Hoofdstuk 14

Migreren van C naar Java

In dit hoofdstuk vindt u een overzicht van de programmeertaal Java. Het biedt tevens een introductie in de toepassing van Java als objectgeoriënteerde programmeertaal. Het is net zo georganiseerd als het hoofdstuk over C++; er wordt een reeks programma's gepresenteerd en de elementen van elk programma worden uitgelegd. De programma's worden geleidelijk aan ingewikkelder en de voorbeelden in de latere paragrafen illustreren enkele concepten van objectgeoriënteerd programmeren. Het kan onafhankelijk van het hoofdstuk over C++ worden gelezen.
De voorbeelden in dit hoofdstuk geven eenvoudige, directe, praktische ervaring met belangrijke functies van de taal Java. Het hoofdstuk laat u kennismaken met Java I/O, klassen, overerving, graphics, threads en excepties. Beheersing van de afzonderlijke onderwerpen vereist een grondige bestudering van een begeleidend boek, zoals Arnold en Gosling, *The Java Programming Language* (Reading, MA: Addison-Wesley, 1996).
Objectgeoriënteerd programmeren wordt geïmplementeerd door de **class**-constructie. De **class**-constructie in Java is een uitbreiding van **struct** in C. De latere voorbeelden in dit hoofdstuk laten zien hoe Java OOP- (objectgeoriënteerde programmeer)concepten verwezenlijkt, zoals data-hiding, ADT's, overerving en typehiërarchieën. Java is ontwikkeld om te worden ingezet op het World Wide Web. Het heeft speciale libraries die zijn ontworpen voor graphics en communicatie via internet. Het is geschreven om op een machine- en systeemonafhankelijke manier te draaien. Dit houdt in dat het Java-programma dezelfde resultaten voortbrengt op een pc waarop Windows 95 draait als op een werkstation waarop SUN Solaris is geïnstalleerd. Dit is mogelijk omdat de semantiek van Java volledig als een virtuele machine wordt gedefinieerd. De taak voor een systeem dat Java wil draaien, is de virtuele machine te poorten. Dit betekent dat er een afweging moet worden gemaakt tussen overdraagbaarheid en efficiëntie. Een machine die een simulator van een andere architectuur draait, kent zonder twijfel extra overhead. Een deel van deze inefficiëntie kan worden verholpen door het gebruik van just-in-time compilers of het gebruik van systeemeigen code die in C is geschreven en die wordt ingevoegd op plaatsen waar efficiëntie van groot belang is. Op veel platforms is het ook mogelijk om een direct-to-native codecompiler in te zetten voor maximale efficiëntie tijdens runtime.

14.1Uitvoer

Programma's moeten communiceren om van nut te zijn. Ons eerste voorbeeld is een programma dat de zin 'Java is an improved C.' in het scherm afdrukt. Het volledige programma staat in het bestand Improved.java:

```
// A first Java program illustrating output.
//    Title:  Improved
//    Author: Richmond Q. Programmer
```

```
class Improved {
    public static void main (String[] args)
    {
       System.out.println("Java is an improved C.");
    }
}
```

Het programma drukt in het scherm af:

```
Java is an improved C.
```

Dit programma wordt gecompileerd door gebruik te maken van de opdracht *javac improved.java*; resulterend in een codebestand met de naam *Improved.class*. Dit kan worden uitgevoerd met de opdracht *java Improved*.

Bespreking van het programma *improved*

- `// A first Java program illustrating output.`

De dubbele schuine streep is een nieuw commentaarsymbool. Het commentaar loopt tot het eind van de regel. De oude commentaarsymbolen in C, /* */, zijn nog steeds beschikbaar voor meerregelig commentaar. Java kent ook de commentaarsymbolen /** */ voor doc-commentaar. Een programma zoals *javadoc* gebruikt doc-commentaar en genereert een HTML-bestand.

- `class Improved {`

Java-programma's zijn klassen. Een **class** heeft een syntactische vorm die wordt afgeleid van de C-vorm **struct**, die niet in Java wordt toegepast. In Java beginnen namen van klassenidentifiers, zoals **Improved**, volgens afspraak met een hoofdletter. Data en code worden binnen klassen geplaatst.

