Java

Recap 2 - Electric Boogaloo

Marcus Köhler

18. Januar 2019

Java-Kurs

Übersicht

1. Abstrakte Klassen
2. Exceptions
3. Generics
4. Collections
5. Andere Programmierkonzepte
   - Casting
   - Rekursion
6. JavaDoc

Abstrakte Klassen

Abstrakte Klassen

Abstrakte Klassen sind Klassen, in denen nicht alle Methoden auch eine Definition haben. Das ist praktisch, wenn man in einer Klasse sowohl gemeinsames Verhalten als auch sogenannte *Verhaltensvorschriften* (d.h. Vorgabe von Methodensignaturen) haben will.
Abstrakte Klassen & Methoden sind durch das Keyword `abstract` gekennzeichnet.

Abstrakte Klassen

Abstrakte Klassen sind Klassen, in denen nicht alle Methoden auch eine Definition haben. Das ist praktisch, wenn man in einer Klasse sowohl gemeinsames Verhalten als auch sogenannte *Verhaltensvorschriften* (d.h. Vorgabe von Methodensignaturen) haben will.
Abstrakte Klassen & Methoden sind durch das Keyword `abstract` gekennzeichnet.

```
1  public abstract class UniEmployee {
2      private String name;
3      private long pNumber;
4
5      public UniEmployee(String name, long pNumber) {
6          this.name = name;
7          this.pNumber = pNumber;
8      }
9
10     public String getName() {return this.name;}
11     //abstract because different employees have different salary
        calculations
12     public abstract int calculateSalary();
13 }
```

Abstrakte Klassen

Wichtige Eigenschaften von abstrakten Klassen:

- Sie können *nicht* instanziert werden:

```
1 UniEmployee employee = new UniEmployee("T. Est", 42);
2 //this is not allowed!
```

Abstrakte Klassen

Wichtige Eigenschaften von abstrakten Klassen:

- Sie können *nicht* instanziert werden:

```
1 UniEmployee employee = new UniEmployee("T. Est", 42);
2 //this is not allowed!
```

- Erbende Klassen müssen entweder *alle* abstrakten Methoden implementieren ODER auch abstrakt sein:

```
1 public abstract class ServiceEmployee extends UniEmployee{ ... }
2
3 public class ResearchEmployee extends UniEmployee {
4     @Override
5     public int calculateSalary { ... }
6 }
```

Abstrakte Klassen

Wichtige Eigenschaften von abstrakten Klassen:

- Sie können *nicht* instanziert werden:

```
1 UniEmployee employee = new UniEmployee("T. Est", 42);
2 //this is not allowed!
```

- Erbende Klassen müssen entweder *alle* abstrakten Methoden implementieren ODER auch abstrakt sein:

```
1 public abstract class ServiceEmployee extends UniEmployee{ ... }
2
3 public class ResearchEmployee extends UniEmployee {
4     @Override
5     public int calculateSalary { ... }
6 }
```

- Wenn eine Klasse eine abstrakte Methode enthält, muss sie als abstrakt markiert werden:

```
1 public class Truck {
2     public abstract void drive();
3 } //this is not permitted, the class must be abstract
```

Exceptions

Exceptions

Exceptions sind spezielle Java-Klassen, welche Ausnahmezustände im Verlauf eines Programms darstellen und *geworfen* werden, wenn die zugehörigen Zustände eintreten.
Normalerweise beendet eine Exception ein Programm, wenn sie nicht abgefangen und behandelt wird. Zustände, in denen Exceptions auftreten, sind u.a.:

- Arithmetische Fehler(z.B. Division durch 0)

Exceptions

Exceptions sind spezielle Java-Klassen, welche Ausnahmezustände im Verlauf eines Programms darstellen und *geworfen* werden, wenn die zugehörigen Zustände eintreten.
Normalerweise beendet eine Exception ein Programm, wenn sie nicht abgefangen und behandelt wird. Zustände, in denen Exceptions auftreten, sind u.a.:

- Arithmetische Fehler(z.B. Division durch 0)
- Ungenügende Ressourcen(z.B. Nicht genug RAM)

Exceptions

Exceptions sind spezielle Java-Klassen, welche Ausnahmezustände im Verlauf eines Programms darstellen und *geworfen* werden, wenn die zugehörigen Zustände eintreten.
Normalerweise beendet eine Exception ein Programm, wenn sie nicht abgefangen und behandelt wird. Zustände, in denen Exceptions auftreten, sind u.a.:

- Arithmetische Fehler(z.B. Division durch 0)
- Ungenügende Ressourcen(z.B. Nicht genug RAM)
- Fehlerhafter Programmcode(z.B. Null-Pointer Dereferenzierung)

Exceptions

Exceptions sind spezielle Java-Klassen, welche Ausnahmezustände im Verlauf eines Programms darstellen und *geworfen* werden, wenn die zugehörigen Zustände eintreten.
Normalerweise beendet eine Exception ein Programm, wenn sie nicht abgefangen und behandelt wird. Zustände, in denen Exceptions auftreten, sind u.a.:

- Arithmetische Fehler(z.B. Division durch 0)
- Ungenügende Ressourcen(z.B. Nicht genug RAM)
- Fehlerhafter Programmcode(z.B. Null-Pointer Dereferenzierung)
- ...

Exceptions

Eine Exception generiert einen sogenannten *Stacktrace* wenn sie geworfen wird. Ein Stacktrace beinhaltet die „Position“ im Programm, in dem die Exception aufgetreten ist.

Exceptions

Eine Exception generiert einen sogenannten *Stacktrace* wenn sie geworfen wird. Ein Stacktrace beinhaltet die „Position“ im Programm, in dem die Exception aufgetreten ist.

Ein Stacktrace sieht folgendermaßen aus:

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.myproject.Book.getTitle(Book.java:16)
    at com.example.myproject.Author.getBookTitles(Author.java:25)
    at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

Exceptions

Eine Exception generiert einen sogenannten *Stacktrace* wenn sie geworfen wird. Ein Stacktrace beinhaltet die „Position“ im Programm, in dem die Exception aufgetreten ist.
Ein Stacktrace sieht folgendermaßen aus:

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.myproject.Book.getTitle(Book.java:16)
    at com.example.myproject.Author.getBookTitles(Author.java:25)
    at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

Im vorangegangenen Beispiel ist die Exception im `main`-Thread innerhalb der Methode `Book.getTitle` aufgetreten.

Exceptions

Eine Exception generiert einen sogenannten *Stacktrace* wenn sie geworfen wird. Ein Stacktrace beinhaltet die „Position“ im Programm, in dem die Exception aufgetreten ist.
Ein Stacktrace sieht folgendermaßen aus:

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.myproject.Book.getTitle(Book.java:16)
    at com.example.myproject.Author.getBookTitles(Author.java:25)
    at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

Im vorangegangenen Beispiel ist die Exception im `main`-Thread innerhalb der Methode `Book.getTitle` aufgetreten.
In den Zeilen darunter sind die Methoden aufgelistet, die die jeweils darüberliegende Methode aufgerufen haben.

Exceptions

Eine Exception generiert einen sogenannten *Stacktrace* wenn sie geworfen wird. Ein Stacktrace beinhaltet die „Position“ im Programm, in dem die Exception aufgetreten ist.
Ein Stacktrace sieht folgendermaßen aus:

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.myproject.Book.getTitle(Book.java:16)
    at com.example.myproject.Author.getBookTitles(Author.java:25)
    at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

Im vorangegangenen Beispiel ist die Exception im `main`-Thread innerhalb der Methode `Book.getTitle` aufgetreten.
In den Zeilen darunter sind die Methoden aufgelistet, die die jeweils darüberliegende Methode aufgerufen haben.
Die Dateinamen und Zahlen in den Klammern in jeder Zeile geben an, wo genau der Aufruf stattgefunden hat(im Format *<quelldatei:zeile>*).

