

# 1 Herhaling

Programmeren 2 – Java

2017 - 2018

**KdG** Karel de Grote  
Hogeschool

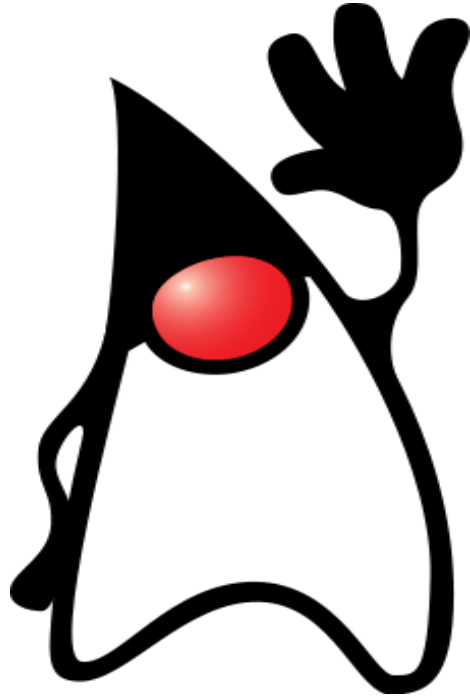
Kris Behiels  
Jan De Rijke  
Mark Goovaerts

# Programmeren 2 - Java

---

1. Herhaling en Collections
2. Generics en documenteren
3. Annotations en Reflection
4. Testen en logging
5. Design patterns (deel 1)
6. Design patterns (deel 2)
7. Lambda's en streams
8. Persistentie (JDBC)
9. XML en JSON
10. Threads
11. Synchronization
12. Concurrency





---

# Herhaling

# Agenda

---

## 1. Herhaling

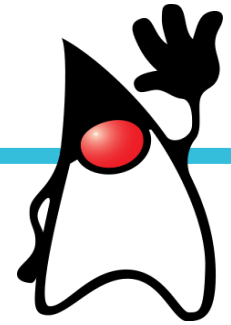
- Inleiding
- OO concepten
  - Inkapseling en overerving
  - Polymorfisme, overloading en overriding
  - Abstract en interface
  - final en static
  - Toegankelijkheid

## 2. Collections

- List
- Set
- Map



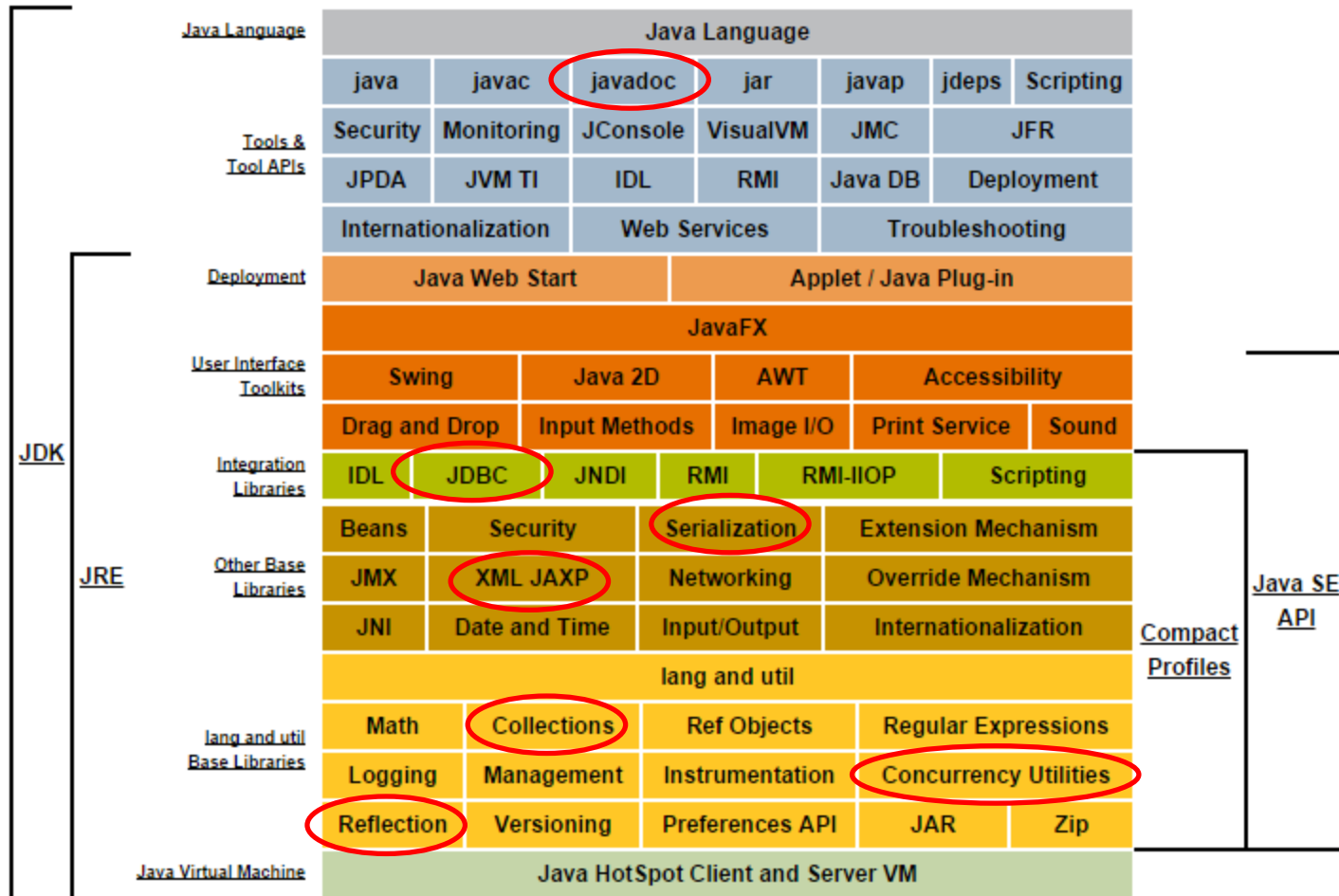
# Evolutie van Java



- JDK 1.0 → 1996 (Oak)
- JDK 1.1 → 1997
- J2SE 1.2 → 1998 (Playground)
- J2SE 1.3 → 2000 (Kestrel)
- J2SE 1.4 → 2002 (Merlin)
- J2SE 5.0 → 2004 (Tiger)
- Java SE 6 → 2006 (Mustang)
- Java SE 7 → 2011 (Dolphin)
- Java SE 8 → 2014 (Spider)
- Java SE 9 → 2017

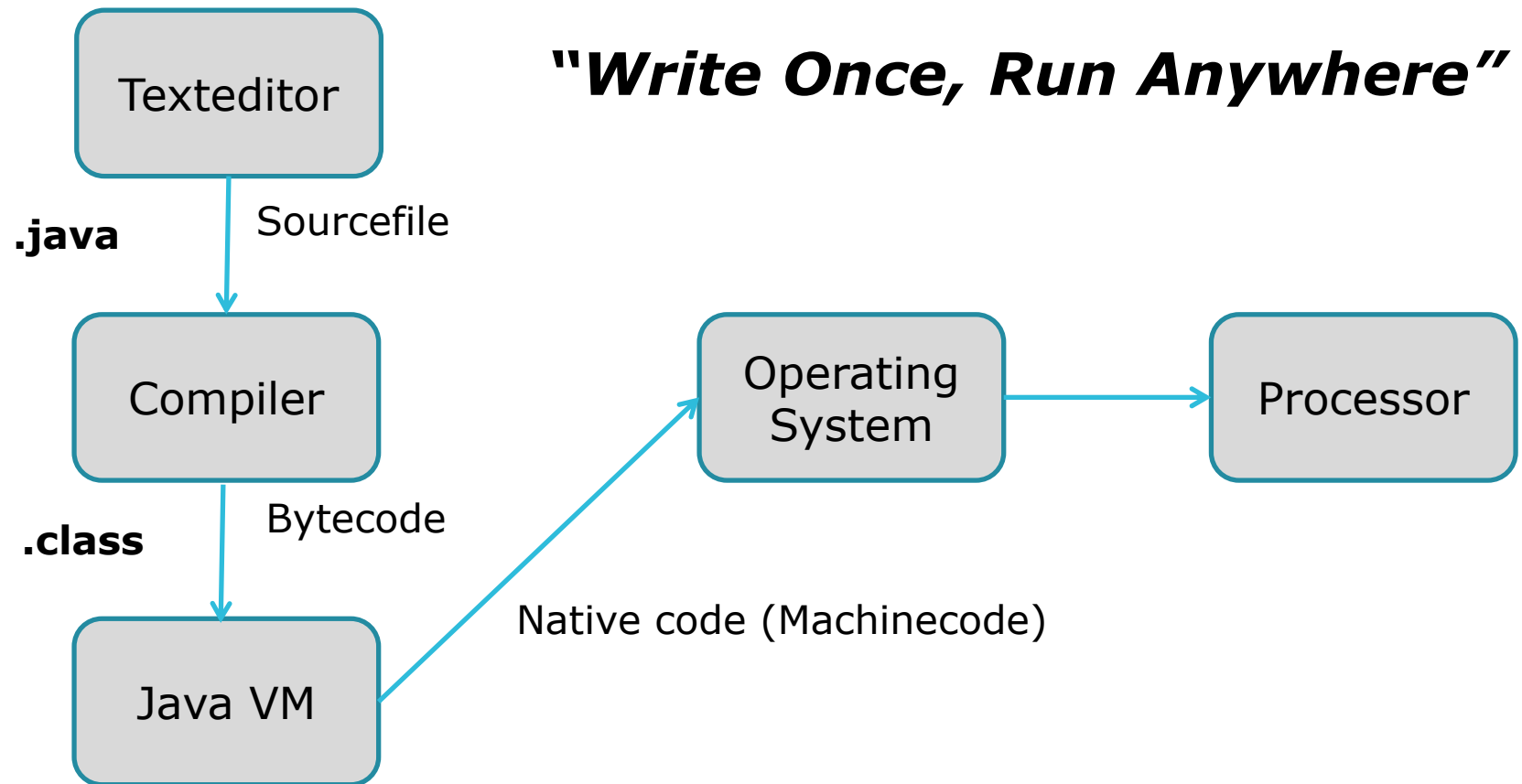
Gebruiken wij!

