de verschillende URL's in de applicatie die verwijzen naar het servlet niet aan te passen. Bovendien scherm je de interne structuur van je applicatie af van de buitenwereld. Je maakt het moeilijker voor de gebruiker om URL's te raden en in te tikken in de browser.

Het bestand web.xml ziet er als volgt uit.

```
<web-app>
    <servlet>
        <servlet-name>HalloServlet1</servlet-name>
        <servlet-class>be.hogent.iii.servlets.HalloServlet1</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HalloServlet1</servlet-name>
        <url-pattern>/hallo1</url-pattern>
    </servlet-mapping>
</web-app>
```

Het basiselement van de *Deployment Descriptor* is `<web-app>`. Het element `<servlet>` koppelt de volledige klassenaam aan de interne naam van de servlet, m.a.w. een servlet wordt gekenmerkt door een naam en een klasse. Voor elk servlet van de webapplicatie wordt een `<servlet>`-element aangemaakt. De link tussen de interne naam en de alias, een stuk van de publieke URL, wordt bepaald door het element `<servlet-mapping>`.

De Deployment Descriptor wordt niet alleen gebruikt voor het koppelen van servlets aan URL's. Ook andere aspecten van je webapplicatie wordt geconfigureerd in de Deployment Descriptor: beveiliging, foutpagina's, configuratieparameters, ...

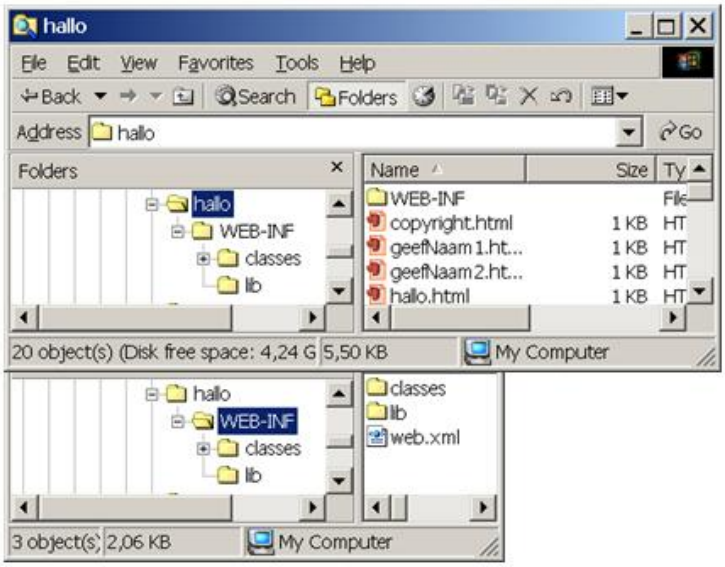
17.5 De structuur van een webapplicatie

Een webapplicatie die je wil *deployen* in een webcontainer moet een welbepaalde vastgelegde structuur hebben. (zie figuur 17.9) De basismap van een webapplicatie bevat JSP-pagina's, statische HTML-bestanden, figuren, ... en een submap *WEB-INF* met de volgende structuur.

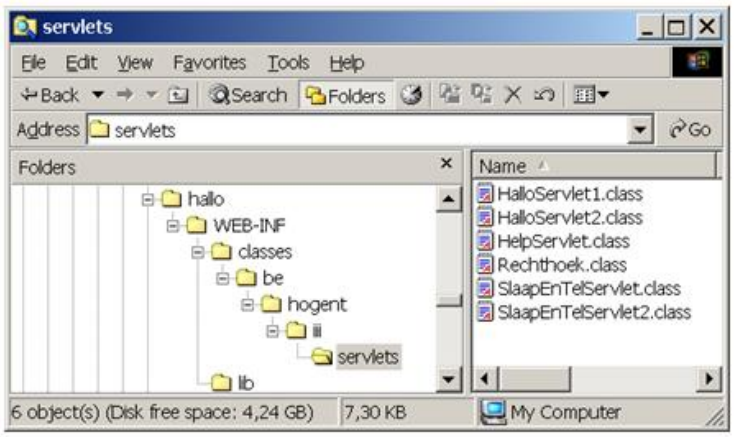
Het bestand web.xml : de deployment descriptor van de webapplicatie

De map classes bevat alle Java-klassen van de webapplicatie: servlets, ondersteunende klassen, JavaBeans, ... (bv. zie figuur 17.10)

De map lib bevat JAR-bestanden van bibliotheken die gebruikt worden door de klassen in de webapplicatie.



Figuur 17.9: De structuur van de webapplicatie



Figuur 17.10: Inhoud van de map classes

Een webapplicatie kan op twee manieren gepubliceerd worden in een webcontainer: als map of als JAR-bestand. In het eerste geval wordt de basismap van de webapplicatie naar de juiste plaats op de server gekopieerd. In het tweede geval wordt de basismap gecomprimeerd tot een JAR-bestand met extensie WAR (web archive). De extensie WAR wordt gebruikt i.p.v. JAR omdat het bestand een specifieke structuur heeft.

17.6 Levensloop van een servlet

De levensloop van een servlet wordt geregeld door de webcontainer en bestaat uit verschillende fasen die we hieronder uitgebreid behandelen. Als de webcontainer een HTTP-aanvraag krijgt voor een servlet, dan voert hij de volgende stappen uit.

1. Als er geen servletobject bestaat, dan
   (a) laat de webcontainer de servletklasse,
   (b) maakt een servletobject aan
   (c) en roept de *init*-methode aan om het servletobject te initialiseren
2. Hij voert de *service*-methode van het servletobject uit waarbij hij het aanvraag- en antwoordobject als argumenten meegeeft.

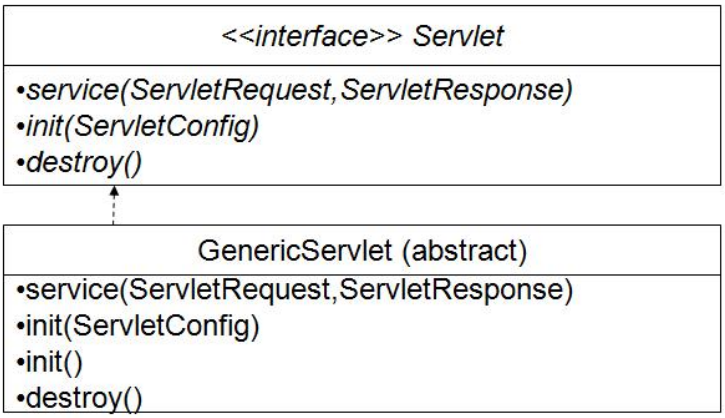
Indien de webcontainer het servletobject moet verwijderen, dan roept hij de methode *destroy* aan.

17.6.1 Initialiseren van het servlet

Het is mogelijk om de standaardinitialisatie van een servlet uit te breiden en zo te voldoen aan specifieke noden van een applicatie. Zo kan je bijvoorbeeld configuratiegegevens inlezen, gegevensbanken of log-bestanden registeren, ... Elke actie die slechts uitgevoerd dient te worden tijdens de opstartfase van het servletobject kan opgenomen worden in de initialisatiefase. Om de initialisatie van een servlet aan te passen moet je de *init*-methode van de abstracte klasse *GenericServlet* overschrijven (zie figuur 17.6.1). De initialisatiefaze maakt geen deel uit van de constructor van het servletobject omdat tijdens de constructie nog niet alle informatie, zoals bv. configuratiegegevens, aanwezig is.

