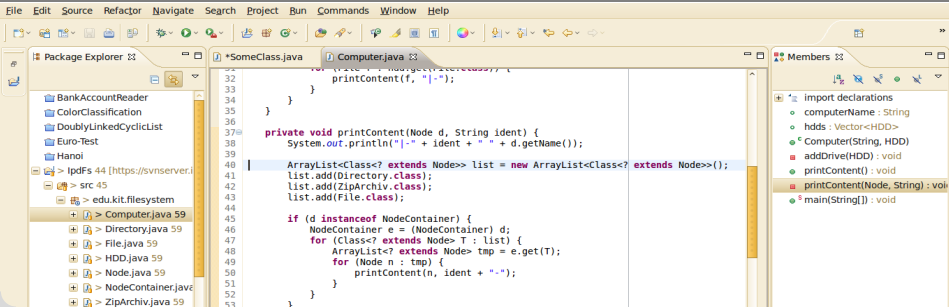


# Programmieren-Tutorium Nr. 10 bei Martin Thoma

Sortieren, equals(), hashCode(), abstrakte Klassen, finale Klassen

Martin Thoma | 12. Februar 2013

FAKULTÄT FÜR INFORMATIK



The screenshot shows an IDE with the following components:

- Package Explorer:** Lists packages like `edu.kit.filesystem` and files like `Computer.java`, `Directory.java`, `File.java`, `HDD.java`, `Node.java`, `NodeContainer.java`, and `ZipArchiv.java`.
- Editor:** Displays the code for `Computer.java`. The code includes a `printContent` method and a `main` method. The `printContent` method uses `ArrayList` and `Node` classes. The `main` method uses `NodeContainer` and `Node` classes.
- Members:** Shows the class members for `Computer`, including `computerName`, `hdds`, `addDrive`, `printContent`, and `main`.

```
32         printContent(f, "|-");
33     }
34 }
35
36
37 private void printContent(Node d, String ident) {
38     System.out.println(ident + " " + d.getName());
39
40     ArrayList<Class? extends Node> list = new ArrayList<Class? extends Node>();
41     list.add(Directory.class);
42     list.add(ZipArchiv.class);
43     list.add(File.class);
44
45     if (d instanceof NodeContainer) {
46         NodeContainer e = (NodeContainer) d;
47         for (Class? extends Node T : list) {
48             ArrayList? extends Node tmp = e.get(T);
49             for (Node n : tmp) {
50                 printContent(n, ident + "-");
51             }
52         }
53     }
```

- 1 Einleitung
- 2 equals()
- 3 Sortieren
- 4 hashCode()
- 5 Interface
- 6 abstract
- 7 final
- 8 Abspann

```
1 import java.util.Collections;
2 import java.util.LinkedList;
3 import java.util.List;
4
5 public class Main {
6     public static void main(String[] args) {
7         List<List<String>> myList = new LinkedList<List<String>>();
8         List<String> list1 = new LinkedList<String>();
9         myList.add(list1);
10
11         list1.add("I");
12         list1.add("think");
13         list1.add("therefore");
14         list1.add("I");
15         list1.add("am");
16
17         System.out.println(myList);
18         Collections.sort(myList);
19         System.out.println(myList);
20     }
21 }
```

- Gibt es einen Compiler-Fehler?
- Gibt es einen Laufzeit-Fehler?
- Gibt es eine Ausgabe? Welche?

```
7 List<List<String>> myList = new LinkedList<List<String>>();
8 List<String> list1 = new LinkedList<String>();
9 myList.add(list1);
10
11 list1.add("I");
12 list1.add("think");
13 list1.add("therefore");
14 list1.add("I");
15 list1.add("am");
16
17 System.out.println(myList);
18 Collections.sort(myList);
19 System.out.println(myList);
```

## Compiler-Fehler

Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
Bound mismatch: The generic method `sort(List<T>)` of type `Collections` is  
not applicable for the arguments ( `List<List<String>>` ).

The inferred type `List<String>` is not a valid substitute for the bounded  
parameter `<T extends Comparable<? super T>>`  
at Main.main(Main.java:18)

Does it make sense to implement `clone()`, `equals()` or `hashCode()` for an abstract class?