# Java SE8



# Java-programma

---



# Java-programma

---

- Edit
  - Source code (\*.java)
- Compile
  - Byte code (\*.class)
- Load
  - Byte code → memory
- Verify
  - Security
- Execute
  - Interpreter
  - JIT compiler



## IDE

- **IntelliJ**
- Eclipse
- Netbeans
- ...

## Java Virtual Machine (JVM)



# Agenda

---

## 1. Herhaling

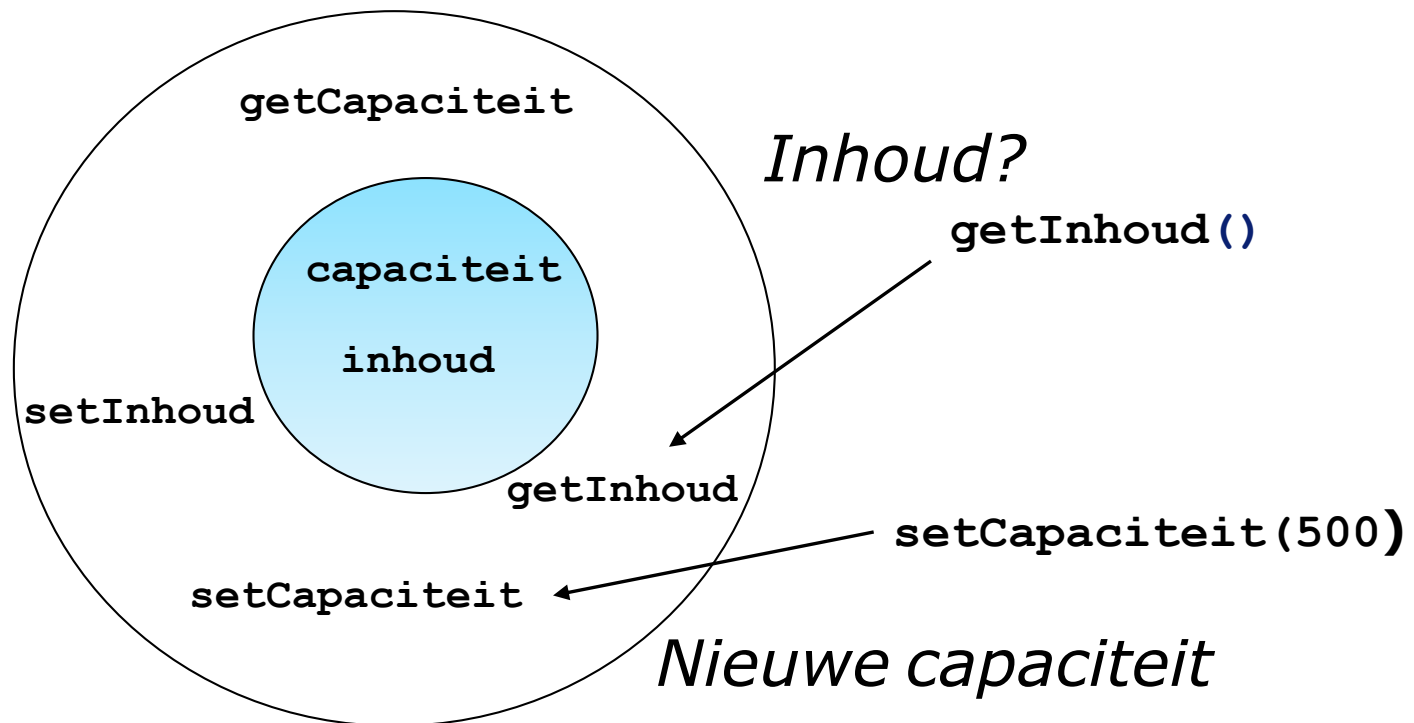
- Inleiding
- OO concepten
  - Inkapseling en overerving
  - Polymorfisme, overloading en overriding
  - Abstract en interface
  - final en static
  - Toegankelijkheid



## 2. Collections

- List
- Set
- Map

## *Object van de klasse Vat*



# Inkapseling

---

- De inhoud van een object is in principe niet rechtstreeks toegankelijk (**encapsulation**), omwille van beveiliging en het principe van information-hiding.
- Er moet gebruikt gemaakt worden van boodschappen (*methods*) om de attributen van een object te benaderen.
- Objecten kunnen ook boodschappen naar andere objecten sturen.

# Inkapseling

---

```
public class Vat {  
    private int inhoud;  
    private int capaciteit;  
    public Vat() {  
    }  
    public int getInhoud() {  
        return inhoud;  
    }  
    public void setCapaciteit(int capaciteit) {  
        if(capaciteit > 0) {  
            this.capaciteit = capaciteit;  
        }  
        ... }  
}
```

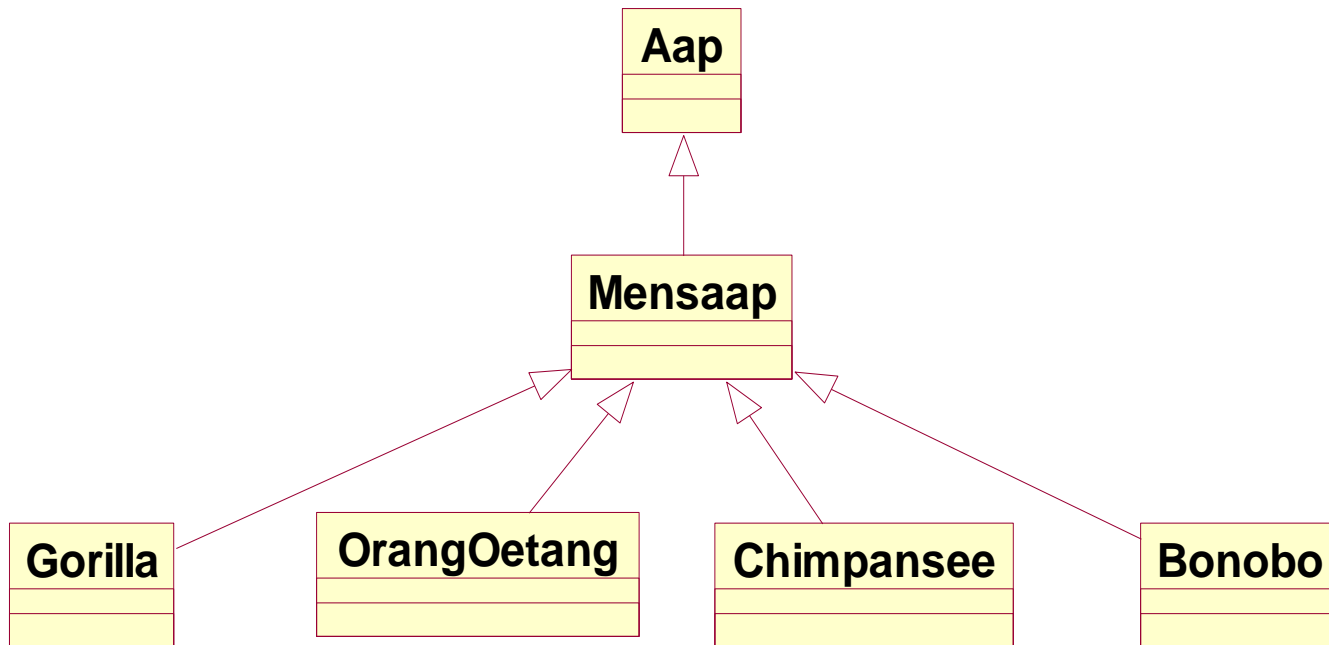
default constructor

getter

setter

# Overerving

- Overerving (**inheritance**) maakt het mogelijk om code te hergebruiken, zo kan je zowel attributen als methoden van een klasse recupereren.



Een nieuwe klasse maken kan op twee manieren:

- **specialisatie:**

- de bestaande klasse wordt de superklasse en de nieuwe klasse wordt de subklasse

- **generalisatie:**

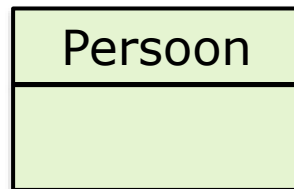
- een nieuwe superklasse wordt gevonden door abstractie te maken van de onderliggende subklassen

# Specialisatie

---

Originele klasse:

```
public class Persoon {  
    private String naam;  
  
    public Persoon(String naam) {  
        this.naam = naam;  
    }  
  
    public String getNaam() {  
        return naam();  
    }  
  
    public String toString() {  
        return ...  
    }  
}
```

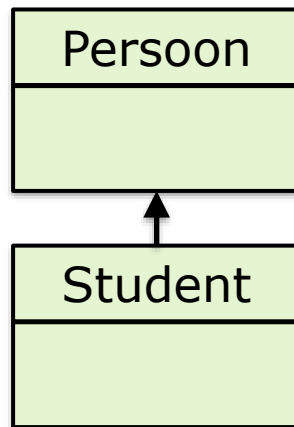


# Specialisatie

## Originele klasse

→ nu superklasse / parent:

```
public class Persoon {  
    private String naam;  
  
    public Persoon(String naam) {  
        this.naam = naam;  
    }  
  
    public String getNaam() {  
        return naam();  
    }  
  
    public String toString() {  
        return ...  
    }  
}
```



## Nieuwe klasse

→ subklasse / child:

```
public class Student  
    extends Persoon {  
        private int nummer;  
  
        public Student(String naam,  
                        int nummer) {  
            super(naam);  
            this.nummer = nummer;  
        }  
  
        public String toString() {  
            return super.toString() ...  
        }  
    }
```

### Specialisatie:

Nieuwe klasse wordt  
eronder geplaatst



# Generalisatie

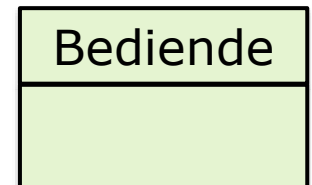
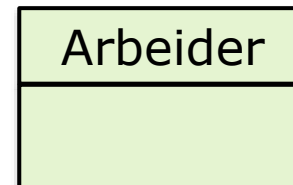
---

## Originele klasse:

```
public class Arbeider {  
    // attributen en methoden...  
}
```

## Originele klasse:

```
public class Bediende {  
    // attributen en methoden...  
}
```



# Generalisatie

**Nieuwe klasse → superklasse / parent:**

```
public class Werknemer {  
    // Gemeenschappelijke attributen  
    // Gemeenschappelijke methoden
```