Het volgend voorbeeld is een servlet dat een webpagina toont waarop een verwijzing naar een pagina met meer informatie en een emailadres staat. Het pagina- en het emailadres worden niet opgenomen in de broncode van de servletklasse, maar in het configuratiebestand web.xml van de webcontainer dat wordt geraadpleegd via een *ServletConfig*-object. Dit object wordt aangesproken met behulp van de methode *getInitParameter*.

```
public class HelpServlet extends HttpServlet
{
    private String helppagina;
    private String helpemail;

    public void init() throws ServletException
```



Figuur 17.11: API levensloop methodes servlet

```
    {
            helppagina = getInitParameter ("helppagina");
            helpemail  = getInitParameter ("helpemail");
    }

    public void doGet (HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
            response.setContentType ("text/html");

            PrintWriter out = response.getWriter ();

            out.println ("<html>");
            out.println ("<head><title>Help!</title></head>");
            out.println ("<body>");
            out.println ("Hallo!<br>");
            out.println ("Welkom op onze informatiepagina.");
            out.print ("Voor meer hulp klik <a href=\"");
            out.print (helppagina);
            out.println ("\">hier</a> of stuur een mail naar ");
            out.print ("<a href=\"mailto:");
            out.print (helpemail);
            out.println ("\">onze helpdesk</a>.");
            out.println ("</body></html>");

            out.close ();
    }
}
```

De parameters *helppagina* en *helpemail* worden geconfigureerd in het bestand web.xml.

```
<servlet>
    <servlet-name>info</servlet-name>
    <servlet-class> be.hogent.iii.servlets.HelpServlet</servlet-class>
    <init-param>
        <param-name>helppagina</param-name>
        <param-value>index.html</param-value>
    </init-param>
    <init-param>
        <param-name>helpemail</param-name>
        <param-value>help@info.be</param-value>
    </init-param>
```

```
</servlet>
```

Om initialisatieparameters te gebruiken die specifiek zijn voor één welbepaalde servlet, gebruik je een configuratie-object van het type *ServletConfig*. Als de webcontainer een servletobject aanmaakt, leest hij de deployment descriptor en maakt de parameters aan en voegt ze toe aan het *ServletConfig*-object. Deze parameters worden dus slechts éénmaal gelezen. *ServletConfig* is een interface met de volgende methodes.

```
public String getInitParameter(String name);
public Enumeration getInitParameterNames();
public ServletContext getServletContext();
public String getServletName();
```

De methode *getInitParameter* geeft de waarde van de initialisatieparameter met naam *name* terug of *null* als de parameter niet bestaat. Aangezien de interface *ServletConfig* ook door *GenericServlet* wordt geïmplementeerd, moet je nooit expliciet een configuratie-object op te halen. Je kan dus rechtstreeks de methode *getInitParameter* oproepen zoals in het voorbeeld en zo de initialisatieparameters opvragen.

Om de namen van alle initialisatieparameters van een servlet te kennen, gebruik je de methode *getInitParameterNames*.

Met de methode *getServletContext* krijg je een verwijzing naar het *ServletContext*-object (zie §17.8). De methode *getServletName* geeft de interne naam van de servlet weer.

Het *ServletConfig*-object ophalen doe je met de methode *getServletConfig* van *GenericServlet*.

Indien een servlet het initialisatieproces niet correct kan afwerken en bijgevolg niet in staat is om aanvragen te behandelen, moet het een *UnavailableException* genereren. Zo'n fout kan bijvoorbeeld optreden omdat het servlet geen verbinding kan maken met een bepaalde gegevensbank, of omdat het geen logbestanden kan registreren, ... De container zal dan geen aanvragen meer doorgeven aan het servlet en de gebruiker melden dat het servlet niet beschikbaar is.

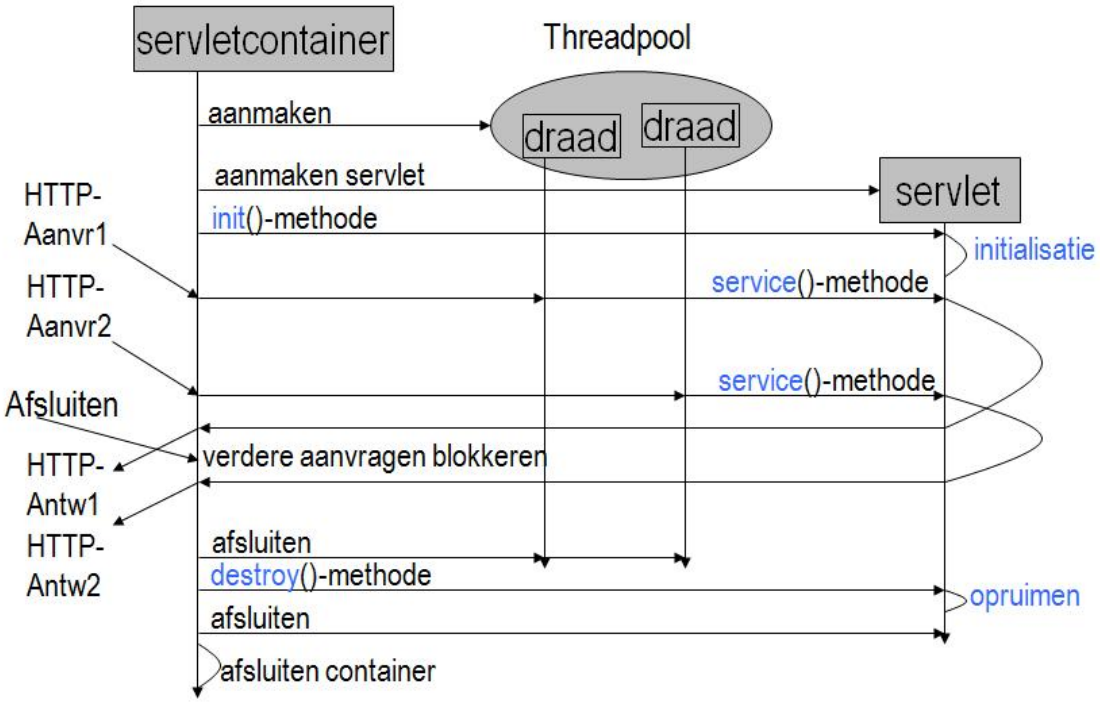
```
public class MyServlet extends HttpServlet
{   ...

    public void init() throws ServletException
    {
      try {
        ... // maak een verbinding met een gegevensbank
      } catch (SQLException e) {
        throw new UnavailableException ("Kan geen verbinding maken met de gegevensbank"
            + e.getMessage());
      }
    }
    ...
}
```

UnavailableException is een afgeleide klasse van *ServletException* die aangeeft dat een servlet tijdelijk of permanent niet beschikbaar is. Om andere fouten in een servlet af te handelen kan de bovenklasse *ServletException* gebruikt worden. Bij het optreden van een fout in een servlet mag je in *geen* geval de methode *System.exit* oproepen. Deze methode zou namelijk veroorzaken dat de servletcontainer zelf wordt afgesloten, samen met alle servlets die erin gehuisvest zijn.

17.6.2 Afhandelen van de aanvragen

Als het servlet geïnitialiseerd is dan is hij klaar om aanvragen af te handelen. Elke HTTP-aanvraag wordt toegekend aan een afzonderlijke draad (*thread*). In deze draad wordt de *service*-methode van het servlet uitgevoerd. De *service*-methode zorgt ervoor dat afhankelijk van de aanvraagmethode de aanvraag behandeld wordt door de methode *doGet* of *doPost* (en analoge methoden zoals *doPut* die we verder niet meer zullen vermelden). De *service*-methode is reeds geïmplementeerd in de klasse *HttpServlet* en het is niet aangewezen om deze methode te overschrijven. De code voor het aanmaken van webpagina's hoort thuis in de methoden *doGet* en *doPost*.

Het is mogelijk dat de *service*-methode van één servletobject tegelijkertijd in verschillende draden wordt uitgevoerd (zie figuur 17.6.2). Dit betekent dat de zelfgeschreven methoden *doGet* en *doPost*, *thread safe* moeten zijn. Er moet dus op een veilige manier worden omgegaan met attributen van de servletklasse en met uitwendige bronnen zoals bestanden. Ook moet men er rekening meehouden dat de webcontainer in sommige gevallen tegelijkertijd meerdere instanties van één zelfde servletklasse kan aanmaken.



Figuur 17.12: Levensloop van een servlet

Het servlet construeert het HTTP-antwoord in twee stappen: eerst stel je de HTTP-headers in en daarna stuur je de inhoud van het corpus door. Dit gebeurt via het argument *response* van de *doGet* of *doPost*.

