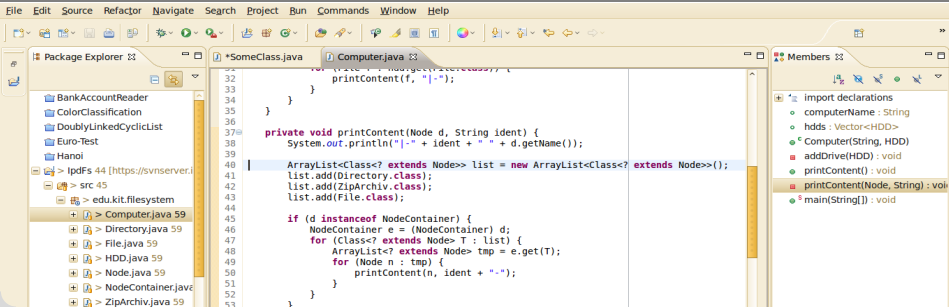


Programmieren-Tutorium Nr. 10 bei Martin Thoma

Wildcards, equals(), Exceptions

Martin Thoma | 3. Januar 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 that recursively prints the contents of a `Node` and its children. It uses wildcards (`Class extends Node`) and the `ArrayList` class.
- Members:** Shows the class members for `Computer`, including `import declarations`, `computerName : String`, `hdds : Vector<HDD>`, `Computer(String, HDD)`, `addDrive(HDD) : void`, `printContent() : void`, `printContent(Node, String) : void`, and `main(String[]) : void`.

```
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 Generics

3 equals

4 Exceptions

5 Praxis

6 Abspann

```
1 import java.util.LinkedList;
2 import java.util.List;
3
4 public class Main {
5     public static void main(String[] args) {
6         List<Fruit> myFruits = new LinkedList<Fruit>();
7         List<Apple> myApples = new LinkedList<Apple>();
8
9         myFruits.add(new Fruit());
10        myFruits.add(new Apple());
11
12        myApples.add(new Apple());
13
14        System.out.println(myFruits.getClass());
15        System.out.println(myFruits.getClass());
16        myFruits = myApples;
17    }
18 }
```

```
_____ Fruit.java _____
public class Fruit { }
```

```
_____ Apple.java _____
public class Apple extends Fruit { }
```

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

Compiler-Fehler

Type mismatch: cannot convert from List<Apple> to List<Fruit>

- Ohne Zeile 16 gibt es folgende Ausgabe:

```
class java.util.LinkedList
```

```
class java.util.LinkedList
```

- Sowohl `myFruits = myApples;` als auch `myApples = myFruits;` geben einen Compiler-Fehler

Compiler-Fehler

Type mismatch: cannot convert from List<Apple> to List<Fruit>

- Ohne Zeile 16 gibt es folgende Ausgabe:

```
class java.util.LinkedList
```

```
class java.util.LinkedList
```

- Sowohl `myFruits = myApples;` als auch `myApples = myFruits;` geben einen Compiler-Fehler

```
LinkedList<Apple> apples = new LinkedList<Apple>();  
LinkedList<Fruit> fruits = apples;  
fruits.add(new Banana());
```

```
// Safe at compile time, but it's a Banana!  
Apple apple = apples.getFirst();
```

Quiz: Lösung #1

```
1 import java.util.LinkedList;
2
3 public class Main {
4     public static void main(String[] args) {
5         LinkedList<? super Apple> apples = new LinkedList<Fruit>()
6         apples.add(new Apple());
7
8         // I can't get apples out
9         for (Object o : apples) {
10             Apple a = (Apple) o;
11             System.out.println(a);
12         }
13     }
14 }
```

Quiz: Lösung #2

```
1 import java.util.LinkedList;
2
3 public class Main {
4     public static void main(String[] args) {
5         LinkedList<? extends Fruit> apples = new LinkedList<Apple>
6
7         // I can't get apples in
8         // this gives an error
9         apples.add(new Apple());
10        // In fact, you can only add null!
11    }
12 }
```



