Inleiding Programmeren
College 11

Universiteit van Amsterdam

Robin de Vries

11 oktober 2017

Inhoud

Final modifier voor variabelen

Voorheen hebben we de **final** modifier al gebruikt om variabelen immutable te maken:

```
class Paar {
    final double coefficient;
    final int macht;

    Paar(double c, int m) {
        coefficient = c;
        macht = m;
    }
}
```

Final modifier voor variabelen

`final` instantiatievariabelen mogen enkel en éénmalig in de constructor een waarde krijgen:

```
Paar p = new Paar(2, 3);
p.macht = 3;
```

```
error: cannot assign a value to final variable macht
        p.macht = 3;
          ^
1 error
```

Final modifier voor klassen en methoden

De `final` modifier heeft nog een andere functie met betrekking tot overerving:

- een `final` klasse kan niet worden uitgebreid
- een `final` methode kan niet worden overriden

Final modifier voor klassen

```
final class A {

}

class B extends A {

}
```

```
error: cannot inherit from final A
class B extends A {
                ^
1 error
```

Final modifier voor methode

```
class Paar {
    int coefficient, macht;
    final int getMacht() {
        return macht;
    }
}
class ComplexPaar extends Paar {
    int getMacht() {
        return macht;
    }
}
```

```
error: getMacht() in ComplexPaar cannot override getMacht() in Paar
    int getMacht() { return macht; }
        ^
  overridden method is final
```

Inhoud

String formatting

Voorheen als we een bericht wilden printen als “Je mag nog 1 keer raden” waarbij **1** als integer is opgeslagen, deden we iets als:

```
int aantal = 1;
System.out.println("Je mag nog " + aantal + " keer raden");
```

Echter zitten er wat nadelen aan deze methode:

- Het is redelijk complex (overal plusjes om te concateneren)
- het is foutgevoelig (spaties voor het einde van de strings)
- Het is niet erg flexibel

Daarom is er een alternatief: Strings gebruiken met een sjabloon!

String formatting (2)

Met behulp van de functies:

- `String.format(String format, ...)`
- `System.out.format(String format, ...)`
- `System.out.printf(String format, ...)`

Kun je in een sjabloon (format) aangeven welke meegegeven parameters hoe in de uiteindelijke string moeten komen, bijvoorbeeld:

```
int aantal = 1;
System.out.printf("Je mag nog %d keer raden", aantal);
```

String formatting (3)

Het format kan parameters bevatten die vervolgens als argumenten moeten worden meegegeven.

Iedere parameter begint met een %-teken en geeft het type van de parameter aan.

```
System.out.printf("Datum van vandaag: %d-%d-%d", 13, 10, 2016);
```

```
Datum van vandaag: 13-10-2016
```

Mogelijke parameters

Parameter	Betekenis	Voorbeeld	Uitvoer
%d	Geheel getal	System.out.printf("%d", 10)	10
%f	Double/Floating point	System.out.printf("%f", 5.4);	5.400000
%c	Karakter		
%s	String	System.out.printf("Hi %s Ho", "Ha");	Hi Ha Ho
%n	Newline		
%%	Procent	System.out.printf("%%");	%

Hoofdletters

Als je de parameter naar een hoofdletter verandert, bijvoorbeeld van %s naar %S, dan zal Java de volledige uitvoer van dat stuk in hoofdletters zetten:

```
System.out.printf("%S Wereld", "Hallo");
```

```
HALLO Wereld
```

Grootte

Je kunt ook aangeven hoe breed een stuk mag worden, handig als je dingen wilt uitlijnen:

```
String[] teksten = {"Klaslokaal", "Pikachu", "Snorlax"};

for(String tekst : teksten)
    System.out.printf("Tekstje: %10s\n", tekst);
```

```
Tekstje: Klaslokaal
Tekstje:    Pikachu
Tekstje:    Snorlax
```

Als je "%-10s" opgeeft lijnt links ipv rechts uit.