Headers instellen

Je stelt de headers in met de volgende basismethode van *HttpServletResponse*:

```
public void setHeader (String name, String value);
```

Er bestaan ook nog een aantal varianten die je kan gebruiken voor headers waarvan de waarde een geheel getal is of een tijdstip.

```
public void setDateHeader (String name, long date);
public void setIntHeader  (String name, int value);
```

Een *doGet*-methode van een servlet begint dus misschien op de volgende manier:

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    response.setHeader ("Pragma", "no-cache");
    response.setHeader ("Cache-Control", "no-cache");
    response.setContentType ("text/html");
    ...
}
```

We hebben de methode *setContentType*, waarmee we hier het MIME-type van het bericht instellen, reeds eerder aangehaald. Je zou dit natuurlijk ook rechtstreeks met behulp van *setHeader* kunnen instellen.

Wanneer je een tweede keer *setHeader* oproept met dezelfde headernaam maar met een andere waarde, dan heeft de nieuwe waarde voorrang. Er bestaan echter ook methoden *addHeader*, *addDateHeader* en *addIntHeader* waarmee je tegelijkertijd verschillende waarden aan dezelfde naam kunt hechten. Tot slot vermelden we ook nog de methode *addCookie* waarmee je cookies aan het bericht kunt toevoegen.

Je kan de status van je bericht instellen met behulp van de methode *setStatus* met een numerieke statuscode als parameter. (De defaultstatus is '`200 OK`'.) De klasse *HttpServletResponse* bevat verschillende symbolische constanten (zoals *SC_OK* en *SC_SEE_OTHER*) die je daarbij kunt gebruiken. Gebruik *setStatus* echter enkel voor condities die *niet* op een fout wijzen, want voor dat geval is er een andere methode voorzien:

```
public void sendError (int sc, String msg) throws IOException;
```

De parameter *sc* is opnieuw de statuscode en *msg* is een corresponderend bericht dat in de foutmelding zal worden verwerkt.

Corpus aanmaken

Bij een gewoon bericht wil je na de headers ook een corpus zenden. Hiertoe moet je met behulp van *getWriter* eerst een *PrintWriter*-object van het antwoord ophalen. Dit object zorgt ervoor dat je naar de netwerkverbinding tussen client en webserver kan schrijven. Je gebruikt dus de verschillende

print-methoden van *PrintWriter* om het corpus van het HTTP-antwoord uit te schrijven. We hebben hiervan reeds verschillende voorbeelden gegeven.

In antwoord op een POST-bericht is het soms aangewezen de client als antwoord naar een andere URL door te verwijzen in plaats van zelf een corpus te genereren. Dit doe je met de volgende methode:

```
public void sendRedirect (String location) throws IOException;
```

De parameter *location* bevat het nieuwe adres. Dit mag ook een relatieve URL zijn. Een relatieve URL die met een schuine streep begint verwijst naar een andere webapplicatie in dezelfde webcontainer. Stel dat de gebruiker oorspronkelijk de volgende pagina intikte:

```
http://mijnserver.com/hallo/intern/welkom
```

Vervolgens roept de servlet met alias *welkom* de methode *sendRedirect* op met een relatieve URL die NIET start met een schuine streep:

```
response.sendRedirect("opdrachten/login");
```

Dan voegt de webcontainer de volgende volledige URL toe als *Location*-header van HTTP-antwoordbericht:

```
http://mijnserver.com/hallo/intern/opdrachten/login
```

Indien het antwoord WEL met een schuine streep begon:

```
response.sendRedirect("/opdrachten/login");
```

Dan wordt de waarde van de *Location*-header:

```
http://mijnserver.com/opdrachten/login
```

Deze URL verwijst naar een andere webapplicatie met context `/opdrachten` op dezelfde webserver.

Wanneer de methoden *doGet* en *doPost* niet overschreven worden dan geven ze standaard een `BAD_REQUEST` (400) fout weer in het geval van HTTP/1.0 en een `NOT_IMPLEMENTED` (501) fout bij HTTP/1.1.

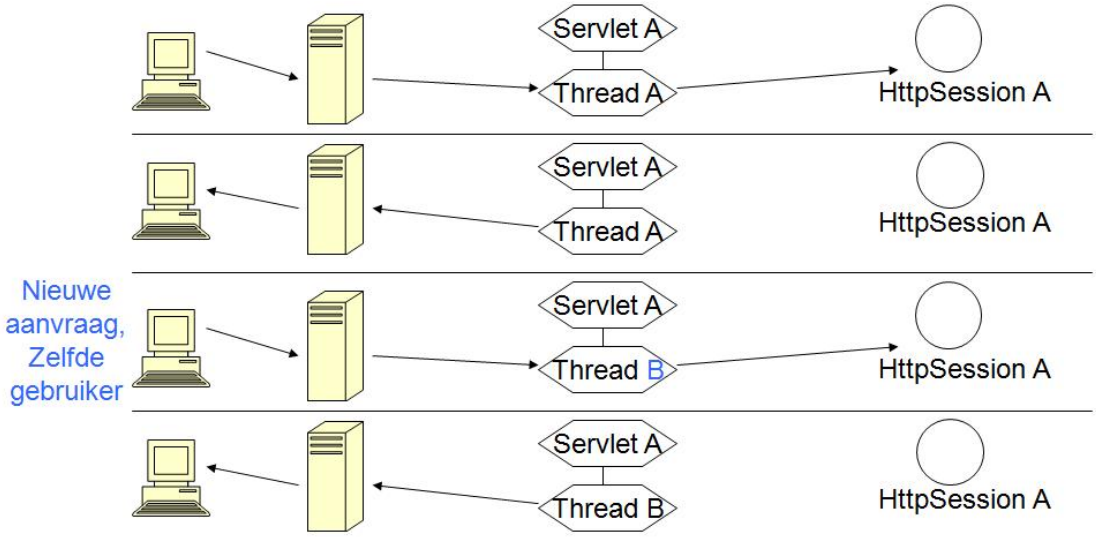
17.6.3 Opruimen van het servlet

De servletcontainer is niet verplicht om een servlet voor eeuwig in het geheugen te houden. Wanneer de container beslist dat een servlet moet worden opgeruimd, wacht hij eerst tot alle actieve draden zijn beëindigd die de *service*-methode van het servlet afhandelen (of snijdt ze desnoods zelf af, na een ruime wachttijd). Daarna roept hij de methode *destroy* van dit servlet op zodat het servlet al zijn bronnen kan vrijgeven (verbindingen met gegevensbanken, hulpdraden, en dergelijke). Je kan de methode *destroy* dus met dit doel overschrijven. De container aanvaardt geen aanvragen meer voor het servlet en verwijdert alle referenties naar het servlet-object zodat het klaar staat voor garbage collection.

17.7 Sessies

Het HTTP-protocol bevat geen toestandsinformatie over de aanvrager. Dit wil zeggen dat er geen verband is tussen de verschillende aanvragen van een zelfde gebruiker. De meeste webapplicaties willen de verschillende aanvragen van een zelfde klant echter wel met elkaar verbinden. Bijvoorbeeld een webapplicatie die een online-winkel voorstelt en uit verscheidene servlets bestaat, zal informatie (zoals reeds bestelde artikels en dergelijke) over een bepaalde klant willen gebruiken in alle servlets van de applicatie.

Aangezien het HTTP-protocol een toestandsloos protocol is zal deze informatie aan de serverkant bewaard moeten worden. De verschillende HTTP-aanvragen van een zelfde gebruiker worden een *sessie* genoemd, de informatie die bij een sessie hoort is de toestand van deze gebruiker (zie figuur 17.13).



Figuur 17.13: Sessie-objecten op server

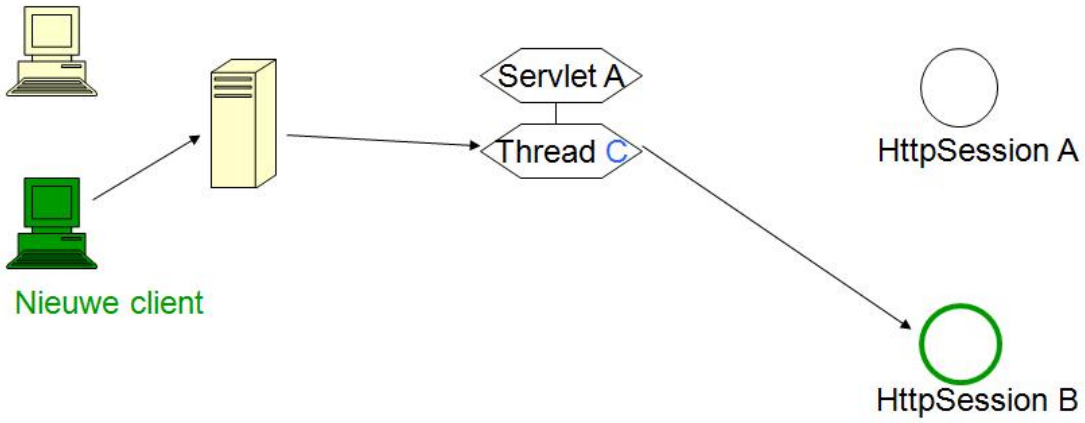
17.7.1 Een voorbeeld

Ter illustratie van het gebruik van sessies bespreken we een heel eenvoudige toepassing, geïnspireerd op het elektronische winkelwagentje dat veel gebruikt wordt bij online-winkels. We hebben ons hier beperkt tot een toepassing die bestaat uit slechts twee verschillende pagina's geïmplementeerd in de vorm van drie afzonderlijke servlets.

Het servlet *WinkelServlet* toont een lijst van goederen waaruit de klant er één kan selecteren en toevoegen aan zijn winkelwagentje. Bij het drukken op de submitknop wordt het servlet *NieuwProductServlet* geactiveerd. Dit servlet doet weinig meer dan het product daadwerkelijk aan het winkelwagentje toe te voegen en de browser doorverwijzen naar het derde servlet *WagenServlet* die de huidige in-

houd van het wagentje toont en een terugverwijzing naar de eerste pagina bevat. (We laten het aan de lezer om deze toepassing verder uit te breiden zodat de klant ook producten uit het wagentje kan verwijderen of het wagentje volledig leeg maken met één druk op de knop.)

Opdat de drie servlets zouden kunnen samenwerken moeten ze toegang hebben tot een aantal gemeenschappelijke gegevens: de inhoud van het winkelwagentje. Deze gegevens vormen de *toestand* die we als sessieinformatie zullen opslaan. Merk op dat twee verschillende klanten tegelijkertijd een winkelwagentje moeten kunnen opvullen, en dat dit niet hetzelfde winkelwagentje mag zijn (zie figuur 17.14). Door gebruik te maken van de sessiemogelijkheden van de webcontainer zijn we hiervan automatisch verzekerd.



Figuur 17.14: Sessie-objecten voor verschillende clients

We stellen de producten van onze winkel in onze applicatie voor als strings en het een winkelwagentje als een lijst van strings (een object van het type *List*). In een professionele toepassing zou je hiervoor wellicht aparte klassen ontwikkelen.

De implementatie van *WinkelServlet* is heel eenvoudig: hij moet er enkel voor zorgen dat de juiste HTML-code in de pagina terecht komt. De producten waaruit de klant kan kiezen, bevinden zich in een constante tabel *boeken* van strings. (Onze winkel is een boekenwinkel.)

