

Afbeelding 75: Invullen van array van Strings

Ook hier kunnen we de *array* initialiseren tijdens de declaratie:

```
String[] lines = {"Hello World",
                  "Hello Mars",
                  "Hello Venus",
                  "Hello Jupiter",
                  "Hello Saturn"};
```

We komen hier nog even terug op de klasse *String*. We hebben reeds een aantal methoden van deze klasse gebruikt. In de onderstaande tabel geven we nog een aantal extra methoden die telkens een *array* van gegevens teruggeven:

Return	Methode	Beschrijving
String[]	split(String regex)	Splitst de <i>string</i> op in deel- <i>strings</i> op basis van een scheidingstekst die wordt opgegeven door middel van een reguliere uitdrukking.
char[]	toCharArray()	Geeft de <i>array</i> van karakters van deze <i>string</i> terug.
byte[]	getBytes()	Geeft de <i>array</i> van <i>bytes</i> van deze <i>string</i> terug (standaard karakterset).

Tabel 28: Methoden van de klasse *String* die een *array* teruggeven

We illustreren even het gebruik van de methode *split()*.

```
public class SplitSample {
    public static void main(String[] args) {
        String text = "I just want to say hello!";
        String[] words = text.split(" ");
        for(String word: words) {
            System.out.println(word);
        }
    }
}
```

We gaan de tekst opsplitsen in woorden. De scheidingstekst is hier dus een spatie. We kunnen ook een complexe scheidingstekst meegeven in de vorm van een reguliere uitdrukking (*regular expression*), maar dat valt buiten het bestek van deze cursus. We verwijzen hiervoor naar de documentatie van de klasse *Pattern*.

Het resultaat van deze `split`-methode is een *array* van *strings* die we vervolgens afdrukken.

We kunnen deze `split`-methode ook gebruiken om bijvoorbeeld tekst op te splitsen waarbij de verschillende delen gescheiden zijn door een komma of puntkomma. Dit komt voor bij bestanden die opgemaakt zijn met de CSV-indeling (*comma separated value*).

Ten slotte nog een woordje over de methode `main()` die we in elke programma terugvinden.

```
public static void main(String[] args)
```

Deze heeft telkens als argument een *array* van *strings*. Deze *array* bevat de woorden die zijn meegegeven bij het opstarten van het programma.

Bijvoorbeeld:

```
package hello;

public class HelloApp {
    public static void main(String[] args) {
        for(String arg: args) {
            System.out.println(arg);
        }
    }
}
```

We kunnen deze toepassing als volgt opstarten van de commandolijn:

```
java hello.HelloApp Hello World!
```

De woorden die volgen op de volledige klassennaam zullen aan de methode `main()` worden doorgegeven via de *array*.

Het meegeven van argumenten aan een programma kan ook als we het opstarten via de IDE. We verwijzen hiervoor naar de documentatie van de IDE.

Opdracht 2: Arrays van objecten

- Maak een programma dat een regel tekst aan de gebruiker vraagt en de woorden afdrukt op het scherm. Lees de regel tekst als volgt:

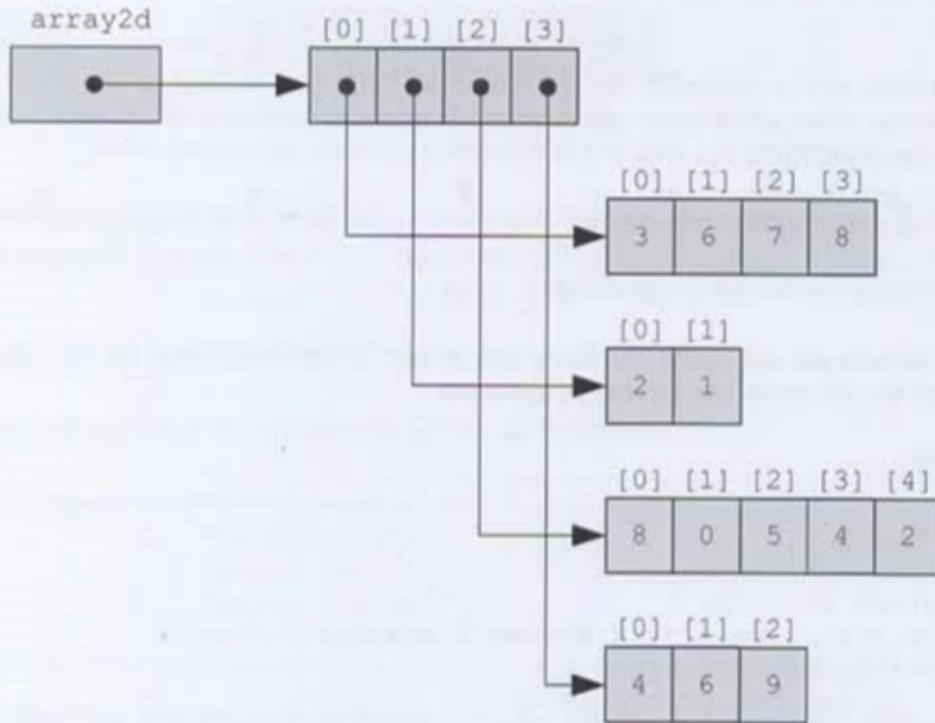
```
Scanner keyboard = new Scanner(System.in);
String text = keyboard.nextLine();
```

7.6 Arrays van arrays

Vermits een *array* ook een object is, kan men ook een *array* van *arrays* maken. In het volgende voorbeeld maken we een *array* van *arrays* van integers:

```
int[][] array2d = {{3,6,7,8},{2,1},{8,0,5,4,2},{4,6,9}};
```

Deze *array* bevat vier referenties naar vier *arrays* van integers. Merk op dat de *arrays* van integers niet noodzakelijk dezelfde lengte hebben.



Afbeelding 76: Een array van arrays

We gebruiken bij de declaratie dubbele vierkante haken om aan te geven dat het om een *array van arrays* gaat.

In het vorige voorbeeld hebben we de *array van arrays* onmiddellijk geïnitialiseerd. We kunnen deze stappen ook afzonderlijk zetten:

```

int[][] array2d = new int[4][];
array2d[0] = new int[4];
array2d[1] = new int[2];
array2d[2] = new int[5];
array2d[3] = new int[3];
array2d[0][0] = 3;
array2d[0][1] = 6;
array2d[0][2] = 7;
array2d[0][3] = 8;
array2d[1][0] = 2;
array2d[1][1] = 1;
array2d[2][0] = 8;
array2d[2][1] = 0;
array2d[2][2] = 5;
array2d[2][3] = 4;
array2d[2][4] = 2;
array2d[3][0] = 4;
array2d[3][1] = 6;
array2d[3][2] = 9;
  
```

Eerst wordt de *array van arrays* gecreëerd. Vervolgens wordt voor elk element uit die *array* een *array* van integers gecreëerd en de referentie wordt toegewezen aan het element. Ten slotte krijgen de getallen een waarde.

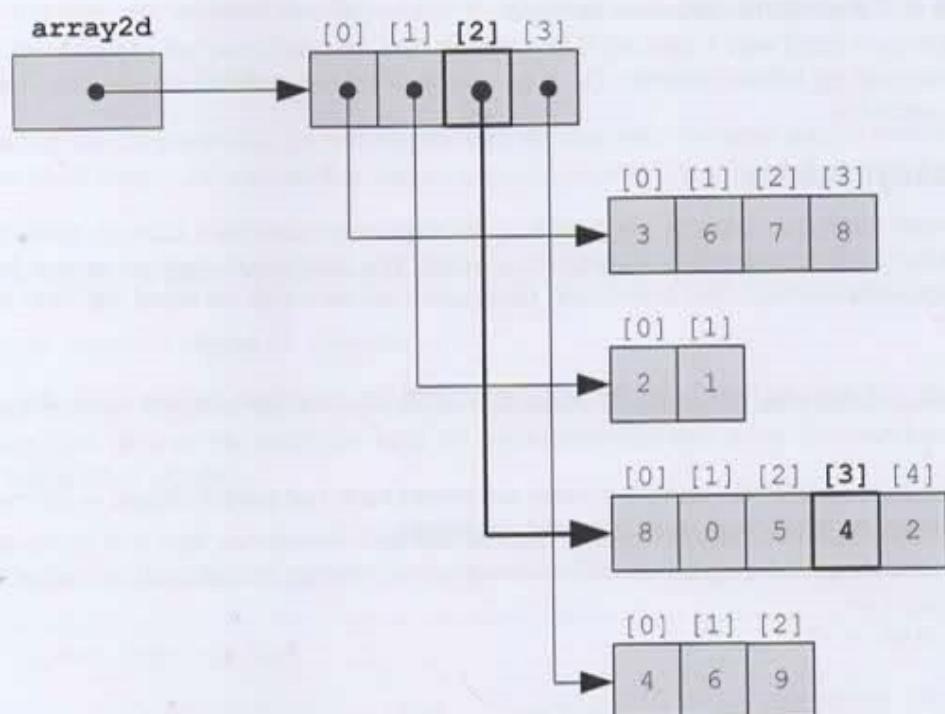
Om een bepaald element uit de tweedimensionale *array* aan te duiden, gebruiken we tweemaal de vierkante haken: het eerste stel om de index in de eerste *array* aan te geven en het tweede stel om de index in de tweede *array* aan te geven.

Bijvoorbeeld:

array2d[2][3]

De waarde van dit element is 4.

Schematisch:



Afbeelding 77: Een element in een tweedimensionale array aanduiden

Het afdrukken van alle elementen kan men als volgt doen:

```
for(int i = 0; i < array2d.length ; i ++){  
    for(int j = 0; j < array2d[i].length; j++){  
        System.out.println(array2d[i][j]);  
    }  
}
```

Of door middel van een *for each*-lus:

```
for(int[] row: array2d) {  
    for(int el: row) {  
        System.out.print(el);  
    }  
}
```

Indien de *arrays* allemaal dezelfde lengte hebben, kan men een tweedimensionale *array* als

volgt declareren:

```
int[][] table = new int[4][7];
```

We maken hier een tweedimensionale *array* van vier rijen en zeven kolommen.

Zowel de hoofd-*array* als de sub-*array* worden tegelijkertijd gecreëerd. De hoofd-*array* wordt ingevuld met referenties naar de sub-*arrays*.

Op dezelfde manier kunnen we ook een driedimensionale *array* maken:

```
int[][][] cube = new int[4][7][9];
```

Opdracht 3: Tweedimensionale arrays

- Maak een tabel van 4 rijen en 6 kolommen. Vul de tabel met het product van het rijnummer en kolomnummer. Druk de matrix af op het scherm en gebruik hierbij een *for each*-lus.

7.7 Lookup tables

Arrays worden vaak gebruikt om gegevens op te zoeken en daarmee kunnen soms lange *if else*- of *switch case*-constructies vermeden worden. We noemen dergelijke *arrays lookup tables*. Gegevens kunnen dan eenvoudig opgezocht worden aan de hand van hun index in de array.

We illustreren dit aan de hand van het voorbeeld met de maanden dat we al eerder gezien hebben.

We moesten hier de waarde van de maand omzetten naar het aantal dagen in de maand, wat resulteerde in deze lange *switch case*-constructie.

```
int month;
int days = 0;

switch (month) {
    default: days = 0; break;
    case 2: days = 28; break;
    case 4:
    case 6:
    case 9:
    case 11: days = 30; break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: days = 31;
}
System.out.println(days);
```

Met een *lookup table* kan dat ook als volgt:

```
public class LookupTableSample {
    public static void main(String[] args) {
```

```

int[] months = {31,28,31,30,31,30,31,31,30,31,30,31};
int month = 5;
System.out.println(months[month-1]);
}
    
```

Opdracht 4: Lookup Tables

- Maak een programma dat aan de gebruiker een woord vraagt en dat vervolgens de Scrabble-woordwaarde berekent. Gebruik hiervoor een *lookup table*. Hint: maak gebruik van de methode `toCharArray()` van de klasse `String`.

7.8 Methoden met een variabel aantal parameters

Methoden kunnen een variabel aantal parameters hebben. De algemene syntax is als volgt:

```
returntype method(type... params);
```

Met ... wordt aangegeven dat de parameter van dit type een variabel aantal keren kan voorkomen (0 of meer). Dit kan echter alleen gebruikt worden voor de laatste parameter in de lijst.

In feite is dit hetzelfde alsof de parameter een *array* zou zijn:

```
returntype method(type[] params);
```

De *compiler* zal een variabel aantal parameters gewoon omzetten in een *array* van het aangegeven type. Binnen de methode stelt de parameter dus een *array* voor en kan die als dusdanig behandeld worden.

We illustreren dit met een voorbeeld. Stel dat we een klasse hebben met een methode om het gemiddelde van een aantal getallen te berekenen. Dit aantal getallen kan variëren.

```

public class AverageApp {

    public static int average(int... values) {
        int total = 0;
        for (int el : values) {
            total += el;
        }
        int avg = 0;
        if(values.length != 0) {
            avg = total/values.length;
        }
        return avg;
    }

    public static void main(String[] args) {
        System.out.println(average(4,7,9));
        System.out.println(average(7,9,2,3,5));
        System.out.println(average(3,8,6,9,4,7));
        System.out.println(average());
        int[] values = {3,8,6,9,4,7};
        System.out.println(average(values));
    }
}
    
```

We kunnen nu de methode oproepen met een variabel aantal argumenten of met een *array* van waarden. Dit komt op hetzelfde neer. De eerste manier wordt door de compiler namelijk omgezet in de tweede.

Indien twee methoden dezelfde naam dragen (*method name overloading*) kan er evenwel een dubbelzinnigheid ontstaan.

We illustreren dit met een voorbeeld:

```
public class AverageApp {  
  
    public static int average(int... values) {  
        int total = 0;  
        for (int el : values) {  
            total += el;  
        }  
        int avg = 0;  
        if(values.length != 0) {  
            avg = total/values.length;  
        }  
        return avg;  
    }  
  
    public static int average(int value1, int value2) {  
        return (value1 + value2)/2;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(average(4,6));  
    }  
}
```

De methode `average()` heeft twee *overloaded* varianten, de ene met een variabel aantal parameters en de andere met expliciet twee parameters. Indien we de methode oproepen met effectief twee argumenten, zal de methode met twee parameters opgeroepen worden. De reden hiervoor is dat de mogelijkheid tot een variabel aantal parameters later in Java is toegevoegd (sinds Java 5) en dat dergelijke toevoeging nooit het gedrag van oudere code mag wijzigen.

Opdracht 5: Methoden met variabel aantal parameters

- Maak een klasse `Statistics` die volgende methoden heeft om het gemiddelde, minimum en maximum te berekenen:

```
public static int average(int... values)  
public static int min(int... values)  
public static int max(int... values)
```

- Maak een hoofdprogramma waarin je deze methoden gebruikt.