Er is nog veel meer, lees vooral de documentatie!
https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

Inhoud

Reguliere expressies (regexes)

- Met reguliere expressies is het mogelijk een verzameling van Strings te beschrijven.
- Reguliere expressies kunnen gebruikt worden bij zoeken, bewerken en het manipuleren van tekst en data.
- Een **reguliere expressie** is een patroon dat toegepast kan worden op tekst (String)
- Een reguliere expressie **matcht** de tekst (of een gedeelte), of **matcht niet**
- Sinds Java 1.4 **regular expression package**: `java.util.regex`

Bepalen of een String 10 opeenvolgende cijfers heeft

Onderstaande methode zegt van een String of deze wel of niet 10 opeenvolgende cijfers bevat.

```
static boolean hasTenDigits(String s) {
    int noDigits = 0;
    for (char c : s.toCharArray()) {
        if (Character.isDigit(c)) {
            if (++noDigits == 10)
                return true;
        }
        else
            noDigits = 0;
    }
    return false;
}
```

Bepalen of een String 10 opeenvolgende cijfers heeft (2)

De code van de vorige slide is identiek aan dit:

```
static boolean hasTenDigits(String s) {
    return s.matches(".*[0-9]{10}.*");
}
```

En dit is precies de reden dat we graag reguliere expressies gebruiken!

String.matches()

`boolean matches(String regex)` van de String-klasse bepaalt of de gegeven String voldoet aan de opgegeven expressie:

```
String waar = "waar";
System.out.println(waar.matches("waar"));
System.out.println(waar.matches("Waar"));
```

```
true
false
```

String.matches()

Met behulp van blokhaken kun je voor één karakter meerdere mogelijkheden opgeven:

```
System.out.println("waar".matches("[wW]aar"));
System.out.println("Waar".matches("[wW]aar"));
```

```
true
true
```

In dit geval voldoet dus zowel “waar” als “Waar”.

Ranges

Je kunt ook kijken of een karakter binnen een bepaalde range valt, bijvoorbeeld:

```
System.out.println("waar".matches("[a-z]aar"));
System.out.println("xaar".matches("[a-z]aar"));
System.out.println("Zaar".matches("[a-z]aar"));
```

```
true
true
false
```

Beperkt overzicht van mogelijkheden

	Meaning	Example	Meaning
$\cdot$	any character	$[0-9]\cdot$	digit + any other character
$*$	zero or more...	$\cdot*$	any character sequence
		$[0-9]*$	any number of digits
		$a*$	any number of letter a's
?	zero or one...	$[0-9]?$	an optional digit
		$X?$	an optional letter X
$+$	one or more...	$[0-9]+$	one or more digits
$\{x\}$	x instances of...	$[0-9]\{10\}$	ten digits
		$n\{4\}$	4 instances of the letter n
		$\cdot\{3\}$	3 instances of any character
$\{x,y\}$	between x and y inst. of	$[0-9]\{10,14\}$	between 10 and 14 digits
$\{x,\}$	at least x instances of...	$\cdot\{5,\}$	at least 5 characters

Datum patroon

```
String str;

String patroon = "[0-9]{2}-[0-9]{2}-[0-9]{4}"; // dd-mm-jjjj
System.out.println("Patroon is: " + patroon);

str = "27-07-1998";
System.out.println(str + ": " + str.matches(patroon));

str = "51-07-1998";
System.out.println(str + ": " + str.matches(patroon));
```

```
Patroon is: [0-9]{2}-[0-9]{2}-[0-9]{4}
27-07-1998: true
51-07-1998: true
```

Inhoud

Geneste klassen

Naast dat klassen en objecten naast elkaar kunnen bestaan, kunnen ze ook genest worden.
Een klasse is dan onderdeel van een andere klasse. Dit kan op drie manieren:

- Inner klassen (klasse binnen een klasse)
- Lokale klassen (klasse binnen een methode)
- Anonieme klassen

Lees verder:
`https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`

Anonieme klassen

Zoals de naam al doet vermoeden heeft een anonieme klasse geen naam, hij wordt direct aangemaakt en geïnstantieerd op de desbetreffende plek in de code.