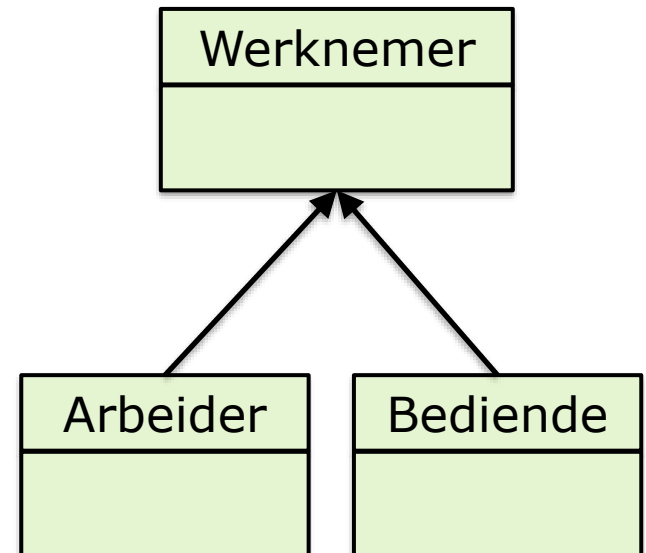
**Originele klasse → subklasse / child:**

```
public class Arbeider extends Werknemer {  
    // attributen en methoden...  
}
```

**Originele klasse → subklasse / child:**

```
public class Bediende extends Werknemer {  
    // attributen en methoden...  
}
```

**Generaliserende**  
klasse wordt erboven  
geplaatst



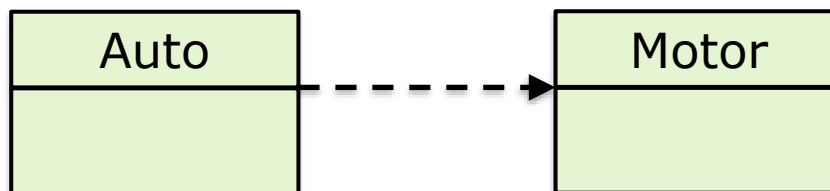
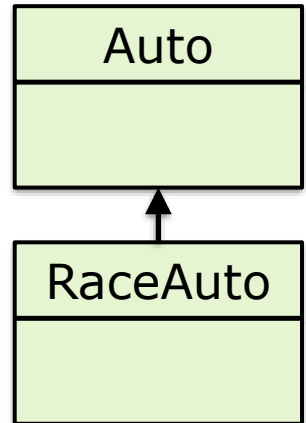
# Wanneer overerving?

- Alleen als er een "**is-een**" relatie tussen beide klassen bestaat

```
public class RaceAuto extends Auto {...
```

- Een alternatief voor overerving is **delegatie**, in dit geval is er sprake van een "**heeft-een**" relatie

```
public class Auto {  
    private Motor motor;  
}
```





# Overerving of delegatie?

---

- Vierkant – Veelhoek
- Detaillijn – Factuur
- Auto – Stuur
- Kleuter – Kind
- Persoon – Adres
- Cirkel – Straal

# Overloading

- meer dan 1 methode met dezelfde naam in één klasse
- voorwaarde: verschillende parameterset
- ook mogelijk bij constructor
- Voorbeeld:

```
System.out.println("Hello world!");  
System.out.println(3.1415);  
System.out.println(42);
```

AClass
method() method(int i) method(double d)

de methode println is  
verschillende keren  
**overloaded**, voor elk  
type parameter.

# Overloading

- meer dan 1 methode met dezelfde naam in één klasse
- voorwaarde: verschillende parameterset
- ook mogelijk bij constructor

AClass
method() method(int i) method(double d)

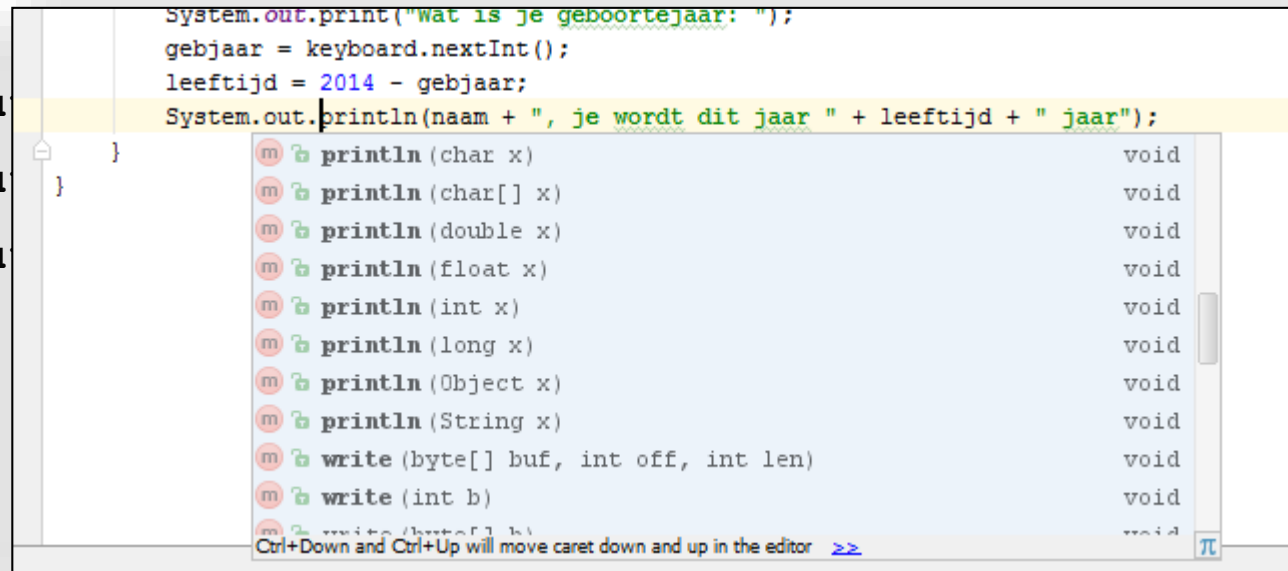
- Voorbeeld:

System.out

System.out

System.out

```
System.out.print("wat is je geboortejaar: ");
gebjaar = keyboard.nextInt();
leeftijd = 2014 - gebjaar;
System.out.println(naam + ", je wordt dit jaar " + leeftijd + " jaar");
}
```



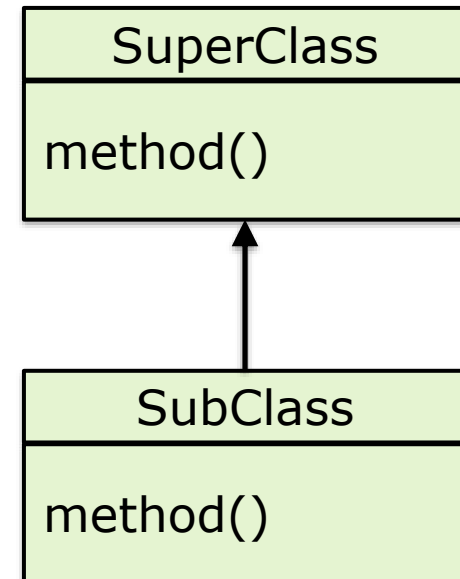
The screenshot shows a code editor with a Java program snippet. The code uses `System.out.print` and `System.out.println`. Below the code, a dropdown menu is open, displaying a list of `println` and `write` methods with their signatures and return types. The list includes overloads for various data types: `char`, `char[]`, `double`, `float`, `int`, `long`, `Object`, and `String`, all returning `void`. It also includes `write` methods for `byte[]` and `int`, also returning `void`. The IDE interface includes a search icon, a list of methods, and a scroll bar.

m	println (char x)	void
m	println (char[] x)	void
m	println (double x)	void
m	println (float x)	void
m	println (int x)	void
m	println (long x)	void
m	println (Object x)	void
m	println (String x)	void
m	write (byte[] buf, int off, int len)	void
m	write (int b)	void
m	write (byte[] b)	void

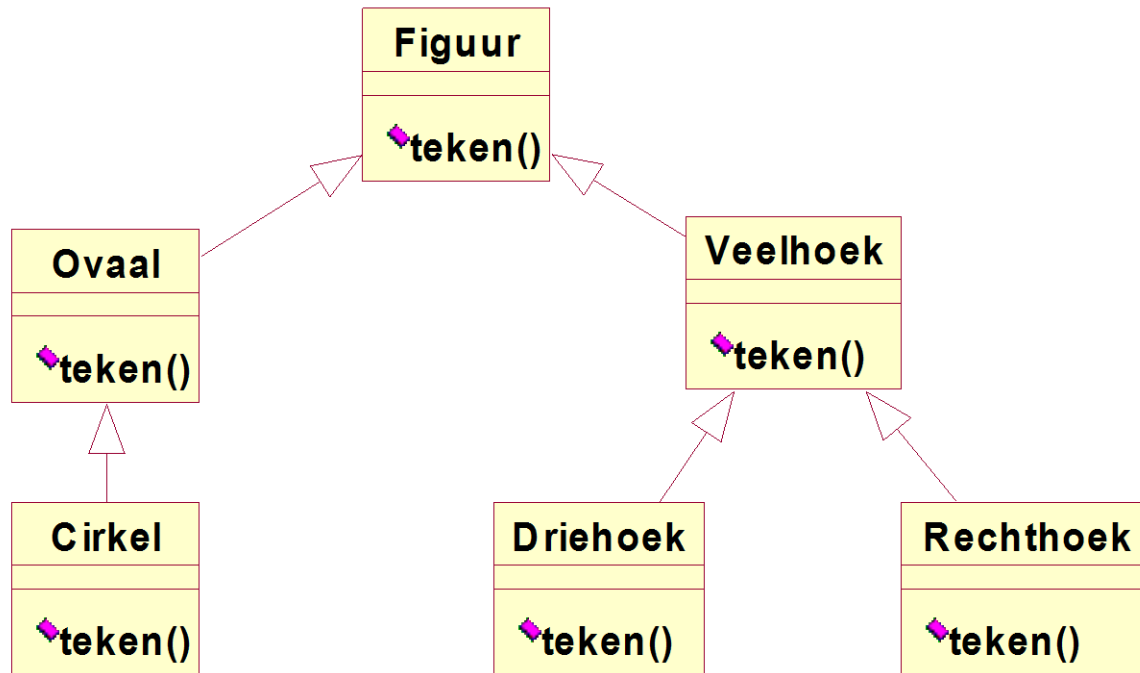
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>>

# Overriding

- vervangt de overgeërfde methode door een nieuwe methode in een kind-klasse van dezelfde hiërarchie.
- voorwaarde: zelfde naam en parameterset
- Voorbeeld:  
De `toString` methode van de klasse `Object` wordt heel vaak *overriden*.