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

Als een klasse wordt uitgevoerd als een programma, roept het als eerste de lidfunctie **main()** aan. In dit geval is **main()** een lid van **Improved**. In Java worden opdrachtregelargumenten ingevoegd in een array met **String**s. In C hebben we een **argc**-variabele nodig om het aantal opdrachtregelargumenten door te geven aan het programma. In Java wordt deze arraylengte gevonden met **args.length**.

- `System.out.println("Java is an improved C.");`

Dit statement drukt af naar het scherm. Het **Systemout**-object gebruikt de lidfunctie **println()** om af te drukken. De functie drukt de string af en voegt een nieuwe regel toe, die de schermcursor naar de volgende regel verplaatst. In tegenstelling tot **printf()** in C maakt **println()** gebruik van opmaakbesturingselementen.

In Java staan alle functies in klassen. In dit geval is de functie **main()** een lid van de klasse **Improved**. Een lidfunctie wordt een *methode* genoemd.

14.2Variabelen en typen

We gaan een programma schrijven om de afstand van de aarde naar de maan in mijlen om te zetten naar kilometers. In mijlen is deze afstand ongeveer 238.857 Dit getal is een integer. Om mijlen te converteren naar kilometers vermenigvuldigen we met de conversiefactor 1,609, een reëel getal.
Ons conversieprogramma maakt gebruik van variabelen voor de opslag van integerwaarden en reële waarden. De variabelen in het volgende programma worden gedeclareerd in **main()**. In Java kunnen variabelen niet worden gedeclareerd als **extern** (met andere woorden, als variabelen met een globaal of bestandsbereik).
De primitieve typen in een Java-programma kunnen **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** en **double** zijn. Deze typen worden altijd op dezelfde wijze gedefinieerd, ongeacht de machine of het systeem waarop ze draaien. Zo is het **int**-type altijd een signed 32-bits integer, in tegenstelling tot in C, waar dit per systeem kan variëren. Het boolean-type is geen rekenkundig type en kan niet worden gebruikt in gemengde rekenkundige expressies. Het **char**-type gebruikt 16-bits Unicode-waarden. De typen **byte**, **short**, **int** en **long** zijn allemaal signed integer-typen, die respectievelijk een lengte hebben van 8, 16, 32 en 64 bits. In tegenstelling tot C kent Java geen unsigned typen. De floating-typen voldoen aan IEEE754-standaarden en zijn **float**, een 32-bits type en **double**, een 64-bits type. De niet-primitieve typen zijn klassentypen en arraytypen, en variabelen van deze typen gebruiken referenties als hun waarde.

In het bestand Moon.java:

```
// The distance to the moon converted to kilometers.
//    Title: moon

public class Moon {
   public static void main(String[] s) {
      int moon = 238857;
      int moon kilo;

      System.out.println("Earth to moon = " + moon + " mi.");
      moon kilo = (int)(moon * 1.609);
      System.out.println("Kilometers = " + moon kilo +"km.");
    }
```

De uitvoer van het programma is:

```
Earth to moon = 238857 mi.
Kilometers = 384320 km.
```

Analyse van het programma *moon*

```
   int moon = 238857;
```

Variabelen van het type **int** zijn signed 32-bits integers. Ze kunnen net als in C worden geïnitialiseerd.

```
   System.out.println("Earth to moon = " + moon + " mi.");
```

De **println()**-methode kan onderscheid maken tussen een verscheidenheid aan eenvoudige waarden zonder extra opmaakinformatie nodig te hebben. In dit voorbeeld wordt de waarde van **moon** afgedrukt als een integer. Het symbool + staat voor een samenvoeging van strings. Het gebruik van 'plus' **println()** kan een lijst met argumenten afdrukken. Wat er gebeurt, is dat elk argument wordt geconverteerd van zijn specifieke type naar een uitvoerstring die wordt samengevoegd en samen met een nieuwe regel wordt afgedrukt.

```
moon kilo = (int)(moon * 1.609);
```

De gemengde expressie **moon * 1.609** is een **double**. Deze moet expliciet worden geconverteerd naar **int**.

Merk op dat conversies naar een type dat uit minder bits bestaat, die in C impliciet plaatsvinden, niet in Java worden toegepast.

14.3Klassen en abstracte datatypen

Nieuw in Java is het type **class**. Een **class** is een uitbreiding van het concept van **struct** in traditioneel C. Een **class** zorgt voor de middelen om een user-defined datatype en geassocieerde functies te implementeren. Daarom kan een **class** worden gebruikt om een ADT te implementeren. We gaan een klasse, **Person** genaamd, schrijven om informatie over personen op te slaan.