Gegeven de volgende interface:

```
interface KlikbaarInterface {
    void klik();
}
```

En de volgende methode:

```
static void klik(int aantal, KlikbaarInterface klikbaar) {
    for(int i = 0; i < aantal; i++)
       klikbaar.klik();
}
```

Wat hebben we nodig om de methode `klik` aan te roepen?

Anonieme klassen (2)

We kunnen de definitie van de klasse dus direct schrijven als argument van de methode:

```
public static void main(String[] args) {
    // andere code
    klik(3, new KlikbaarInterface() {
        public void klik() {
            System.out.println("Klik!");

        }
    });
}
```

```
Klik!
Klik!
Klik!
```

Anonieme klassen (3)

Daarnaast kan een anonieme klasse variabelen raadplegen uit de omliggende context:

```
public static void main(String[] args){
    String wattes = "muis";

    klik(3, new KlikbaarInterface() {
        public void klik() {
            System.out.printf("%sklik\n", wattes);

        }
    });
}
```

```
muisklik
muisklik
muisklik
```

Inhoud

Iterators

Naast de `for`-loop hebben we ook al gekeken naar de `foreach`-loop, welke alle elementen uit een lijst haalde:

```
ArrayList<String> dna_sequenties = new ArrayList<String>();
dna_sequenties.add("AAGGCT");
// ..

for(String sequentie : dna_sequenties) {
    // ...
}
```

En zoals we hebben gezien werkt dit ook bij Arrays. Maar hoe werkt dit?

Iterators (2)

Dit werkt met behulp van **Iterators**. Een Iterator object zorgt ervoor dat je van een lijst of andere datastructuur, één voor één alle elementen kan opvragen.

In Java is er een `Iterator<T>`-interface welke definieert hoe de **Iterator** van een object eruit moet zien. Deze heeft de volgende methoden[1]:

- `boolean hasNext()`: Returns true if the iteration has more elements.
- `T next()`: Returns the next element in the iteration.

[1] Zie ook: `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`

Zelf itereren met een iterator object

Je kunt zelf van een ArrayList dus ook een Iterator-object opvragen:

```
ArrayList<String> dna_sequenties = new ArrayList<String>();
dna_sequenties.add("AAGGCT");
dna_sequenties.add("GGCTAC");
Iterator<String> dna_iterator = dna_sequenties.iterator();

while(dna_iterator.hasNext()) {
    String sequentie = dna_iterator.next();
    System.out.println(sequentie);
}
```

Dit is dus wat Java intern (ongeveer) doet voor de foreach-loop!

Inhoud

Eigen iterators maken

Je kunt ook zelf een klasse schrijven die de `Iterator<T>`-interface implementeert en daarmee je eigen iterator schrijven.

Laten we eens kijken naar een Iterator die de Lucasreeks ophoest!

LucasIterator

```
class LucasIterator implements Iterator<Integer> {

    int a = 0;
    int b = 1;

    public boolean hasNext() {
        return true;
    }

    public Integer next() {
        int temp = a;
        a = b;
        b = temp + b;
        return a;
    }
}
```

LucasIterator (2)

Nu kunnen we deze Iterator gebruiken om Lucasgetallen uit te printen:

```
Iterator<Integer> lucas_iterator = new LucasIterator();
while(lucas_iterator.hasNext())
{
    int lucas = lucas_iterator.next();
    if(lucas > 20)
        break;
    System.out.format("%d ", lucas);
}
```

```
1 1 2 3 5 8 13
```

Van iterator naar de foreach-loop met de iterable interface

Maar dit werkt nog niet met een foreach-loop, want we geven daar tenslotte het object waarover we willen itereren mee en niet de iterator.

Java kent een `Iterable<T>`-interface[2] :

- `Iterator<T> iterator()`: Returns an iterator over elements of type T.

De foreach-loop zal de **iterator**-methode aanroepen van het object dat je meegeeft.