Exceptions

Um Exceptions zu behandeln, gibt es in Java den `try-catch`-Mechanismus. Dieser führt den Code im `try`-Block aus und fängt jede Exception ab, welche dann im `catch`-Block behandelt wird. Man kann den `catch`-Block auch anweisen, nur bestimmmte Arten von Exceptions zu behandeln.

Exceptions

Um Exceptions zu behandeln, gibt es in Java den `try-catch`-Mechanismus. Dieser führt den Code im `try`-Block aus und fängt jede Exception ab, welche dann im `catch`-Block behandelt wird. Man kann den `catch`-Block auch anweisen, nur bestimmmte Arten von Exceptions zu behandeln.

```
...
try {
    int result = numerator/denominator;
} catch(ArithmeticException e) {
    System.out.println("Illegal division occurred!");
}
```

Exceptions

Um Exceptions zu behandeln, gibt es in Java den `try-catch`-Mechanismus. Dieser führt den Code im `try`-Block aus und fängt jede Exception ab, welche dann im `catch`-Block behandelt wird. Man kann den `catch`-Block auch anweisen, nur bestimmmte Arten von Exceptions zu behandeln.

```
...
try {
    int result = numerator/denominator;
} catch(ArithmeticException e) {
    System.out.println("Illegal division occurred!");
}
```

Wenn man selber Exceptions „werfen“ will, kann das mithilfe des `throw`-Keywords machen.

```
public void setValue(int val) {
    if(value <= 0) throw new IllegalArgumentException();
    ...
}
```

Generics

Generics

Generics sind Javas Implementierung von generischer Programmierung.

Generics

Generics sind Javas Implementierung von generischer Programmierung. Generische Programmierung ist ein Programmierkonzept, welches ermöglicht, verschiedene Ausprägungen einer Klasse bzw. eines Objektes mithilfe einer sogenannten *Typvariable* zu generieren:

Generics

Generics sind Javas Implementierung von generischer Programmierung. Generische Programmierung ist ein Programmierkonzept, welches ermöglicht, verschiedene Ausprägungen einer Klasse bzw. eines Objektes mithilfe einer sogenannten *Typvariable* zu generieren:

```
1 public class Node<T> { //T for type
2     private T data;
3     private Node<T> leftChild, rightChild;
4     public Node(T data, Node<T> leftChild, Node<T> rightChild) {...}
5     public Node(T data);
6     public T getData() { return this.data; }
7     public Node<T> getLeftChild() { return this.leftChild; }
8 }
```

Generics

Generics sind Javas Implementierung von generischer Programmierung. Generische Programmierung ist ein Programmierkonzept, welches ermöglicht, verschiedene Ausprägungen einer Klasse bzw. eines Objektes mithilfe einer sogenannten *Typvariable* zu generieren:

```
1 public class Node<T> { //T for type
2     private T data;
3     private Node<T> leftChild, rightChild;
4     public Node(T data, Node<T> leftChild, Node<T> rightChild) {...}
5     public Node(T data);
6     public T getData() { return this.data; }
7     public Node<T> getLeftChild() { return this.leftChild; }
8 }
```

In diesem Beispiel ist T die Typvariable der Klasse `Node`. Die Typvariable wird innerhalb der Klasse verwendet, um „unbekannte“ Typen festzulegen und zu garantieren, dass diese dem vorgegebenen Typen T entsprechen.

Generics

Die Typvariablen werden dann bei der Deklaration und Instanzierung konkret festgelegt:

Generics

Die Typvariablen werden dann bei der Deklaration und Instanzierung konkret festgelegt:

```
1 Node<Integer> leftChild = new Node<>(42); //using the <> Operator
2 Node<Integer> rightChild = new Node<Integer>(1337); //specifying T in
      instantiation as well
3 Node<Integer> root = new Node<>(10, leftChild, rightChild);
```

Generics

Die Typvariablen werden dann bei der Deklaration und Instanzierung konkret festgelegt:

```
1 Node<Integer> leftChild = new Node<>(42); //using the <> Operator
2 Node<Integer> rightChild = new Node<Integer>(1337); //specifying T in
      instantiation as well
3 Node<Integer> root = new Node<>(10, leftChild, rightChild);
```

Wenn der Compiler nicht automatisch aus den Argumenten des Konstruktors ableiten kann, welchen Typ die Typvariable annimmt, **muss** T auch im Aufruf des Konstruktors angegeben werden.