```

1 public static class Cage.java
2 public static class Cage<T extends Animal> {
3     private Set<T> pen = new HashSet<T>();
4
5     public void add(T animal) {
6         pen.add(animal);
7     }
8
9     /* It's OK to put subclasses into a cage of
10      super class
11      */
12     public void transferTo(Cage<? super T> cage) {
13         cage.pen.addAll(this.pen);
14     }
15
16     public void showAnimals() {
17         System.out.println(pen);
18     }
19 }

```

```

1 public static class Animal {
2     public String toString() {
3         return getClass().getSimpleName();
4     }
5 }
6 public static class Rat extends Animal {}
7 public static class Lion extends Animal {}
8 public static class Cage<T extends Animal> {
9     /* above */
10 }
11
12 public static void main(String[] args) {
13     Cage<Animal> animals = new Cage<Animal>();
14     Cage<Lion> lions = new Cage<Lion>();
15
16     // OK to put a Rat into a Cage<Animal>
17     animals.add(new Rat());
18
19     lions.add(new Lion());
20
21     // invoke the super generic method
22     lions.transferTo(animals);
23     animals.showAnimals();
24 }

```

Source: [StackOverflow](#)

- Das `?` in `List<?> myList` wird Wildcard genannt
- `?` steht immer nur in der Deklaration, nie in der Initialisierung

⇒ `?` nur links vom `=`

- `List<?> myList` ist eine „unbounded Wildcard“

- `List<?> myList = new LinkedList<Fruit>();`
`myList.add(null); // ok`
`myList.add(new Fruit()); // Compiler error`

- `?` ein bestimmter, aber nicht angegebener Parameter

- ⇒ kann zur Compile-Zeit nicht überprüft werden

- ⇒ Liste darf nicht modifiziert werden

- `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
- `?` steht immer nur in der Deklaration, nie in der Initialisierung

⇒ `?` nur links vom `=`

- `List<?> myList` ist eine „unbounded Wildcard“

- `List<?> myList = new LinkedList<Fruit>();`
`myList.add(null); // ok`
`myList.add(new Fruit()); // Compiler error`

- `?` ein bestimmter, aber nicht angegebener Parameter

- ⇒ kann zur Compile-Zeit nicht überprüft werden

- ⇒ Liste darf nicht modifiziert werden

- `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
 - `?` steht immer nur in der Deklaration, nie in der Initialisierung
- ⇒ `?` nur links vom `=`
- `List<?> myList` ist eine „unbounded Wildcard“
 - `List<?> myList = new LinkedList<Fruit>();`
`myList.add(null); // ok`
`myList.add(new Fruit()); // Compiler error`
 - `?` ein bestimmter, aber nicht angegebener Parameter
 - ⇒ kann zur Compile-Zeit nicht überprüft werden
 - ⇒ Liste darf nicht modifiziert werden
 - `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
 - `?` steht immer nur in der Deklaration, nie in der Initialisierung
- ⇒ `?` nur links vom `=`
- `List<?> myList` ist eine „unbounded Wildcard“
 - ```
List<?> myList = new LinkedList<Fruit>();
myList.add(null); // ok
myList.add(new Fruit()); // Compiler error
```
    - `?` ein bestimmter, aber nicht angegebener Parameter
      - ⇒ kann zur Compile-Zeit nicht überprüft werden
      - ⇒ Liste darf nicht modifiziert werden
  - `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
  - `?` steht immer nur in der Deklaration, nie in der Initialisierung
- ⇒ `?` nur links vom `=`
- `List<?> myList` ist eine „unbounded Wildcard“
    - ```
List<?> myList = new LinkedList<Fruit>();  
myList.add(null); // ok  
myList.add(new Fruit()); // Compiler error
```
 - `?` ein bestimmter, aber nicht angegebener Parameter
 - ⇒ kann zur Compile-Zeit nicht überprüft werden
 - ⇒ Liste darf nicht modifiziert werden
 - `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
 - `?` steht immer nur in der Deklaration, nie in der Initialisierung
- ⇒ `?` nur links vom `=`
- `List<?> myList` ist eine „unbounded Wildcard“
 - ```
List<?> myList = new LinkedList<Fruit>();
myList.add(null); // ok
myList.add(new Fruit()); // Compiler error
```
    - `?` ein bestimmter, aber nicht angegebener Parameter
      - ⇒ kann zur Compile-Zeit nicht überprüft werden
      - ⇒ Liste darf nicht modifiziert werden
  - `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
  - `?` steht immer nur in der Deklaration, nie in der Initialisierung
- ⇒ `?` nur links vom `=`
- `List<?> myList` ist eine „unbounded Wildcard“
    - ```
List<?> myList = new LinkedList<Fruit>();  
myList.add(null); // ok  
myList.add(new Fruit()); // Compiler error
```
 - `?` ein bestimmter, aber nicht angegebener Parameter
- ⇒ kann zur Compile-Zeit nicht überprüft werden
- ⇒ Liste darf nicht modifiziert werden
- `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
 - `?` steht immer nur in der Deklaration, nie in der Initialisierung
- ⇒ `?` nur links vom `=`
- `List<?> myList` ist eine „unbounded Wildcard“
 - ```
List<?> myList = new LinkedList<Fruit>();
myList.add(null); // ok
myList.add(new Fruit()); // Compiler error
```
    - `?` ein bestimmter, aber nicht angegebener Parameter
      - ⇒ kann zur Compile-Zeit nicht überprüft werden
      - ⇒ Liste darf nicht modifiziert werden
  - `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- Das `?` in `List<?> myList` wird Wildcard genannt
  - `?` steht immer nur in der Deklaration, nie in der Initialisierung
- ⇒ `?` nur links vom `=`
- `List<?> myList` ist eine „unbounded Wildcard“
    - ```
List<?> myList = new LinkedList<Fruit>();  
myList.add(null); // ok  
myList.add(new Fruit()); // Compiler error
```
 - `?` ein bestimmter, aber nicht angegebener Parameter
 - ⇒ kann zur Compile-Zeit nicht überprüft werden
 - ⇒ Liste darf nicht modifiziert werden
 - `List<? extends Fruit> myList` und `List<? super Fruit> myList` sind „bounded Wildcards“