# Polymorfisme



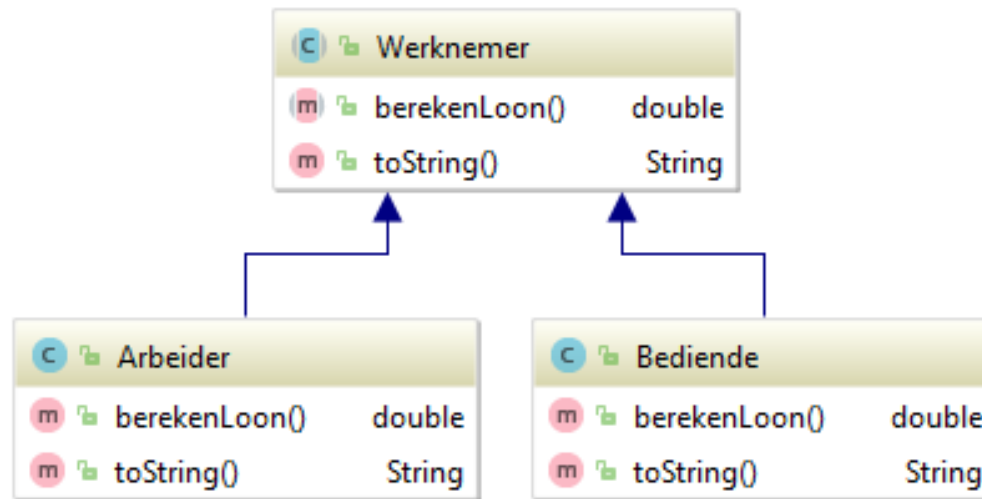
```
List<Figuur> figuren = new ArrayList<>();  
// De List wordt gevuld met allerlei Figuur-objecten...
```

```
for(Figuur figuur : figuren) {  
    figuur.teken();  
}
```

Hier spreken we van  
**polymorfisme**



# Probleem?



- Waar zet ik de methode **berekenLoon**?
  - In de superklasse heb ik te weinig informatie om de methode uit te werken
  - Enkel in de subklassen? Dan kan ik geen polymorfisme toepassen:

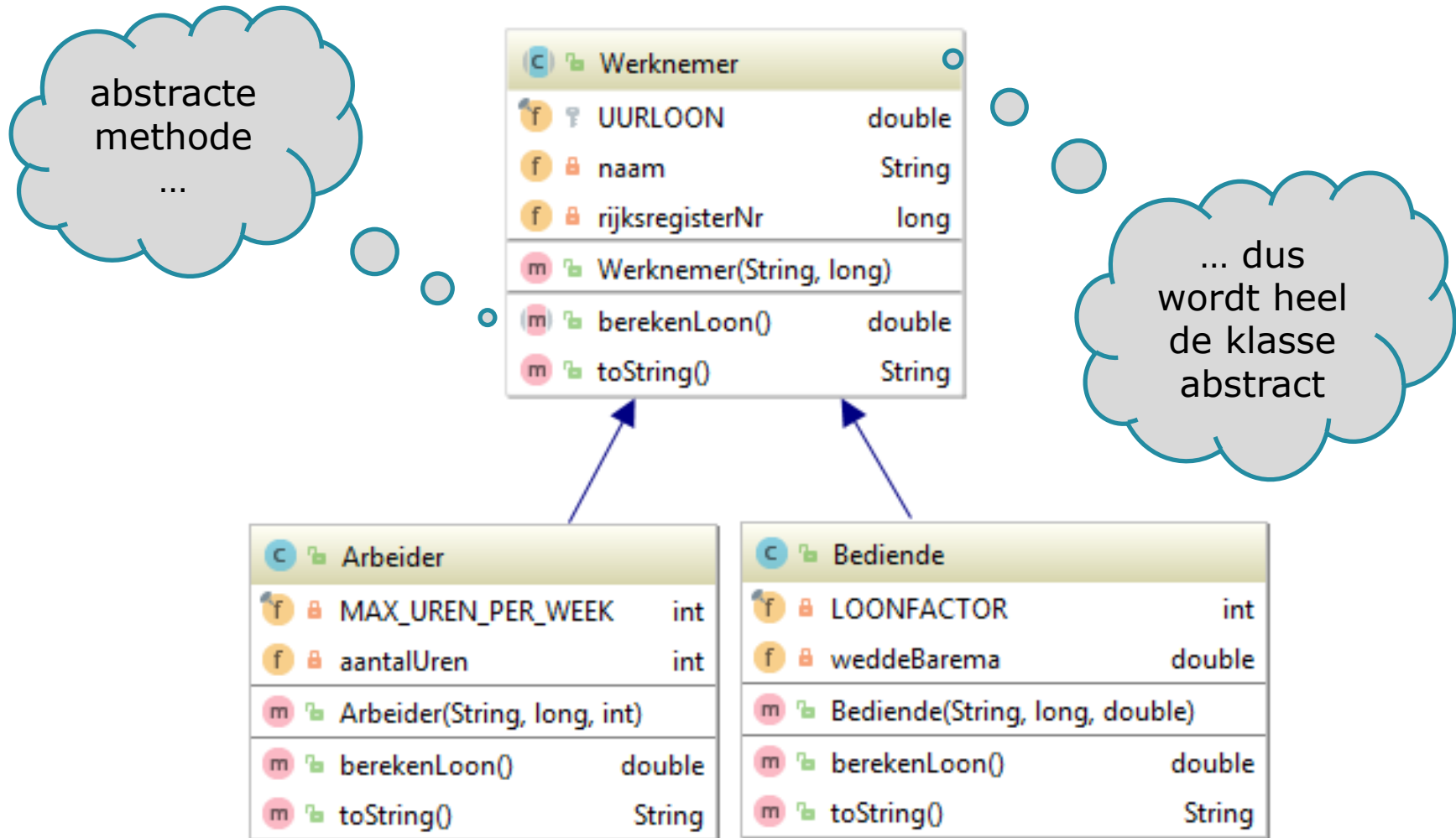
```
for (Werknemer werknemer : werknemers) {
    System.out.println(werknemer.berekenLoon());
}
```

# Abstracte klassen en methoden

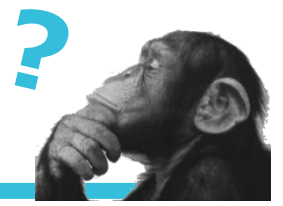
---

- Soms kan je in een superklasse een bepaalde methode niet uitwerken, omdat je niet over alle informatie beschikt. Toch wil je polymorfisme toepassen.
  - Maak die methode abstract
  - Daardoor wordt de hele klasse abstract
  - De subklassen zijn verplicht om de abstracte methode te overriden.
  - De abstracte klasse kan nu wel niet meer rechtstreeks geïntantieerd worden

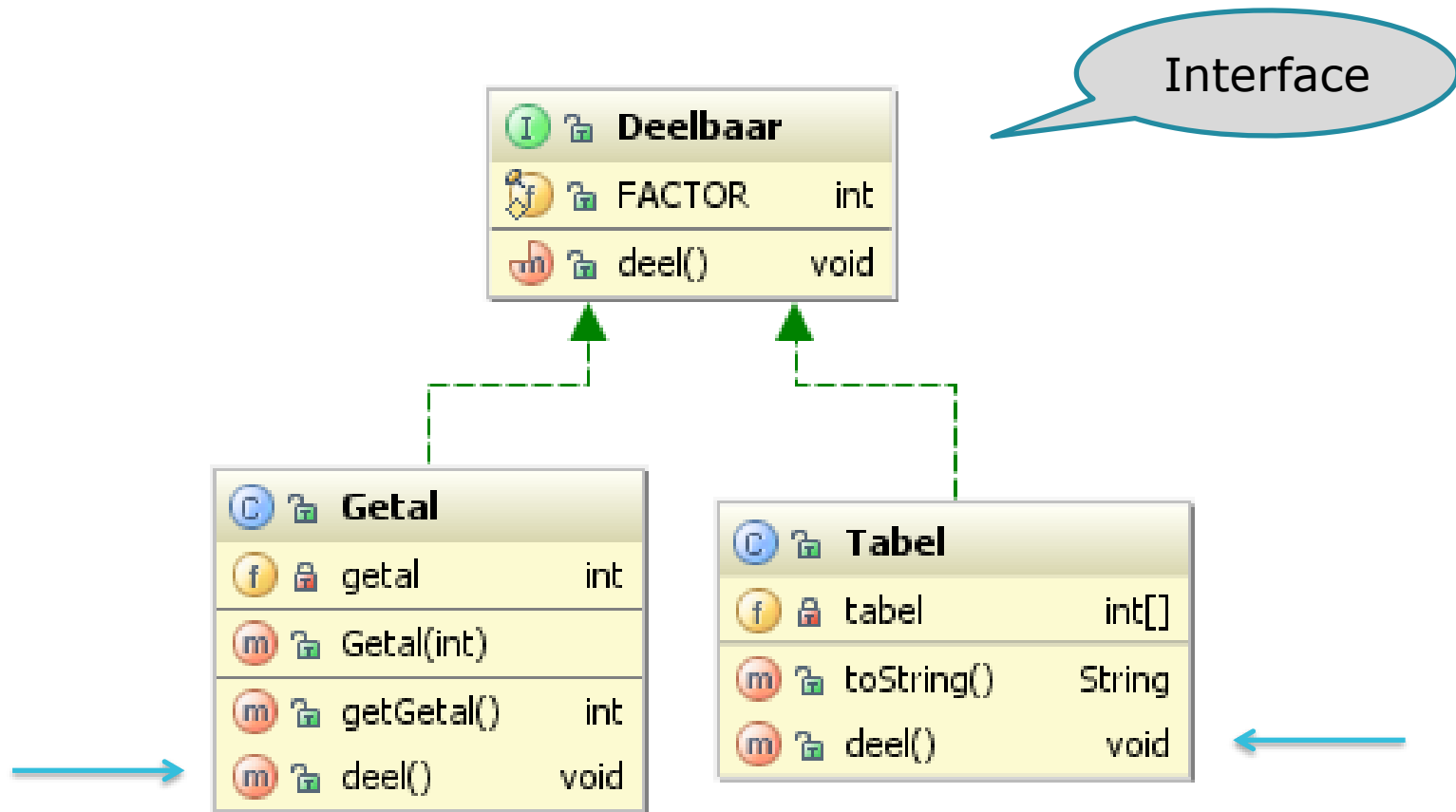
# Abstracte klasse: voorbeeld



# Interface?



# Interface: voorbeeld



# Interface: voorbeeld

```
public interface Deelbaar {  
  
    int FACTOR = 2; // public static final  
  
    void deel(); // public abstract  
}  
  
public class Getal implements Deelbaar {  
    ...  
    public void deel() {  
        getal /= FACTOR; //getal wordt gehalveerd  
    }  
}  
  
public class Tabel implements Deelbaar {  
    ...  
    public void deel() {  
        //tabel wordt gehalveerd  
    }  
}
```

