Allgemeine Informatik I

Java Cheatsheet

Maximilian Kratz, Christopher Katins, Jannis Blüml

Version: 14. April 2020, 18:34 Uhr

Hinweis

Dieses Cheatsheet fasst die meisten Grundlagen der Programmiersprache Java zusammen, ist aber keinesfalls allumfassend. Es wurde in den vergangenen Jahren von Tutor*innen zusammengestellt und basiert größtenteils auf dem Repetitorium zu Java der Veranstaltung *Softwarepraktikum* aus dem Wintersemester 2017/2018.

Des Weiteren möchten wir an dieser Stelle noch einmal explizit darauf hinweisen, dass **dieses Dokument nicht zur Klausur zugelassen ist**!

Eigenschaften von Java

- Klassenbasierte und objektorientierte Programmierung.
- Typsicher.
- Hohe Bekanntheit (meistgenutzte Programmiersprache).
- Betriebssystemunabhängigkeit.
- Einfache Fehler-/Ausnahmebehandlung.
- Großes Angebot an Bibliotheken und Werkzeugen.
- Einfache Syntax.

Datentypen

Typen	Bedeutung	Operatoren	Beispiele
Primitive Datentypen			
`int`	Ganzzahlen von -2^{31} bis $2^{31}-1$	+ - · / %	2, 3, -5, 21221213
`float`	Fließkommazahlen bis 32 Bit	+ - · /	2.3, 3.4, -1.3
`double`	Fließkommazahlen bis 64 Bit	+ - · /	2.3, 3.4, -1.3
`char`	einzelnes Zeichen/Zeichenwert		'A', '1', '%'
`boolean`	Wahrheitswerte	&& || !	`true`, `false`
(Komplexere) Datentypen			
`String`	Zeichenkette aus Characters	+ (Konkatenation)	"Hallo", "2.5", "CB"
Selbst erzeugte Objekte			Ein Objekt der Klasse Graph hat den Typ Graph.

Häufige Fehler in Java

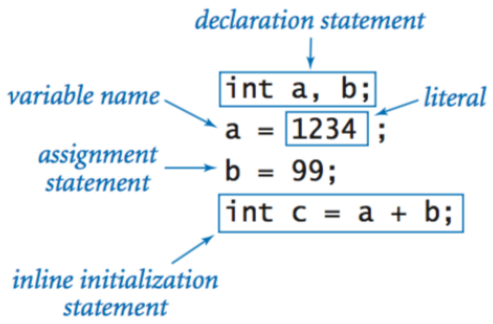
1. Java ist case sensitive, sprich Groß- und Kleinbuchstaben spielen eine Rolle.
2. Für jede öffnende Klammer muss es auch eine passende schließende Klammer geben.
3. Semikola nicht vergessen.
4. statische Variablen/Methoden und nicht statische Variablen. In statischen Methoden können nur statische Variablen verwendet werden.
5. =, == und `equals()` nicht verwechseln.
6. Der Index startet bei 0 nicht bei 1.
7. Wenn Exceptions zu erwarten sind/geworfen werden, müssen diese behandelt werden. Zum Beispiel mit `try-catch`-Blöcken.
8. Modifiziert keine Listen, während über sie iteriert wird.
9. Vergesst nicht Übergaben zu überprüfen (z. B. auf `null`).

Operatoren

Operatoren – Übersicht –

Zeichen	Bedeutung	Kommentar
Arithmetische Operatoren und Zuweisung		
=	Zuweisung	
+	Addition	
-	Subtraktion	
*	Multiplikation	
/	Division	Keine Division durch 0
%	Modulo	Rest einer Division
Inkrementierung und Dekrementierung		
++	Inkrementieren um 1	
−−	Dekrementieren um 1	
a += b	Entspricht a=a+b	
*=, /=, ...	Siehe +=	
Logische Operatoren		
<, >	Kleiner, Größer	
<=, >=	Kleiner Gleich, ...	
==	Gleich	Vorsicht bei nicht primitiven Datentypen
!=	Ungleich	siehe ==
&&	Und	
||	Oder	

Operatoren – Beispiel –




```
int a = 5;
int b = 7;
boolean c = a < b;
// Ergebnis eines logischen Operators
// ist immer ein Wahrheitswert.

int a = 5;
int b = a++; // b = 5 und a=6
b += ++a; // b = 12 und a = 7
// Achtung a++ und ++a machen
// verschiedene Dinge.
```