## Answer

I wouldn't implement `clone()`.

But it makes sense to implement `equals()`, `hashCode()`, and `toString()` to provide the default behavior for all subclasses. Children can choose to use it if they add no new class members or supplement as needed.

## Übungsleiter

generics werden wir für die Abschlusssaufgaben vermeiden.

# instanceof vs. getClass()

- instanceof akzeptiert auf Untertypen:

```
public class Main {  
    public static void main(String[] args) {  
        Apple myApple = new Apple();  
        if (myApple instanceof Fruit) {  
            System.out.println("It's true!");  
        }  
    }  
}
```

- getClass nicht:

```
if (this.getClass() != other.getClass())  
    return false;
```

⇒ Bei equals() eher getClass verwenden

- instanceof funktioniert auch mit null
- null.getClass() gibt NullPointerException nicht

⇒ zuerst auf null überprüfen

# instanceof vs. getClass()

- instanceof akzeptiert auf Untertypen:

```
public class Main {  
    public static void main(String[] args) {  
        Apple myApple = new Apple();  
        if (myApple instanceof Fruit) {  
            System.out.println("It's true!");  
        }  
    }  
}
```

- getClass nicht:

```
if (this.getClass() != other.getClass())  
    return false;
```

⇒ Bei equals() eher getClass verwenden

- instanceof funktioniert auch mit null
- null.getClass() gibt NullPointerException nicht

⇒ zuerst auf null überprüfen



# instanceof vs. getClass()

- instanceof akzeptiert auf Untertypen:

```
public class Main {  
    public static void main(String[] args) {  
        Apple myApple = new Apple();  
        if (myApple instanceof Fruit) {  
            System.out.println("It's true!");  
        }  
    }  
}
```

- getClass nicht:

```
if (this.getClass() != other.getClass())  
    return false;
```

⇒ Bei equals() eher getClass verwenden

- instanceof funktioniert auch mit null
- null.getClass() gibt NullPointerException nicht

⇒ zuerst auf null überprüfen

# instanceof vs. getClass()

- instanceof akzeptiert auf Untertypen:

```
public class Main {  
    public static void main(String[] args) {  
        Apple myApple = new Apple();  
        if (myApple instanceof Fruit) {  
            System.out.println("It's true!");  
        }  
    }  
}
```

- getClass nicht:

```
if (this.getClass() != other.getClass())  
    return false;
```

⇒ Bei equals() eher getClass verwenden

- instanceof funktioniert auch mit null

- null.getClass() gibt NullPointerException nicht

⇒ zuerst auf null überprüfen

# instanceof vs. getClass()

- instanceof akzeptiert auf Untertypen:

```
public class Main {  
    public static void main(String[] args) {  
        Apple myApple = new Apple();  
        if (myApple instanceof Fruit) {  
            System.out.println("It's true!");  
        }  
    }  
}
```

- getClass nicht:

```
if (this.getClass() != other.getClass())  
    return false;
```

⇒ Bei equals() eher getClass verwenden

- instanceof funktioniert auch mit null
- null.getClass() gibt NullPointerException nicht

⇒ zuerst auf null überprüfen

# instanceof vs. getClass()

- instanceof akzeptiert auf Untertypen:

```
public class Main {  
    public static void main(String[] args) {  
        Apple myApple = new Apple();  
        if (myApple instanceof Fruit) {  
            System.out.println("It's true!");  
        }  
    }  
}
```

- getClass nicht:

```
if (this.getClass() != other.getClass())  
    return false;
```

⇒ Bei equals() eher getClass verwenden

- instanceof funktioniert auch mit null
- null.getClass() gibt NullPointerException nicht

⇒ zuerst auf null überprüfen

# instanceof vs. getClass()

Aber ...

- Sehr viele ziehen instanceof in equals() der getClass() Variante vor
- Es gibt Argumente für beides
  - pro-instanceof: Debug-Klassen
  - pro-instanceof: Liskov substitution principle
  - pro-getClass(): Die Klassen stimmen wirklich überein
- Achtung: Andere Semantik!