7.9 Samenvatting

In dit hoofdstuk hebben we leren werken met **arrays**. We hebben gezien hoe we een **array** van een bepaald datatype kunnen **aanmaken** en de elementen kunnen **invullen** en **opvragen**.

Arrays kunnen primitieve datatypes bevatten maar ook objecten. We krijgen dan een verzameling van objecten. Vermits een *arrays* zelf ook een object is, kunnen we dan ook een array van *arrays* maken. Zo is het mogelijk **meerdimensionale arrays** te maken.

Lookup tables zijn een handige toepassing van *arrays* om snel een keuze te kunnen maken tussen elementen.

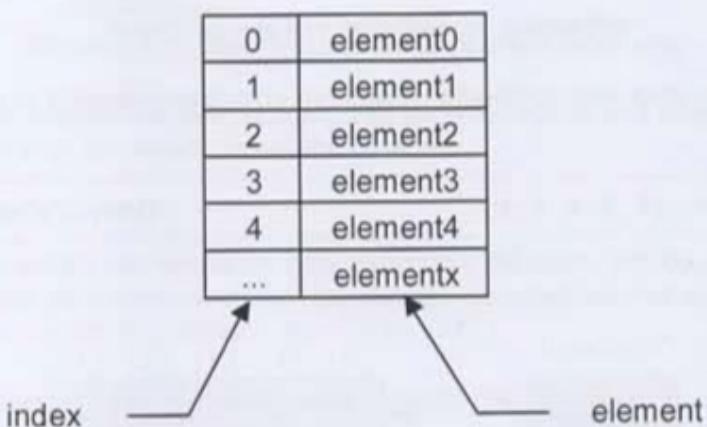
Methoden kunnen een **variabel aantal parameters** hebben; dit is eigenlijk een parameter van het type *array*.

Hoofdstuk 7: Arrays

7.1 Inleiding

Een bijzonder type object is de *array*. Daarom besteden we er hier een afzonderlijk hoofdstuk aan.

Een *array* is een verzameling van elementen van hetzelfde type waarbij ieder element voorzien is van een nummer (index) dat de plaats van dat element in de *array* aanduidt.



Afbeelding 67: Een array met index en element

De elementen in een *array* kunnen zowel primitieve datatypen als referentietypen zijn. In het laatste geval heeft men een reeks van verwijzingen naar objecten.

7.2 Arrays maken

In Java is een *array* op zich ook een object. *Arrays* worden net als alle andere objecten gecreëerd met de *new*-operator. De algemene syntax ziet er als volgt uit:

```
datatype[] arrayName = new datatype[length];
```

Om aan te geven wat het datatype van de elementen in de *array* is, laten we de *new*-operator volgen door het datatype van de elementen. Tussen vierkante haken wordt vervolgens het aantal elementen in de *array* vastgelegd. *Arrays* hebben namelijk een vaste grootte die nadien niet meer gewijzigd kan worden. Dit aantal wordt tijdens de uitvoering van het programma (*runtime*) bepaald en mag dus een variabele zijn.

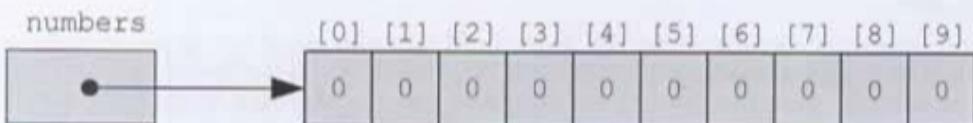
Er wordt een *array-object* gecreëerd en het resultaat is een referentie naar deze *array*. Deze referentie kennen we toe aan de referentievariabele *arrayName*. De referentievariabele wordt gedeclareerd als zijnde van het type *array van datatype*, hetgeen wordt aangegeven door *datatype[]*. We kunnen de rechthoekige haken ook achter de naam van de *array* zetten, maar deze notatie wordt niet aanbevolen:

```
datatype arrayName[] = new datatype[length];
```

In het volgende voorbeeld maken we een *array* van tien getallen:

```
int[] numbers = new int[10];
```

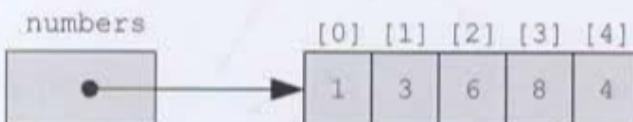
Na het creëren van de *array* worden de waarden van de elementen automatisch geïnitialiseerd met de respectievelijke waarde 0, null of false.



Afbeelding 68: Een array met tien elementen

Java beschikt tevens over een compacte notatie om *arrays* tegelijkertijd te creëren en in te vullen:

```
int[] numbers = {1,3,6,8,4};
```



Afbeelding 69: Een array met vijf elementen

De initiële waarden van de elementen worden opgesomd tussen accolades en gescheiden door een komma. De lengte van de *array* wordt automatisch bepaald door het aantal elementen tussen de accolades.

Indien de initialisatie niet op dezelfde regel als de declaratie gebeurt, moeten we volgende notatie gebruiken:

```
int[] numbers;  
numbers = new int[] {1,3,6,8,4};
```

We noemen dit een anonieme *array*.

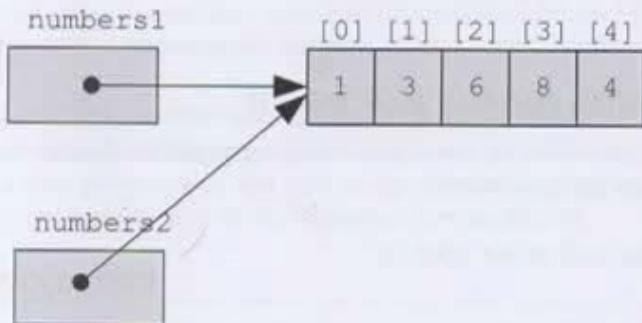
In dit geval mag de lengte van de *array* niet gespecificeerd worden: ze wordt namelijk bepaald door het aantal elementen.

Vermits *arrays* gewoon objecten zijn, is de variabele van dit type ook een referentie naar dit object. Dit impliceert dat we meerdere variabelen kunnen hebben die naar hetzelfde object of dezelfde *array* verwijzen.

Dit illustreren we in de volgende code:

```
int[] numbers1 = {1,3,6,8,4};  
int[] numbers2 = numbers1;
```

Schematisch:



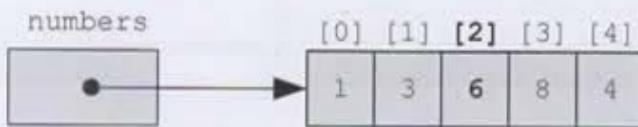
Afbeelding 70: Meerdere referenties naar eenzelfde array

Concreet wil dit ook zeggen dat een wijziging van de *array* via de ene referentievariabele dus ook gevolgen heeft voor de tweede referentievariabele.

7.3 Arrays gebruiken

We kunnen de elementen van een *array* gebruiken door de naam van de *array* (referentie) te nemen, gevolgd door vierkante haken met daartussen de index van het element dat we willen gebruiken: `arrayName[index]` vb. `numbers[2]`.

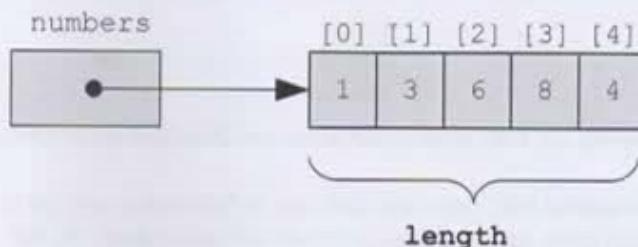
De index van een element van een *array* begint steeds met 0. `numbers[2]` is dus het derde element uit de *array*.



Afbeelding 71: Een element uit een array met de index aanduiden

```
int numbers = new int[5];
numbers[2] = 6;
System.out.println(numbers[2]);
```

Alle *arrays* hebben een eigenschap die het aantal elementen van de *array* aangeeft: `length` vb. `numbers.length`



Afbeelding 72: De lengte van een array

Om alle elementen in een *array* te overlopen kunnen we de volgende code gebruiken:

```
for(int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

7.4 De uitgebreide for-lus (for each)

Om makkelijk een verzameling te overlopen, kan men gebruikmaken van de uitgebreide for-lus, of ook wel *for each*-lus genoemd.

De syntax van deze lus ziet er als volgt uit:

```
for(type element: array)
    statement;
```

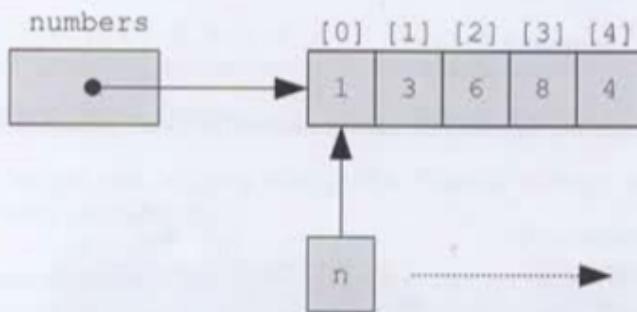
De variabele *element* neemt tijdens de iteratie een voor een de waarde aan van de elementen uit de *array*.

We illustreren dit met een voorbeeld:

```
int[] numbers = {1,2,3,4};
for(int n: numbers) {
    System.out.println(n);
}
```

Voor de initialisatie van de iteratievariabele kunnen we ook hier gebruikmaken van het type *var*:

```
int[] numbers = {1,2,3,4};
for(var n: numbers) {
    System.out.println(n);
}
```



Afbeelding 73: Een array overlopen met de uitgebreide for-lus

Een dergelijke lus kan evenwel niet gebruikt worden indien men ook de index nodig heeft tijdens de iteratie of indien men andere stapgroottes wil gebruiken. In dat geval moet men gebruikmaken van de klassieke *for*-lus.

Opdracht 1: Arrays gebruiken

- Maak een *array* van 20 getallen en vul deze met veelvouden van 7.
- Druk de *array* in volgorde af op het scherm met een *for each*-lus.

- Druk de *array* ook in omgekeerde volgorde af met een gewone *for-lus*.
- Maak een *array* van *boolean*-waarden en druk deze *array* af met een *for each-lus*.
- Optioneel: Pas het programma aan voor het berekenen van priemgetallen zodat de gevonden priemgetallen opgeslagen worden in een *array*. Maak deze *array* voldoende groot.
- Optioneel: Maak het programma voor de priemgetallen sneller door enkel te delen door de priemgetallen die reeds gevonden zijn.
- Optioneel: maak een programma dat een *array* getallen ordent van klein naar groot. Gebruik hiervoor de *bubble sort* of de *selection sort* methode.

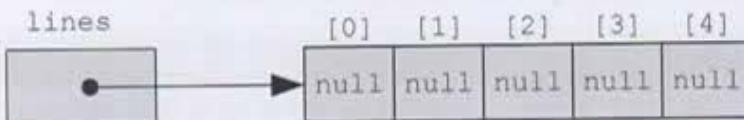
7.5 Arrays van objecten

Arrays kunnen zowel primitieve datatypen als referentietypen bevatten. In het laatste geval spreken we van een *array* van objecten, omdat de *array* een aantal verwijzingen naar objecten van hetzelfde type bevat.

In het volgende voorbeeld maken we een *array* van *String*-objecten:

```
String[] lines = new String[5];
```

In dit voorbeeld worden geen vijf *String*-objecten gecreëerd, maar wel een *array* van vijf referenties naar een *String*-object. Deze referenties worden geïnitialiseerd met de waarde **null**.



Afbeelding 74: Initialisatie van een array van Strings met null

We moeten daarom de vijf *String*-objecten afzonderlijk creëren en de referenties toekennen aan de elementen van de *array*:

```
String[] lines = new String[5];
lines[0] = "Hello World";
lines[1] = "Hello Mars";
lines[2] = "Hello Venus";
lines[3] = "Hello Jupiter";
lines[4] = "Hello Saturn";
```

Hoofdstuk 6: Objectgeoriënteerd programmeren

6.1 Inleiding

In dit hoofdstuk vangen we aan met het objectgeoriënteerd programmeren. We geven eerst een korte inleiding hierop om een aantal concepten en begrippen te duiden. Vervolgens leren we gebruik te maken van bestaande klassen en objecten die reeds aanwezig zijn in het Java-platform. In een volgend hoofdstuk zullen we dan leren hoe we zelf de code moeten schrijven voor de klassen.

6.2 Inleiding in het objectgeoriënteerd programmeren

Java is een objectgeoriënteerde programmeertaal. Om met Java aan de slag te kunnen gaan, is het onontbeerlijk om inzicht te hebben in de basisprincipes van het objectgeoriënteerd programmeren (OOP). In de volgende paragrafen doen we een aantal basisbegrippen van het objectgeoriënteerd programmeren uit de doeken. We introduceren hier dus enkel het begrippenkader dat gebruikt wordt bij OOP.

6.2.1 Objecten

Het objectgeoriënteerd programmeren gebruikt het object als basis. Softwareobjecten kunnen vergelijken met reële objecten of voorwerpen uit ons dagelijks leven. Zo is de auto waarmee ik rij een voorwerp.

Een auto heeft een aantal **kenmerken** zoals kleur, afmetingen, bouwjaar, cilinderinhoud, maximumsnelheid en noem maar op.

Daarnaast heeft een auto ook een bepaald **gedrag**: als ik op het gaspedaal duw, dan gaat hij sneller; als ik op het rempedaal duw, dan remt hij; als ik aan het stuur draai, dan verandert mijn auto van richting.

Met softwareobjecten is dat net hetzelfde. De kenmerken van een softwareobject noemt men **eigenschappen (properties)** en de gedragingen noemt men **methoden (methods)**.

De eigenschappen worden ondergebracht in variabelen¹ die verbonden zijn aan het object.

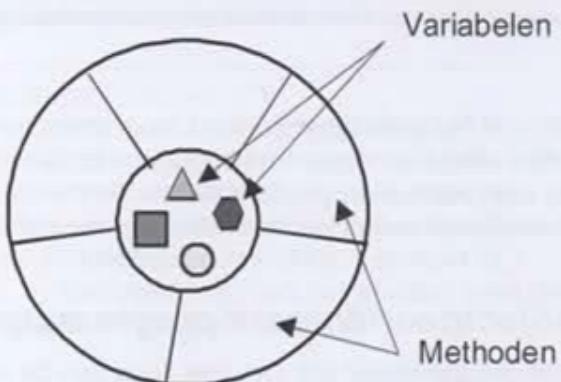
Methoden zijn codeblokken die verbonden zijn aan het object. De methoden bepalen de manier waarop ik met een object kan omgaan.

In het voorbeeld van de auto kan ik de auto van richting doen veranderen door aan het stuur te draaien; ik kan sneller rijden door op het gaspedaal te drukken enzovoort.

Definitie: Een object is een groepering van variabelen (eigenschappen) en gerelateerde codeblokken (methoden).

Men kan een object als volgt visueel voorstellen:

¹ De termen *eigenschap* en *variabele* worden om deze reden vaak door elkaar gebruikt. Ook in deze cursus wordt soms de term *eigenschap* gebruikt en op andere plaatsen de term *variabele*. Het gaat telkens om hetzelfde.



Afbeelding 49: Visuele voorstelling van een object

6.2.1.1 Private en publieke eigenschappen

Uit de tekening blijkt dat de eigenschappen in de kern van het object zitten, terwijl de methoden zich aan de buitenkant bevinden. Het uiteindelijke streefdoel van goed objectgeoriënteerd programmeren is de eigenschappen van een object af te schermen van de buitenwereld. Dit noemt men ook *inkapseling (encapsulation)*. De eigenschappen kunnen dan niet rechtstreeks benaderd worden van buitenaf. Men moet dan steeds passeren via een of andere methode om de inhoud van een eigenschap op te vragen of aan te passen. Hoewel het een streefdoel is om eigenschappen af te schermen, komt het toch vaak voor dat eigenschappen wel rechtstreeks benaderbaar zijn van buitenaf. Men maakt daarom een onderscheid tussen **publieke** eigenschappen en **private** eigenschappen. Publieke eigenschappen zijn wel van buitenaf rechtstreeks benaderbaar terwijl private eigenschappen alleen intern in het object gebruikt kunnen worden.

We nemen nogmaals het voorbeeld van de auto: het brandstofpeil is een publieke eigenschap, omdat iedere gebruiker op het dashboard deze eigenschap rechtstreeks kan aflezen. Bovendien kan iedere gebruiker het brandstofpeil veranderen door brandstof bij te tanken.

Het oliepeil daarentegen zou men kunnen beschouwen als een private eigenschap, omdat die niet zo eenvoudig te achterhalen is. Hiervoor moet men meestal de motorkap openmaken en met een peilstok het niveau van de olie meten.

De eigenschap oliepeil wordt wel intern door de auto zelf gebruikt. Als het peil te laag is, kan er een waarschuwingslampje gaan branden op het dashboard. Het oliepeil aanpassen door olie toe te voegen doet de gewone gebruiker meestal niet zelf, maar laat hij over aan een automechanicien.

In de Java-syntaxis worden eigenschappen aangegeven met de naam van het object gevolgd door een punt en de naam van de eigenschap.

objectName.property

Voorbeeld:

```
car.oilLevel  
car.fuelLevel
```

Deze eigenschappen kunnen als gewone variabelen gemanipuleerd worden. De variabele is in dit geval eigendom van een object.

6.2.1.2 Private en publieke methoden

Uit de tekening blijkt verder dat de methoden een schil vormen rond de eigenschappen. Via de methoden kan de buitenwereld met het object omgaan. Ook bij de methoden kan men onderscheid maken tussen **publieke** methoden en **private** methoden.

De **publieke** methoden staan ter beschikking van de buitenwereld om met het object om te gaan. De **private** methoden daarentegen kunnen alleen gebruikt worden door het object zelf. Het zijn als het ware methoden waarmee het object met zichzelf kan omgaan.

Het draaien aan het stuur van een auto is een publieke methode, de werking van het differentieel daarentegen zou men kunnen beschouwen als een private methode.

In de Java-syntax worden methoden aangegeven door de naam van het object gevolgd door een punt en de naam van de methode met twee ronde haken daarachter. Tussen de ronde haken kunnen parameters aan de methode worden meegegeven.

```
objectName.method()
```

Voorbeeld:

```
car.turn()  
car.accelerate()
```

De codeblokken verbonden aan een object kunnen via deze notatie opgeroepen worden.

6.2.1.3 Voordelen van OOP

Het objectgeoriënteerd programmeren heeft een aantal voordelen ten opzichte van het procedureel programmeren:

1. **Modulariteit:** De software wordt keurig opgedeeld in afzonderlijke modules (de objecten) met hun eigen variabelen en methoden. Dergelijke modules kunnen in verschillende programma's gebruikt worden. Een goed ontworpen module of object is herbruikbaar voor andere toepassingen.

Voorbeeld: ik kan mijn auto uitleen aan een vriend en die kan daar ook mee rijden.

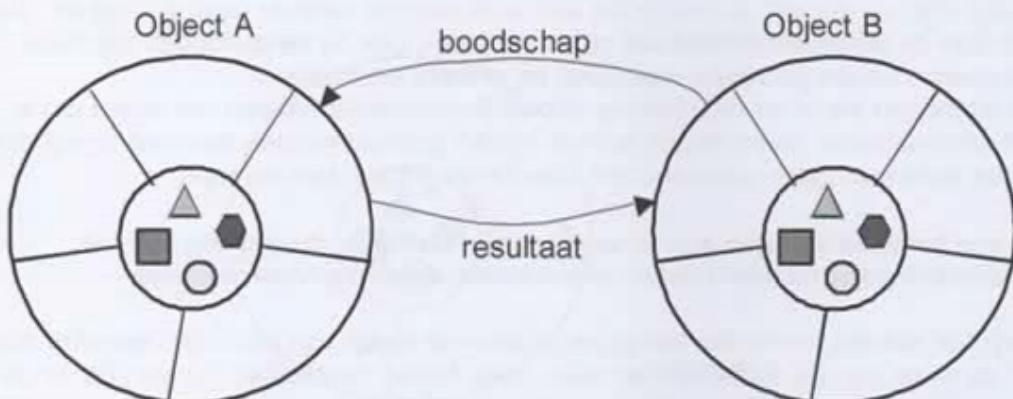
2. **Afscherming van informatie:** Ieder object heeft een interne structuur die afgeschermd is van de buitenwereld. De communicatie met die buitenwereld verloopt via een publieke interface (zie verder). Men kan dus vrij de interne opbouw of implementatie van het object veranderen zolang de publieke interface dezelfde blijft.

Voorbeeld: De autoconstructeur kan een geheel nieuw type motor in een auto steken terwijl de eindgebruiker nog steeds op dezelfde manier met de auto kan rijden. Men hoeft geen nieuw rijbewijs te halen.

6.2.2 Boodschappen

Een object maakt nog geen programma. Een programma bestaat meestal uit verschillende objecten die met elkaar samenwerken en communiceren. Deze communicatie verloopt via *boodschappen*. Het ene object stuurt een boodschap naar het andere object om dat object iets te laten doen. Na afloop kan het ontvangende object het resultaat van de actie terugsturen.

Het zenden van boodschappen verloopt via het aanroepen van methoden van een object. Dit zijn blokken code die verbonden zijn aan het object.



Afbeelding 50: Boodschappen tussen objecten

Boodschappen hebben vier kenmerken:

1. De **bestemming**: dit is het object waaraan de boodschap gericht is.
2. De **methode** van het object die wordt aangeroepen.
3. Bijkomende **parameters** die extra informatie geven over de boodschap (optioneel).
4. Het **resultaat** van de boodschap (optioneel).

We nemen nogmaals het voorbeeld van de auto. Als de bestuurder aan het stuur draait, dan verandert de auto van richting. We hebben hier een interactie tussen twee objecten: de bestuurder en de auto.

De bestuurder zendt een boodschap naar de auto. De bestemming van de boodschap is de 'auto', de methode die wordt aangeroepen is 'draaien' en de parameter is de 'hoek' waarover de bestuurder het stuur draait. Het resultaat van deze boodschap is dat de auto, inclusief bestuurder van richting verandert.

Het komt er eigenlijk op neer dat de bestuurder een stuk interne code van de auto laat uitvoeren, nl. de code om de richting van de auto te veranderen.

In de Java-syntaxis ziet deze boodschap er als volgt uit:

```
direction = car.turn(angle);
           ↑       ↑       ↑
           |       |       |
           +-----+       parameter
                   |
                   method
                   |
                   +-----+   adressee
                           |
                           result
```

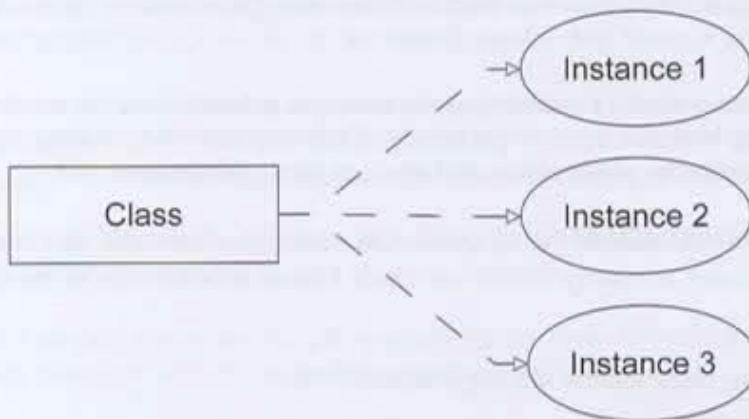
We hebben net de interactie tussen bestuurder en auto beschreven; maar ook in de auto zelf is er voortdurend interactie tussen verschillende objecten. Zo zal het stuur de informatie die het bekomt doorgeven aan de wielen. Het stuur-object stuurt dus een boodschap naar het wiel-object. De auto kunnen we dus ook beschouwen als een verzameling objecten die onderling boodschappen sturen.

6.2.3 Klassen

Veel voorwerpen uit ons dagelijks leven zijn van dezelfde soort. Het voorwerp waarmee mijn buurman 's morgens naar zijn werk rijdt, is ook een auto, maar het is niet mijn auto. Beide voorwerpen behoren tot de *klasse* auto. Ze zijn gemaakt met hetzelfde grondplan, op basis van dezelfde blauwdruk.

Definitie: Een klasse is een blauwdruk die de eigenschappen en methoden van objecten van dezelfde soort bepaalt.

De autoconstructeur gebruikt deze blauwdruk om een hele reeks identieke auto's te fabriceren. Die afzonderlijke auto's zijn concrete realisaties van die blauwdruk; het zijn concrete objecten die gebouwd zijn volgens hetzelfde grondschaal. Objecten noemt men ook instanties (*instances*) van een klasse.



Afbeelding 51: Klassen en instanties

Door het onderscheid tussen klassen en objecten kan men ook een onderscheid maken tussen het soort variabelen en methoden:

1. *Instance*-variabelen en methoden
2. Klassenvariabelen en methoden

In de klasse *auto* wordt bijvoorbeeld bepaald dat iedere auto een bepaalde kleur heeft. De concrete kleur kan verschillend zijn voor iedere instantie van de klasse, voor ieder afzonderlijk object van die klasse. De kleur noemt men daarom een ***instance variable***. Het is een eigenschap van het concrete object van een bepaalde klasse.

Nu zijn er ook eigenschappen die gemeenschappelijk zijn voor alle objecten van eenzelfde klasse. Het aantal wielen is bijvoorbeeld vier voor alle auto's van een bepaalde klasse. Deze eigenschap noemt men daarom een ***class variable*** (klassenvariabele).

Hetzelfde geldt voor methoden: sommige methoden zijn ***instance methods*** en sommige methoden zijn ***class methods***. *Instance methods* zijn gekoppeld aan een concreet object van een bepaalde klasse terwijl *class methods* eerder te maken hebben met de klasse op zich. De methode `turn()` bijvoorbeeld is een *instance-methode* omdat ze betrekking heeft op een welbepaald auto-object.

Een en ander zal duidelijker worden in het verdere verloop van deze cursus.

6.3 Werken met bestaande objecten

6.3.1 Inleiding

Java-programma's bestaan uit een aantal objecten die met elkaar samenwerken. Dit hoofdstuk behandelt de creatie en het gebruik van objecten. We maken hier gebruik van bestaande klassen die reeds aanwezig zijn in het Java-platform. Voorlopig gaan we nog geen klassen zelf schrijven; dit is voor het volgende hoofdstuk.

6.3.2 Objecten maken van een bestaande klasse

Objecten worden gemaakt op basis van een blauwdruk. Deze blauwdruk is de klasse waartoe het object behoort.

Vooraleer een object gemaakt kan worden, moet dus eerst die klasse gedefinieerd worden. Het definiëren van klassen wordt in een volgend hoofdstuk beschreven. In dit hoofdstuk gaan we gebruikmaken van klassen die reeds gedefinieerd zijn en die we gewoon in ons programma kunnen gebruiken.

In de JDK zijn reeds een hele reeks klassen gedefinieerd die gebruikt kunnen worden. Wij zullen in ons voorbeeld de klasse `Random` gebruiken. Objecten van deze klasse zijn in staat willekeurige (*random*) getallen te produceren: het zijn *random*-generatoren.

Deze klasse bevindt zich in het pakket `java.util`. De volledige naam van de klasse is daarom `java.util.Random` en de *bytecode* van deze klasse bevindt zich in het bestand `Random.class`.

We kunnen een object van deze klasse als volgt maken:

```
java.util.Random rand = new java.util.Random();
```

We creëren hier een *random*-object op basis van de klasse `Random`.

Deze programmeerregel bevat drie onderdelen:

1. De creatie van het object.
2. De initialisatie van het object.
3. De declaratie en initialisatie van de referentievariabele.

6.3.2.1 De creatie van het object

De creatie gebeurt met de `new` operator gevolgd door de naam van de klasse. Zo maken we een nieuw object van die klasse. Men noemt dit ook wel 'instantiatie', het creëren van een instantie van een klasse.

Tijdens de creatie wordt er geheugen gereserveerd voor het object. Het resultaat van de creatie is een verwijzing (referentie) naar het object in het geheugen.

6.3.2.2 De initialisatie van het object

De naam van de klasse wordt gevolgd door ronde haken met eventueel een aantal parameters ertussen, bijvoorbeeld: `Random(10)`. Met deze parameters wordt het nieuwe object geïnitialiseerd.

In feite wordt er een speciale methode van het object aangeroepen, namelijk de constructor. De constructor is een methode die dezelfde naam draagt als de klasse en die enkel gebruikt wordt om het object tijdens de creatie te initialiseren. Iedere klasse heeft een of meerdere

constructors die van elkaar verschillen in het aantal en type van de parameters. Een object kan daardoor op verschillende manieren geïnitialiseerd worden. Concreet wil dit zeggen dat bij het aanmaken van het object dit object al een aantal gegevens meekrijgt om zijn interne gegevens vorm te geven. Het zijn startwaarden die worden meegegeven.

In de onderstaande tabel worden de constructors van de klasse weergegeven:

Constructor	Beschrijving
Random()	Creëert een nieuwe <i>random</i> -generator met willekeurige <i>seed</i> .
Random(long seed)	Creëert een nieuwe <i>random</i> -generator met de opgegeven <i>seed</i> .

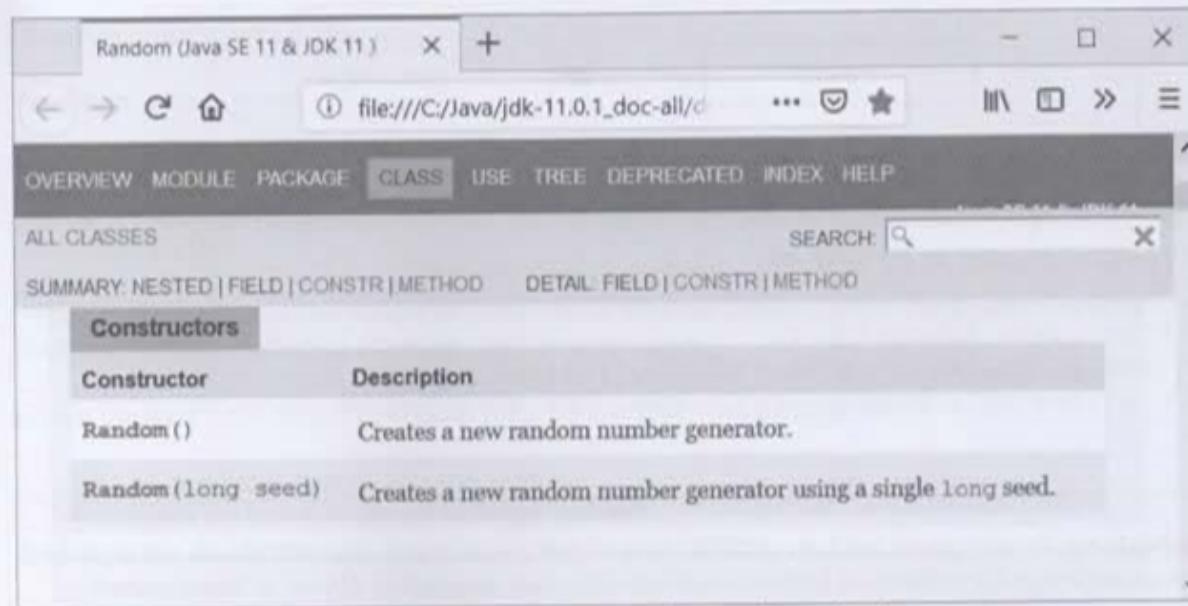
Tabel 20: Constructors van de klasse Random

De *seed* is een getal dat gebruikt wordt bij de berekening van de opeenvolgende willekeurige getallen. *Random*-generatoren met dezelfde *seed* zullen eenzelfde reeks willekeurige getallen produceren en zijn in die zin voorspelbaar.

Indien we de constructor zonder argumenten gebruiken, zal intern een willekeurige *seed* gebruikt worden zodat het hoogstwaarschijnlijk is dat twee *random*-generatoren die gecreëerd zijn met deze constructor toch niet dezelfde reeks willekeurige getallen opleveren.

De betekenis van die *seed* is hier niet zo belangrijk, wel het feit dat we een startwaarde aan een object kunnen meegeven.

We kunnen hier ook al een eerste blik werpen op de Java-API-documentatie. Die geeft namelijk een overzicht van de mogelijke constructors:



Afbeelding 52: API-documentatie van de constructors

6.3.2.3 De declaratie en initialisatie van de referentievariabele

Tijdens de creatie van het object wordt er geheugen gereserveerd voor het object. Het resultaat van de creatie is een verwijzing naar dit nieuwe object in het geheugen. Die verwijzing hebben we nodig om dit object later in ons programma te kunnen gebruiken. Deze verwijzing kennen we toe aan een variabele van het *referentietype*.

We hebben reeds vermeld dat variabelen op de volgende manier gedeclareerd worden:

```
type name = initialValue;
```

Het referentietype wordt op dezelfde manier gedeclareerd. Het type is in dit geval geen primitief datatype maar de naam van een klasse. In ons voorbeeld is dat de klasse `java.util.Random`. De initiële waarde is de referentie naar het object dat gecreëerd werd.

We hebben dus een *statement* met tegelijkertijd een declaratie en een objectcreatie. In dit geval kunnen we het datatype van de referentievariabele ook laten afleiden door de compiler:

```
var name = initialValue;
```

Concreet:

```
var rand = new java.util.Random();
```

We zouden dit ook in twee stappen kunnen doen:

```
java.util.Random rand; // Declaration of the reference variable  
rand = new java.util.Random(); // Creation and initialisation
```

In dit geval kunnen we geen gebruikmaken van `var` en moeten we het juiste datatype gebruiken bij de declaratie van de referentievariabele.



Afbeelding 53: Een object en een referentie naar het object

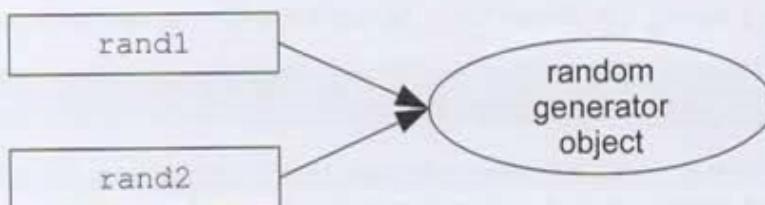
Ons programma ziet er als volgt uit:

```
public class RandomApp {  
    public static void main(String[] args) {  
        java.util.Random rand = new java.util.Random();  
    }  
}
```

Opmerking: In ons geval hebben we één referentievariabele die naar het object verwijst. Het is evenwel mogelijk meerdere referentievariabelen naar eenzelfde object te laten wijzen.

```
java.util.Random rand1; // Declaration of the reference variable  
java.util.Random rand2; // Declaration of the reference variable  
rand1 = new java.util.Random(); // Creation and initialisation  
rand2 = rand1;
```

Een referentievariabele die naar geen enkel object verwijst, heeft de waarde `null`.



Afbeelding 54: Meerdere referenties naar eenzelfde object

6.3.2.4 Pakketten importeren

De klasse Random behoort tot het pakket `java.util`. De volledige naam van de klasse is dus `java.util.Random`. Het gebruik van de volledige naam kan echter vrij omslachtig zijn, vooral als de klassennaam vaak wordt gebruikt. Men kan daarom ook de korte naam van de klasse gebruiken. Om verwarring met andere klassen te vermijden, **importeert** men eerst de desbetreffende klasse met zijn volledige naam. Dit importeren gebeurt in het begin van het broncodebestand. Men kan vervolgens in het broncodebestand gebruikmaken van de korte naam van de klasse.

```

import java.util.Random;
...
Random rand = new Random();
...
  
```

Indien meerdere klassen uit eenzelfde pakket geïmporteerd worden, kan men ook het **volledige pakket importeren**. Men gebruikt daarvoor het jokerteken *:

```

import java.util.*;
...
Random rand = new Random();
...
  
```

Hierdoor importeert men alle klassen van het pakket en kan men ze allemaal met hun korte naam aanduiden.

Indien men meerdere pakketten importeert die klassen met dezelfde korte naam bevatten, is men verplicht de volledige klassennaam te gebruiken.

Bij het zoeken naar de juiste klasse volgt de compiler deze volgorde:

1. Eerst de **explicit** geïmporteerde klassen.
2. Vervolgens de klassen uit het **huidige pakket**.
3. Ten slotte de klassen uit de **geïmporteerde pakketten**.

De Java-compiler importeert automatisch een aantal pakketten:

1. **Het standaardpakket**. Indien we een klasse maken en daarbij geen pakket definiëren, wordt deze klasse opgenomen in het standaardpakket. Alle klassen zonder specifieke pakketnaam behoren tot dit standaardpakket. Men kan best geen gebruik maken van dit standaardpakket, tenzij voor tijdelijk gebruik.
2. **Het huidige pakket**. Indien we een klasse als lid van een pakket definiëren, wordt dit pakket automatisch geïmporteerd.

3. Het `java.lang` pakket. Dit pakket bevat de basisklassen die deel uitmaken van de Java *language*.

6.3.3 Objecten gebruiken

Eens het object gecreëerd is, kunnen we het gebruiken in ons programma. Uit de inleiding over objectgeoriënteerd programmeren weten we dat objecten eigenschappen en methoden hebben. We hebben die toen onderverdeeld in **private** en **publieke** eigenschappen en methoden.

We kunnen op dit moment enkel gebruikmaken van de publieke eigenschappen en methoden.

6.3.3.1 Publieke eigenschappen gebruiken

De *random*-generator heeft geen publieke eigenschappen.

Dit is iets wat we vaak zullen tegenkomen en heeft te maken met het principe van *data hiding* dat in objectgeoriënteerd programmeren gebruikelijk is. Interne gegevens worden verborgen voor de buitenwereld, zodat ze niet zomaar rechtstreeks aangepast kunnen worden. Dit laatste zou namelijk de werking van het object kunnen verstoren. Zodra we zelf klassen gaan schrijven, zal het duidelijker worden waarom dat zo is.

6.3.3.2 Publieke methoden gebruiken

Naast publieke eigenschappen kan men ook publieke methoden van een object gebruiken. Methoden worden gebruikt om boodschappen naar objecten te sturen en ze bepaalde acties te laten uitvoeren. In feite wordt er bij het aanroepen van een methode een stukje code van het object uitgevoerd. We geven hier eventueel wat extra gegevens mee en na afloop krijgen we eventueel nieuwe gegevens terug waar we dan verder iets mee kunnen doen.

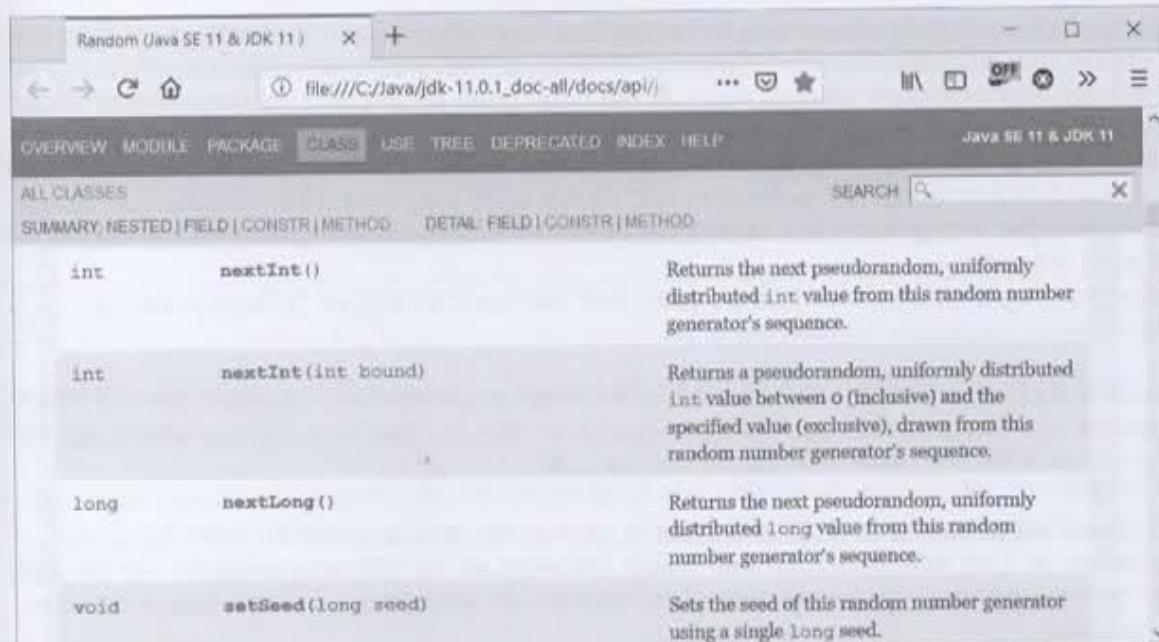
De methoden van een object worden aangeroepen door de naam van het object (van de referentievariabele) gevolgd door een punt en de naam van de methode, gevolgd door ronde haken met daartussen eventueel een aantal parameters. Onze *random*-generator heeft onder andere de volgende publieke methoden:

Return-type	Methode	Beschrijving
boolean	<code>nextBoolean()</code>	Geeft de volgende willekeurige boolean.
int	<code>nextInt()</code>	Geeft de volgende willekeurige integer.
int	<code>nextInt(int bound)</code>	Geeft de volgende willekeurige integer tussen 0 en de opgegeven waarde (<i>bound</i>).
long	<code>nextLong()</code>	Geeft de volgende willekeurige long.
float	<code>nextFloat()</code>	Geeft de volgende willekeurige float.
double	<code>nextDouble()</code>	Geeft de volgende willekeurige double.

Tabel 21: Methoden van de klasse Random

Hierbij wordt telkens aangegeven wat het datatype van de parameters is en wat het datatype van de teruggegeven waarde is. Indien er geen waarde wordt teruggegeven, is het datatype `void`.

In de Java-API-documentatie zien we onder andere dit:



The screenshot shows the Java API documentation for the `Random` class. The `CLASS` tab is selected. The methods listed are:

- `int nextInt()`: Returns the next pseudorandom, uniformly distributed `int` value from this random number generator's sequence.
- `int nextInt(int bound)`: Returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
- `long nextLong()`: Returns the next pseudorandom, uniformly distributed `long` value from this random number generator's sequence.
- `void setSeed(long seed)`: Sets the seed of this random number generator using a single `long` seed.

Afbeelding 55: API-documentatie van de eigenschappen van de klasse Random

Methoden kunnen als volgt aangeroepen worden:

```
Random rand = new Random();
int value1 = rand.nextInt();
int value2 = rand.nextInt(100);
```

In de eerste methodeaanroep geven we geen waarde mee. We schrijven wel de ronde haken om aan te geven dat we een methode oproepen. Na afloop krijgen we een willekeurig getal terug.

Bij de twee methodeaanroep geven we wel een waarde mee. We geven hiermee aan dat de willekeurig waarde moet liggen tussen 0 en de opgegeven waarde, deze waarde zelf niet meegerekend. We krijgen dan een willekeurige waarde terug die tussen deze grenzen ligt.

Opdracht 1: Objecten maken en gebruiken

In deze opdracht gaan we de klasse `Random` gebruiken.

- Maak een nieuw programma met de naam `RandomApp`.
- Importeer het pakket `java.util`.
- Maak een `random`-generator en vraag verschillende willekeurige integers op. Druk deze af op het scherm.
- Vraag ook verschillende willekeurige waarden op die liggen tussen een bepaalde grens.
- Maak een nieuw programma dat zes willekeurige getallen berekent tussen 0 en 45 (Lotto).

6.3.4 Objecten opruimen

Wanneer objecten in een programma gecreëerd worden, wordt er geheugen gereserveerd. Objecten die niet meer gebruikt worden, moeten opgeruimd worden zodat dit geheugen weer vrijkomt en gebruikt kan worden voor andere doeleinden. Zo voorkomt men dat het programma onnodig veel geheugen gaat gebruiken.

Bij programmeertalen zoals C/C++ moet de programmeur nauwgezet bijhouden welke

objecten niet meer gebruikt worden en het gereserveerde geheugen explicit weer vrijgeven. Dit vraagt veel discipline van de programmeur. Programma's die niet consequent het geheugen weer vrijgeven, werken meestal wel maar gaan bij langdurig gebruik steeds meer geheugen gebruiken. Bij gebrek aan RAM-geheugen wordt uiteindelijk het virtueel geheugen (wisselbestand) aangesproken waardoor het systeem enorm zal vertragen. We spreken in dit geval van *memory leaks*. Het geheugen lekt als het ware weg omdat een programma steeds meer geheugen vraagt van het besturingssysteem en het niet (meer) gebruikte geheugen niet meer vrijgeeft.

Dergelijke programmeerfouten komen vaak zeer laat aan het licht en hebben al menig programmeur slapeloze nachten bezorgd.

In Java is de programmeur ontheven van deze verantwoordelijkheid. Objecten die niet meer in gebruik zijn, worden door de JVM zelf opgeruimd. Het mechanisme dat daarvoor zorgt, is de *garbage collection*; de vuilniswagen van de JVM.

Een object wordt beschouwd als niet meer in gebruik als iedere referentie naar dat object verdwenen is. Een referentie naar een object verdwijnt als de referentievariabele die verwijst naar het object buiten zijn bereik gaat of als expliciet de waarde `null` wordt toegekend aan de variabele.

Objecten waarnaar geen enkele referentievariabele meer verwijst, worden gemarkeerd en nadien opgeruimd tijdens de *garbage collection*.

De *garbage collection* gebeurt in een afzonderlijke *thread* die wordt uitgevoerd met lage prioriteit. Dit wil zeggen dat deze activiteit op de achtergrond gebeurt en nagenoeg geen invloed heeft op de algemene prestaties van het programma.

Normaal gezien hoeft de programmeur zich niets aan te trekken van de *garbage collection*. Deze doet zijn werk autonoom op tijdstippen dat het systeem niets te doen heeft (*idle time*). Het is nochtans mogelijk de *garbage collection* een extra duwtje te geven door het aanroepen van de volgende methode: `System.gc()`.

Dit is echter geen garantie dat de *garbage collection* onmiddellijk gebeurt, maar de kans is wel groter.

6.4 Tekenreeksen

6.4.1 Inleiding

We behandelen nu enkele andere voorbeelden van klassen die veel gebruikt worden binnen Java, namelijk klassen voor het opslaan en bewerken van tekst, ook wel tekenreeksen genoemd. De klassen die hiervoor gebruikt worden zijn `String` en `StringBuilder`.

6.4.2 De klasse String

De `String`-klasse wordt gebruikt voor onveranderlijke tekenreeksen. Deze klasse maakt deel uit van het pakket `java.lang`. Vermits dit pakket standaard altijd geïmporteerd wordt, hoeven we in de code hiervoor geen extra `import` toe te voegen.

Bij de creatie van het object wordt een initiële waarde aan de `string` toegekend die nadien niet meer veranderd kan worden.

Een voorbeeld:

```
String text = new String("Hello World!");
```

We creëren hier een object van de klasse `String` en we kennen via de constructor

onmiddellijk de waarde "Hello World!" toe. De string wordt geïnitialiseerd door middel van een *string literal*. *String literals* bestaan uit een reeks karakters tussen dubbele aanhalingstekens.

De variabele `text` is een referentie naar het object.

We kunnen vervolgens deze string op het scherm afdrukken:

```
System.out.println(text);
```

De methode `System.out.println()` aanvaardt als parameter namelijk een object van het type `String`.

In de vorige oefeningen hebben we steeds de tekst onmiddellijk als parameter meegegeven:

```
System.out.println("Hello World!");
```

Op de achtergrond creëert Java een nieuw `String`-object en initialiseert het met de waarde "Hello World!". Dit object wordt vervolgens als parameter aan de methode meegegeven.

Vermits Java elke *string literal* automatisch omzet in een object van de klasse `String`, kan men een *string* ook als volgt declareren:

```
String text = "Hello World!";
```

Er wordt hier een `String`-object gecreëerd en de referentie hiernaar wordt toegekend aan de variabele `text`.

Aangezien een *string literal* behandeld kan worden als een `String`-object, kan men gebruikmaken van de eigenschappen en methoden van het `String`-object.

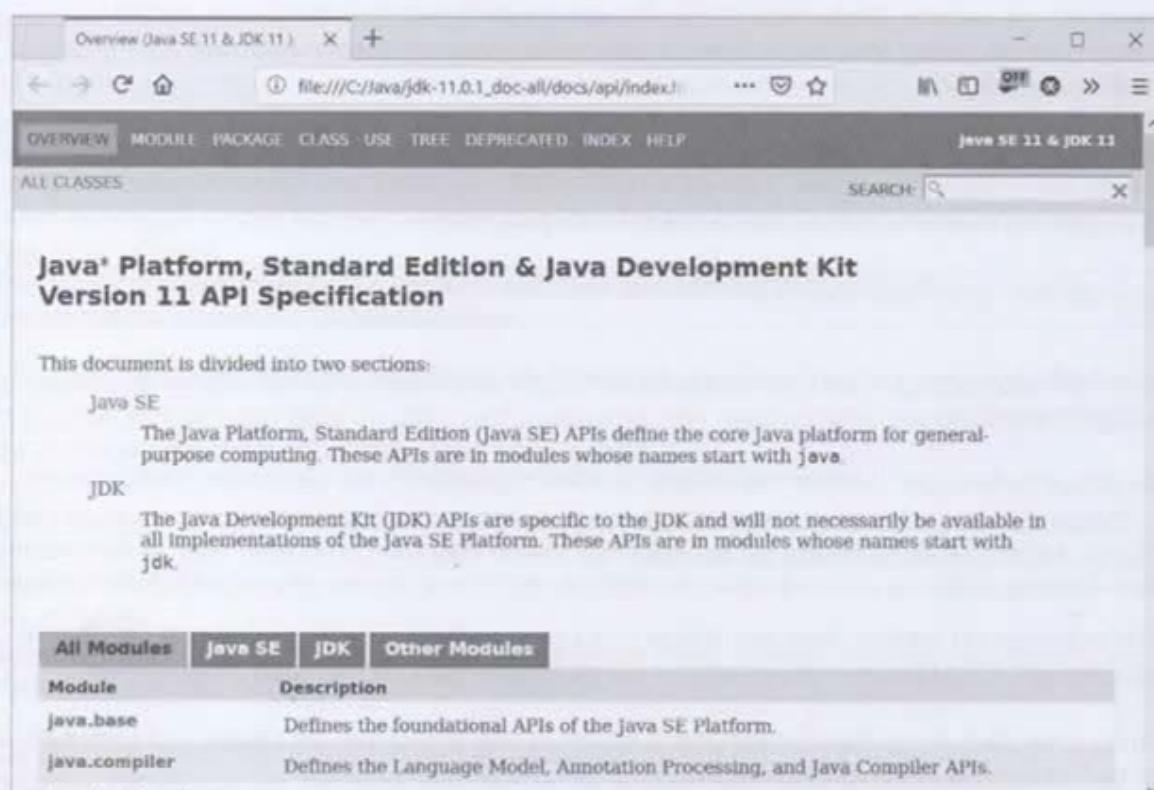
Net als ieder ander object heeft een `String`-object een aantal publieke variabelen en methoden die men kan gebruiken.

De beschrijving van alle publieke variabelen en methoden vindt men in de documentatie van de Java-API.

Opdracht 2: De Java-API-documentatie openen

In deze opdracht openen we de Java-API-documentatie en zoeken we de beschrijving van de klasse `String`.

- Open het bestand ..\docs\api\index.html



**Java® Platform, Standard Edition & Java Development Kit
Version 11 API Specification**

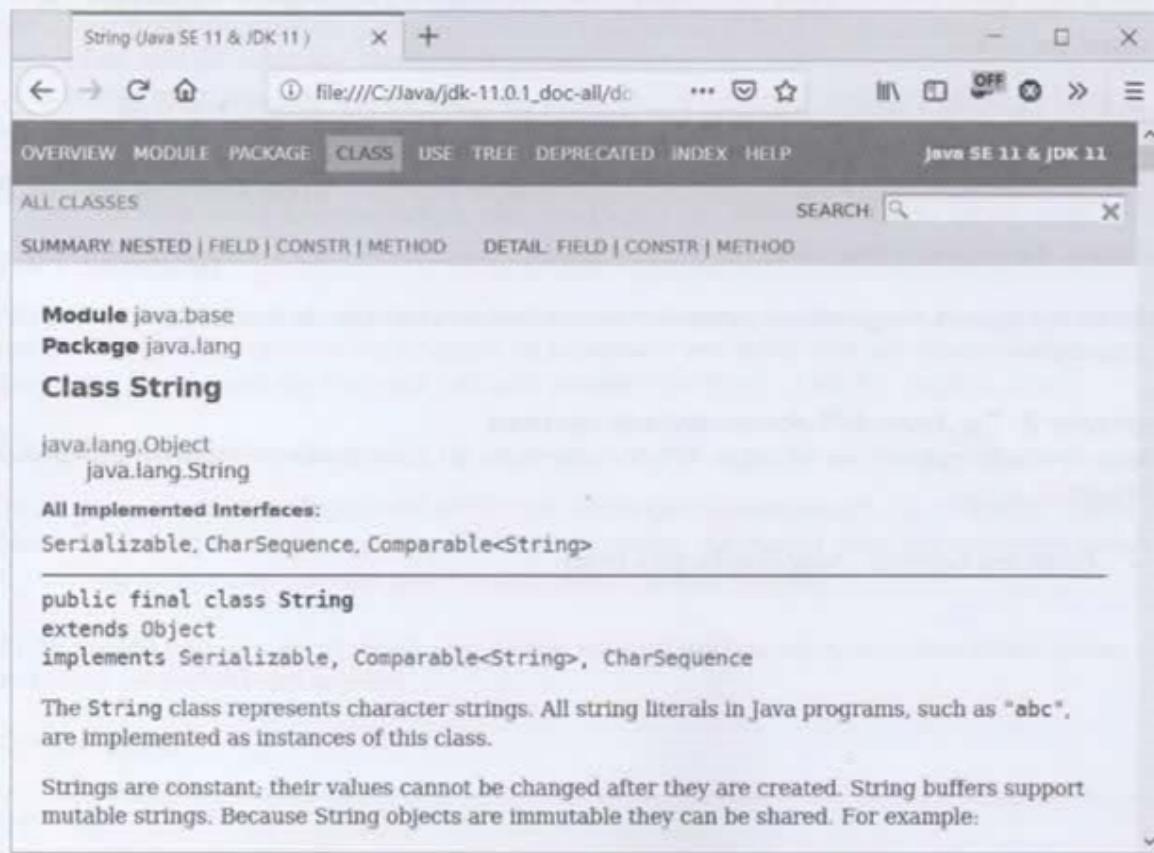
This document is divided into two sections:

- Java SE**
The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.
- JDK**
The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules	Java SE	JDK	Other Modules
Module	Description		
<code>java.base</code>	Defines the foundational APIs of the Java SE Platform.		
<code>java.compiler</code>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.		

Afbeelding 56: API-documentatie

- Tik in het zoekvenster rechtsboven de tekst **String** en ga vervolgens naar de gesuggereerde klasse `java.lang.String`.



String (Java SE 11 & JDK 11)

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP **java SE 11 & JDK 11**

ALL CLASSES **SEARCH:**

Module `java.base`
Package `java.lang`

Class String

`java.lang.Object`
`java.lang.String`

All Implemented Interfaces:
`Serializable, CharSequence, Comparable<String>`

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because `String` objects are immutable they can be shared. For example:

Afbeelding 57: API-documentatie van de klasse `String`

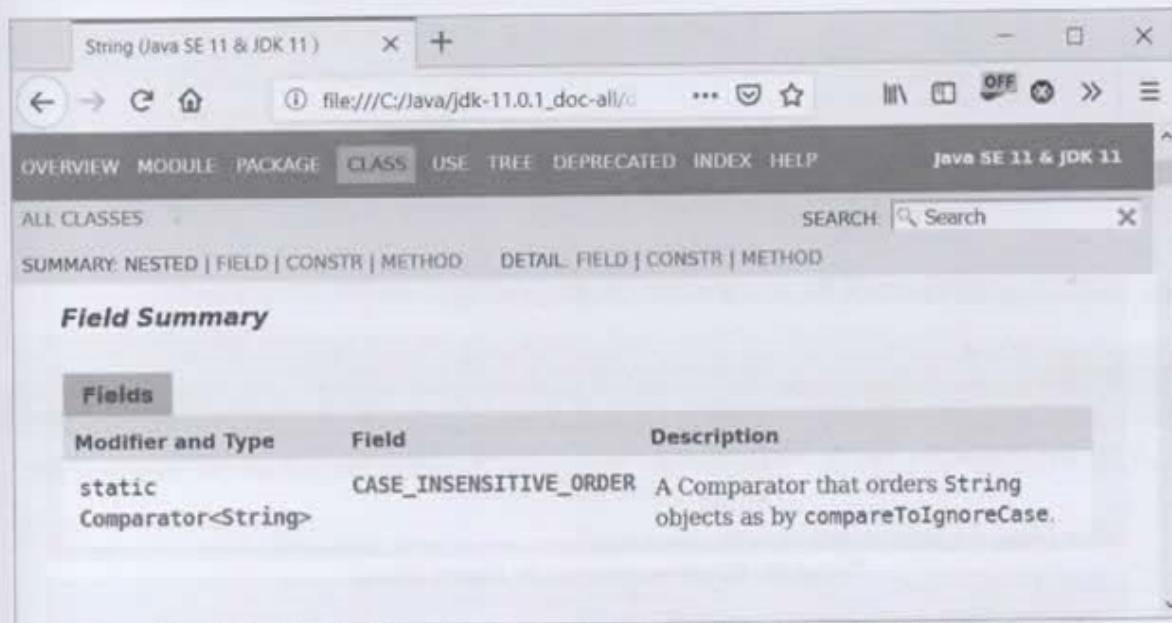
- Overloop de uitleg van de `String`-klasse. Het is op dit moment nog niet nodig alle details van deze uitleg te begrijpen. Naarmate de cursus verdergaat, zullen meer details uitgelegd worden.

Op deze pagina vindt men de volledige beschrijving van de klasse `String`. Deze beschrijving bevat de volgende onderdelen:

6.4.2.1 Eigenschappen

De tabel **Field Summary** bevat de beschrijving van alle eigenschappen van de `String`-klasse.

In de eerste kolom wordt het datatype van de eigenschap gegeven; in de tweede kolom de beschrijving en de naam van de eigenschap.



The screenshot shows the Java API documentation for the `String` class. The title bar says "String (Java SE 11 & JDK 11)". The navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is selected), USE, TREE, DEPRECATED, INDEX, and HELP. The right side of the bar says "java SE 11 & JDK 11". Below the navigation bar, there are tabs for ALL CLASSES, SUMMARY, NESTED, FIELD, CONSTR, and METHOD. The main content area is titled "Field Summary" and contains a table with the following data:

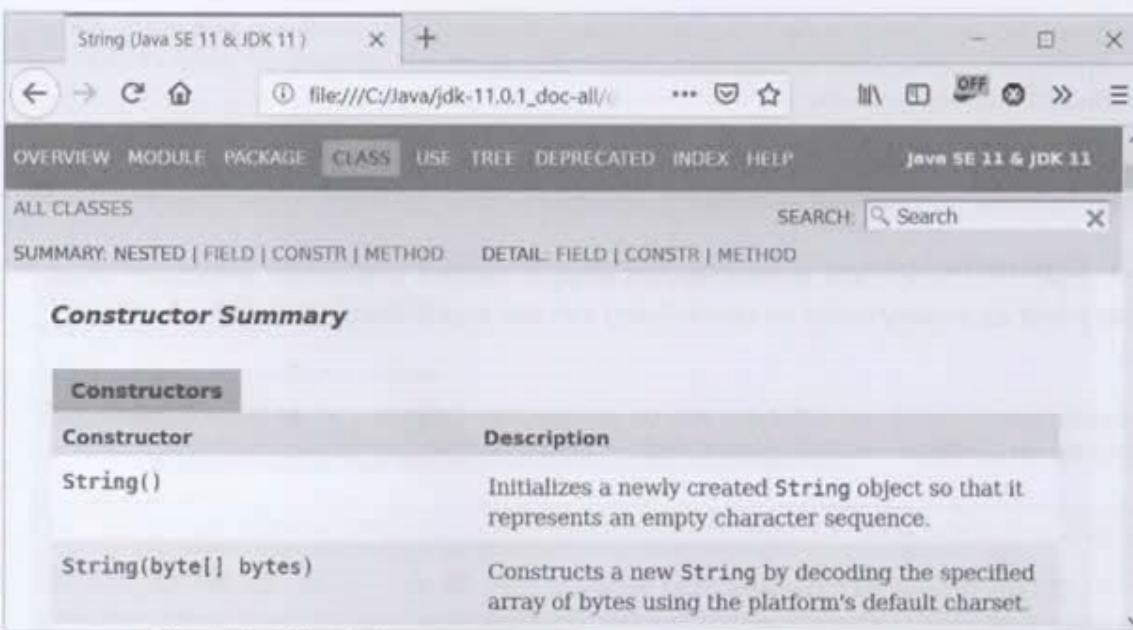
Modifier and Type	Field	Description
static <code>Comparator<String></code>	<code>CASE_INSENSITIVE_ORDER</code>	A Comparator that orders <code>String</code> objects as by <code>compareToIgnoreCase</code> .

Afbeelding 58: API-documentatie van de eigenschappen van de klasse `String`

Ook hier stellen we vast dat de klasse, op één speciale uitzondering na, geen publieke eigenschappen heeft. We laten de vermelde eigenschap voorlopig voor wat ze is en onthouden vooral dat klassen doorgaans geen publieke eigenschappen hebben vanwege de *data hiding*. Zo is dat ook met de klasse `String`.

6.4.2.2 Constructors

De tabel **Constructor Summary** geeft een beschrijving van alle mogelijke constructors. Een `String`-object kan namelijk op verschillende manieren gecreëerd worden. Naargelang het aantal en type van de parameters wordt automatisch de juiste constructor aangeroepen om het object te initialiseren.



The screenshot shows the Java API documentation for the `String` class. The title bar says "String (Java SE 11 & JDK 11)". The navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is selected), USE, TREE, DEPRECATED, INDEX, and HELP. A search bar at the top right contains the text "Search". Below the navigation bar, there are tabs for ALL CLASSES, SUMMARY, NESTED, FIELD, CONSTR, and METHOD. The main content area is titled "Constructor Summary" and contains a table with two rows. The first row has a header "Constructors". The second row shows two constructors: `String()` and `String(byte[] bytes)`. The `String()` constructor is described as initializing a new `String` object to an empty sequence. The `String(byte[] bytes)` constructor is described as constructing a new `String` by decoding the specified array of bytes using the platform's default charset.

Afbeelding 59: API-documentatie van de constructors van de klasse String

In de onderstaande tabel sommen we twee constructors op:

Constructor	Beschrijving
<code>String()</code>	Maakt een lege <code>String</code> .
<code>String(String original)</code>	Maakt een <code>String</code> met dezelfde inhoud als de meegegeven <code>String</code> .

Tabel 22: Constructors van de klasse String

De laatste constructor hebben we gebruikt bij het voorbeeld:

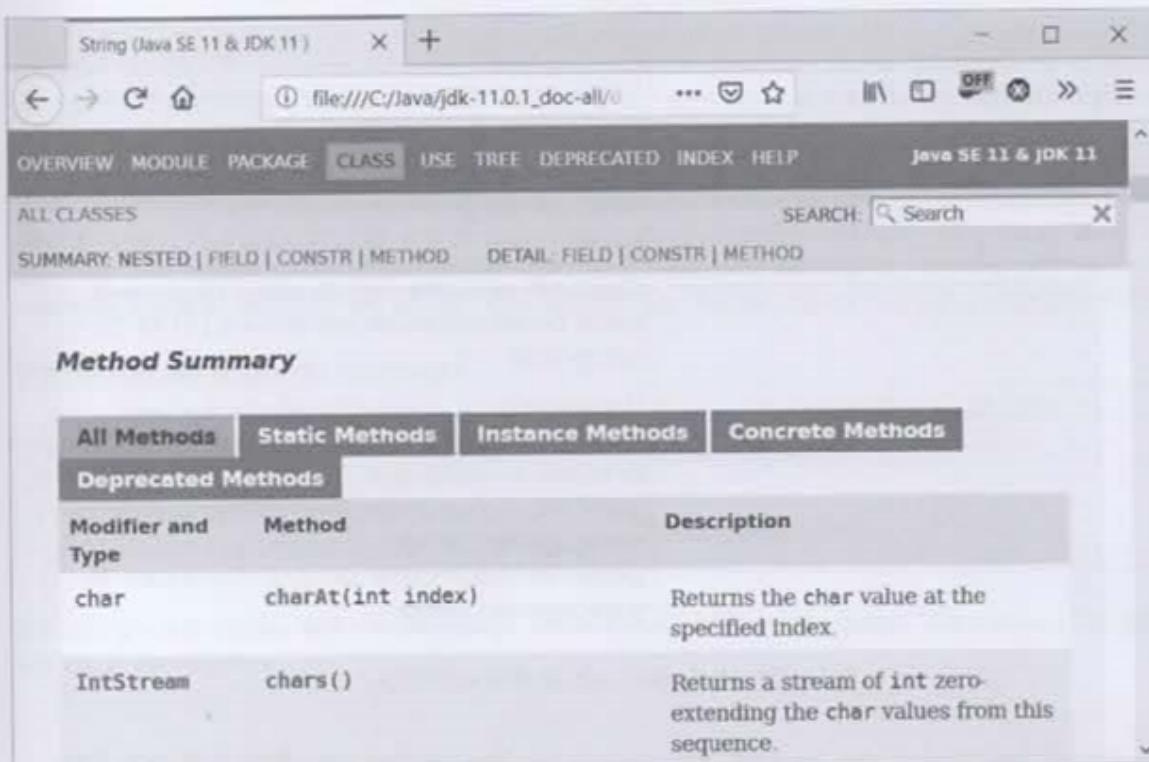
```
String text = new String("Hello World!");
```

We geven hier een *string literal* mee en creëren een nieuwe *string* met dezelfde inhoud.

De eerste constructor maakt een lege *string*. Vermits we die *string* nadien toch niet kunnen wijzigen, heeft deze constructor niet onmiddellijk nut.

6.4.2.3 Methoden

De tabel **Method Summary** geeft een beschrijving van alle methoden van de klasse. Voor iedere methode wordt de naam, de beschrijving en het aantal en het type van de parameters gegeven. In de eerste kolom wordt de resulterende waarde vermeld. Als deze `void` is, wil dit zeggen dat de methode geen resulterende waarde teruggeeft.



Modifier and Type	Method	Description
char	charAt(int index)	Returns the char value at the specified index.
IntStream	chars()	Returns a stream of int zero-extending the char values from this sequence.

Afbeelding 60: API-documentatie van de methoden van de klasse String

In de onderstaande tabel geven we een aantal interessante methoden weer:

Return	Methode	Beschrijving
char	charAt(int index)	Geeft het karakter op een bepaalde positie in de string.
String	concat(String str)	Plakt twee strings aan elkaar en geeft een nieuwe string terug.
boolean	equals(Object o)	Vergelijkt twee strings inhoudelijk en geeft true terug indien ze hetzelfde zijn.
int	indexOf(int ch)	Geeft de eerste positie van het opgegeven karakter in de string.
int	indexOf(int ch, int fromIndex)	Geeft de positie van het opgegeven karakter vanaf de opgegeven startpositie (fromIndex).
int	indexOf(String str)	Geeft de eerste positie van de opgegeven string binnen deze string.
int	length()	Geeft het aantal karakters in de string.
String	replace(char o, char n)	Geeft een nieuwe string terug waarbij overal het karakter o door het karakter n vervangen is.
String	toLowerCase()	Geeft een nieuwe string terug met dezelfde tekst in kleine letters.
String	toUpperCase()	Geeft een nieuwe string terug met dezelfde tekst in hoofdletters.
boolean	startsWith(String s)	Geeft aan of de string begint met een bepaalde tekenreeks.
boolean	endsWith(String s)	Geeft aan of de string eindigt met een bepaalde tekenreeks.

Return	Methode	Beschrijving
String	trim()	Geeft een nieuwe <i>string</i> terug waarbij lege ruimte voor en na is afgeknipt.
String	substring(int s)	Geeft het gedeelte van de <i>string</i> beginnend vanaf de aangegeven startpositie (<i>s</i>) tot het einde.
String	substring(int s, int e)	Geeft het gedeelte van de <i>string</i> beginnend vanaf de aangegeven startpositie (<i>s</i>) tot de eindpositie (<i>e</i>).
int	compareTo(String o)	Vergelijkt deze <i>string</i> alfabetisch met een andere en geeft volgende waarde terug: 0: indien ze gelijk zijn. getal <0: indien deze <i>string</i> kleiner is dan de meegegeven <i>string</i> . getal >0: indien deze <i>string</i> groter is dan de meegegeven <i>string</i> .

Tabel 23: Methoden van de klasse String

We illustreren het gebruik van de methode `charAt()`. We moeten aan deze methode een getal (`int`) meegeven. Het *string*-object gaat dan intern kijken welk karakter op deze positie staat en zal vervolgens dit karakter teruggeven. Met deze teruggegeven waarde kunnen we dan iets doen. Doorgaans steken we die in een andere variabele voor later gebruik.

```
String text = new String("Hello World!");
char c = text.charAt(4);
System.out.println(c); // o
```

De positie van karakters start altijd met 0. Om die reden is de vierde positie gelijk aan het vijfde karakter. Daarom dat in bovenstaand voorbeeld het karakter 'o' wordt teruggegeven.

Op gelijkaardige wijze kunnen we opvragen op welke eerste positie een bepaald karakter staat. Stel dat we willen weten waar het karakter 'l' in de *string* voorkomt, dan kunnen we dit als volgt doen:

```
String text = new String("Hello World!");
int pos = text.indexOf('l');
System.out.println(pos); // 2
```

Als we de andere karakters 'l' willen vinden, moeten we opnieuw zoeken met een andere startpositie. Dit doen we als volgt:

```
System.out.println(pos); // 2
int pos2 = text.indexOf('l', pos + 1);
System.out.println(pos2); // 3
int pos3 = text.indexOf('l', pos2 + 1);
System.out.println(pos3); // 9
```

We gebruiken hier de variant van de methode `indexOf()` die twee parameters heeft. De tweede parameter bepaalt de startpositie om het gevraagde karakter te zoeken. Deze methode heeft weliswaar dezelfde naam, maar heeft een verschillend aantal parameters. We

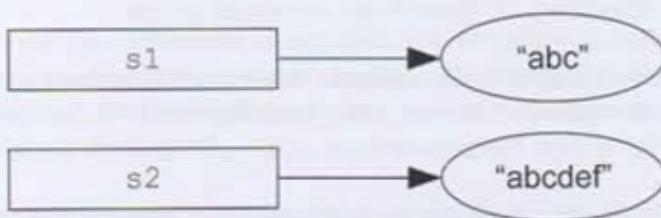
noemen dit *method overloading*. Dergelijke *overloaded* varianten treffen we vaak aan. Het is daarom belangrijk de Java-API-documentatie goed te lezen om zulke varianten op te sporen. Zo heeft de methode `indexOf()` bijvoorbeeld ook een variant waarbij we een andere *string* in plaats van een karakter kunnen meegeven.

Sommige methoden van de klasse `String` geven opnieuw een referentie terug naar een *string-object*. Het is belangrijk om te weten dat iedere methode die een *string* 'verandert' eigenlijk een nieuw *string-object* créeert en diens referentie teruggeeft. *Strings* zelf zijn namelijk onveranderbaar; we krijgen dus telkens een nieuw object met de gewijzigde inhoud.

We nemen het volgende voorbeeld:

```
String s1 = "abc";
String s2 = s1.concat("def");
System.out.println(s1); // abc
System.out.println(s2); // abcdef
```

De *string* `s2` is hierbij een nieuw object. De *string* `s1` blijft ongewijzigd. We kunnen dit visueel als volgt voorstellen:

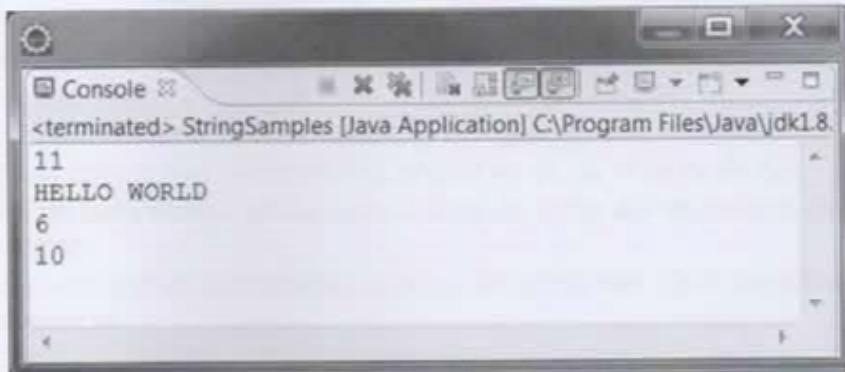


Afbeelding 61: Een nieuwe referentie naar een gewijzigde String

We geven verder nog enkele andere voorbeelden van gebruik van *string*-methoden:

```
String text = "Hello World";
System.out.println(text.length()); // Drukt de lengte af
System.out.println(text.toUpperCase()); // Zet om naar hoofdletters
System.out.println(text.indexOf("World")); // Index van "World"
System.out.println(text.compareTo("Hello Mars")); // getal > 0
```

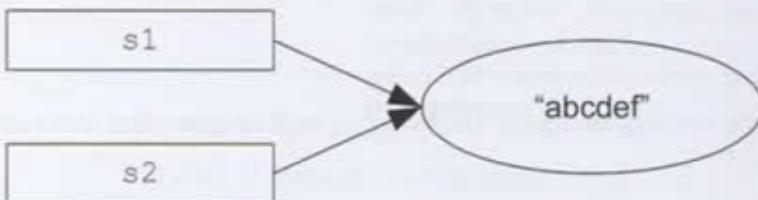
Na uitvoering krijgen we dit:



6.4.2.4 Geheugengebruik bij strings

Doordat strings onveranderbaar zijn, is het mogelijk een *pool* bij te houden van *string-objecten*. Men noemt dit de *string constant pool*. Zodra een nieuw string object gecreëerd wordt door middel van een *string literal*, gaat de VM na of deze *string* zich reeds in de *pool* bevindt en wordt er eventueel een referentie naar de reeds bestaande *string* teruggegeven. We illustreren dit met een voorbeeld:

```
String s1 = "abcdef";
String s2 = "abcdef";
System.out.println(s1==s2); // true
```



Afbeelding 62: Gebruik van canonieke strings

Het lijkt hier te gaan om twee verschillende objecten, maar de VM optimaliseert het geheugengebruik. In feite verwijzen *s1* en *s2* naar hetzelfde object in het geheugen. Dit vormt geen probleem omdat *strings* onveranderbaar zijn.

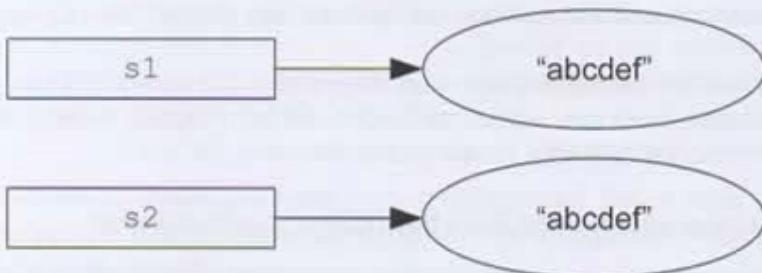
Dergelijke *strings* worden ook wel *canonieke strings* genoemd. *String literals* resulteren steeds in een *canonieke string*.

Indien men echter een *string* maakt door middel van de constructor wordt er wel een nieuw object gecreëerd:

```
String s1 = "abcdef";
String s2 = new String("abcdef");
System.out.println(s1==s2); // false
```

De referentievariabelen *s1* en *s2* verwijzen hier naar twee verschillende objecten in het geheugen. Daarom is de inhoud van de referentie niet dezelfde.

Bij de creatie van *s2* wordt een nieuw object aangemaakt buiten de *string-pool* op basis van een object dat in de *pool* zit, namelijk "abcdef".



Afbeelding 63: Referenties naar verschillende strings

Indien men de inhoud van twee *strings* wil vergelijken, dient men dit steeds te doen met de methode `equals()`. Deze gaat namelijk de inhoud van de twee *strings* vergelijken, ongeacht hun plaats in het geheugen.

```

String s1 = "abcdef";
String s2 = new String("abcdef");
boolean eq = s1.equals(s2);
System.out.println(eq); // true
  
```

Door het bestaan van de canonieke *strings* kan een vergelijking met de `==`-operator soms eenzelfde resultaat geven als de methode `equals()`, maar dit gedrag is onbetrouwbaar omdat in een andere context *strings* dynamisch gemaakt kunnen zijn en deze operator dan mogelijk een ander resultaat geeft.

Kortom, *strings* vergelijken met `==` is uit den boze tenzij men zeer bewust de referentie wil vergelijken en men rekening houdt met het bestaan van canonieke *strings*.

Opdracht 3: Werken met String-objecten

In deze opdracht gaan we werken met *String*-objecten. Zoek bij iedere opdracht een geschikte methode van de klasse *String*.

Om op interactieve wijze tekenreeksen van het toetsenbord te lezen, kan je eventueel het volgende stukje code gebruiken:

```

Scanner keyboard = new Scanner(System.in);
String text = keyboard.next();
  
```

- Maak een programma met een regel tekst. Druk de tekst en de lengte van de tekst af.
- Druk de tekst af met allemaal hoofdletters.
- Druk de tekst af met allemaal kleine letters.
- Vervang alle letters 'a' door de letter 'o' en druk het resultaat af.
- Druk het aantal letters 'e' in de tekst af. Hint: gebruik een `for`-lus.
- Maak twee *strings* met verschillende inhoud en ga na of ze gelijk zijn.
- Vergelijk de twee *strings* alfabetisch en druk de string die alfabetisch eerst komt, ook als eerste af.
- Maak een *string* met extra spaties vooraan en achteraan. Druk de *string* zonder deze extra spaties af.

6.4.3 De klasse `StringBuilder`

Er bestaat in Java een tweede klasse voor het opslaan van *strings*: de `StringBuilder`.

Objecten van deze klassen bevatten intern een tekenreeks die veranderd kan worden. De `StringBuilder`-klasse heeft een aantal methoden die het mogelijk maken de *string* te wijzigen, zoals de methoden `append()` en `insert()`.

In de onderstaande tabel sommen we twee belangrijke constructors op:

Constructor	Beschrijving
<code>StringBuilder()</code>	Maakt een lege <i>stringbuilder</i> .
<code>StringBuilder(String str)</code>	Maakt een <i>stringbuilder</i> met dezelfde inhoud als de meegegeven <i>string</i> .

Tabel 24: Constructors van de klasse `StringBuilder`

Tevens geven we een aantal interessante methoden:

Return-type	Methode	Beschrijving
<code>StringBuilder</code>	<code>append(...)</code>	Voegt iets toe aan de <i>stringbuilder</i> . Er zijn verschillende varianten met elk een eigen type voor de meegegeven parameter.
<code>char</code>	<code>charAt(int index)</code>	Geeft het karakter terug dat zich op een bepaalde positie bevindt in de <i>stringbuilder</i> .
<code>void</code>	<code>setCharAt(int index, char ch)</code>	Stelt het karakter op de aangegeven positie in.
<code>void</code>	<code>deleteCharAt(int index)</code>	Verwijderd het karakter op de aangegeven positie.
<code>boolean</code>	<code>equals(Object o)</code>	Vergelijkt twee <i>stringbuilders</i> en geeft <code>true</code> terug indien het gaat om hetzelfde object in het geheugen.
<code>int</code>	<code>indexOf(String str)</code>	Geeft de eerste index van de opgegeven <i>string</i> binnen deze <i>stringbuilder</i> .
<code>int</code>	<code>length()</code>	Geeft het aantal karakters in de <i>stringbuilder</i> .
<code>StringBuilder</code>	<code>replace(int start, int end, String str)</code>	Vervangt een <i>substring</i> in de <i>stringbuilder</i> tussen de aangegeven indexen.
<code>String</code>	<code>toString()</code>	Geeft de inhoud van de <i>stringbuilder</i> als <i>string</i> terug.
<code>String</code>	<code>substring(int start, int end)</code>	Geeft de <i>string</i> terug die tussen de aangegeven indexen ligt.
<code>StringBuilder</code>	<code>reverse()</code>	Draait de inhoud van de <i>stringbuilder</i> om.

Return-type	Methode	Beschrijving
StringBuilder	insert(int offset,...)	Voegt iets toe op de aangegeven positie. Er zijn varianten voor verschillende datatypes.

Tabel 25: Methoden van de klasse *StringBuilder*

Merk op dat verschillende methoden niets (`void`) teruggeven. Dat is vaak ook niet nodig want de methode zelf verandert intern de inhoud van het *stringbuilder*-object en er hoeft niet onmiddellijk iets terug te komen.

We illustreren dit met de methode `setCharAt()`. Hiermee kunnen we een bepaald karakter op een bepaalde plaats instellen.

```
StringBuilder text = new StringBuilder("Hello World");
text.setCharAt(4, 'a');
System.out.println(text);
```

Op positie 4 plaatsen we het karakter 'a'. Omdat de positie ook hier met 0 begint, is dit dus het vijfde karakter. Het resultaat is dan "Hella World". De *stringbuilder* heeft zijn interne gegevens aangepast na het aanroepen van de methode. We kunnen zo blijven verdergaan en telkens nieuwe wijzigingen aanbrengen aan het object.

Sommige methoden hebben meerdere parameters. Nemen we als voorbeeld de methode `substring()`. Hiermee zoeken we een *substring* tussen een start- en eindpositie. Deze posities moeten we daarom meegeven aan de methode:

```
StringBuilder text = new StringBuilder("Hello World");
String sub = text.substring(6,11);
System.out.println(sub); // "World"
```

De twee gegevens worden gescheiden door een komma. De methode geeft de *string* terug die zich tussen deze posities bevindt.

In tegenstelling tot de methoden van *strings* geven de methoden van de *stringbuilder* vaak een referentie terug naar het (gewijzigde) *stringbuilder*-object zelf. Er wordt bij een wijziging geen nieuw object gecreëerd; de inhoud van het object wijzigt namelijk zelf.

```
StringBuilder sb1 = new StringBuilder("abc");
StringBuilder sb2 = sb1.append("def");
System.out.println(sb1); // abcdef
System.out.println(sb2); // abcdef
System.out.println(sb1 == sb2); // true
```

Doordat dergelijke methoden het object zelf teruggeven, kunnen we een aaneenschakeling van methoden hebben:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").append("ghi").append("jkl");
```

De teruggegeven referentie wordt hier onmiddellijk opnieuw gebruikt om een methode van hetzelfde object op te roepen.

We geven ten slotte nog enkele andere voorbeelden voor het gebruik van de *stringbuilder*.

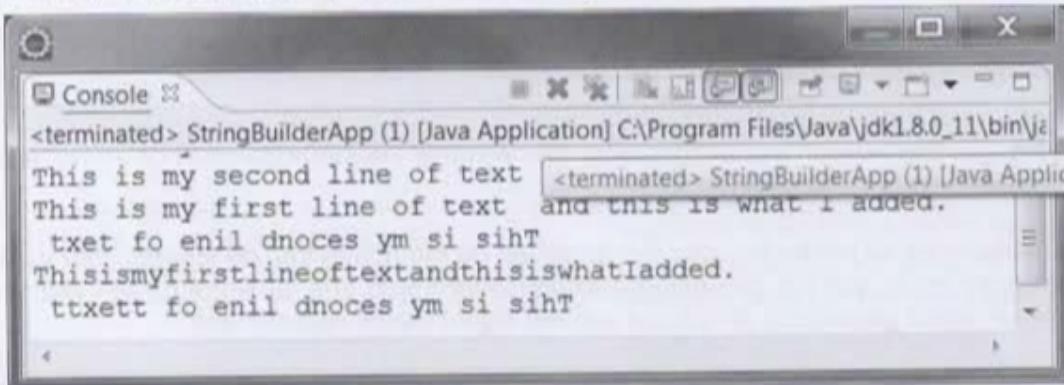
```
StringBuilder sb = new StringBuilder("Hello"); // "Hello"
sb.append(" World"); // "Hello World"
int offset = sb.indexOf("World");
sb.insert(offset,"Big "); // "Hello Big World"
sb.reverse(); // "dlroW gib olleH"
String str = sb.toString();
System.out.println(str.toUpperCase()); // "DLROW GIB OLLEH"
```

Merk op dat sommige methoden die we hadden in de klasse *String* niet beschikbaar zijn in de klasse *StringBuilder*. Voor bepaalde bewerkingen moeten we daarom de *stringbuilder* eerst omzetten naar een *string* met de methode *toString()*. In het voorbeeld is dat onder andere voor het omzetten naar hoofdletters.

Opdracht 4: Werken met *StringBuilder*-objecten

In deze opdracht gaan we werken met *StringBuilder* objecten. Zoek bij iedere opdracht een geschikte methode van de klasse *StringBuilder*.

- Zoek in de Java-API-documentatie de beschrijving van de klasse *StringBuilder*. Deze bevindt zich in het pakket *java.lang*.
- Maak een programma met twee regels tekst. Druk de regels af.
- Voeg aan de eerste *stringbuilder* een stukje tekst toe.
- Keer de tekens van de tweede *stringbuilder* om.
- Verwijder alle spaties uit de eerste *stringbuilder*.
- Verdubbel iedere letter 't' in de tweede *stringbuilder*.



6.4.4 Strings samenvoegen met de + operator

Stringbuilders kunnen worden samengevoegd met de *append*-methode. Het is echter ook mogelijk strings samen te voegen met de *+*-operator. Dit is een stuk makkelijker en korter om te schrijven.

```
text3 = text1 + text2;
```

Door de compiler wordt dit op de volgende manier behandeld:

```
text3 = new StringBuilder().append(text1).append(text2).toString();
```

Er wordt een nieuwe *StringBuilder* gecreëerd zonder inhoud en vervolgens worden de



twee strings hieraan toegevoegd. De uiteindelijke *stringbuilder* wordt ten slotte omgezet in een *String*-object met de methode *toString()*.

De *+*-operator voor *strings* zijn we al herhaaldelijk tegengekomen toen we waarden wilden afdrukken op het scherm:

```
System.out.println("A number :" + number);
```

De methode *println* neemt een object van de klasse *String* als parameter. Daarom wordt de opgegeven parameter door de compiler geconverteerd naar een *string*.