```
public class WinkelServlet extends HttpServlet
{
    private static final String[] boeken    = { "Thinking in Java",
        "The Java Tutorial", "Java Servlet Programming", "Java Server Programming",
      "The Swing Tutorial" };

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
            PrintWriter out = response.getWriter();

            out.println ("<html>");
            out.println ("<head><title>Boekenwinkel</title></head>");
            out.println ("<body>");
            out.println ("<h1>Boekenwinkel</h1>");
```

```
            out.println ("Kies een boek:");
            out.println ("<form action=\"NieuwProductServlet\" method=\"POST\">");
            out.println ("<select name=\"item\">");
            for (int i = 0; i < boeken.length ; i++ )
            out.println ("  <option>" + boeken[i]);
            out.println ("</select>");
            out.println ("<input type=\"submit\" value=\"Toevoegen \">");
            out.println ("</form>");

            out.println("</body></html>");

            out.close();
    }
}
```

Op het eerste zicht zou je de *WinkelServlet* kunnen vervangen door een statische HTML-pagina. We zullen later echter een kleine wijziging moeten aanbrengen waardoor het noodzakelijk wordt dat deze HTML-code wel door een echt servlet wordt gegenereerd (zie §17.7.3).

17.7.2 De interface *HttpSession*

Het servlet *NieuwProductServlet* voegt het geselecteerde product (de waarde van de parameter `item`) toe aan het winkelwagentje uit de sessie. We drukken hieronder de *doPost*-methode af waarin dit gebeurt.

```
public void doPost (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    HttpSession session = request.getSession();
    List wagentje = (List)session.getAttribute ("winkelwagen");
    if (wagentje == null) {
    wagentje = new ArrayList ();
    session.setAttribute ("winkelwagen", wagentje);
    }
    String item = request.getParameter ("item");
    if (item != null)
    wagentje.add (item);
    response.sendRedirect ("WagenServlet");
}
```

De eerste opdracht in deze methode zorgt voor het ophalen van de HTTP-sessie. Hiervoor wordt de methode *getSession* van *HttpServletRequest* gebruikt. Deze methode haalt de sessie op geassocieerd met de huidige aanvraag. Is er geen sessie verbonden met de aanvraag dan wordt er een nieuwe aangemaakt. Er bestaat een tweede versie van deze methode waarbij je kan aangeven of je wenst dat een niet bestaande sessie automatisch wordt aangemaakt:

```
public HttpSession getSession (boolean aanmaken);
```

Deze methode haalt ook de sessie geassocieerd met de huidige aanvraag op. Is er geen actieve sessie dan bepaalt het argument of er nieuwe sessie (ingeval van **true**) of **null** wordt teruggegeven (indien **false**).

De methode *getSession* moet opgeroepen worden vooraleer de *Writer* of de *OutputStream* van het HTTP-antwoord opgehaald worden. Het gebruiken van een sessie voegt namelijk extra headerinfor-

matie toe aan het HTTP-antwoord.

Een sessie kan je gebruiken om objecten te bewaren. Elk object wordt gekoppeld aan een sleutel (type *String*). Deze *String*/*Object*-paren worden *attributen* genoemd. Merk het verschil op met parameters(*String*/*String*-paren)! Om attributen toe te voegen, op te halen of te verwijderen van de sessie zijn de volgende methoden van *HttpServletSession* beschikbaar:

```
public Object getAttribute (String naam);
public void removeAttribute (String naam);
public void setAttribute (String naam, Object waarde);
```

De methode *getAttribute* haalt het object op dat verbonden is met de sleutel *naam*. Is er geen object verbonden met de opgegeven naam dan wordt **null** teruggegeven. De waarde van de methode *getAttribute* is van het type *Object*. Om de methoden van het object te kunnen gebruiken moet het nog gecast worden.

De methode *removeAttribute* verwijdert het object verbonden met de opgegeven naam en doet niets als er geen object bij de opgegeven naam hoort.

De methode *setAttribute* plaatst een object op de sessie en verbindt het met de opgegeven naam. Indien er reeds een object bestaat dat bij de gegeven naam hoort dan wordt dat object vervangen door het meegegeven object.

In ons voorbeeld hebben we het winkelwagentje opgeslagen onder de naam `winkelwagen`. Het servlet haalt dus eerst dit attribuut op, of maakt indien nodig een nieuw attribuut aan met een lege lijst als waarde. Uiteindelijk komt het huidig winkelwagenobject dus terecht in de lokale variabele *wagentje*. Daarna wordt het product dat door de gebruiker werd geselecteerd aan *wagentje* toegevoegd en tot slot gebruiken we *sendRedirect* om de browser door te verwijzen naar het *WagenServlet*.

Het *WagenServlet* gebruikt dezelfde techniek om de sessie te hanteren als hierboven. Dit is zijn *doGet*-methode:

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    HttpSession session = request.getSession ();
    List wagentje = (List) session.getAttribute ("winkelwagen");
    PrintWriter out = response.getWriter();

    out.println ("<html>\n<head>");
    out.println ("<title>Winkelwagentje</title>");
    out.println ("</head>\n<body>");
    out.println ("<h1>Winkelwagentje</h1>");

    if (wagentje == null || wagentje.size() == 0)
        out.println ("<p>Je winkelwagentje is leeg!</p>");
    else {
        int size = wagentje.size ();
        out.println ("<ol>");
        for (int i=0; i < size; i++)
                out.println ("<li>" + wagentje.get(i) + "</li>");
        out.println ("</ol>");
    }
    out.println ("<a href=\"WinkelServlet\"> Terug naar de winkel</a>");
    out.println ("</body></html>");

    out.close();
```