In het bestand Person.java:

```
// An elementary implementation of type Person.

class Person {
    private String name;
    private int age;
    private char gender;          //male == 'M' , female == 'F'

    public void assignName(String nm) { name = nm; }
    public void assignAge(int a) { age = a; }
    public void assignGender(char b) { gender = b; }
    public String toString(
        return (name + " age is " + age +
        " sex is " + gender );
    }
};
```

In dit voorbeeld staan twee belangrijke toevoegingen aan het structuurconcept van C: ten eerste heeft het leden, *klassenmethoden* genaamd, die functies zijn, zoals **AssignAge()**. Ten tweede beschikt het zowel over public als private leden. Het keyword **public** duidt op de zichtbaarheid van de leden die erop volgen. Zonder dit keyword zijn de leden private voor de klasse. Private leden zijn alleen beschikbaar voor gebruik door andere lidfuncties van de klasse. Public leden zijn overal beschikbaar waar de klasse

beschikbaar is. Als gevolg van privacy kan een deel van de implementatie van een klassentype ‘hidden’ zijn. Door deze beperking wordt voorkomen dat er onaangekondigd aanpassingen aan de datastructuur plaatsvinden. Beperkte toegang, of *data-hiding*, is een functie van objectgeoriënteerd programmeren.
Door de declaratie van methoden in een klasse kan het ADT over acties of gedrag beschikken die kunnen inwerken op zijn private representatie. De lidfunctie **toString()** heeft bijvoorbeeld toegang tot private leden en geeft **Person** een string-representatie die in de uitvoer wordt gebruikt. Deze methode is gebruikelijk voor vele klassentypen.
We kunnen dit datatype **Person** nu inzetten alsof het standaard deel uitmaakt van de taal. Andere code die dit type gebruikt, is een *client*. De client kan alleen gebruikmaken van de public leden om variabelen van het type **Person** te beïnvloeden.

```
//PersonTest.java uses Person.

public class PersonTest {
     public static void main (String[] args )
     {
          System.out.println("Person test:");
          Person p1 = new Person();    //create a Person object
          p1.assignAge(20);
          p1.assignName("Alan Turing");
          p1.assignGender(false);
          System.out.println(p1.toString());
     }
}
```

De uitvoer van dit voorbeeldprogramma is:

```
Person test:
Alan Turing age is 20 sex is M
```

Let op het gebruik van **new Person()** om een werkelijke instantie van **Person** in het leven te roepen. De **new**-operator gaat naar de heap, ongeveer net zoals **malloc()** in C, en reserveert geheugen voor het creëren van een echte instantie van het object **Person**. De waarde van **p1** is een verwijzing naar dit object. Dit is het adres van het object.

14.4Overloading

De term *overloading* verwijst naar het geven van meerdere betekenissen aan een methode. De geselecteerde betekenis hangt af van de typen argumenten die worden doorgegeven aan de methode, oftewel de *signatuur* van de methode. We gaan de functie **assignGender()** uit het vorige voorbeeld overloaden. Dit wordt een tweede definitie van de methode **assignGender()**.

```
class Person {
.....
       public void assignGender(char b) { gender = b; }
       public void assignGender(String b)
```

```
        { gender = ((b == "M")? 'M': 'F'); }
}
```

Deze versie van **assignGender()** gebruikt één argument van het type **String**. Hij converteert en bewaart dit argument als een **gender**-tekenwaarde. Nu kan een gebruiker een **char-** of een **String**-waarde gebruiken bij de toekenning van het geslacht.

14.5Constructie en destructie van klassentypen

Een *constructor* is een functie die als taak heeft om een object van zijn klasse te *initialiseren*. Constructors worden aangeroepen nadat aan de instantievariabelen van een zojuist gecreëerd klassenobject standaardbeginwaarden zijn toegekend en expliciete initializers worden aangeroepen. Constructors worden vaak overloaded.
Een constructor is een lidfunctie waarvan de naam gelijk is aan die van de klasse. De constructor is geen methode en kent geen returntype. We gaan het **Person**-voorbeeld zo aanpassen dat constructors de naaminstantievariabele initialiseren.

```
//constructor to be placed in Person

public Person() {name = "Unknown";}
public Person(String nm) { name =nm;}
public Person(String nm, int a, char b)
    { name =nm; age =a; gender = b;}
```