In feite gebeurt er dit:

```
System.out.println(new StringBuilder().append("A number : ")
    .append(number)
    .toString());
```

Er wordt eerst een nieuw object gecreëerd van de klasse *StringBuilder*. Dit object is aanvankelijk leeg. Vervolgens wordt de *string* "A number :" toegevoegd met de methode *append()*. Daarna wordt het getal toegevoegd met de methode *append()*. Vermits de functie *println* een *String* als parameter neemt, wordt de *StringBuilder* op zijn beurt omgezet naar een *String* met de methode *toString()*.

Het is mogelijk om het even welk object bij een *string* op te tellen door middel van het *+*-teken. De compiler roept dan de methode *toString()* op van het object. Ieder object beschikt over deze methode omdat ze geïmplementeerd is in de klasse *Object* waar elke klasse impliciet van afgeleid is.

Indien men getallen bij een *string* wil optellen, zal het getal steeds omgezet worden in een *string* en vervolgens aan de *string* toegevoegd worden:

```
System.out.println(3+7+"Hello World!"+3+7);
```

Dit geeft als resultaat: 10Hello World!37

6.4.5 Gegevens formatteren met de klasse *Formatter*

Vaak willen we allerlei gegevens op een elegante manier afdrukken. Denk maar aan getallen met komma: soms willen we dat er een minimaal of maximaal aantal cijfers na de komma is, of dat er spaties worden toegevoegd aan de cijfers voor de komma.

Om dit alles te doen, kunnen we gebruikmaken van de klasse *Formatter* die bij het pakket *java.util* hoort. Objecten van deze klasse helpen ons bij het formatteren van de gegevens.

We kunnen zo'n *formatter* maken met de gewone constructor zonder parameters:

```
Formatter formatter = new Formatter();
```

De belangrijkste methode van deze klasse is *format()* waarmee we een meegegeven *string* verder kunnen formatteren.

We illustreren dit met een voorbeeld. Stel dat we de leeftijd van een persoon moeten afdrukken. Deze leeftijd is opgeslagen in een variabele van het type `int` en wordt ergens in de code bepaald.

Gebruikmakend van een *formatter* kan dit als volgt:

```
Formatter formatter = new Formatter();
String text = "I'm %d years old!";
formatter.format(text, 21);
System.out.println(formatter.toString());
formatter.close();
```

Laten we eerst naar de *string* `text` kijken: "I'm %d years old!". Op de plaats waar we de leeftijd willen invullen, zetten we een *format specifier* `%d`. Dit is eigenlijk een gemarkeerd 'gat' in de *string* dat we later zullen invullen. Het `%`-teken markeert deze *format specifier* en met `d` geven we aan dat het gaat om een *decimaal* getal.

Het invullen van dat 'gat' gebeurt in de methode `format()`. We geven hier als eerste parameter de '*string* met gaten' mee en de volgende parameter is de waarde die in het eerste gat moet ingevuld worden, namelijk de leeftijd.

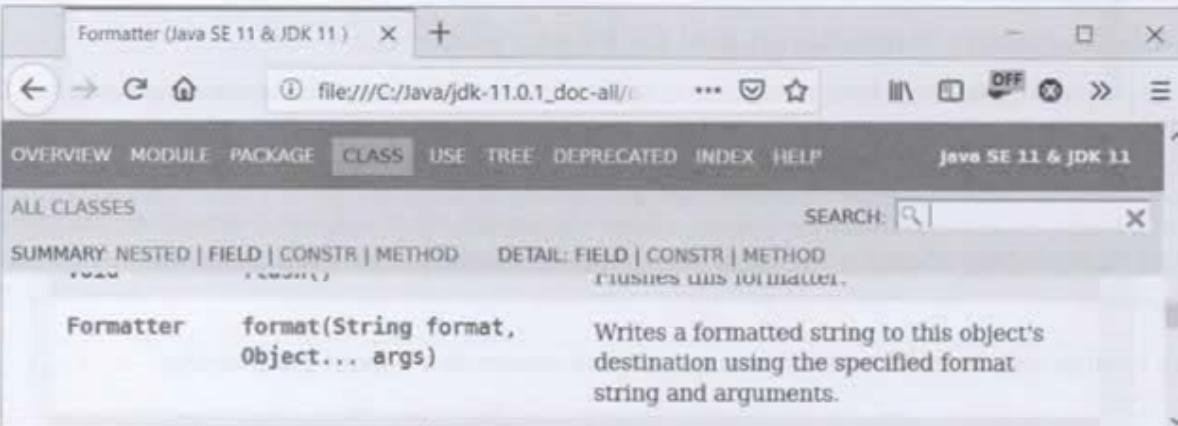
Intern zal deze *formatter* alles opslaan in een *stringbuilder*. We kunnen het eindresultaat evenwel gewoon opvragen met de methode `toString()`.

We kunnen ook *strings* met meerdere *format specifiers* hebben:

```
String text = "I'm %d years old and I'm %f m tall!";
formatter.format(text, 21, 1.75);
```

De tweede *format specifier* is een getal met komma. We duiden dit aan met `%` gevolgd door de letter `f` (*float*). Dit tweede getal geven we daarom ook mee als derde parameter.

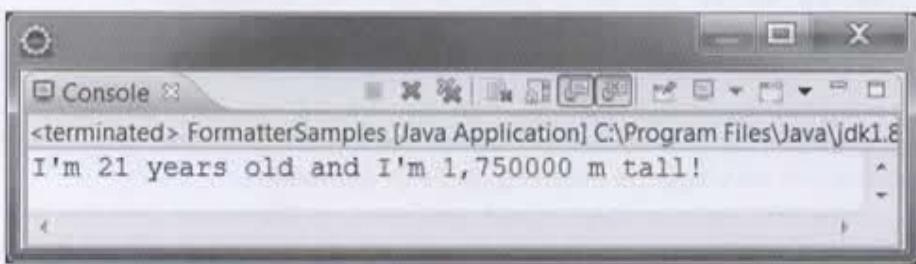
Merk op dat de methode `format()` een variabel aantal parameters kan hebben! Als we de Java-API-documentatie bekijken, zien we inderdaad dit:



Afbeelding 64: API-documentatie van de methode `format()`

De drie puntjes achter `Object...` wijzen erop dat hier meerderere parameters mogelijk zijn.

Als we de inhoud afdrukken, krijgen we:



Het decimale punt is veranderd in een komma, want dat is in onze contreien de juiste notatie. Maar die vier extra nullen zijn een beetje overbodig. Tot op de centimer volstaat doorgaans wel.

We kunnen nu de formatting van dit getal wat aanpassen, zodat slechts twee cijfers na de komma getoond worden. Dit doen we als volgt:

```
String text = "I'm %d years old and I'm %.2f m tall!";
formatter.format(text, 21, 1.75);
```

We zetten .2 tussen % en f om aan te geven dat twee cijfers na de komma volstaan. Dit noemen we de precisie.

We kunnen nu de gegevens verder formatteren. De mogelijkheden zijn zeer uitgebreid en worden uitvoerig beschreven in de documentatie van de klasse `Formatter`.

We geven hier de algemene syntax van een *format specifier*:

`%[index$][flags][width][.precision]conversion`

De gegevens tussen [] zijn optioneel, de andere delen zijn verplicht.

We overlopen de verschillende onderdelen.

Met **conversion** geven we het datatype aan van de parameter. Enkele mogelijke waarden zijn:

b	boolean
c	character
d	integer
f	floating point
s	string
e	wetenschappelijke notatie
n	nieuwe regel

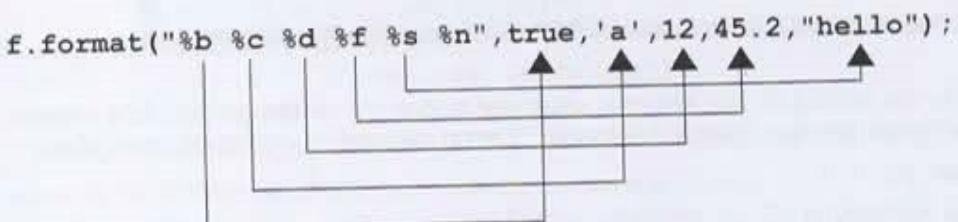
Tabel 26: Tekens voor 'conversion'

We illustreren dit met nog een voorbeeld:

```
Formatter f = new Formatter();
```

```
f.format("%b %c %d %f %s %n",true,'a',12,45.2,"hello");
System.out.println(f.toString());
```

We gebruiken volgende formatteringsstring: "%b %c %d %f %s %n", gevuld door een variabel aantal waarden die overeenkomen met de verschillende *format specifiers*. De overeenkomst wordt in de volgende tekening duidelijk gemaakt:



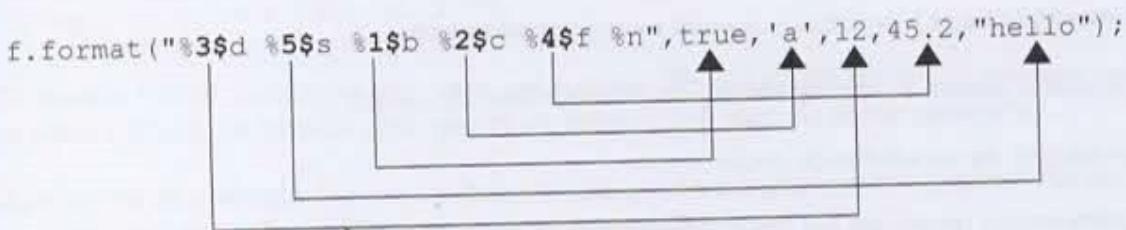
Afbeelding 65: De formatteringsstring

Merk op dat we voor %n geen parameter definiëren; deze voegt gewoon een *newline* (nieuwe regel) toe. Het uiteindelijke resultaat is:

```
true a 12 45,200000 hello
```

In dit voorbeeld dient de volgorde van de parameters overeen te komen met de volgorde van de *format specifiers*.

Met de optionele **index** kunnen we een bepaalde waarde selecteren uit de reeks parameters. De volgorde hoeft dan niet overeen te komen.



Afbeelding 66: Het gebruik van een index in de formatteringsstring

Het resultaat is dan:

```
12 hello true a 45,200000
```

Het is hier tevens mogelijk een parameter meermaals in de formatteringsstring op te nemen.

Voorts kan men **flags** toevoegen die extra informatie bevatten voor de formatering van de waarde. We sommen er enkele op:

-	Lijnt de waarde links uit
+	Voorziet een getal van een + of - teken
0	Vult een getal vooraan aan met nullen
,	Gebruikt scheidingstekens bij duizentallen
(Plaatst negatieve getallen tussen haakjes

Tabel 27: Tekens voor 'flags'

```
f.format("%(d %d %,d", -125, 145, 12356987);
```

resulteert in:

```
(125) +145 12.356.987
```

Ten slotte kan ook nog een **width** en **precision** toegevoegd worden. Met **width** geven we het minimum aantal karakters aan dat afgedrukt moet worden en **precision** kan onder andere gebruikt worden voor het aantal cijfers na de komma te bepalen.

```
f.format("%-6d %06d %.2f", -125, 145, 123.56987);
```

resulteert in:

```
-125 000145 123,57
```

Bij het afdrukken van gegevens op de console kan men gebruikmaken van de methoden `System.out.printf()` of `System.out.format()` die impliciet gebruikmaken van een **formatter**. Ook hier kunnen we een formatteringsstring meegeven gevolgd door een variabel aantal parameters.

```
System.out.printf("%-6d %06d %.2f", -125, 145, 123.56987);
```

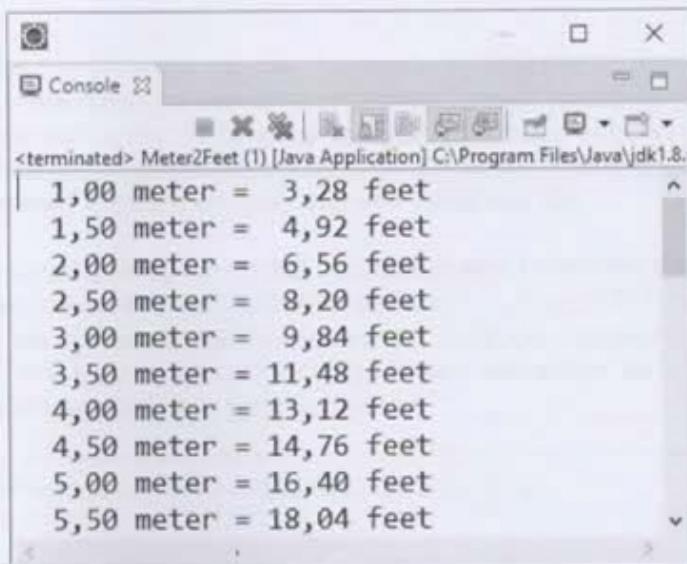
of

```
System.out.format("%-6d %06d %.2f", -125, 145, 123.56987);
```

Opdracht 5: Strings formatteren

In deze opdracht experimenteren we met de **formatter**.

- Maak een programma dat de eenheid 'meter' omzet naar de eenheid 'voet'. Toon de waarden van 1 meter tot en met 20 meter (toename van 0,5 meter) en de overeenkomstige waarden in 'voet'. Zorg dat slechts twee cijfers na de komma getoond worden en dat alle getallen mooi uitgelijnd zijn.



Meter	Feet
1,00	3,28
1,50	4,92
2,00	6,56
2,50	8,20
3,00	9,84
3,50	11,48
4,00	13,12
4,50	14,76
5,00	16,40
5,50	18,04

6.5 Samenvatting

In dit hoofdstuk hebben we kennis gemaakt met het objectgeoriënteerd programmeren. We hebben eerst het algemene concept en begrippenkader gesitueerd. Objecten bestaan uit eigenschappen (**properties**) en bijhorende gedragingen (**methods**). De eigenschappen zijn variabelen en de gedragingen zijn codeblokken die iets kunnen doen met die variabelen. Samen vormen ze een object.

Objecten worden gemaakt op basis van een blauwdruk die we de **klasse** noemen. Sommige eigenschappen en methoden horen toe aan een object en andere horen eerder toe aan de klasse zelf. Dat is een belangrijk onderscheid.

Vervolgens hebben we gezien hoe we concreet objecten kunnen **maken** en **gebruiken** in een Java-toepassing. Als voorbeeld namen we de klasse `Random`. We hebben geleerd de publieke methoden van een object te gebruiken en ook de verschillende constructors die vorhanden zijn.

Ten slotte hebben we de klassen `String` en `StringBuilder` onder de loep genomen. Ook hier hebben we geleerd gebruik te maken van de constructors en publieke methoden. Tekst formatteren hebben we gedaan met de klasse `Formatter`.