- `List<? extends Fruit> myList` kann als Elemente `Fruit` und `Apple` haben, nicht jedoch `Object`
- Hinweis: „extends“ ist hier nicht exakt das gleiche wie bei der Vererbung. Es kann entweder wirklich „extends“ oder „implements“ bedeuten
- Sowohl in `List<Fruit>` als auch in `List<? extends Fruit>` können `Fruit` und `Apple` beinhalten

- `List<? extends Fruit> myList` kann als Elemente `Fruit` und `Apple` haben, nicht jedoch `Object`
- Hinweis: „extends“ ist hier nicht exakt das gleiche wie bei der Vererbung. Es kann entweder wirklich „extends“ oder „implements“ bedeuten
- Sowohl in `List<Fruit>` als auch in `List<? extends Fruit>` können `Fruit` und `Apple` beinhalten

- `List<? extends Fruit> myList` kann als Elemente `Fruit` und `Apple` haben, nicht jedoch `Object`
- Hinweis: „extends“ ist hier nicht exakt das gleiche wie bei der Vererbung. Es kann entweder wirklich „extends“ oder „implements“ bedeuten
- Sowohl in `List<Fruit>` als auch in `List<? extends Fruit>` können `Fruit` und `Apple` beinhalten

- `List<? super Fruit> myList` kann als Elemente `Fruit` und `Object` haben, nicht jedoch