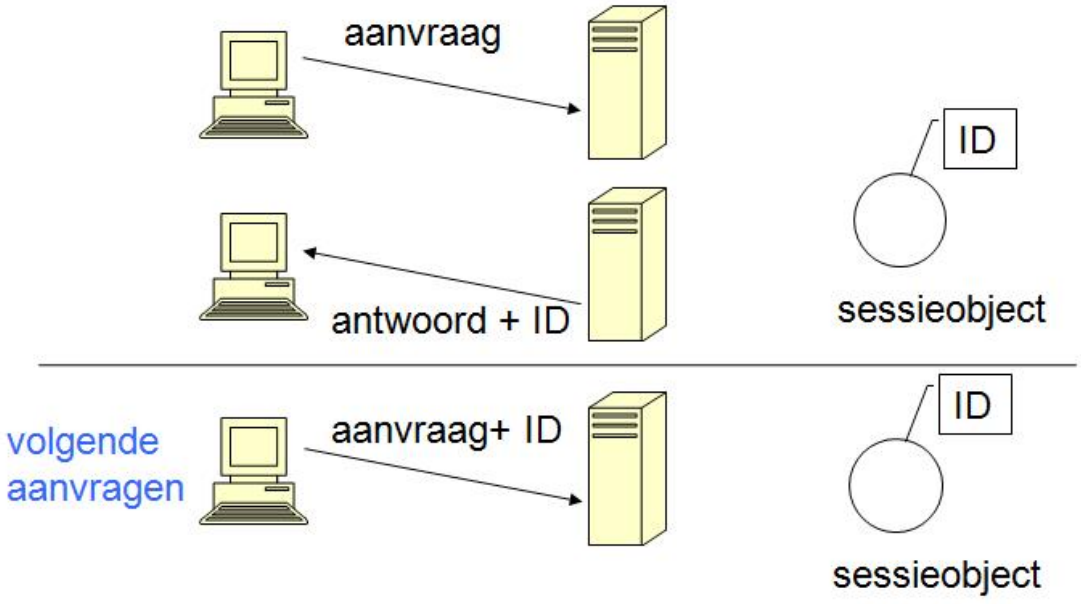
```
}
```

Je hebt je misschien afgevraagd waarom we hier drie servlets gebruiken in plaats van twee, er worden immers slechts twee verschillende pagina's getoond. Het is inderdaad mogelijk om *NieuwProductServlet* en *WagenServlet* te combineren tot één enkel servlet, dat eerst het nieuwe product aan het winkelwagentje toevoegd en daarna de inhoud van het winkelwagentje als HTML-pagina naar de client doorstuurt.

Bedenk echter wat er in dat geval gebeurt wanneer de gebruiker de winkelwagenpagina op zijn browser 'reload'. Niet alleen wordt de pagina opnieuw afgebeeld, maar het product wordt voor een tweede maal aan het wagentje toegevoegd. Door drie servlets te gebruiken doet dit ongewenste bijeffect zich echter niet voor.

17.7.3 URL-herschrijven

Drie mogelijke manieren om sessie-opvolging te realiseren zijn: *URL-herschrijven*, *cookies* en *verborgen velden* in een formulier. Deze technieken gebruiken alle dezelfde werkwijze om sessies te implementeren, namelijk het uitwisselen van een kenteken tussen de client en de server. Als een gebruiker een eerste keer een aanvraag doet, wordt met het antwoord een uniek kenteken meegegeven. Met elke volgende aanvraag wordt dit unieke kenteken terug meegegeven. Op deze wijze kan de server de client herkennen (zie figuur 17.15).



Figuur 17.15: Sessiekenteken uitwisselen tussen client en server

Bij URL-herschrijven wordt extra padinformatie toegevoegd aan elke URL die in het HTTP-antwoord voorkomt. In het geval van verborgen formuliervelden wordt extra informatie toegevoegd aan het

formulier van het HTTP-antwoord in de vorm van onzichtbare velden. Bij het gebruik van cookies wordt het kenteken doorgegeven via de HTTP-headers van de vraag en het antwoord.

Webcontainers realiseren het opvolgen van sessies meestal door gebruik te maken van cookies (zie §13.4). Bij servlets zal dit een cookie zijn van de volgende vorm.

```
JSESSIONID=215266E2484832741237D43179502854
```

De waarde van deze cookie wordt door de webcontainer geassocieerd met een object van het type *HttpSession*. Dit betekent dat de methode *getSession* bij de eerste aanvraag van de client niet alleen het sessie-object aanmaakt, maar ook de `Set-Cookie`-header toevoegt aan het antwoord. Bij volgende aanvragen wordt de sessie-id uit de headers gehaald. Doordat de client dit cookie automatisch opstuurt voor elke aanvraag die bij deze webapplicatie hoort, kan de webcontainer sessie-objecten eenduidig met een bepaalde client associëren.

Indien de client geen cookies ondersteunt dan is de bovenstaande werkwijze niet mogelijk. In dat geval moet je het `JSESSIONID`-kenteken meegegeven als onderdeel van de URL van het servlet. Om dit te verkrijgen moet men gebruik maken van de methode *encodeURL* van *HttpServletResponse*. Dit noemt men de URL *herschrijven*.

Bij *WinkelServlet* moeten we de actie-URL van het formulier herschrijven. In plaats van

```
out.println ("<form action=\"NieuwProductServlet\" method=\"POST\">");
```

staat er nu het volgende:

```
out.println ("<form action=\"" + response.encodeURL ("NieuwProductServlet")
    + "\" method=\"POST\">");
```

Het effect hiervan is dat de actie-URL van het formulier nu automatisch het sessiekenteken bevat:

```
<form action="NieuwProductServlet;JSESSIONID=2152795...54" method="POST">
```

Dit verklaart waarom *WinkelServlet* niet zomaar door een statische HTML-pagina kan worden vervangen.

De link naar *WinkelServlet* in *WagenServlet* moet op dezelfde manier worden aangepast:

```
out.println("<a href=\"" + response.encodeURL ("WinkelServlet")
    + "\">Terug naar de winkel</a>");
```

En ook in *NieuwProductServlet* moeten we de URL in de laatste *sendRedirect* herschrijven.

```
response.sendRedirect(response.encodeRedirectURL("WagenServlet"));
```

Merk op dat je in dit geval de methode *encodeRedirectURL* moet gebruiken in plaats van *encodeURL*.

De methoden *encodeURL* en *encodeRedirectURL* zullen de URL niet aanpassen als de browser wel cookies toelaat. Dit betekent dat het aangeraden is om steeds alle URLs te herschrijven.