# Was kann man sortieren?

- Zahlen
- Wörter
- Länder nach Anzahl der Einwohner
- Spielkarten
- ...

Totale Ordnungsrelation  $\preceq$  auf einer Menge  $C$ :

- Totalität:  $\forall x, y \in C : x \preceq y \vee y \preceq x$
- Antisymmetrie:  $\forall x, y \in C : x \preceq y \wedge y \preceq x \Rightarrow x = y$
- Transitivität:  $\forall x, y, z \in C : x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$

# Wo ist das nicht gegeben?

- Totalität:  $\forall x, y \in C : x \preceq y \vee y \preceq x$ ?



# Wo ist das nicht gegeben?

■ Totalität:  $\forall x, y \in C : x \preceq y \vee y \preceq x$ ?

⇒ Menge  $\mathbb{C}$ , Relation  $\leq$ :  $i$  und  $1$  stehen nicht in Relation!

⇒ Menge  $\mathcal{P}(\{1, 2, 3\})$ , Relation  $\subseteq$ :  $\{1\}$  und  $\{2\}$  stehen nicht in Relation!

# Wo ist das nicht gegeben?

- Totalität:  $\forall x, y \in C : x \preceq y \vee y \preceq x$ ?

⇒ Menge  $\mathbb{C}$ , Relation  $\leq$ :  $i$  und  $1$  stehen nicht in Relation!

⇒ Menge  $\mathcal{P}(\{1, 2, 3\})$ , Relation  $\subseteq$ :  $\{1\}$  und  $\{2\}$  stehen nicht in Relation!

- Antisymmetrie:  $\forall x, y \in C : x \preceq y \wedge y \preceq x \Rightarrow x = y$ ?

# Wo ist das nicht gegeben?

- Totalität:  $\forall x, y \in C : x \preceq y \vee y \preceq x$ ?

⇒ Menge  $\mathbb{C}$ , Relation  $\leq$ :  $i$  und  $1$  stehen nicht in Relation!

⇒ Menge  $\mathcal{P}(\{1, 2, 3\})$ , Relation  $\subseteq$ :  $\{1\}$  und  $\{2\}$  stehen nicht in Relation!

- Antisymmetrie:  $\forall x, y \in C : x \preceq y \wedge y \preceq x \Rightarrow x = y$ ?

⇒ Menge  $\mathbb{R}$ , Relation  $\preceq$ :  $x \preceq y \Leftrightarrow x, y \in \mathbb{R}$  (vgl. [SO](#))

# Wo ist das nicht gegeben?

- Totalität:  $\forall x, y \in C : x \preceq y \vee y \preceq x$ ?

⇒ Menge  $\mathbb{C}$ , Relation  $\leq$ :  $i$  und  $1$  stehen nicht in Relation!

⇒ Menge  $\mathcal{P}(\{1, 2, 3\})$ , Relation  $\subseteq$ :  $\{1\}$  und  $\{2\}$  stehen nicht in Relation!

- Antisymmetrie:  $\forall x, y \in C : x \preceq y \wedge y \preceq x \Rightarrow x = y$ ?

⇒ Menge  $\mathbb{R}$ , Relation  $\preceq$ :  $x \preceq y \Leftrightarrow x, y \in \mathbb{R}$  (vgl. [SO](#))

- Transitivität:  $\forall x, y, z \in C : x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$ ?

# Wo ist das nicht gegeben?

- Totalität:  $\forall x, y \in C : x \preceq y \vee y \preceq x$ ?

⇒ Menge  $\mathbb{C}$ , Relation  $\leq$ :  $i$  und  $1$  stehen nicht in Relation!

⇒ Menge  $\mathcal{P}(\{1, 2, 3\})$ , Relation  $\subseteq$ :  $\{1\}$  und  $\{2\}$  stehen nicht in Relation!

- Antisymmetrie:  $\forall x, y \in C : x \preceq y \wedge y \preceq x \Rightarrow x = y$ ?