De klassen `Getal` en `Tabel` zijn verplicht om de methode `deel` te implementeren. Elke klasse doet dat op zijn manier



# Interface

---

- Met behulp van een interface kan je andere klassen verplichten om een set van gemeenschappelijke methoden te implementeren.
- Alle methoden in een interface zijn automatisch **public abstract**.
- Alle attributen zijn automatisch **public static final**.
- Standaard interfaces in Java:  
**Comparable, Serializable, Runnable, List, Set, Map,...**
- Multiple inheritance met interfaces is mogelijk:

```
public MyClass extends SuperClass implements  
    Comparable, Serializable, Runnable {  
    // ...  
}
```

# Static

---

- Een **static attribuut** (of *klasse attribuut*) is een attribuut dat verbonden is met de klasse, staat dus los van de gecreëerde objecten.

–Een voorbeeld uit Java zelf:

`Math.PI`

- Een **static methode** benader je door de naam van de klasse voor de methodcall te plaatsen

–Een voorbeeld uit Java zelf:

`Math.max(double a, double b)`



# Static voorbeeld (1)

```
public class Werknemer {  
    private String naam;  
    private static int aantal = 0;  
  
    public Werknemer(String naam) {  
        this.naam = naam;  
        aantal++;  
    }  
  
    public String getNaam() {  
        return naam;  
    }  
  
    public static int getAantal() {  
        return aantal;  
    }  
}
```

← instance attribuut

← klasse attribuut

← Bij elke creatie van een nieuwe Werknemer, wordt het aantal verhoogd (voor heel de klasse)

← gewone getter om een instance variabele te bereiken

← een getter om een static variabele te bereiken is zelf ook static

## Static voorbeeld (2)

```
public class DemoStatic {  
    public static void main(String[] args) {  
        Werknemer wn1 = new Werknemer("Fred");  
        Werknemer wn2 = new Werknemer("Annit");  
  
        System.out.printf("Werknemer wn1 heet: %s\n", wn1.getNaam());  
        System.out.printf("Werknemer wn2 heet: %s\n", wn2.getNaam());  
        System.out.println("Aantal werknemers:" + Werknemer.getAantal());  
    }  
}
```

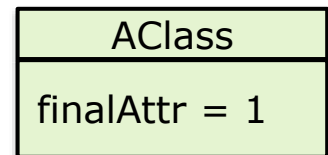
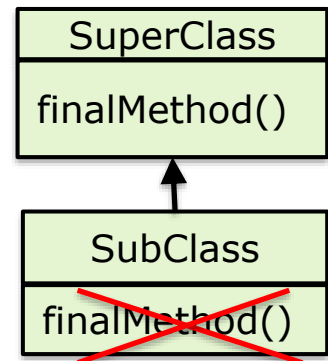
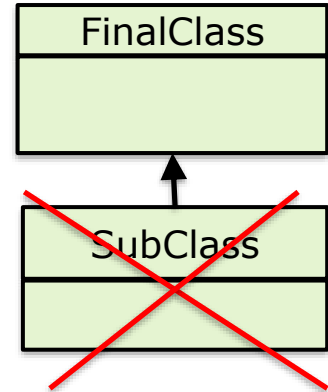
instance attribuut  
bereikbaar via het object

static attribuut bereikbaar  
via de klasse



# Final

- final klasse
  - Van een **final** klasse kan je geen andere klassen meer afleiden  
(<> abstracte klasse!!!)
- final methode
  - Een final methode kan niet overriden worden in een afgeleide klasse  
(<> abstracte methode!!!)
- final attribuut
  - Een final attribuut kan slechts één keer een waarde krijgen



~~Aclass.finalAttr++;~~

# Final

---

- Waarom een klasse final maken?
  - Veiligheid
  - Snelheid (geen polymorfisme mogelijk)
- Voorbeeld:
  - de klasse `String` is `final` omwille van performantieredenen
- meer weten?



<http://stackoverflow.com/questions/2068804/why-is-string-final-in-java>

# Toegankelijkheid

---

## 4 toegankelijkheidsniveaus:

- **private**: alleen toegankelijk binnen de klasse
- (*geen*): toegankelijk vanuit elke klasse binnen de package
- **protected**: toegankelijk vanuit elke klasse binnen de package en voor subklassen buiten de package
- **public**: vrij toegankelijk voor alle klassen

Het "***Principle of least privilege***" luidt dat je een variabele / methode altijd **zo privaat mogelijk** declareert!

# Overzicht toegankelijkheid

---

Aanduiding	klasse	package	subklasse	iedereen
<b>private</b>	x			
<i>(geen)</i>	x	x		
<b>protected</b>	x	x	x	
<b>public</b>	x	x	x	x

# Welke toegankelijkheid gebruiken?



- Attributen:
  - `private`, soms (package)
  - `protected`: eerder zelden
  - `public` alleen bij ...
- Methoden:
  - `public` als ze ...
  - `private` als ze ...
- Klassen:
  - per java-file slechts één public klasse = naam van bestand!
  - `private` of (package) voor *inner classes*

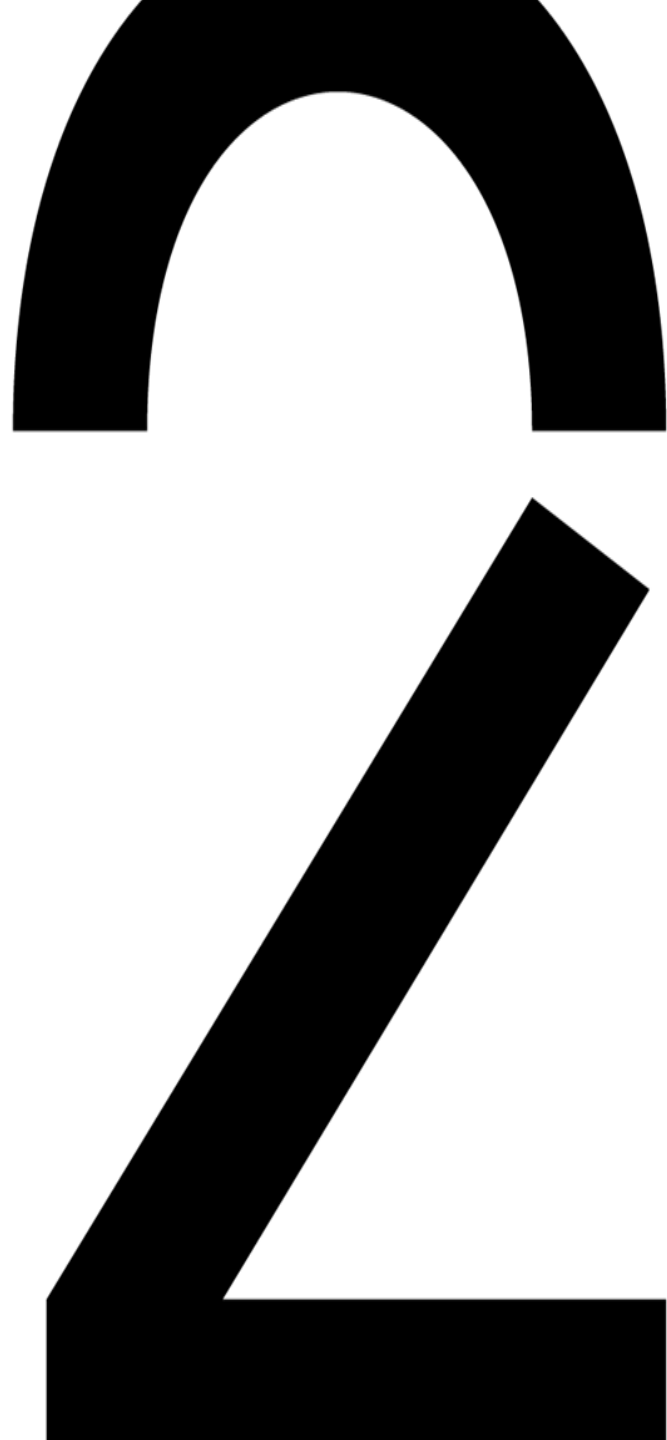
# Opdrachten

---



- Opdrachten op 
  - Interface opdracht "Deelbaar"
  - Herhalingsopdracht (Team)





---

# Collections

# Agenda

---



## 1. Herhaling

- Inleiding
- OO concepten
  - Inkapseling en overerving
  - Polymorfisme, overloading en overriding
  - Abstract en interface
  - final en static
  - Toegankelijkheid

## 2. Collections

- List
- Set
- Map

# Wat is een collection class?

---

- Een collection class is een *dynamische* container
- Is geen klassieke *statische* array
- Eerste Java-versie → alleen **Vector**, **Stack** en **Hashtable**
- Vanaf Java 2 → volwaardig *collections framework*
- Vanaf Java 5 → uitbreidingen in het framework (o.a. **Queue**) en **generics**!
- Belangrijk: een collection bevat NIET de objecten zelf, maar WEL de referenties naar die objecten (pointers!)

# Collections Framework

---

Het *framework* bestaat uit drie onderdelen:

- **Interfaces**

abstracte datatypes om de verzamelingen voor te stellen, onafhankelijk van hun interne uitwerking bewerken.

- vb: `List`, `Set`, `Map`

- **Implementaties**

de concrete implementaties van de verzamelingenklassen.