17.7.4 Andere methodes van *HttpSession*

Wil je weten of de sessie al bestond voor de huidige aanvraag, gebruik dan de methode *isNew*. Deze methode geeft **true** weer als de client dit sessiekenteken (-ID) nog niet doorstuurde.

```
public boolean isNew();
```

Sessie-objecten gebruiken systeembronnen. We willen dus niet dat sessie-objecten langer bestaan dan nodig is. Hoe weet de webcontainer dat het veilig is om een sessie-object te verwijderen? Er zijn drie manieren waarop een sessie afgesloten kan worden:

- Een bepaalde timeout-periode wordt overschreden. Deze periode kan ingesteld worden in web.xml en wordt uitgedrukt in minuten. In het volgend voorbeeld worden sessie-objecten waarvoor 30 minuten geen HTTP-aanvraag ontvangen is, verwijderd.

```
<web-app ... >
    ...
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

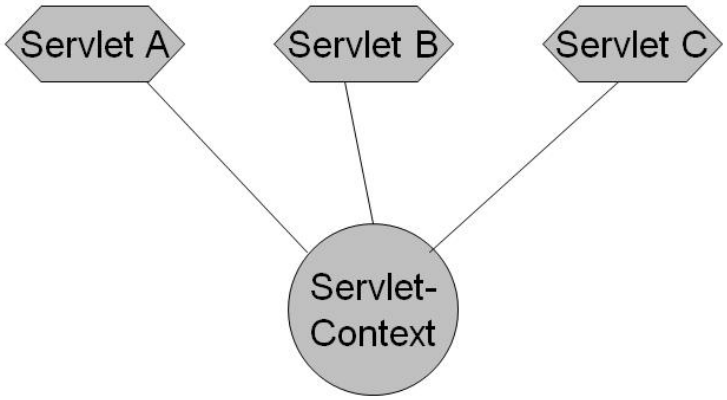
- De applicatie wordt afgesloten, bv. omdat ze crasht of omdat ze van de server verwijderd wordt.
- Je roept de methode *invalidate* op op het sessie-object. Deze methode verwijdert alle attributen van het sessie-object en geeft het vrij. Er bestaat geen sessie-object meer voor het bijhorende sessie-ID. Je gebruikt de methode *invalidate* als je weet dat de sessie afgerond is, bv. omdat de gebruiker uitlogt.

17.8 *ServletContext*

Sessies zorgen ervoor dat er objecten geassocieerd kunnen worden met clients. Om objecten te associëren met de webtoepassing zelf, gevormd door een aantal servlets, voorziet de Java Servlet API de interface *ServletContext*. De servletcontext kan gebruikt worden om informatie te verkrijgen over de applicatieomgeving: naam van de server, methoden om naar log-bestanden te schrijven, parameters voor de webapplicatie en dergelijke meer. Een andere toepassing is het delen van objecten tussen verschillende servlets van dezelfde webapplicatie. Er is **één** *ServletContext*-object voor de volledige webapplicatie. Alle servlets van deze webapplicatie delen hetzelfde *ServletContext*-object. De webcontainer maakt het *ServletContext*-object aan wanneer de webapplicatie gepubliceerd (*gedeployed*) wordt op de server en stelt het ter beschikking van alle servlets in de applicatie. (zie figuur 17.16)

Een object *ServletContext* kan verkregen worden door de methode *getServletContext* op te roepen. Deze methode hoort bij de interface *ServletConfig* die door *GenericServlet*, en dus door elke servlet wordt geïmplementeerd.

De interface *ServletContext* voorziet onder andere de volgende methoden:



Figuur 17.16: ServletContext

```
public Object getAttribute (String name);
public Enumeration getAttributeNames ();
public void removeAttribute (String name);
public void setAttribute (String name, Object object);
```

Deze methoden bieden de mogelijkheid om de status van de applicatie te bewaren. Net zoals bij sessies kan je hiermee attributen (sleutel/object-paren) bewaren. Merk op dat als verschillende servlets tegelijkertijd objecten van de servletcontext kunnen veranderen de methoden van die objecten gesynchroniseerd moeten zijn.

In het volgend voorbeeld wordt een referentie naar een *Catalogus*-object op de *ServletContext* opgehaald (zie figuur 17.17). Hetzelfde *Catalogus*-object wordt door twee servlets gebruikt: het *OverzichtBoekenServlet* en het *BoekDetailServlet*. Het eerste servlet gebruikt het *Catalogus*-object om een webpagina met een lijst van boeken te genereren. Het *Catalogus*-object biedt ook de mogelijkheid om detailinformatie over een bepaald boek op te zoeken. Deze functionaliteit wordt opgeroepen door het *BoekDetailServlet* om een webpagina door te sturen met uitgebreide informatie over een welbepaald boek. In beide servlets wordt de referentie naar het *Catalogus*-object als volgt verkregen.

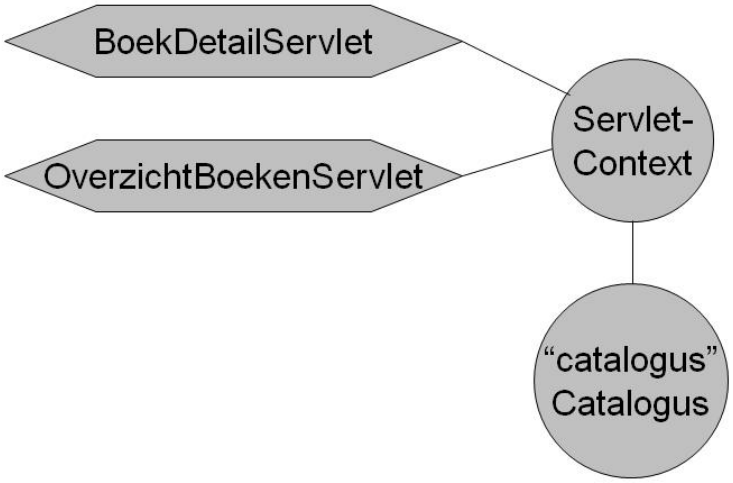
```
Catalogus catalogus = (Catalogus) getServletContext().getAttribute(``catalogus'');
```

17.8.1 Contextparameters

Een servletcontext bevat ook *parameters* die door de webadministrator kunnen worden geconfigureerd in het bestand web.xml en door de servlets worden geraadpleegd met de volgende methoden:

```
public String getInitParameter (String naam);
public Enumeration getInitParameterNames ();
```

De methode *getInitParameter* geeft de waarde terug van de parameter met de gegeven *naam*, of **null** wanneer er geen dergelijke parameter bestaat. Met de methode *getInitParameterNames* kan je alle geldige parameternamen overlopen.



Figuur 17.17: Catalogus-object op ServletContext

De parameters uit de servletcontext zijn dezelfde voor alle servlets van de webtoepassing. Je vindt ze dan ook terug als kindelementen van het `<web-app>`-element in het bestand web.xml. Merk het verschil op met de initialisatieparameters van een servlet (zie §17.6.1).

```
<web-app ... >
    <context-param>
        <param-name>naam</param-name>
        <param-value>waarde</param-value>
        ...
    </context-param>
</web-app>
```

17.9 Samenwerking tussen servlets

In meer complexe webtoepassingen kan het nuttig zijn dat het ene servlet gebruik maakt van de diensten van een ander servlet. Bijvoorbeeld:

- Wanneer alle webpagina's gebruik maken van dezelfde, of van een gelijkaardige hoofding, dan kan je deze hoofding telkens door hetzelfde servlet laten genereren, terwijl de verschillende paginainhouden door verschillende servlets worden voorgebracht.
- Je kan de logica en de presentatie van je toepassing van elkaar scheiden door een servlet te voorzien dat reeds gedeeltelijk de aanvragen verwerkt, en dan voor de presentatie beroep doet op een ander servlet. (Deze techniek wordt onder andere toegepast bij het zogenaamde MVC-patroon, ook wel Model 2 genoemd, zie hoofdstuk **??**, [MVC],[Ses99].)

Bij het eerste voorbeeld wordt de uitvoer van het ene servlet *tussengevoegd* tussen de uitvoer van het andere servlet. Bij het tweede voorbeeld wordt de verwerking van de aanvraaggegevens *doorgegeven* aan een ander servlet.

Om deze technieken toe te passen, maak je gebruik van een object van het type *RequestDispatcher*. Een dergelijk object stelt een webcomponent voor die door dezelfde webcontainer wordt beheerd als het servlet waarin dit object wordt gebruikt. Deze webcomponent kan een servlet zijn, maar ook een statische webpagina of een JSP-pagina (zie hoofdstuk **??**).

Er zijn drie manieren waarop je een *RequestDispatcher* kan bekomen. Enerzijds zijn er twee methoden uit de interface *ServletContext*:

```
public RequestDispatcher getRequestDispatcher (String path)
public RequestDispatcher getNamedDispatcher (String name)
```

Bij *getRequestDispatcher* geef je als argument het pad op van de webcomponent die je wenst te gebruiken. Dit pad begint steeds met een '/' en wordt gerekend ten opzichte van de *context root* (de basis-URL) van de webtoepassing (zie 17.4). Bij *getNamedDispatcher* geef je de interne *naam* op van de webcomponent (bv. een servlet). Deze interne naam wordt aan een bepaalde webcomponent verbonden in het configuratiebestand web.xml van de webtoepassing.

Een derde methode is wellicht iets gebruiksvriendelijker en hoort bij de klasse *ServletRequest*:

```
public RequestDispatcher getRequestDispatcher (String path)
```

Het enige verschil met de gelijknamige methode uit *ServletContext* is dat je hier ook een relatief pad mag opgeven. De gevraagde webcomponent wordt dan bepaald op basis de URL geassocieerd met het *ServletRequest*-object en het relatieve pad.

De interface *RequestDispatcher* bezit twee methoden die je kan gebruiken om aanvragen naar de aangegeven bron uit te voeren. Wens je de inhoud van deze bron op te nemen als onderdeel van de uitvoer van je eigen servlet, dan roep je de volgende methode op:

```
public void include (ServletRequest request, ServletResponse response)
    throws ServletException, IOException
```

Wens je je aanvraag door te geven aan de nieuwe bron, dan gebruik je de tweede methode:

```
public void forward (ServletRequest request, ServletResponse response)
    throws ServletException, IOException
```

In de praktijk roep je de methoden *include* en *forward* op met als argumenten dezelfde *request*- en *response*-objecten die je als parameters binnenkreeg van je eigen *doGet* of *doPost*. Is de webcomponent die je oproept een servlet, dan kan je eventueel extra informatie meegeven via de zogenaamde *attributen* van *request*. Deze zijn sleutel/object-paren vergelijkbaar met de attributen van *ServletContext* of van *HttpSession*. Hiermee kan je informatie delen tussen de servlets die éénzelfde aanvraag verwerken. Verwar ze niet met de HTTP-aanvraagparameters (*String*/*String*-paren) zelf.

Als voorbeeld passen we het *WagenServlet* aan zodat het een foutboodschap genereert wanneer het wordt opgeroepen zonder dat er reeds een winkelwagentje werd aangemaakt. De foutboodschap is een statische webpagina die we zullen oproepen met behulp van *forward*.

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
```

```
    HttpSession session = request.getSession ();
    List wagentje = (List) session.getAttribute ("winkelwagen");

    if (wagentje == null) {
        RequestDispatcher rd = request.getRequestDispatcher ("/error.html");
        rd.forward (request, response);
    } else {
        ...
  }
}
```

Andere voorbeelden vind je later in deze tekst (zie §**??**) en in de uitwerking het MVC-model.

Tot slot merken we nog op dat er een belangrijk verschil is tussen het gebruik van een *sendRedirect* zoals we in *NieuwProductServlet* hebben toegepast (§17.7.2), en het gebruik van een *forward* zoals hierboven. In het eerste geval is er een tussenkomst van de client nodig: er wordt een HTTP-antwoordbericht naar de client gestuurd en de browser doet een nieuwe HTTP-aanvraag op basis van de `Location`-header. In het tweede geval gebeurt alles aan de kant van de server: het *request*-object wordt doorgegeven van het ene servlet naar het andere.

17.10 Luisteraars (*Listeners*)

17.10.1 Attributen hebben een *scope*

In de paragrafen 17.7.2, 17.8 en 17.9 hebben we gezien dat we attributen kunnen toevoegen aan en opvragen van sessie-objecten, de servletcontext en *request*-objecten. Attributen zijn sleutel/objecten-paren, waarbij de sleutel van het type *String* is. Wie deze attributen kan zien en gebruiken en hoe lang ze bestaan, hangt af van het object waarop ze bewaard worden: de servletcontext, de sessie of de aanvraag. De *scope* (reikwijdte) van een attribuut is één van de volgende waarden: **context**, **session** of **request**. De toegang tot de attributen en hun levensduur hangt dus af van hun scope (zie tabel 17.1).

Merk op dat attributen met scope context en session NIET thread-safe zijn. Ze kunnen dus vanuit verschillende threads gelijktijdig gemanipuleerd worden. Enkel attributen met scope request zijn thread-safe omdat ze maar vanuit één thread gebruikt worden: de thread waarin de *service-methode* voor deze aanvraag wordt uitgevoerd.

17.10.2 Luisteren naar events (gebeurtenissen)

In het voorbeeld uit §17.8 illustreerden we dat een attribuut van het type *Catalogus* met scope *context* gebruikt werd in twee servlets nl. *OverzichtBoekenServlet* en *BoekDetailServlet* (zie figuur 17.18).

Waar en wanneer wordt het attribuut “catalogus” aangemaakt en bewaard? Je zou ervoor kunnen kiezen om dit te doen in de initialisatiefaze (i.e. in de *init*-methode) van *OverzichtBoekenServlet*.

Scope	Toegang	Levensduur	Gebruik
Context	Alle webcomponenten, luisteraars, ...	Zolang de webapplicatie bestaat. Verwijderd als de server of de applicatie afsluit.	Bronnen die gedeeld worden voor de volledige applicatie.
Session	Elke webcomponent die toegang heeft tot deze sessie, m.a.w. die een aanvraag afhandelt die gelinkt is aan deze sessie.	Zolang dit sessie-object bestaat.	Data die specifiek is voor deze gebruiker en die moet behouden blijven tussen verschillende aanvragen.
Request	Alle webcomponenten die de huidige aanvraag afhandelen, d.i. de verschillende servlets waaraan de aanvraag wordt doorgeven d.m.v. een *RequestDispatcher*.	Zolang het aanvraag-object bestaat, m.a.w. tijdens het uitvoeren van de *service*-methode.	Data doorgeven die specifiek is voor één enkele aanvraag.

Tabel 17.1: Kenmerken attributen in de verschillende scopes

Dit betekent echter dat *BoekDetailServlet* of een andere servlet dat gebruikt maakt van het attribuut "catalogus" niet opgeroepen mag worden voordat *OverzichtBoekenServlet* geïnitialiseerd is en het attribuut "catalogus" op de *ServletContext* geplaatst is. Een beter en robuuster alternatief is het gebruik van luisteraars. De servlet API voorziet verschillende types luisteraars die luisteren naar events van de levensloop van *ServletContext*-, *HttpSession*- en *ServletRequest*-objecten.

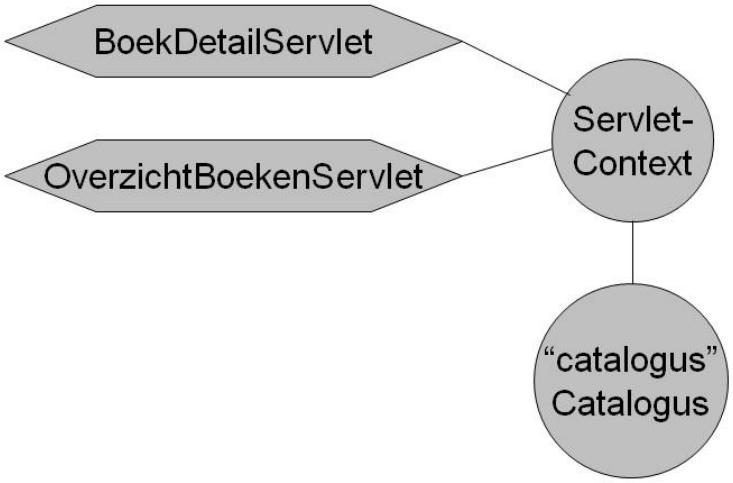
ServletContextListener

In het volgend voorbeeld zorgen we ervoor dat het attribuut "catalogus" aangemaakt wordt bij het opstarten van de webapplicatie, meer specifieker bij het aanmaken van het *ServletContext*-object. Hiervoor schrijven we een implementatie van de interface *ServletContextListener* uit het pakket *javax.servlet*. Deze luisteraar kan op twee gebeurtenissen reageren: het aanmaken en het vernietigen van het *ServletContext*-object. De interface *ServletContextListener* bestaat uit de twee volgende methodes.

```
public void contextInitialized(ServletContextEvent event);
public void contextDestroyed(ServletContextEvent event);
```

In het voorbeeld ondernemen we alleen iets bij het creëren van het *ServletContext*-object.