⇒ Menge  $\mathbb{R}$ , Relation  $\preceq$ :  $x \preceq y \Leftrightarrow x, y \in \mathbb{R}$  (vgl. [SO](#))

- Transitivität:  $\forall x, y, z \in C : x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$ ?

⇒ ?

# Hilfe, ich komme mit Relationen nicht zurecht!

Don't Panic!

- Meist vergleicht man indirekt Zahlen
- Bei `double` und `float` den Epsilon-Vergleich machen!
- Sonst vergleicht man Strings
- `myString.compareTo(myOtherString)`
- Die JavaDoc von `compareTo(other)` sind weniger mathematisch formuliert

# Wie sortiert man?

## Vergleichsbasierte Sortialgorithmen:

- Selectionsort
- Bubblesort
- Quicksort
- ...

## Nicht vergleichsbasierte Algorithmen:

- Radixsort
- Countingsort

Implementierungen und Vergleiche dieser und weiterer Algorithmen sind [hier](#) zu finden.

## Info am Rande

`Collections.sort()` verwendet Mergesort-Variante (vermutlich Timsort)

# Sortieren in Java: Arrays

```
1 import java.util.Arrays;
2
3 public class Main {
4     public static void main(String[] args) {
5         String[] myStrings = new String[5];
6         myStrings[0] = "I";
7         myStrings[1] = "think";
8         myStrings[2] = "therefore";
9         myStrings[3] = "I";
10        myStrings[4] = "am";
11
12        System.out.println(Arrays.asList(myStrings));
13        Arrays.sort(myStrings);
14        System.out.println(Arrays.asList(myStrings));
15    }
16 }
```



# Sortieren in Java: Collections

```
1 import java.util.Collections;
2 import java.util.LinkedList;
3 import java.util.List;
4
5 public class Main {
6     public static void main(String[] args) {
7         List<String> myList = new LinkedList<String>();
8         myList.add("I");
9         myList.add("think");
10        myList.add("therefore");
11        myList.add("I");
12        myList.add("am");
13
14        System.out.println(myList);
15        Collections.sort(myList);
16        System.out.println(myList);
17    }
18 }
```

# Sortieren in Java: Comparator

```
1 import java.util.Comparator;
2
3 public class PopulationDensityComperator implements
4     Comparator<Country> {
5
6     @Override
7     public int compare(Country o1, Country o2) {
8         double o1Density = o1.population / o1.area;
9         double o2Density = o2.population / o2.area;
10
11         if (Math.abs(o1Density - o2Density) < 0.00001) {
12             return 0;
13         } else {
14             return o1Density - o2Density;
15         }
16     }
17
18 }
```

# Sortieren in Java: Comparator benutzen

```
1 import java.util.ArrayList;
2 import java.util.Collections;
3 import java.util.List;
4
5 public class Main {
6     public static void main(String[] args) {
7         List<Country> europe = new ArrayList<Country>();
8         europe.add(new Country(81903000,357121.41,"Germany"));
9         europe.add(new Country(64667000,668763, "France"));
10        europe.add(new Country( 4985900,385199, "Norway"));
11        europe.add(new Country( 9514406,450295, "Sweden"));
12        europe.add(new Country(47212990,504645, "Spain"));
13        europe.add(new Country( 8014000, 41285, "Switzerland"));
14        europe.add(new Country( 36371, 2.02, "Monaco"));
15        System.out.println(europe);
16        Collections.sort(europe);
17        System.out.println(europe);
18        Collections.sort(europe, new PopulationDensityComperator());
19        System.out.println(europe);
20    }
21 }
```

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange



## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie viele Situationen gibt es auf einem  $7 \times 6$ -Feld bei „4 Gewinn“?