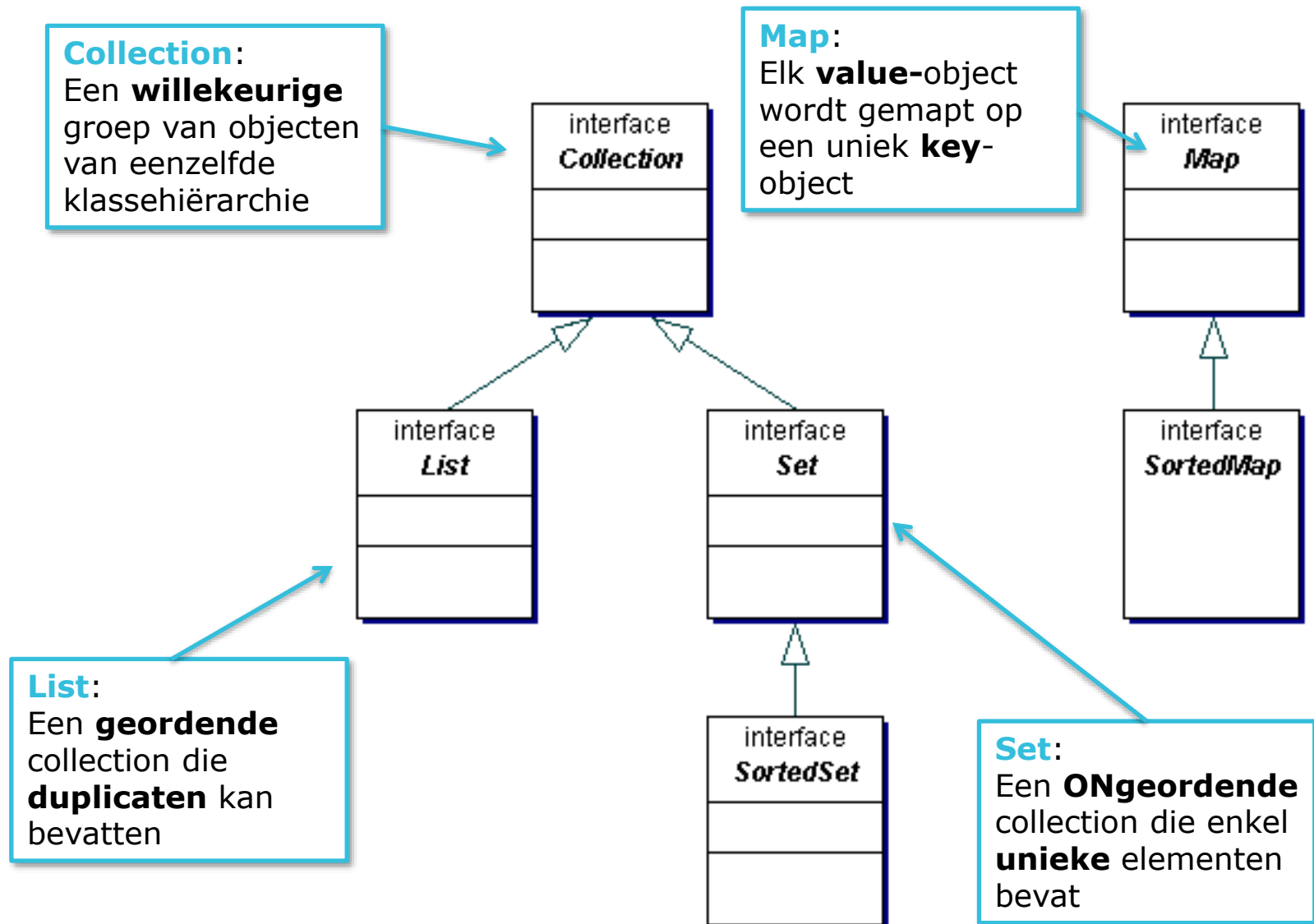
- vb: `LinkedList`, `HashSet`, `TreeMap`

- **Algoritmen**

polymorfe, static methoden die algemene bewerkingen op verzamelingen kunnen uitvoeren.

- vb: `Collections.sort`, `Collections.shuffle`, ...

# De belangrijkste interfaces



static methods van de klasse **Collections** :

- void **sort**(List list);
- int **binarySearch**(List list, Object key);
- void **reverse**(List list);
- void **shuffle**(List list);
- Object **min**(Collection col);
- Object **max**(Collection col);
- void **copy**(List dest, List src);
- List **unmodifiableList**(List list);
- Set **unmodifiableSet**(Set set);
  
- enz...

# Met een lus doorheen een List

```
List<Student> myList = new ArrayList<>()
```

*//... myList wordt gevuld ...*

1

```
for(int i = 0; i < myList.size(); i++) {  
    System.out.println(myList.get(i));  
}
```

shortcut IntelliJ:  
**itli <TAB>**

2

```
for(Student student : myList) {  
    System.out.println(student);  
}
```

shortcut IntelliJ:  
**iter <TAB>**

3

```
for(Iterator<Student> it = myList.iterator(); it.hasNext();) {  
    System.out.println(it.next());  
}
```

shortcut IntelliJ:  
**itco <TAB>**

# Iterator

---

- Om de elementen van een collection te doorlopen
- Kan op verschillende soorten collecties toegepast worden
- Opgelet: tijdens het gebruik van een iterator mag je niets aan de collection wijzigen (o.a. geen add() of remove()).  
→ Wel via de iterator zelf

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```





# Verwijderen: wat is het probleem?

```
public class Probleem {  
    private static int[] data = {5, 6, 3, 2, 4, 1};  
  
    public static void main(String[] args) {  
        List<Integer> myList= new ArrayList<>();  
        for (int i : data) {  
            myList.add(i);    // Autoboxing  
        }  
        for (int i = 0; i < myList.size(); i++) {  
            if(myList.get(i) % 2 == 0) {  
                myList.remove(i); // Verwijder de even getallen  
            }  
        }  
  
        for (Integer i : myList) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }  
}
```

**ONVEILIG**  
verwijderen!

5 3 4 1

# Verwijderen (oplossing)

```
public class Probleem {  
    private static int[] data = {5, 6, 3, 2, 4, 1};  
  
    public static void main(String[] args) {  
        List<Integer> myList = new ArrayList<>();  
        for(int i : data) {  
            myList.add(i);    // Autoboxing  
        }  
        for(Iterator<Integer> it = myList.iterator();  
            it.hasNext();) {  
            if(it.next() % 2 == 0) {  
                it.remove(); // Verwijder de even getallen  
            }  
        }  
  
        for (Integer i : myList) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }  
}
```

**VEILIG**  
verwijderen  
via de iterator

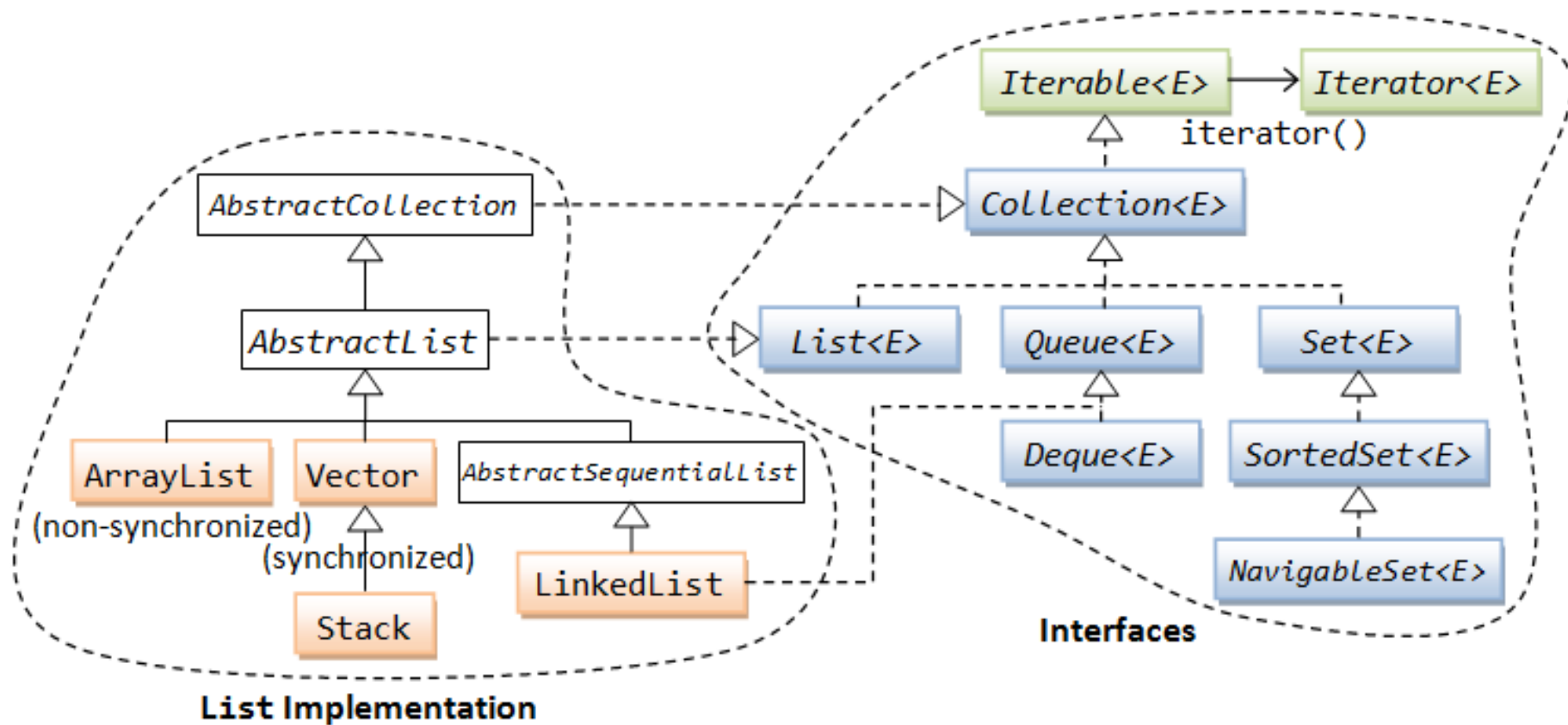
5 3 1

# De interface List

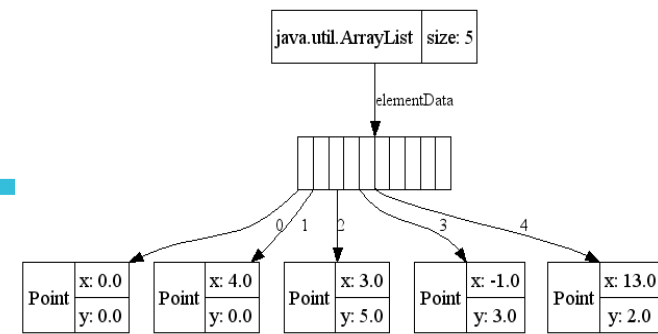
---

```
public interface List<E> extends Collection<E> {  
    // Toegang  
    E get(int index);  
    E set(int index, E element);  
    boolean add(Object element);  
    void add(int index, E element);  
    E remove(int index);  
    boolean addAll(int index, Collection<? extends E> c);  
    // Zoeken  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    // Iteratie  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
    // Deellijst  
    List<E> subList(int from, int to);  
}
```