Deze zouden worden aangeroepen als **new** gewend raakt aan het associëren van een gecreëerde instantie met het juiste type referentievariabele. Bijvoorbeeld,

```
p1 = new Person();                      //creates "unknown 0 M
p1 = new Person("Laura Pohl");          //creates Laura Pohl 0 M
p1 = new Person("Laura Pohl" 9, 'F');   //creates Laura Pohl 9 F
```

De overloaded constructor wordt geselecteerd door de reeks argumenten die overeenkomt met de constructorsparameterlijst.
Destructie wordt automatisch door het systeem uitgevoerd via automatische garbage collection. Als er niet langer naar het object kan worden verwezen, bijvoorbeeld wanneer aan de bestaande verwijzing een nieuw object wordt toegekend, wordt het nu ontoegankelijke object garbage genoemd. Het systeem doorloopt regelmatig het geheugen en haalt deze ‘dode’ objecten op. De programmeur hoeft zich niet bezig te houden met dat soort geheugenlekken.

14.6Objectgeoriënteerd programmeren en overerving

In Java is overerving het mechanisme van het *uitbreiden* van een nieuwe klasse van een bestaande klasse, de *superklasse* genaamd. De uitgebreide klasse voegt iets toe aan of wijzigt de overgeërfde superklassenmethoden. Deze methode wordt gebruikt om de interface te delen en om een hiërarchie van verwante typen te creëren.
We gaan ons hoofd eens buigen over een database voor een universiteit. Het administratief hoofd moet de verschillende typen studenten bijhouden. De superklasse

waarmee we beginnen, is **Person1**. Deze klasse is exact gelijk aan **Person**, behalve dat de **private** instantievariabelen zo worden aangepast dat ze als toegang **protected** hebben. Deze toegang staat hun gebruik in de subklasse toe, maar gedraagt zich anders als **private**.
Hier ziet u een voorbeeld van het afleiden van een klasse:

```
// Note Person1 is Person with private instance variables
// made protected

class Student extends Person1 {
    private String college;
    private byte year;          //1 = fr, 2 = so, 3 = jr, 4 = sr
    private double gpa;         //0.0 to 4.0
    public void assignCollege(String nm) { college = nm; }
    public void assignYear(byte a) { year = a; }
    public void assignGpa(double g) { gpa = g; }
    public String toString()
       { return (super.toString() + "  College is " + college); }
    public Student()
       {super.assignName("Unknown"); college = "Unknown";}
    public Student(String nm)
        { super(nm); college = "Unknown"; }
    public Student(String nm, int a, char b)
        { name =nm; age =a; gender = b; }
};
```

In dit voorbeeld is **Student** de subklasse en **Person1** de superklasse. Let op het gebruik van het keyword **super**. Het biedt een manier om de instantievariabelen of methoden in de superklasse te benaderen.
De overervingsstructuur biedt een ontwerp voor het totale systeem. De superklasse **Person1** leidt tot een ontwerp waar de subklasse **Student** ervan wordt afgeleid. Andere subklassen, zoals **GradStudent** of **Employee**, kunnen aan deze overervingshiërarchie worden toegevoegd.

14.7Polymorfisme en overschrijfmethoden

In Java is polymorfisme zowel het resultaat van het overloaden als van het overschrijven van methoden. Overloading is al besproken. Overschrijven treedt op als een methode opnieuw in de subklasse wordt gedefinieerd. De **toString()**-methode bevindt zich in **Person1** en wordt opnieuw gedefinieerd in **Student**, die is uitgebreid vanaf **Person1**.

```
//Overriding the printName() method
class Person1 {
    protected String name;
    protected int age;
    protected char gender;          //male == 'M' , female == 'F'
    public toString() {
         return(name + " age is " + age +
                 " sex is " + (gender == 'F' ? "F": "M") );
```

```
    }
.....
};
```

De overschreven methode **toString()** heeft dezelfde naam en signatuur in de superklasse **Person1** en de subklasse **Student**. Welke daarvan wordt geselecteerd, hangt tijdens runtime af van waarnaar wordt verwezen. In de code

```
//StudentTest.java use Person1

public class StudentTest {
    public static void main (String[] args )
    {
        Person1 q1;
        q1 = new Student();
        q1.assignName("Charles Babbage");
        System.out.println(q1.toString());
        q1 = new Person1();
        q1.assignName("Charles Babbage");
        System.out.println(q1.toString());
    }
}
```

kan de variabele **q1** bijvoorbeeld verwijzen naar het **Person1**-object of het subtype **Student**-object. Tijdens runtime wordt de juiste **toString()** geselecteerd. De **assignName()**-methode is tijdens runtime bekend, omdat het de superklasse **Person1**-methode is.