- maximal:  $3^{7 \cdot 6} = 3^{42} = 109418989131512359209 \approx 109 \cdot 10^{18}$
  - minimal: schwer zu sagen
  - Idee: Brute-Force
    - Alle möglichen Spielentscheidungen durchgehen
    - Kommt man auf eine bereits bekannte Situation, ist es keine neue
    - Man muss also alte Situationen speichern (z.B. ein `char[42]` pro Spielsituation)
    - Man muss eine alte Situationen finden können
    - Vermutung: min. 20 000 000 Spielsituationen
- ⇒ lineare Suche nach bekannten Situationen dauert zu lange

## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
  - Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
  - Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
  - $h$  ist also nicht injektiv
  - Sobald der Array voll ist, können wir aufhören
  - Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$
- ⇒ wird „lineares Sondieren genannt“

## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
  - Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
  - Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
  - $h$  ist also nicht injektiv
  - Sobald der Array voll ist, können wir aufhören
  - Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$
- ⇒ wird „lineares Sondieren genannt“

## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
  - Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
  - Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
  - $h$  ist also nicht injektiv
  - Sobald der Array voll ist, können wir aufhören
  - Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$
- ⇒ wird „lineares Sondieren genannt“

## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
  - Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
  - Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
  - $h$  ist also nicht injektiv
  - Sobald der Array voll ist, können wir aufhören
  - Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$
- ⇒ wird „lineares Sondieren genannt“



## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
  - Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
  - Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
  - $h$  ist also nicht injektiv
  - Sobald der Array voll ist, können wir aufhören
  - Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$
- ⇒ wird „lineares Sondieren genannt“

## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
  - Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
  - Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
  - $h$  ist also nicht injektiv
  - Sobald der Array voll ist, können wir aufhören
  - Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$
- ⇒ wird „lineares Sondieren genannt“

## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
- Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
- Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
- $h$  ist also nicht injektiv
- Sobald der Array voll ist, können wir aufhören
- Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$

⇒ wird „lineares Sondieren genannt“

## Frage

Wie kann ich schnell eine Spielsituation speichern und wieder finden?

Antwort: Hash-Funktion mit linearer Sondierung!

- Ich will eine Funktion:  $h : \text{Spielsituationen} \rightarrow \text{Array-Index}$
  - Die Spiel-Situation kann ich als Zahl  $x$  auffassen mit  $0 \leq x \leq 110 \cdot 10^{18}$
  - Für den Array-Index  $i$  gilt:  $0 \leq i \leq 20\,000\,000$
  - $h$  ist also nicht injektiv
  - Sobald der Array voll ist, können wir aufhören
  - Falls  $h(x)$  ein Array-Index ist, der bereits belegt ist, aber der Array nicht voll ist, müssen wir die nächste freie Stelle suchen.  
Dazu gehen wir einfach auf  $(h(x) + 1) \% 20\,000\,000$
- ⇒ wird „lineares Sondieren genannt“

## Frage

Wie sieht eine gute hash-Funktion aus?

- Sie sollte surjektiv sein
- Sie sollte gleichmäßig auf die Bildmenge abbilden
- Vorschlag:

```
1 unsigned int getFirstIndex(char board[BOARD_WIDTH][BOARD_HEIGHT]) {  
2     unsigned int index = 0;  
3     for (int x=0; x<BOARD_WIDTH; x++) {  
4         for (int y=0; y<BOARD_HEIGHT; y++) {  
5             index += charToInt(board[x][y])*myPow(3, ((x*y*BOARD_WIDTH)%HASH_MODULO));  
6         }  
7     }  
8     index = index % MAXIMUM_SITUATIONS;  
9     return index;  
10 }
```

- Beispiel-Code ist auf [GitHub](#)

## Frage

Wie sieht eine gute hash-Funktion aus?

- Sie sollte surjektiv sein
- Sie sollte gleichmäßig auf die Bildmenge abbilden

### ■ Vorschlag:

```
1 unsigned int getFirstIndex(char board[BOARD_WIDTH][BOARD_HEIGHT]) {
2     unsigned int index = 0;
3     for (int x=0; x<BOARD_WIDTH; x++) {
4         for (int y=0; y<BOARD_HEIGHT; y++) {
5             index += charToInt(board[x][y])*myPow(3, ((x*y*BOARD_WIDTH)%HASH_MODULO));
6         }
7     }
8     index = index % MAXIMUM_SITUATIONS;
9     return index;
10 }
```

- Beispiel-Code ist auf [GitHub](#)

## Frage

Wie sieht eine gute hash-Funktion aus?