- [JavaDoc Tutorial - Wildcards](#)
- [JavaDoc Tutorial - Wildcards \(extra\)](#)
- [What does the question mark in Java generics' type parameter mean?](#)
- [What's the difference between List<Object> and List<?>](#)
- [Java: Wildcards again](#)
- [Incompatible type with Arrays.asList\(\)](#)
- [Java Generics \(Wildcards\)](#)

- Wildcards sind schwer
- Wildcards werdet ihr vermutlich bei den Abschlussaufgaben nicht benötigen

- Man will ein beliebiges Objekt mit dem momentanen Objekt auf Gleichheit vergleichen
- Dazu nutzt man `myObject.equals(otherObject);`
- `myObject` muss dann die `equals(Object obj)` implementieren

Die Implementierung läuft fast immer gleich ab:

- ist `obj == null` → `return false;`
- ist `!(obj instanceof MyClass)` → `return false;`
- `other = (MyClass) obj;`
- vergleich der (relevanten) Attribute

- Man will ein beliebiges Objekt mit dem momentanen Objekt auf Gleichheit vergleichen
- Dazu nutzt man `myObject.equals(otherObject);`
- `myObject` muss dann die `equals(Object obj)` implementieren

Die Implementierung läuft fast immer gleich ab:

- ist `obj == null` → `return false;`
- ist `!(obj instanceof MyClass)` → `return false;`
- `other = (MyClass) obj;`
- vergleich der (relevanten) Attribute

- Man will ein beliebiges Objekt mit dem momentanen Objekt auf Gleichheit vergleichen
- Dazu nutzt man `myObject.equals(otherObject);`
- `myObject` muss dann die `equals(Object obj)` implementieren

Die Implementierung läuft fast immer gleich ab:

- ist `obj == null` → `return false;`
- ist `!(obj instanceof MyClass)` → `return false;`
- `other = (MyClass) obj;`
- vergleich der (relevanten) Attribute

- Man will ein beliebiges Objekt mit dem momentanen Objekt auf Gleichheit vergleichen
- Dazu nutzt man `myObject.equals(otherObject);`
- `myObject` muss dann die `equals(Object obj)` implementieren

Die Implementierung läuft fast immer gleich ab:

- ist `obj == null` → `return false;`
- ist `!(obj instanceof MyClass)` → `return false;`
- `other = (MyClass) obj;`
- vergleich der (relevanten) Attribute

- Man will ein beliebiges Objekt mit dem momentanen Objekt auf Gleichheit vergleichen
- Dazu nutzt man `myObject.equals(otherObject);`
- `myObject` muss dann die `equals(Object obj)` implementieren

Die Implementierung läuft fast immer gleich ab:

- ist `obj == null` → `return false;`
- ist `!(obj instanceof MyClass)` → `return false;`
- `other = (MyClass) obj;`
- `vergleich der (relevanten) Attribute`

- Man will ein beliebiges Objekt mit dem momentanen Objekt auf Gleichheit vergleichen
- Dazu nutzt man `myObject.equals(otherObject);`
- `myObject` muss dann die `equals(Object obj)` implementieren

Die Implementierung läuft fast immer gleich ab:

- ist `obj == null` → `return false;`
- ist `!(obj instanceof MyClass)` → `return false;`
- `other = (MyClass) obj;`
- vergleich der (relevanten) Attribute

- Man will ein beliebiges Objekt mit dem momentanen Objekt auf Gleichheit vergleichen
- Dazu nutzt man `myObject.equals(otherObject);`
- `myObject` muss dann die `equals(Object obj)` implementieren

Die Implementierung läuft fast immer gleich ab:

- ist `obj == null` → `return false;`
- ist `!(obj instanceof MyClass)` → `return false;`
- `other = (MyClass) obj;`
- vergleich der (relevanten) Attribute

- Eclipse kann die equals()-Methode generieren
- Source Generate hashCode() and equals()...
- Felder auswählen, die für den Vergleich wichtig sind
- nochmals drüber schauen

Exceptions ...