- Contextparameters inlezen. Zij bepalen de klasse van het *Catalogus*-object. (*Catalogus* is een interface waarvan de implementatie kan verschillen)
- Het *Catalogus*-object aanmaken
- Het *Catalogus*-object bewaren op de *ServletContext*



Figuur 17.18: Catalogus-object op ServletContext

```
package be.hogent.iii.boeken.web;
import javax.servlet.*;
import be.hogent.iii.boeken.bo.Catalogus;

public class AanmaakCatalogus implements ServletContextListener{

    public static final String CATALOGUS = "catalogus";
    public static final String KLASSE_CATALOGUS = "catalogusKlasse";

    public void contextInitialized (ServletContextEvent event) {
        ServletContext ctx = event.getServletContext();
        String catalogusKlasse = ctx.getInitParameter(KLASSE_CATALOGUS);
        try {
            // aanmaken Catalogus-object
            Catalogus catalogus = (Catalogus) Class.forName(catalogusKlasse).newInstance();
            ctx.setAttribute(CATALOGUS,  catalogus);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public void contextDestroyed (ServletContextEvent event) {
    }
}
```

```
<web-app...>
  <context-param>
    <param-name>catalogusKlasse</param-name>
    <param-value>be.hogent.iii.boeken.implMemory.GeheugenCatalogus</param-value>
  </context-param>
  ...
</web-app>
```

Een referentie naar het *ServletContext*-object verkrijg je via de *ServletContextEvent*, een argument van de event-methodes. Het gooien van een *RuntimeException* zorgt ervoor dat de webapplicatie niet opstart indien het *Catalogus*-object niet aangemaakt kan worden.

Nu we een luisterklasse geschreven hebben, resten nog de volgende vragen: Waar moet het .class-bestand staan en hoe moet de luisteraar geregistreerd worden zodat hij opgeroepen wordt als de event optreedt? Het volgende stappenplan geeft hierop een antwoord.

1. Maak een luisterklasse die de interface *ServletContextListener* implementeert. Schrijf ook de klassen en interfaces die de luisterklasse gebruikt (in het voorbeeld *Catalogus* en *GeheugenCatalogus*).

2. Plaats de klasse(n) in de map `WEB-INF/classes` van de webapplicaties. In het voorbeeld komt het bestand *AanmaakCatalogus.class* in de map `WEB-INF/classes/be/hogent/iii/boeken/web/` omdat de klasse tot het pakket *be.hogent.iii.boeken.web* behoort. In het voorbeeld moeten ook de .class-bestanden van *Catalogus* en *GeheugenCatalogus* op de juiste plaats gezet worden.

3. Registreer de luisteraar. Hiervoor voeg je een `<listener>`-element toe aan de deployment descriptor. De webcontainer zal ervoor zorgen dat de luisteraar op het juiste moment aangemaakt en opgeroepen wordt.

```
<web-app...>
  ...
  <listener>
    <listener-class>be.hogent.iii.boeken.web.AanmaakCatalogus</listener-class>
  </listener>
  ...
</web-app>
```

Andere luisteraars

Je kan niet alleen luisteren naar events van de levensloop van het *ServletContext*-object. Ook voor events gerelateerd met aanvragen en sessies, zijn er luisteraars. Bovendien is het mogelijk om te luisteren naar het toevoegen, aanpassen en verwijderen van attributen in de verschillende scopes. Al deze luisteraars worden geregistreerd in deployment descriptor m.b.v. een `<listener>`-element. De webcontainer inspecteert de klasse en kent zo het type van de luisterklasse.

HttpSessionListener luistert naar het aanmaken of verwijderen van sessie-objecten.

```
public void sessionCreated(HttpSessionEvent event);
public void sessionDestroyed(HttpSessionEvent event);
```

ServletRequestListener luistert naar het binnenkomen van aanvragen.

```
public void requestDestroyed(ServletRequestEvent event);
public void requestInitialized(ServletRequestEvent event);
```

ServletContextAttributeListener luistert naar het toevoegen, vervangen en verwijderen van attributen op de *ServletContext*.

```
public void attributeAdded(ServletContextAttributeEvent event);
public void attributeRemoved(ServletContextAttributeEvent event);
public void attributeReplaced(ServletContextAttributeEvent event);
```

ServletRequestAttributeListener luistert naar het toevoegen, vervangen en verwijderen van attributen op de *ServletRequest*.

```
public void attributeAdded(ServletRequestAttributeEvent event);
public void attributeRemoved(ServletRequestAttributeEvent event);
public void attributeReplaced(ServletRequestAttributeEvent event);
```

HttpSessionAttributeListener luistert naar het toevoegen, vervangen en verwijderen van attributen op de *HttpSession*.

```
public void attributeAdded(HttpSessionBindingEvent event);
public void attributeRemoved(HttpSessionBindingEvent event);
public void attributeReplaced(HttpSessionBindingEvent event);
```

Bibliografie

[APA] Apache http server project. http://httpd.apache.org/.

[BLMF+98] T. Berners-Lee, MIT/LCS, R. Fielding, UC Irvine, L. Masinter, and Xerox Corporation. Uniform resource identifiers (uri): Generic syntax. http://www.ietf.org/rfc/rfc2396.txt, August 1998.

[BSB04] Bryan Basham, Kathy Sierra, and Bert Bates. *Head First Servlets & JSP*. O'Reilly, August 2004.

[Cha01] Steve Champeon. Javascript: How did we get here? http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html, June 2001.

[CSSa] Cascading style sheets. http://www.w3.org/Style/CSS/.

[CSSb] Cascading style sheets level 2 revision 1 (css 2.1) specification, full property table. http://www.w3.org/TR/CSS21/propidx.html.

[DOM] Document object model (dom). http://www.w3.org/DOM/.

[DYNa] Dhtml tutorial. http://www.w3schools.com/dhtml/default.asp.

[DYNb] Dynamic html. http://nl.wikipedia.org/wiki/Dynamic_HTML.

[FIG+99] R. Fielding, UC Irvine, J. Gettys, Compaq/W3C, J. Mogul, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. ftp://ftp.isi.edu/in-notes/rfc2616.txt, June 1999.

[Fis99] Maydene Fisher. The jdbc tutorial: Chapter 3 - advanced tutorial. http://java.sun.com/developer/Books/JDBCTutorial/index.html, May 1999.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide, Fifth Edition*. O'Reilly, August 2006.

[GMW+99] Y. Goland, Microsoft, E. Whitehead, UC Irvine, A. Faizi, Netscape, S.R. Carter, D. Jensen, and Novell. Http extensions for distributed authoring – webdav. http://www.webdav.org/specs/rfc2518.pdf, February 1999.

[HTMa] Handleiding html. http://www.handleidinghtml.nl/.

[HTMb] Html 4.01 specification. http://www.w3.org/TR/html401/.

[IIS] Internet information services. http://www.iis.net/.

[ISO] Ecmascript language specification, iso/iec 16262. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33835.

[JBC+07] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase. The java ee 5 tutorial. http://java.sun.com/javaee/5/docs/tutorial/doc/, September 2007.

[KLT+97] D. Kristol, Bell Laboratories, Lucent Technologies, L. Montulli, and Netscape Communications. Http state management mechanism. http://www.ietf.org/rfc/rfc2109.txt, February 1997.

[MVC] Java blueprints: Model-view-controller. http://java.sun.com/blueprints/patterns/MVC-detailed.html.

[Seb08] Robert W. Sebesta. *Programming the World Wide Web, 4th edition*. Pearson Education, 2008.

[Ses99] Govind Seshadri. Server-side java: Understanding javaserver pages model 2 architecture, exploring the mvc design pattern. *JavaWorld.com*, December 1999. http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html.

[W3C] World wide web consortium. http://www.w3.org.

[WEBa] Webdevelopment. http://nl.wikipedia.org/wiki/Webdevelopment.

[WEBb] World wide web distributed authoring and versioning. http://www.webdav.org/.

[Whi98] Jim Whitehead. Collaborative authoring on the web: Introducing webdav. *Bulletin of the American Society for Information Science*, Vol. 25(No. 1):25–29, October 1998. http://www.asis.org/Bulletin/Oct-98/webdav.html.