14.8Applets

Java staat erom bekend dat het voor applets op webpagina's zorgt. Er wordt een browser gebruikt om de applet weer te geven en uit te voeren. Normaliter biedt de applet een grafische gebruikersinterface voor de code. Het volgende stukje code is een applet voor het berekenen van de grootste gemene deler van twee getallen.

```
//GCD applet implementations

import java.applet.*; //gets the applet superclass
import java.awt.*;    //abstract windowing toolkit
import java.io.*;

//derived from the class Applet

public class wgcd extends Applet {
    int x, y, z, r;
    TextField a = new TextField(10);        //input box
    TextField b = new TextField(10);        //input box
    TextField c = new TextField(10);        //output box
    Label l1 = new Label("Value1: ");
```

```
    Label l2 = new Label("Value2: ");
    Button gcd = new Button("   GCD: ");

//draws the screen layout such as the TextFields

    public void init() {
       setLayout(new FlowLayout());
       c.setEditable(false);
       add(l1); add(a);
       add(l2); add(b);
       add(gcd); add(c);
    }

//computes the greatest common divisor

    public int gcd(int m, int n) {
       while (n !=0) {
          r = m % n;
          m = n;
          n = r;
       }
       return m;
    }

//looks for screen events to interact with

    public boolean action(Event e, Object o) {
       if ("   GCD: ".equals(o)) { //press button
          x = Integer.parseInt(a.getText());
          y = Integer.parseInt(b.getText());
          z = gcd(x,y);
         //place answer in output TextField
          c.setText(Integer.toString(z));
       }
       return true;
    }
};
```

De code gebruikt de graphics-library awt en de appletklasse om een interactieve interface te tekenen die kan worden uitgevoerd door een speciaal programma, de *appletviewer* genaamd, of door een Java-aware browser, zoals Microsoft Explorer of Netscape Navigator. In tegenstelling tot gangbare Java-programma's maakt dit programma niet gebruik van een **main()**-methode om de berekening te starten. In plaats daarvan tekent de **init()**-methode het scherm. Verdere berekening is gebeurtenisgestuurd en wordt verwerkt door de **action()**-methode. De gebruiker beëindigt de applet door op de opdracht Quit in het vervolgkeuzemenu van de applet te klikken.

14.9Java-excepties

Java kent een mechanisme voor de afhandeling van excepties, dat deel uitmaakt van de taal en vaak wordt gebruikt voor foutdetectie tijdens runtime. Het lijkt op dat in C++. Er wordt een exceptie geworpen door een methode als deze een foutconditie ontdekt. De exceptie wordt afgehandeld door het aanroepen van een correcte *handler*, die wordt geselecteerd uit een lijst met handlers, catches genaamd. Deze expliciete catches treden op aan het eind van een omhullend **try**-blok. Een niet-gevangen exceptie wordt afgehandeld door een standaard Java-handler die een boodschap verstuurt en het programma beëindigt. Een exceptie is zelf een object dat moet worden afgeleid van de superklasse **Throwable**. Als een eenvoudig voorbeeld hiervan voegen we een exceptie **NoSuchNameException** toe aan onze **Person**-voorbeeldklasse.

```
class NoSuchNameException extends Exception {
    public String str() { return name; }
    public String name;
    NoSuchNameException(String p) { name = p; }
};
```

Het doel van deze exceptie is een incorrecte of onjuist geformeerde naam te rapporteren. In veel gevallen gedragen excepties zich in de taal C als assertions. Ze bepalen of er een illegale actie heeft plaatsgevonden en rapporteren deze. We passen de **Person**-code aan om gebruik te maken van de exceptie.

```
//Person2.class: Person with exceptions added

class Person2 {
    private String name;
    public Person2(String p)throws NoSuchNameException {
       if (p == "")
          throw new NoSuchNameException(p);
       name = p;  }
    public String toString(){ return name;}
    public static void main(String[] args)
          throws NoSuchNameException
    {
        try{
              Person2 p = new Person2("ira pohl");
              System.out.println("PERSONS");
              System.out.println(p.toString());
              p = new Person2("");
        }
    catch(NoSuchNameException t)
       { System.out.println(" exception with name " + t.str()); }
    finally
       { System.out.println("finally clause"); }
};
```