- ...sind Objekte vom Typ Throwable
- ...unterbrechen den normalen Ablauf eines Programms
- Mit dem Schlüsselwort `throw` werden Exceptions geworfen und mit `catch` kann man sie abfangen.

Beispiele für Exceptions

- NullPointerException
- ArrayIndexOutOfBoundsException
- IllegalArgumentException
- IllegalStateException
- IOException
- ...

Exceptions ...

- ...sind Objekte vom Typ Throwable
- ...unterbrechen den normalen Ablauf eines Programms
- Mit dem Schlüsselwort `throw` werden Exceptions geworfen und mit `catch` kann man sie abfangen.

Beispiele für Exceptions

- NullPointerException
- ArrayIndexOutOfBoundsException
- IllegalArgumentException
- IllegalStateException
- IOException
- ...

Exceptions ...

- ...sind Objekte vom Typ Throwable
- ...unterbrechen den normalen Ablauf eines Programms
- Mit dem Schlüsselwort `throw` werden Exceptions geworfen und mit `catch` kann man sie abfangen.

Beispiele für Exceptions

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `IllegalArgumentException`
- `IllegalStateException`
- `IOException`
- ...

Exceptions ...

- ... sind Objekte vom Typ Throwable
- ... unterbrechen den normalen Ablauf eines Programms
- Mit dem Schlüsselwort `throw` werden Exceptions geworfen und mit `catch` kann man sie abfangen.

Beispiele für Exceptions

- NullPointerException
- ArrayIndexOutOfBoundsException
- IllegalArgumentException
- IllegalStateException
- IOException
- ...

Beispiel: Fibonacci.java

```
1 import java.util.HashMap;
2 import java.util.Map;
3
4 public class Fibonacci {
5     private final Map<Integer, Integer> functionValues;
6
7     public Fibonacci() {
8         functionValues = new HashMap<Integer, Integer>();
9         functionValues.put(0, 0);
10        functionValues.put(1, 1);
11    }
12
13    private int calculate(int x) {
14        return getFunctionValue(x - 1) + getFunctionValue(x - 2);
15    }
16
17    public int getFunctionValue(int x) {
18        if (x < 0) {
19            /* Exception werfen */
20            throw new IllegalArgumentException(
21                "Fibonacci is not defined for negative values");
22        }
23
24        if (functionValues.containsKey(x)) {
25            return functionValues.get(x);
26        } else {
27            int functionValue = calculate(x);
28            functionValues.put(x, functionValue);
29            return functionValue;
30        }
31    }
32 }
```

Beispiel: Main.java

```
1 public class Main {
2
3     /**
4      * @param args
5      */
6     public static void main(String[] args) {
7         Fibonacci f = new Fibonacci();
8         for (int i = 0; i < 10; i++) {
9             System.out.println(f.getFunctionValue(i));
10        }
11
12        /* Fehlerbehandlung */
13        try {
14            f.getFunctionValue(-2);
15        } catch (IllegalArgumentException e) {
16            System.out.println("Your Error: ");
17            System.out.println(e);
18        }
19    }
20
21 }
```

Anti-Pattern: Pokémon Exception Handling



For when you just Gotta Catch 'Em All.

```
try {  
    // your code  
} catch (Exception ex) {  
    // Gotcha!  
}
```

Anti-Pattern: Pokémon Exception Handling

Niemals Pokémon Exception Handling anwenden!

- Die Fehlerbehandlung mit `catch` wird verwendet, um den Programmablauf nach einem Fehler zu definieren
- Bei unterschiedlichen Fehlern will man meist unterschiedlich weiter machen, z.B.
 - `IOException`: nochmals versuchen
 - `NullPointerException`: Fehlerbericht an den Entwickler schicken
 - `IllegalArgumentException`: Fehlerausgabe an den Nutzer
- Durch die verschiedenen `catch`-Blöcke zeigst du, dass du an die verschiedenen Fehlerarten gedacht hast

Anti-Pattern: Pokémon Exception Handling

Niemals Pokémon Exception Handling anwenden!