- Sie sollte surjektiv sein
- Sie sollte gleichmäßig auf die Bildmenge abbilden
- Vorschlag:

```
1 unsigned int getFirstIndex(char board[BOARD_WIDTH][BOARD_HEIGHT]) {  
2     unsigned int index = 0;  
3     for (int x=0; x<BOARD_WIDTH; x++) {  
4         for (int y=0; y<BOARD_HEIGHT; y++) {  
5             index += charToInt(board[x][y])*myPow(3, ((x*y*BOARD_WIDTH)%HASH_MODULO));  
6         }  
7     }  
8     index = index % MAXIMUM_SITUATIONS;  
9     return index;  
10 }
```

- Beispiel-Code ist auf [GitHub](#)

## Frage

Wie sieht eine gute hash-Funktion aus?

- Sie sollte surjektiv sein
- Sie sollte gleichmäßig auf die Bildmenge abbilden
- Vorschlag:

```
1 unsigned int getFirstIndex(char board[BOARD_WIDTH][BOARD_HEIGHT]) {
2     unsigned int index = 0;
3     for (int x=0; x<BOARD_WIDTH; x++) {
4         for (int y=0; y<BOARD_HEIGHT; y++) {
5             index += charToInt(board[x][y])*myPow(3, ((x*y*BOARD_WIDTH)%HASH_MODULO));
6         }
7     }
8     index = index % MAXIMUM_SITUATIONS;
9     return index;
10 }
```

- Beispiel-Code ist auf [GitHub](#)



## Frage

Was ist nun eine Hash-Funktion im Allgemeinen?

## Antwort

Eine Hash-Funktion  $h$  bildet von einem sehr großem Definitionsbereich auf einen deutlich kleineren Wertebereich ab.

- Meist ist der Wertebereich ein `int`, also  
 $[-2147483648, 2147483647] = [-2^{31}, 2^{31} - 1] \approx [-2 \cdot 10^9, 2 \cdot 10^9]$
- Der Definitionsbereich kann alles mögliche sein
- Normalerweise ist es nicht möglich, eine injektive Funktion zu finden

## Frage

Was ist nun eine Hash-Funktion im Allgemeinen?

## Antwort

Eine Hash-Funktion  $h$  bildet von einem sehr großem Definitionsbereich auf einen deutlich kleineren Wertebereich ab.

- Meist ist der Wertebereich ein `int`, also  
 $[-2147483648, 2147483647] = [-2^{31}, 2^{31} - 1] \approx [-2 \cdot 10^9, 2 \cdot 10^9]$
- Der Definitionsbereich kann alles mögliche sein
- Normalerweise ist es nicht möglich, eine injektive Funktion zu finden

## Frage

Was ist nun eine Hash-Funktion im Allgemeinen?

## Antwort

Eine Hash-Funktion  $h$  bildet von einem sehr großem Definitionsbereich auf einen deutlich kleineren Wertebereich ab.

- Meist ist der Wertebereich ein `int`, also  
 $[-2147483648, 2147483647] = [-2^{31}, 2^{31} - 1] \approx [-2 \cdot 10^9, 2 \cdot 10^9]$
- Der Definitionsbereich kann alles mögliche sein
- Normalerweise ist es nicht möglich, eine injektive Funktion zu finden

## Frage

Was ist nun eine Hash-Funktion im Allgemeinen?

## Antwort

Eine Hash-Funktion  $h$  bildet von einem sehr großem Definitionsbereich auf einen deutlich kleineren Wertebereich ab.

- Meist ist der Wertebereich ein `int`, also  
 $[-2147483648, 2147483647] = [-2^{31}, 2^{31} - 1] \approx [-2 \cdot 10^9, 2 \cdot 10^9]$
- Der Definitionsbereich kann alles mögliche sein
- Normalerweise ist es nicht möglich, eine injektive Funktion zu finden

## Signatur

```
public int hashCode()
```

## Bedingung 1

Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method **must consistently return the same integer**, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

source: [JavaDoc](#)

## Signatur

```
public int hashCode()
```

## Bedingung 1

Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method **must consistently return the same integer**, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

source: [JavaDoc](#)

## Bedingung 2

If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects **must** produce the same integer result.

source: [JavaDoc](#)

## Klarstellung 1

Es muss gelten:

$A.equals(B) \Rightarrow A.hashCode() == B.hashCode()$

Aber nicht:

$A.hashCode() == B.hashCode() \Rightarrow A.equals(B)$

Das ist meist auch nicht möglich. Beispiel:

Eine Klasse mit einem `double` als Rückgabewert.