Generics

Die Typvariablen werden dann bei der Deklaration und Instanzierung konkret festgelegt:

```
1 Node<Integer> leftChild = new Node<>(42); //using the <> Operator
2 Node<Integer> rightChild = new Node<Integer>(1337); //specifying T in
      instantiation as well
3 Node<Integer> root = new Node<>(10, leftChild, rightChild);
```

Wenn der Compiler nicht automatisch aus den Argumenten des Konstruktors ableiten kann, welchen Typ die Typvariable annimmt, **muss** T auch im Aufruf des Konstruktors angegeben werden.

>DEMO<

Collections

Collections

Java bietet im Package `java.util` mit der *Collections-API* Datenstrukturen und Mechanismen an, die die Arbeit mit Datensätzen von variabler Größe wesentlich erleichtert.

Collections

Java bietet im Package `java.util` mit der *Collections-API* Datenstrukturen und Mechanismen an, die die Arbeit mit Datensätzen von variabler Größe wesentlich erleichtert.
Arrays sind für solche Datensätze ungeeignet, da man ihre Größe nach der Erstellung nicht mehr ändern kann.

Collections

Java bietet im Package `java.util` mit der *Collections-API* Datenstrukturen und Mechanismen an, die die Arbeit mit Datensätzen von variabler Größe wesentlich erleichtert.
Arrays sind für solche Datensätze ungeeignet, da man ihre Größe nach der Erstellung nicht mehr ändern kann.
Außerdem legen die verschiedenen Datenstrukturen teilweise auch Beschränkungen auf die in ihnen enthaltenen Daten, damit man aufwändige Prüfungen nicht selbst erledigen muss.

Collections

Die Collections-API verwendet Generics, um festzulegen, welchen Typ die enthaltenen Daten haben:

```
1 List<String> stringList = new ArrayList<>();
2 stringList.add("foo");
3 stringList.add("bar");
```

Collections

Die Collections-API verwendet Generics, um festzulegen, welchen Typ die enthaltenen Daten haben:

```
List<String> stringList = new ArrayList<>();
stringList.add("foo");
stringList.add("bar");
```

Außerdem „erzwingt“ die Verwendung von Generics, dass eine Collection nur Elemente des gleichen oder eines erbenden Typs enthält:

```
Set<Number> numberSet = new HashSet<Number>();
numberList.add(1.07f);
numberList.add(0815);
numberList.add("8998"); //not permitted, as String does not inherit from
    Number
```

Collections

Um über die Elemente einer Collection zu iterieren, gibt es in der Collections-API den `Iterator`.

Collections

Um über die Elemente einer Collection zu iterieren, gibt es in der Collections-API den `Iterator`.
Ein Iterator für eine Collection kann über die `iterator()`-Methode der Collection erstellt werden. Das Iterator-Interface definiert u.a. die folgenden Methoden:

Collections

Um über die Elemente einer Collection zu iterieren, gibt es in der Collections-API den `Iterator`.
Ein Iterator für eine Collection kann über die `iterator()`-Methode der Collection erstellt werden. Das Iterator-Interface definiert u.a. die folgenden Methoden:

- hasNext() - [`boolean`] Sind noch Elemente in der Collection vorhanden?

Collections

Um über die Elemente einer Collection zu iterieren, gibt es in der Collections-API den `Iterator`.
Ein Iterator für eine Collection kann über die `iterator()`-Methode der Collection erstellt werden. Das Iterator-Interface definiert u.a. die folgenden Methoden:

- hasNext() - [`boolean`] Sind noch Elemente in der Collection vorhanden?
- next() - [T] Gibt das nächste Element zurück.