- Die Fehlerbehandlung mit `catch` wird verwendet, um den Programmablauf nach einem Fehler zu definieren
- Bei unterschiedlichen Fehlern will man meist unterschiedlich weiter machen, z.B.
 - `IOException`: nochmals versuchen
 - `NullPointerException`: Fehlerbericht an den Entwickler schicken
 - `IllegalArgumentException`: Fehlerausgabe an den Nutzer
- Durch die verschiedenen `catch`-Blöcke zeigst du, dass du an die verschiedenen Fehlerarten gedacht hast

Anti-Pattern: Pokémon Exception Handling

Niemals Pokémon Exception Handling anwenden!

- Die Fehlerbehandlung mit `catch` wird verwendet, um den Programmablauf nach einem Fehler zu definieren
- Bei unterschiedlichen Fehlern will man meist unterschiedlich weiter machen, z.B.
 - `IOException`: nochmals versuchen
 - `NullPointerException`: Fehlerbericht an den Entwickler schicken
 - `IllegalArgumentException`: Fehlerausgabe an den Nutzer
- Durch die verschiedenen `catch`-Blöcke zeigst du, dass du an die verschiedenen Fehlerarten gedacht hast

Anti-Pattern: Pokémon Exception Handling

Niemals Pokémon Exception Handling anwenden!

- Die Fehlerbehandlung mit `catch` wird verwendet, um den Programmablauf nach einem Fehler zu definieren
- Bei unterschiedlichen Fehlern will man meist unterschiedlich weiter machen, z.B.
 - `IOException`: nochmals versuchen
 - `NullPointerException`: Fehlerbericht an den Entwickler schicken
 - `IllegalArgumentException`: Fehlerausgabe an den Nutzer
- Durch die verschiedenen `catch`-Blöcke zeigst du, dass du an die verschiedenen Fehlerarten gedacht hast

Anti-Pattern: Pokémon Exception Handling

Niemals Pokémon Exception Handling anwenden!

- Die Fehlerbehandlung mit `catch` wird verwendet, um den Programmablauf nach einem Fehler zu definieren
- Bei unterschiedlichen Fehlern will man meist unterschiedlich weiter machen, z.B.
 - `IOException`: nochmals versuchen
 - `NullPointerException`: Fehlerbericht an den Entwickler schicken
 - `IllegalArgumentException`: Fehlerausgabe an den Nutzer
- Durch die verschiedenen `catch`-Blöcke zeigst du, dass du an die verschiedenen Fehlerarten gedacht hast

Anti-Pattern: Pokémon Exception Handling

Niemals Pokémon Exception Handling anwenden!

- Die Fehlerbehandlung mit `catch` wird verwendet, um den Programmablauf nach einem Fehler zu definieren
- Bei unterschiedlichen Fehlern will man meist unterschiedlich weiter machen, z.B.
 - `IOException`: nochmals versuchen
 - `NullPointerException`: Fehlerbericht an den Entwickler schicken
 - `IllegalArgumentException`: Fehlerausgabe an den Nutzer
- Durch die verschiedenen `catch`-Blöcke zeigst du, dass du an die verschiedenen Fehlerarten gedacht hast

Wichtig

Der try-Block sollte so klein wie möglich sein.

Gründe:

- Beim lesen eures Codes wird klarer, wo das Problem auftreten kann
- Effizienz

```
_____ UniverseExplodeException.java _____  
1 public class UniverseExplodeException extends RuntimeException {  
2     public UniverseExplodeException() {  
3         super("The universe will explode!");  
4     }  
5 }
```

- Exceptions, die nicht von `RuntimeException` erben, müssen angekündigt werden
- Ankündigen funktioniert über JavaDoc-Annotation `@throws` und Methodensignatur mit `throws`

```
/* *  
 * The foo method.  
 *  
 * @throws UniverseExplodeException when the universe  
 *       is going to explode  
 */  
public void foo() throws UniverseExplodeException {  
    if (true) {  
        throw new UniverseExplodeException();  
    }  
}
```


- Catching and Handling Exceptions
- JLS 7, Kapitel 14.20

Das Spiel besteht aus drei Stäben A, B und C, auf die mehrere gelochte Scheiben gelegt werden, alle verschieden groß.