## Bedingung 2

If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects **must** produce the same integer result.

source: [JavaDoc](#)

## Klarstellung 1

Es muss gelten:

$$A.equals(B) \Rightarrow A.hashCode() == B.hashCode()$$

Aber nicht:

$$A.hashCode() == B.hashCode() \Rightarrow A.equals(B)$$

Das ist meist auch nicht möglich. Beispiel:

Eine Klasse mit einem `double` als Rückgabewert.



## Klarstellung 2

It is **not required** that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

source: [JavaDoc](#)

# hashCode(): Quiz

Ist das eine korrekte Hash-Funktion?

```
1 public class Country {
2     int population;
3     double area;
4     String name;
5
6
7     @Override
8     public int hashCode() {
9         return 0;
10    }
11
12    @Override
13    public boolean equals(Object obj) {
14        if (this == obj)
15            return true;
16        if (obj == null)
17            return false;
18        if (getClass() != obj.getClass())
19            return false;
20        Country other = (Country) obj;
21        if (Double.doubleToLongBits(area) != Double
22            .doubleToLongBits(other.area))
23            return false;
24        if (name == null) {
25            if (other.name != null)
26                return false;
27        } else if (!name.equals(other.name))
28            return false;
29        if (population != other.population)
```

Einleitung

○○○○

equals()

○○

Sortieren

○○○○○○○○○

hashCode()

○○○○○○○●○○

Interface

○

abstract

○○○○

final

○○

Abspann

○○○

# hashCode(): Antwort

- Ja, ist nach [JavaDoc](#) eine korrekte Hash-Funktion
- Aber: Es ist die wohl schlechteste korrekte Hash-Funktion
- Sogar praktisch nutzlos
- NIEMALS so machen!

# hashCode(): Antwort

- Ja, ist nach [JavaDoc](#) eine korrekte Hash-Funktion
- Aber: Es ist die wohl schlechteste korrekte Hash-Funktion
- Sogar praktisch nutzlos
- NIEMALS so machen!

# hashCode(): Antwort

- Ja, ist nach [JavaDoc](#) eine korrekte Hash-Funktion
- Aber: Es ist die wohl schlechteste korrekte Hash-Funktion
- Sogar praktisch nutzlos
- NIEMALS so machen!

# hashCode(): Antwort

- Ja, ist nach [JavaDoc](#) eine korrekte Hash-Funktion
- Aber: Es ist die wohl schlechteste korrekte Hash-Funktion
- Sogar praktisch nutzlos
- NIEMALS so machen!

# hashCode(): Antwort

- Ja, ist nach [JavaDoc](#) eine korrekte Hash-Funktion
- Aber: Es ist die wohl schlechteste korrekte Hash-Funktion
- Sogar praktisch nutzlos
- **NIEMALS** so machen!

# hashCode(): Set

hashCode() und equals() sind für Set wichtig.



War Thema in Tutorium Nr. 8

## docs.oracle.com Tutorial

Eine abstrakte Klasse ist eine Klasse mit dem **abstract**-Schlüsselwort. Sie kann abstrakte Methoden beinhalten. Abstrakte Klassen können nicht instanziiert werden, aber man kann Unterklassen bilden.

### Abstrakte Klassen

- ... müssen keine abstrakten Methoden beinhalten

Quelle: [Defining an abstract class without any abstract methods](#)

- ... sollten eine abstrakte Methode beinhalten

Quelle: [Should an abstract class have at least one abstract method?](#)

- ... können Konstruktoren haben

Quelle: [Abstract class is using it's own abstract method?](#)

- ... können konkret impementierte Methoden haben

Quelle: [Can an abstract class have concrete\(non-abstract method\) methods?](#)

## docs.oracle.com Tutorial

Eine abstrakte Klasse ist eine Klasse mit dem **abstract**-Schlüsselwort. Sie kann abstrakte Methoden beinhalten. Abstrakte Klassen können nicht instanziiert werden, aber man kann Unterklassen bilden.