Collections

Um über die Elemente einer Collection zu iterieren, gibt es in der Collections-API den `Iterator`.
Ein Iterator für eine Collection kann über die `iterator()`-Methode der Collection erstellt werden. Das Iterator-Interface definiert u.a. die folgenden Methoden:

- hasNext() - [`boolean`] Sind noch Elemente in der Collection vorhanden?
- next() - [T] Gibt das nächste Element zurück.
- delete() - [`void`] Entfernt das zuletzt zurückgegebene Element.

Collections

Um über die Elemente einer Collection zu iterieren, gibt es in der Collections-API den `Iterator`.
Ein Iterator für eine Collection kann über die `iterator()`-Methode der Collection erstellt werden. Das Iterator-Interface definiert u.a. die folgenden Methoden:

- hasNext() - [`boolean`] Sind noch Elemente in der Collection vorhanden?
- next() - [T] Gibt das nächste Element zurück.
- delete() - [`void`] Entfernt das zuletzt zurückgegebene Element.

Für einen detaillierten Workflow mit Iterators siehe 07-collections.

Andere Programmierkonzepte

Casting

Casting bezeichnet in vielen Programmiersprachen die Umwandlung eines Datentypen in einen anderen, kompatiblen, Datentypen.
In Java kann man zusätzlich ein Objekt in ein anderes Objekt umwandeln, sofern bestimmte Bedingungen erfüllt sind.[1]

[1] Für Details siehe 08-progconcepts

Casting

Casting bezeichnet in vielen Programmiersprachen die Umwandlung eines Datentypen in einen anderen, kompatiblen, Datentypen.
In Java kann man zusätzlich ein Objekt in ein anderes Objekt umwandeln, sofern bestimmte Bedingungen erfüllt sind.[1]
Ein Cast hat die folgende Form:

```
1 int castedInt = 17;
2 float intCast = (float)castedInt;
3 System.out.println(intCast); //prints 17.0
```

[1] Für Details siehe 08-progconcepts

Casting

Casting bezeichnet in vielen Programmiersprachen die Umwandlung eines Datentypen in einen anderen, kompatiblen, Datentypen.
In Java kann man zusätzlich ein Objekt in ein anderes Objekt umwandeln, sofern bestimmte Bedingungen erfüllt sind.[1]
Ein Cast hat die folgende Form:

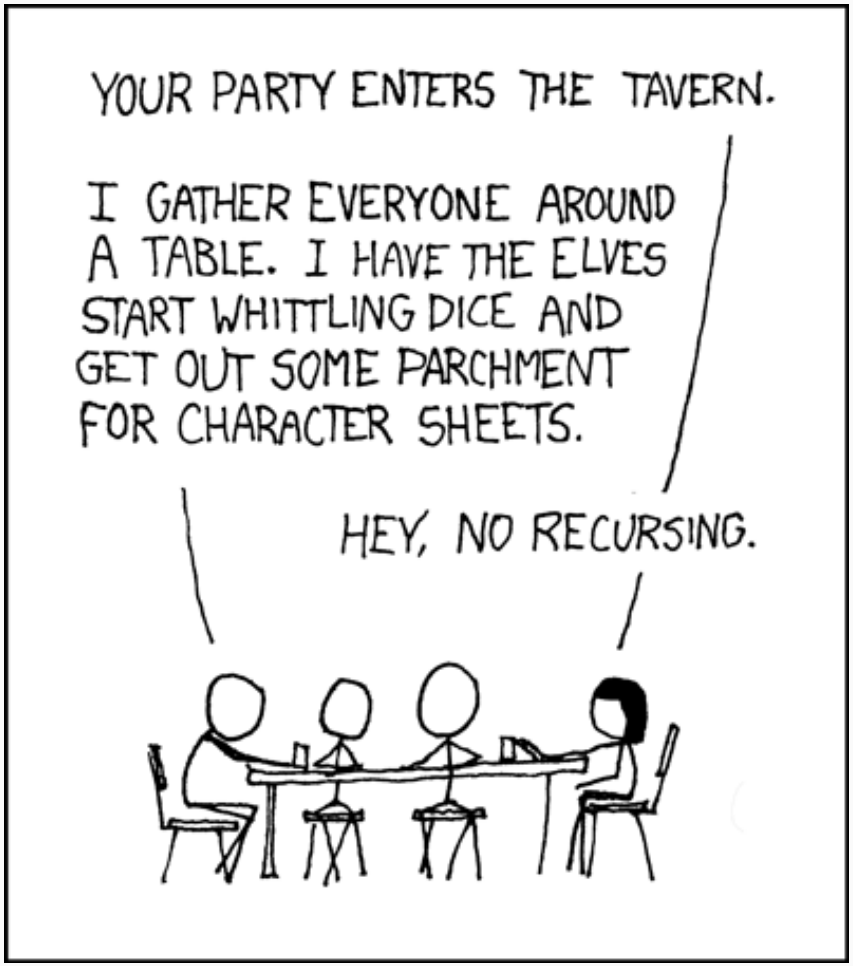
```java
int castedInt = 17;
float intCast = (float)castedInt;
System.out.println(intCast); //prints 17.0
```