# List implementaties



# ArrayList

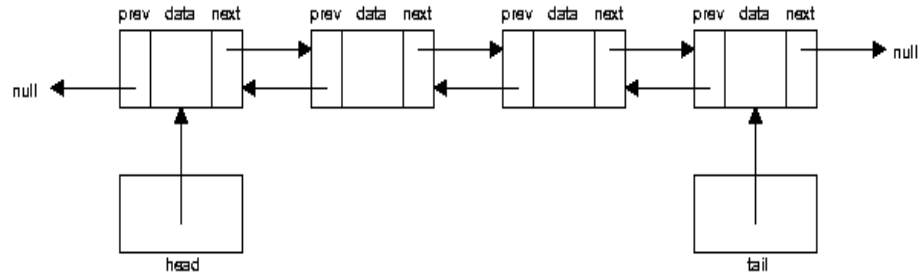


- Implementatie van **List** interface  
→ alle methoden van **Collection** en **List**.
- = dynamische array
- Elk object is toegelaten (ook **null**)
- Primitieve types: automatisch auto(un)boxing
- Capacity:
  - 10 by default of vastleggen via de constructor
  - breidt automatisch uit:  

```
if(capacity < size) {capacity = ((capacity * 3) / 2) + 1;}
```
- **ensureCapacity** methode.

Dus bijvoorbeeld:  
van 10 naar 16

# LinkedList



- Implementatie van **List** interface
  - ➔ alle methoden van **Collection** en **List**.
- uitgewerkt als *double linked list*
- elk object is toegelaten (ook **null**)
- geen default capacity: niet nodig (waarom?)
- extra methoden om bewerkingen te doen aan het begin of het einde van de lijst:
  - `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, `removeLast`

# ArrayList of LinkedList

---

- ArrayList
  - snel bij positioneren (get en set)
  - traag bij `add(int index, Object o)` en `remove`
- LinkedList
  - snel bij alle `add` en `remove` methoden
  - traag bij positioneren (get en set)

# List – sorteren (1)

```
public class SortDemo1 {  
    public static final String[] woorden = {  
        "Een", "twee", "Drie", "vier", "Vijf",  
        "zes", "Zeven", "acht", "Negen", "tien"  
    };  
  
    public static void main(String[] args) {  
        List<String> woordenlijst = new LinkedList<>(  
            Arrays.asList(woorden));  
  
        System.out.println(woordenlijst);  
  
        List<String> sortedlist = new LinkedList<>(woordenlijst);  
        Collections.sort(sortedlist);  
  
        System.out.println(sortedlist);  
  
    }  
}
```

Een, twee, Drie, vier, Vijf, zes,  
Zeven, acht, Negen, tien

Sortering via **compareTo**  
van klasse String

Drie, Een, Negen, Vijf, Zeven,  
acht, tien, twee, vier, zes





## List – sorteren (2)

```
public class SortDemo2 {  
    public static final String[] woorden = {  
        "Een", "twee", "Drie", "vier", "Vijf",  
        "zes", "Zeven", "acht", "Negen", "tien"  
    };  
  
    public static void main(String[] args) {  
        List<String> woordenlijst = new LinkedList<>(  
            Arrays.asList(woorden));  
  
        System.out.println(woordenlijst);  
  
        List<String> sortedlist = new LinkedList<>(woordenlijst);  
        Collections.sort(sortedlist, String.CASE_INSENSITIVE_ORDER);  
  
        System.out.println(sortedlist);  
    }  
}
```

Een, twee, Drie, vier, Vijf, zes,  
Zeven, acht, Negen, tien

Sortering via **Comparator**  
uit de klasse String

acht, Drie, Een, Negen, tien, twee,  
vier, Vijf, zes, Zeven



## List – sorteren (3)

---

```
public class MyStringComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return second.compareToIgnoreCase(first);  
    }  
}
```



Eigen **Comparator** klasse  
sorteert hoofdletterONgevoelig  
en achterstevoren



# List – sorteren (3)

```
public class SortDemo3 {  
    public static final String[] woorden = {  
        "Een", "twee", "Drie", "vier", "Vijf",  
        "zes", "Zeven", "acht", "Negen", "tien"  
    };  
  
    public static void main(String[] args) {  
        List<String> woordenlijst = new LinkedList<>(  
            Arrays.asList(woorden));  
  
        System.out.println(woordenlijst);  
  
        List<String> sortedlist = new LinkedList<>(woordenlijst);  
        Collections.sort(sortedlist, new MyStringComparator());  
  
        System.out.println(sortedlist);  
    }  
}
```

Een, twee, Drie, vier, Vijf, zes,  
Zeven, acht, Negen, tien

Sortering via eigen  
**Comparator** klasse

Zeven, zes, Vijf, vier, twee, tien,  
Negen, Een, Drie, acht



# Conclusie: List sorteren

---

- **Collections.sort**

Enkel mogelijk in combinatie met:

- **Comparable** interface

- natuurlijke ordening volgens **compareTo** methode

- **Comparator** interface

- afzonderlijke klasse, volledige controle via **compare** methode

# List – read only (1)

```
public class Verzameling {  
    private List<Integer> verzameling;  
  
    public Verzameling() {  
        verzameling = new LinkedList<>();  
    }  
  
    public List<Integer> getVerzameling() {  
        return verzameling;  
    }  
}
```

**Gevaarlijk!** de `List` met pointers wordt geretourneerd; alle objecten zijn dus rechtstreeks bereikbaar en kwetsbaar!!!

```
// Beter:  
public List<Integer> getVerzameling() {  
    return Collections.unmodifiableList(verzameling);  
}
```

**Veilig door proxy pattern:**  
de oorspronkelijke list wordt afgeschermd!

# List – read only (1)

---

```
public class Verzameling {  
    private List<Integer> verzameling;  
  
    public Verzameling() {  
        verzameling = new LinkedList<>();  
    }  
  
    public List<Integer> getVerzameling() {  
        return verzameling;  
    }  
}
```

**Veilig:**  
nieuwe **LinkedList**  
op basis van de  
oorspronkelijke

```
// Alternatief:  
public List<Integer> getVerzameling() {  
    return new LinkedList<>(verzameling);  
}
```

# Agenda

---



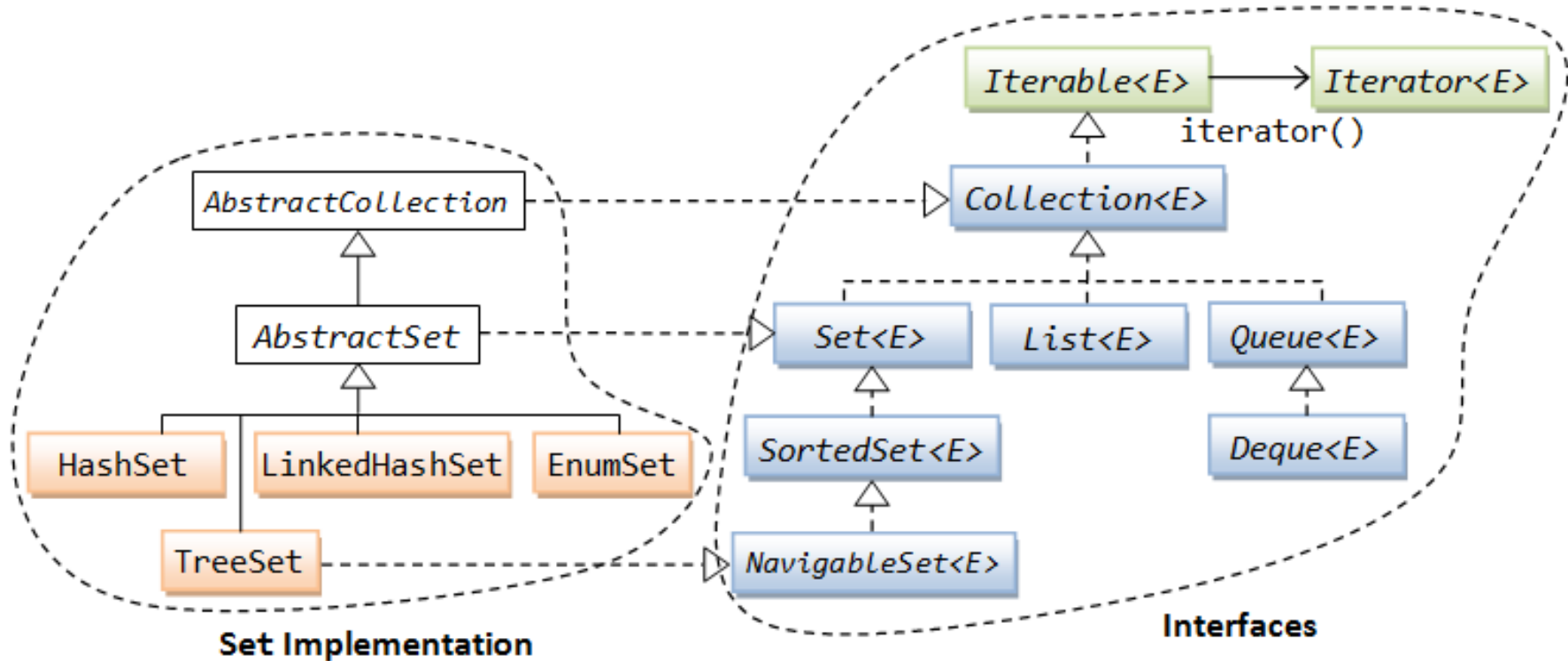
## 1. Herhaling

- Inleiding
- OO concepten
  - Inkapseling en overerving
  - Polymorfisme, overloading en overriding
  - Abstract en interface
  - final en static
  - Toegankelijkheid