### Abstrakte Klassen

- ... müssen keine abstrakten Methoden beinhalten

Quelle: [Defining an abstract class without any abstract methods](#)

- ... sollten eine abstrakte Methode beinhalten

Quelle: [Should an abstract class have at least one abstract method?](#)

- ... können Konstruktoren haben

Quelle: [Abstract class is using it's own abstract method?](#)

- ... können konkret impementierte Methoden haben

Quelle: [Can an abstract class have concrete\(non-abstract method\) methods?](#)

## docs.oracle.com Tutorial

Eine abstrakte Klasse ist eine Klasse mit dem **abstract**-Schlüsselwort. Sie kann abstrakte Methoden beinhalten. Abstrakte Klassen können nicht instanziiert werden, aber man kann Unterklassen bilden.

### Abstrakte Klassen

- ... müssen keine abstrakten Methoden beinhalten

Quelle: [Defining an abstract class without any abstract methods](#)

- ... sollten eine abstrakte Methode beinhalten

Quelle: [Should an abstract class have at least one abstract method?](#)

- ... können Konstruktoren haben

Quelle: [Abstract class is using it's own abstract method?](#)

- ... können konkret impementierte Methoden haben

Quelle: [Can an abstract class have concrete\(non-abstract method\) methods?](#)

## docs.oracle.com Tutorial

Eine abstrakte Klasse ist eine Klasse mit dem **abstract**-Schlüsselwort. Sie kann abstrakte Methoden beinhalten. Abstrakte Klassen können nicht instanziiert werden, aber man kann Unterklassen bilden.

### Abstrakte Klassen

- ... müssen keine abstrakten Methoden beinhalten

Quelle: [Defining an abstract class without any abstract methods](#)

- ... sollten eine abstrakte Methode beinhalten

Quelle: [Should an abstract class have at least one abstract method?](#)

- ... können Konstruktoren haben

Quelle: [Abstract class is using it's own abstract method?](#)

- ... können konkret impementierte Methoden haben

Quelle: [Can an abstract class have concrete\(non-abstract method\) methods?](#)

## What are practical examples of abstract classes in java?

- [StackOverflow](#)
- FileParser, CameraFileParser

Abstrakte Klassen können ...

- Attribute haben, die nicht `static` und `final` sind
- Implementierte Methoden haben

Wenn nutze ich Interfaces?

Wenn ich nur abstrakte Methoden habe

- docs.oracle.com Tutorial: [Abstract Methods and Classes](#)
- codestyle.org: [Java abstract classes FAQ](#)
- openbook.galileodesign.de: [Abstrakte Klassen und abstrakte Methoden](#)



- Finale Klassen können keine Unterklassen haben
- Beispiel: `String`, `StringBuffer`, `StringBuilder`, `Math`:

```
1 public final class Math {  
2  
3     /**  
4      * Don't let anyone instantiate this class.  
5      */  
6     private Math() {}
```

- docs.oracle.com Tutorial: [Writing Final Classes and Methods](#)
- openbook.galileodesign.de: [Finale Klassen](#)

- Anmeldebeginn: 28.1.
- Anmeldeschluss / Abmeldeschluss: 28.2.
- Ausgabetermin für Teil 1: 28.1.
- Ausgabetermin für Teil 2: 11.2.
- Abgabefrist für Teil 1: 11.3.
- Abgabefrist für Teil 2: 25.3.

- 3. 14.01.2013
- 2. 21.01.2013
- 1. 28.01.2013: Abschlussprüfunsvorbereitung
  - 28.01.2013: Ausgabetermin für Teil 1
- 0. 04.02.2013: Abschlussprüfunsvorbereitung
  - 10.02.2013: Ende der Vorlesungszeit des WS 2012/2013 ([Quelle](#))