[2] `https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html`

Iterable en iterator tegelijkertijd

Met een beetje een lelijke truc kunnen we onze iterator iterable maken:

```
class LucasIterator implements Iterable<Integer>, Iterator<Integer> {
    int a = 0;
    int b = 1;

    public Iterator<Integer> iterator() {
        return this;
    }

    public boolean hasNext() {
        return true;
    }
    public Integer next() {
        int temp = a;
        a = b;
        b = temp + b;
        return a;
    }
}
```

Iterable en iterator tegelijkertijd (2)

En vervolgens werkt het dus met de foreach-loop!

```
for(int lucas : new LucasIterator())
{
    if(lucas > 100)
        break;
    System.out.format("%d ", lucas);
}
```

```
1 1 2 3 5 8 13 1 1 2 3 5 8 13 21 34 55 89
```

Itereerbare boekenkast

Tot zover onze lucasreeks iterator, nu tijd om een echte iterator te maken. Laten we de boekenkast van de vorige keer itereerbaar maken. Dit kan op diverse manieren:

- Met een aparte klasse
- Met een inner klasse (klasse binnen een klasse)
- Met een lokale klasse (klasse binnen een methode)
- Met een anonieme klasse

Laten we eens de diverse mogelijkheden bekijken!

Bordspellen

Gegeven de volgende bordspelletjes:

```
class Bordspel { }
class Monopoly extends Bordspel {
    public String toString() {
        return "Monopoly";
    }
}
class Stratego extends Bordspel {
     public String toString() {
         return "Stratego";
     }
}
```

Boekenkast

```
class Boekenkast<T> implements Iterable<T> {
    Object[] planken;
    int laatste_plank;

    Boekenkast(int grootte) {
        planken = new Object[grootte];
    }

    void zetOpPlank(T voorwerp) {
        if(laatste_plank == planken.length)
            return;
        planken[laatste_plank++] = voorwerp;
    }

    public Iterator<T> iterator() {
        // Deze regel is afhankelijk van onze implementatie
    }
}
```

Doel: itereren

Een correcte implementatie doet straks dit:

```
Boekenkast<Bordspel> spelletjeskast = new Boekenkast<Bordspel>(5);
spelletjeskast.zetOpPlank(new Monopoly());
spelletjeskast.zetOpPlank(new Stratego());

for(Bordspel spelletje : spelletjeskast)
    System.out.println(spelletje);
```

```
Monopoly
Stratego
```

Met een aparte klasse

```
class BoekenkastIterator<T> implements Iterator<T> {
    Boekenkast<T> kast;
    int teller;

    public BoekenkastIterator(Boekenkast<T> kast){
        this.kast = kast;
    }

    public boolean hasNext() {
        return teller < kast.laatste_plank;
    }

    @SuppressWarnings("unchecked")
    public T next() {
        return (T) kast.planken[teller++];
    }
}
```

Vervolgens kun je een iterator maken met:

```
return new BoekenkastIterator<T>(this);
```

Nadelen van het gebruiken van een aparte klasse?

Je kunt de interne variabelen niet `private` maken, anders kan de iterator er niet bij!

Oplossing? Geneste klassen!

Als inner klasse

```
class Boekenkast<T> implements Iterable<T> {
    // code van eerder

    public Iterator<T> iterator() {
        return new BoekenkastIterator<T>();
    }

    class BoekenkastIterator<T> implements Iterator<T> {
        int teller;

        public boolean hasNext() {
            return teller < laatste_plank;
        }

        @SuppressWarnings("unchecked")
        public T next() {
            return (T) planken[teller++];
        }
    }
}
```

Als lokale klasse

```
class Boekenkast<T> implements Iterable<T> {
    // code van eerder

    public Iterator<T> iterator() {
        class BoekenkastIterator<E> implements Iterator<E> {
            int teller;

            public boolean hasNext() {
                return teller < laatste_plank;
            }

            @SuppressWarnings("unchecked")
            public E next() {
                return (E) planken[teller++];
            }
        }
        return new BoekenkastIterator<T>();
    }
}
```

Als anonieme klasse

```
class Boekenkast<T> implements Iterable<T> {
    // code van eerder

    public Iterator<T> iterator() {
        return new Iterator<T>() {
            int teller;

            public boolean hasNext() {
                return teller < laatste_plank;
            }

            @SuppressWarnings("unchecked")
            public T next() {
                return (T) planken[teller++];
            }
        };
    }
}
```