Eine wichtige Regel beim Casting ist, dass keine Informationen verloren gehen dürfen.
Deshalb sind Casts wie `int` →`float` erlaubt, `float` →`int` aber nicht.

[1] Für Details siehe 08-progconcepts

Rekursion

Rekursion bezeichnet eine Funktion bzw. eine Lösungsstrategie, die sich selbst wieder aufruft, meistens mit einem Subset der ursprünglichen Daten.

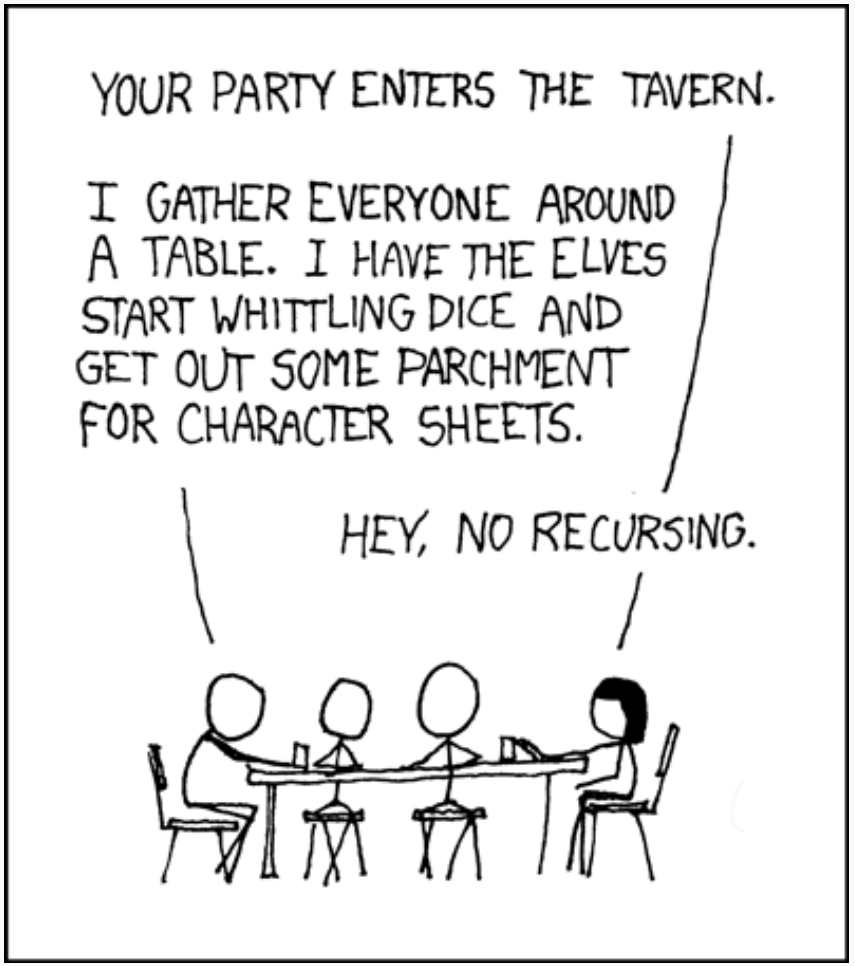


https://xkcd.com/244

Rekursion

Rekursion bezeichnet eine Funktion bzw. eine Lösungsstrategie, die sich selbst wieder aufruft, meistens mit einem Subset der ursprünglichen Daten.

Dies passiert so oft, bis entweder ein sogenannter *Basecase* auftritt, für den eine Lösung bekannt ist, oder aber keine Lösung gefunden werden kann (oder eine `StackOverflowException` auftritt).



https://xkcd.com/244

Rekursion:Beispiel

Ein klassisches Beispiel für Rekursion ist eine Funktion, die die n-te Zahl der Fibonacci-Folge berechnet:

```
1 public int fibonacci(int n) {
2     if(n < 0) throw new IllegalArgumentException();
3     if(n == 0 || n == 1) return 1; //define basecase
4     return fibonacci(n-1) + fibonacci(n-2); //recursive call
5 }
```

JavaDoc

JavaDoc

JavaDoc ist gleichzeitig die „Sprache“ des Java-Dokusystems und das Tool, mit der die eigentliche Dokumentation generiert wird. Javadoc verwendet sogenannte *Tags*, um die Dokumentation zu formatieren.

JavaDoc

JavaDoc ist gleichzeitig die „Sprache“ des Java-Dokusystems und das Tool, mit der die eigentliche Dokumentation generiert wird. Javadoc verwendet sogenannte *Tags*, um die Dokumentation zu formatieren. Dazu wird in einem speziellen mehrzeiligen Kommentar(speziell, weil er mit /** beginnt) mithilfe der Tags Informationen zu Parametern, Rückgabewerten etc. aufgeschrieben und anschließend vom Javadoc-Tool formatiert.