De **throw()** heeft een **NoSuchNameException**-argument en komt overeen met de **catch()**-signatuur. Van deze handler wordt verwacht dat hij een correcte actie onderneemt als er als argument een incorrecte naam wordt doorgegeven aan de **Person2**-constructor. Een foutboodschap en het afbreken van het programma, zoals in

dit voorbeeld, zijn normaal. De **finally**-bepaling is hier weergegeven. Die is code die altijd wordt uitgevoerd, ongeacht hoe het **try**-blok afsluit.

14.10 Voordelen van Java en OOP

Java deelt met C++ het gebruik van klassen en overerving, om op een objectgeoriënteerde manier software te bouwen. Tevens maken beide programmeertalen gebruik van data-hiding en kennen ze methoden die binnen de klasse worden gebundeld.
In tegenstelling tot C++ kan in Java niet op de conventionele wijze worden geprogrammeerd. Alles wordt ingekapseld in een klasse. Hierdoor wordt de programmeur gedwongen alles als object te ontwerpen. Het nadeel hiervan is dat conventionele C-code zich niet zo gemakkelijk aanpast aan Java als aan C++. Java vermijdt de meeste fouten met geheugen-pointers die veel voorkomen in C en C++. Berekening en manipulatie van adressen worden door de compiler en het systeem uitgevoerd, niet door de programmeur. Daarom schrijft de Java-programmeur veiligere code. Verder wordt het vrijgeven van geheugen automatisch uitgevoerd door de garbage collector van Java.
Een ander belangrijk concept in OOP is het bevorderen van het hergebruik van code via het *overerving*smechanisme. In Java is dit het mechanisme van het *uitbreiden* van een nieuwe klasse, een *subklasse* genaamd, van een bestaande klasse, de *superklasse* genaamd. Methoden in de uitgebreide klasse overschrijven de superklassenmethoden. De methodeselectie vindt plaats tijdens runtime en is een enorm flexibele polymorfe stijl van coderen.
Java is volledig overdraagbaar over alle platforms die het ondersteunen. Java wordt gecompileerd naar bytecode, die wordt uitgevoerd op de Java Virtual Machine. Dit is normaliter een interpreter – code die de Java-bytecode-instructies begrijpt. Zulke code is veel langzamer dan systeemeigen code op de meeste systemen. Er moet dus een afweging wprdem gemaakt tussen universeel consistent gedrag versus verlies aan efficiëntie.
Java heeft uitgebreid ontwikkelde library's voor het uitvoeren van webgebaseerde programma's. Het beschikt over de mogelijkheid om grafische gebruikersinterfaces te schrijven die interactief worden gebruikt. Het kent verder een thread-pakket en beveiligde webcommunicatiefuncties waarmee de gebruiker applicaties voor distributie kan schrijven.
Java is veel eenvoudiger dan C++ wat kerntaal en functies betreft. In bepaalde opzichten is dit bedrieglijk, omdat veel van de complexiteit in de libraries staat. Java is veel veiliger vanwege de zeer strikte typering, vermijding van berekeningen met pointers en uitstekend geïntegreerde afhandeling van excepties. Het gedraagt zich systeemonafhankelijk. Deze combinatie van OOP, eenvoud, algemeenheid en web-sensitive libraries maakt het een zeer bruikbare taal.

Samenvatting

1. De dubbele schuine streep, //, is een nieuw commentaarsymbool. Het commentaar loopt tot aan het eind van de regel. De oude C-stijl-commentaarsymbolen /* */ zijn nog steeds beschikbaar voor meerregelig commentaar. Java kent ook de