## 2. Collections

- List
- Set
- Map

# Set: interfaces en implementaties





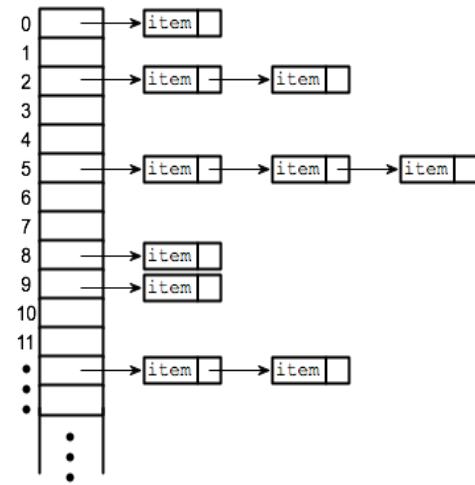
# De interface Set

```
public interface Set<E> extends Collection<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator<E> iterator();  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
  
    // Array Operations  
    Object[] toArray();  
}
```

**Merk op:**  
geen **get-** of **set-**  
**methoden** zoals in  
**List**

# De klasse HashSet

- Een **HashSet** is een ingekapselde hash table van objecten:
  - ongesorteerd (ook geen `Collections.sort`)
  - elk element is **uniek** via 2 methoden:
    - `int hashCode()`
    - `boolean equals(Object)`
- Geen methoden om te positioneren!



bij `List` hebben we dat wel :  
`get(int)`, `getFirst()`,  
`remove(int)` ...

# Hoe objecten UNIEK maken?

---

- 2 methoden zorgen daar SAMEN voor:
  - `public boolean equals (Object object) ;`  
→ bepaalt of **this** *gelijk is aan* een ander object van dezelfde klasse.
  - `public int hashCode () ;`  
→ maakt een *unieke sleutel* op basis van de attributen van **this**.
- Method overriding van **equals** en **hashCode** in de klasse **Object**
- Altijd BEIDE methoden uitwerken!



TIP: Gebruik <ALT>  
<INSERT> in IntelliJ

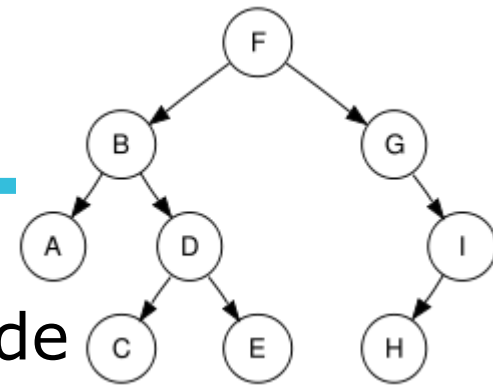
# De methode hashCode

---

- Voorbeelden van hoe de methode **hashCode** is uitgewerkt:
  - in **Object**: op basis van fysisch adres
  - in **String**: op basis van de characters
  - in **Integer**: de ingekapselde int-waarde
  - in **Double**: op basis van ingekapselde double
  - in **Character**: de Unicode-waarde

# De klasse TreeSet

---



- Een **TreeSet** is een binaire, gesorteerde boom

– Elk element = **uniek**

- `int hashCode()`
- `boolean equals(Object)`

– Elk element wordt **gesorteerd** toegevoegd:

- `Comparable` interface
- OF: `Comparator` gebruiken

# Voorbeeld HashSet

```
public class DemoSet1 {  
    public static void main(String[] args) {  
        Set<String> set = new HashSet<>();  
  
        set.add("een");  
        set.add("twee");  
        set.add("drie");  
        set.add("vier");  
        set.add(new String("twee")); // object reeds aanwezig!  
  
        for (String string : set) {  
            System.out.print(string + " ");  
        }  
    }  
}
```

twee drie vier een

Ongeordend!



# Voorbeeld LinkedHashSet

```
public class DemoSet2 {  
    public static void main(String[] args) {  
        Set<String> set = new LinkedHashSet<>();  
  
        set.add("een");  
        set.add("twee");  
        set.add("drie");  
        set.add("vier");  
        set.add(new String("twee")); // object reeds aanwezig!  
  
        for (String string : set) {  
            System.out.print(string + " ");  
        }  
    }  
}
```

een twee drie vier

**Geordend** in  
volgorde van  
toevoegen



# Voorbeeld TreeSet

```
public class DemoSet3 {  
    public static void main(String[] args) {  
        Set<String> set = new TreeSet<>();  
  
        set.add("een");  
        set.add("twee");  
        set.add("drie");  
        set.add("vier");  
        set.add(new String("twee")); // object reeds aanwezig!  
  
        for (String string : set) {  
            System.out.print(string + " ");  
        }  
    }  
}
```

drie een twee vier

**Gesorteerd** volgens  
compareTo van String





# Agenda

---



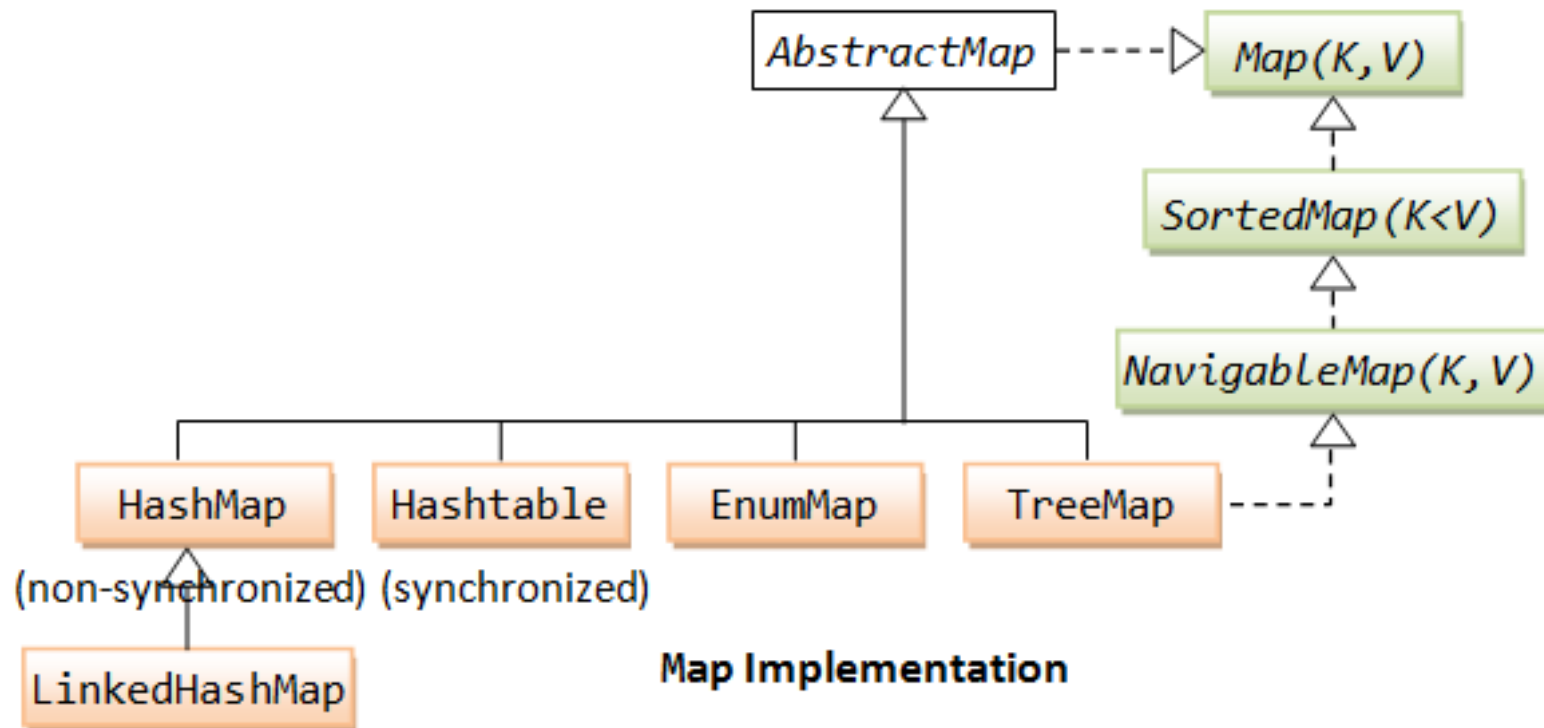
## 1. Herhaling

- Inleiding
- OO concepten
  - Inkapseling en overerving
  - Polymorfisme, overloading en overriding
  - Abstract en interface
  - final en static
  - Toegankelijkheid

## 2. Collections

- List
- Set
- Map

# Map: interfaces en implementaties



# De interface Map



```
public interface Map <K,V> {  
    // Basic Operations  
    V put(K key, V value);  
    V get(Object key);  
    V remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
  
    // Bulk Operations  
    void putAll(Map<? extends K, ? extends V> m);  
    void clear();  
  
    // Collection Views  
    Set<K> keySet();  
    Collection<V> values();  
    Set<Map.Entry<K, V>> entrySet();
```

Belangrijke methoden om **keys**, **values** of **entries** op te halen.  
Let op de **returnwaarde**: verklaring?

# De klasse HashMap

---

- De **HashMap** is een associatieve array van **Entry**-objecten
  - **key** = uniek door `hashCode()`
  - **value** ≠ uniek
- default capacity = 16
- default loadfactor = 0.75
  - if(`size > capacity * loadfactor`) → rehash
  - rehash betekent: verdubbel de capaciteit

# De klasse TreeMap

---

- De **TreeMap** is een binaire boom van **Entry**-objecten
  - **key** = uniek door hashCode()
  - **value** ≠ uniek
- **gesorteerd** opgebouwd volgens key:
  - ofwel: key-elementen zijn **Comparable** (hebben compareTo)
  - ofwel: aparte **Comparator**-klasse meegeven met constructor van TreeMap:

```
TreeMap<String, Klant> myMap = new TreeMap(new MyComparator());
```

# Opdrachten

---



- Groeiproject

- module 1 "Herhaling"



- Opdrachten op BB

- Interface "Deelbaar"
- Herhalingsopgave (Team)
- Optredens (List – Set – Map)