JavaDoc

JavaDoc ist gleichzeitig die „Sprache“ des Java-Dokusystems und das Tool, mit der die eigentliche Dokumentation generiert wird. Javadoc verwendet sogenannte *Tags*, um die Dokumentation zu formatieren. Dazu wird in einem speziellen mehrzeiligen Kommentar(speziell, weil er mit `/**` beginnt) mithilfe der Tags Informationen zu Parametern, Rückgabewerten etc. aufgeschrieben und anschließend vom Javadoc-Tool formatiert.

Einige übliche Tags und ihre Bedeutung:

`@author`	Gibt den Autor an
`@param` *<name>*	Beschreibung v. Param. *<name>* (nur Methoden)
`@return`	Beschreibung des Rückgabewertes (nur Methoden)
`@throws` *<exception>*	Wann *<exception>* auftritt (nur Methoden)
`{@code` *<text>*`}`	Formatiert *<text>* als Code(d.h. monospaced)

JavaDoc

Eine Javadoc Beschreibung sieht z.B. so aus:

```java
/**
 * Generate a String which is formatted for logging output.
 *
 * @param name The name of the process.
 * @param message The actual message to be printed.
 * @param isError Whether the output should be marked as error.
 *
 * @return A String formatted as "[<time>] <name> <level> <message>"
 */
public String getLogFormat(String name, String message, boolean isError)
     {
    String msgLevel = isError ? "CRITICAL" : "INFO";
    String ret = String.format("[%d] %s: %s: %s",
            System.nanoTime(),  name, msgLevel, message);
    return ret;
}
```

Das Statement `<expr> ? <true> : <false>` ist eine Kurzform eines if-Statements