Zu Beginn liegen alle Scheiben auf Stab A, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben.

Ziel des Spiels ist es, den kompletten Scheiben-Stapel von A nach C zu versetzen.

Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der beiden anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Feld der Größe nach geordnet.

Klasse „Disc“

Schreiben Sie zunächst eine Klasse Disc, die eine gelochte Scheibe repräsentiert und als Attribut einen Durchmesser hat.

Klasse „Pole“

Schreiben Sie außerdem eine Klasse Pole, die einen Stab repräsentiert. Ein solcher Stab verwaltet eine Menge von Discs (in einem fest dimensionierten Array) und hat als Attribut einen Namen. Die Klasse Pole stellt dabei sicher, dass die Scheiben immer in geordneter Reihenfolge (wie oben beschrieben) auf dem Stab liegen. Hierfür stellt die Klasse Pole die Methoden `public boolean push(Disc d)` und `public Disc pop()` zur Verfügung.

Methode push

Die Methode `push(Disc d)` legt die Scheibe `d` auf den Stab, falls dieser noch nicht voll ist und der Durchmesser der Scheibe `d` kleiner ist als der Durchmesser der obersten Scheibe des Stabes. Wird die Scheibe erfolgreich auf den Stab gelegt, so ist der Rückgabewert der Methode `true`, andernfalls `false`.

Methode pop

Die Methode `pop()` entfernt die oberste Scheibe des Stabes und liefert diese als Rückgabewert. Falls der Stab leer ist, soll der Rückgabewert `null` sein.










Schreiben Sie, falls nötig, weitere Schnittstellen (z.B. eine Methode `size()`) und `toString()`-Methoden.

Eine weitere Klasse Hanoi soll die main-Methode und eine Methode mit der Signatur `public static void move(Pole from, Pole help, Pole to)` erhalten. Die Methode `move(Pole from, Pole help, Pole to)` legt dabei alle Scheiben des Stabes `from` mit Hilfe des Stabes `help` auf den Stab `to`. Implementieren Sie diese Methode rekursiv. Erzeugen Sie dann in der main-Methode einen Stab A mit mehreren Scheiben und zusätzlich zwei leere Stäbe B und C. Verwenden Sie dann die Methode `move()`, um die Scheiben von Stab A mit Hilfe des Stabes B auf Stab C zu legen.

Ist die Klausuranmeldung schon möglich? Bitte anmelden!

- 4. 07.01.2013
- 3. 14.01.2013
- 2. 21.01.2013
- 1. 28.01.2013: Abschlussprüfunsvorbereitung
- 0. 04.02.2013: Abschlussprüfunsvorbereitung
 - 10.02.2013: Ende der Vorlesungszeit des WS 2012/2013 ([Quelle](#))

Vielen Dank für eure Aufmerksamkeit!

<p>Days 1 - 10 Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...</p> 	<p>Days 11 - 21 Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism,</p> 	<p>Days 22 - 697 Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.</p> 
<p>Days 698 - 3648 Interact with other programmers. Work on programming projects together. Learn from them.</p> 	<p>Days 3649 - 7781 Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.</p> 	<p>Days 7782 - 14611 Teach yourself biochemistry, molecular biology, genetics,...</p> 
<p>Day 14611 Use knowledge of biology to make an age-reversing potion.</p> 	<p>Day 14611 Use knowledge of physics to build flux capacitor and go back in time to day 21.</p> 	<p>Day 21 Replace younger self.</p> 

As far as I know, this is the easiest way to
"Teach Yourself C++ in 21 Days".