commentaarsymbolen /** */ voor doc-commentaar. Een programma als *javadoc* gebruikt doc-commentaar en genereert een HTML-bestand.
2. Java-programma's zijn klassen. Een **class** heeft een syntactische vorm die wordt afgeleid van de C-vorm **struct**, die niet in Java voorkomt. Data en code worden in klassen geplaatst. Als een klasse wordt uitgevoerd als een programma, begint hij met de aanroep van de lidfunctie **main()**.
3. De term *overloading* verwijst naar het geven van meerdere betekenissen aan een methode. De geselecteerde betekenis hangt af van de typen argumenten die worden doorgegeven aan de methode, oftewel de *signatuur* van de methode.
4. Een *constructor* is een functie die als taak heeft om een object van zijn klasse te *initialiseren*. Constructors worden aangeroepen nadat aan de instantievariabelen van een zojuist gecreëerd klassenobject standaardbeginwaarden zijn toegekend en expliciete initializers worden aangeroepen. Constructors worden vaak overloaded. Destructie wordt automatisch door het systeem uitgevoerd via automatische garbage collection.
5. Overerving is het mechanisme van het *uitbreiden* van een nieuwe klasse vanaf een bestaande klasse, de *superklasse* genaamd. De uitgebreide klasse voegt iets toe aan of wijzigt de overgeërfde superklassenmethoden. Dit wordt gebruikt om een interface te delen en om een hiërarchie met verwante typen te creëren.
6. In Java is polymorfisme het resultaat van zowel het overloaden als het overschrijven van methoden. Overschrijven treedt op als een methode opnieuw in de subklasse wordt gedefinieerd. De selectie van de juiste overschreven methodedefinitie wordt tijdens runtime bepaald, afhankelijk van het type object.
7. Java staat erom bekend dat het voor applets op webpagina's zorgt. Er wordt een browser gebruikt om de applet weer te geven en uit te voeren. Normaliter biedt de applet een grafische gebruikersinterface voor de code. In applets reageert een **action()-**methode op een gebeurtenis (event), zoals een muisklik, en bepaalt wat er vervolgens moet gebeuren.
8. Er wordt een exceptie geworpen door een methode als deze een foutconditie ontdekt. De exceptie wordt afgehandeld door het aanroepen van een correcte *handler*, die wordt geselecteerd uit een lijst met handlers, catches genaamd. Deze expliciete catches treden op aan het eind van een omhullend **try**-blok. Een niet-afgevangen exceptie wordt afgehandeld door een standaard Java-handler die een boodschap verstuurt en het programma beëindigt.
9. In tegenstelling tot C++ kan in Java niet op de conventionele wijze worden geprogrammeerd. Alles wordt ingekapseld in een klasse. Hierdoor wordt de programmeur gedwongen alles als object te ontwerpen. Het nadeel hiervan is dat conventionele C-code zich niet zo gemakkelijk aanpast aan Java als aan C++. Het voordeel ervan is dat de Java-programmeur veiligere code schrijft.

Oefeningen

1. Schrijf, gebruikmakend van Java I/O, in het scherm de woorden

   ```
   she sells sea shells by the seashore
   ```

   (a) alle in één regel, (b) in drie regels, (c) binnen een kader.
2. Schrijf een programma dat afstanden in yards converteert naar afstanden in meters. 1 meter komt overeen met 1,0936 yards. Schrijf het programma, indien mogelijk,

zodanig dat afstanden kunnen worden ingelezen; doe het anders via een eenvoudige toekenning aan een instantievariabele binnen de methode **main()**.

3. Schrijf een applet die interactief vraagt om uw *name* en *age* en antwoordt met

   `Hello name, next year you will be next age.`

   waarbij *next_age* gelijk is aan *age + 1*.
4. Schrijf een programma dat een tabel afdrukt met daarin kwadraten, wortels en getallen tot de derde macht. Maak gebruik van tabs of strings met spaties om een keurig uitgelijnde tabel in het scherm te krijgen.

   i i * i square root i * i * i
   _____
   1 1 1.00000 1·····

5. Schrijf een klasse die complexe berekeningen kan uitvoeren. In tegenstelling tot in C++ kunt u in Java geen operatoren overloaden. Schrijf de methode **Complex.plus(Complex)** en de methode **Complex.minus(Complex)** zodat ze een correct **Complex** resultaat teruggeven.
6. Schrijf een klasse **GradStudent()** die **Student** uitbreidt. Voeg aan **Student** extra informatie toe die het onderwerp van het proefschrift van de postdoctoraal student bevat, routines die aan dit deel van de klasse worden toegekend en voor het afdrukken van deze informatie.
7. Voeg excepties toe aan de klasse **GradStudent**, zodat onjuiste initialisatie van **GradStudent**-objecten resulteert in een runtime-exceptie.