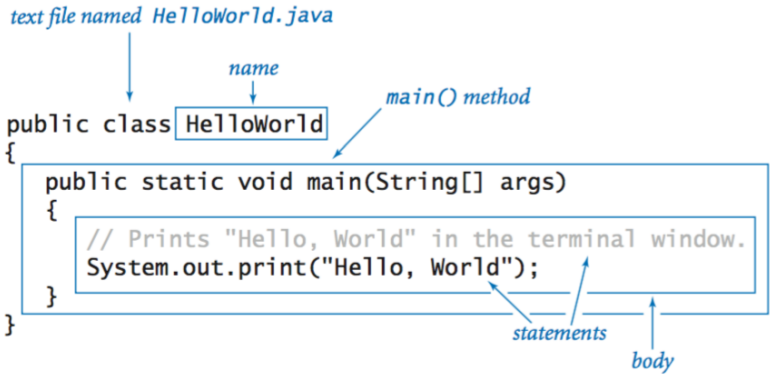
Operatorpräzedenz

++, –	Inkrement und Dekrement
+, -	Unäres Plus und Minus (Vorzeichen)
~	Bitweises Komplement
!	Logisches Komplement
(Typ)	Cast
*, /, %	Multiplikation, Division, Rest (Modulo)
+, -	Addition und Subtraktion
+	String-Konkatenation
<<	Verschiebung links
>>	Rechtsverschiebung mit Vorzeichenerweiterung
>>>	Rechtsverschiebung ohne Vorzeichenerweiterung
<, <=, >, >=	Numerische Vergleiche
`instanceof`	Typvergleich
==, !=	Gleich-/Ungleichheit von Werten/Referenzen
&	Und
^	Xor
|	Oder
&&	Logisches konditionales Und
||	Logisches konditionales Oder
?:	Bedingungsoperator (ternärer Operator)
=	Zuweisung
+=, &=, *=, ...	Zuweisung mit Operation

Hinweis: Wir haben nicht alle dieser Operatoren kennengelernt und verwendet.

Klassenaufbau

Hello World



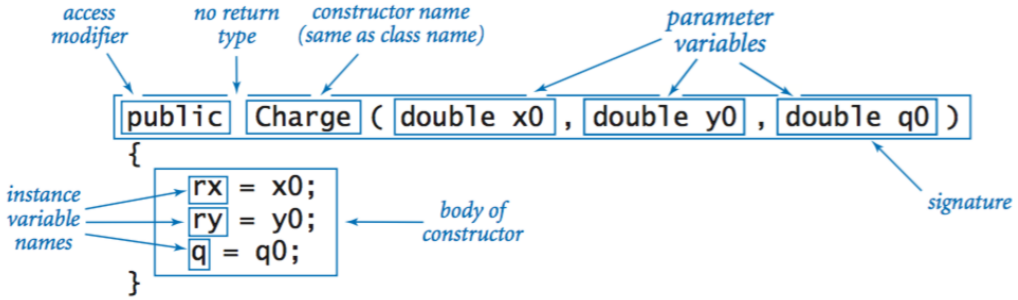
Klassen bestehen immer aus:

- Zugriffsspezifikation (hier `public`).
- Klassenname, entsprechend dem Dateinamen (hier *HelloWorld*).
- Dem Schlüsselwort `class`.

Weiter kann eine Klasse besitzen:

- Objektvariablen.
- Methoden.
- Konstruktor.
- „main“-Methode.

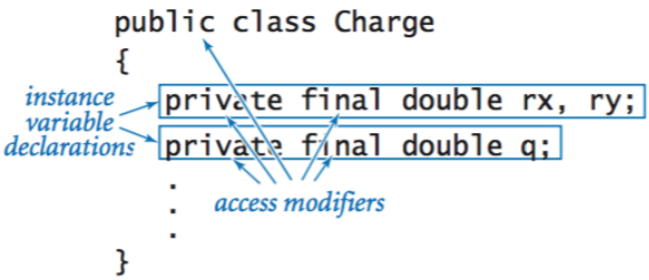
Konstruktor



Ein Konstruktor besteht aus:

- Zugriffsspezifikation (hier `public`).
- **Keinen** Rückgabewert (auch kein `void`).
- Klassenname (hier *Charge*).
- Übergabeparameter(n) (kann auch leer sein ⇒ (), hier *x0, y0, q0*).

Variablen

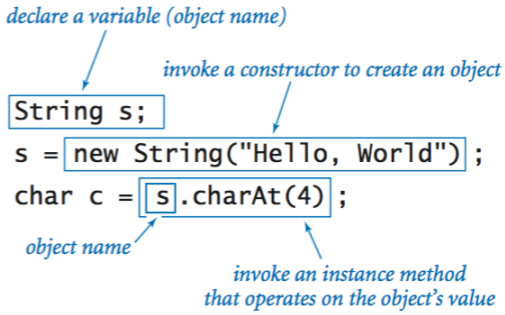


Eine Variable besteht aus:

- Zugriffsspezifikation (hier `private`, `final`).
- Variablenname.

Hinweis: Eine Variable kann bei der Deklarieren bereits initialisiert werden.
Hinweis: Es können mehrere Variablen gleichzeitig deklariert werden.
Hinweis: Variablen können global (Instanzvariable) oder lokal erstellt werden.

Objekte

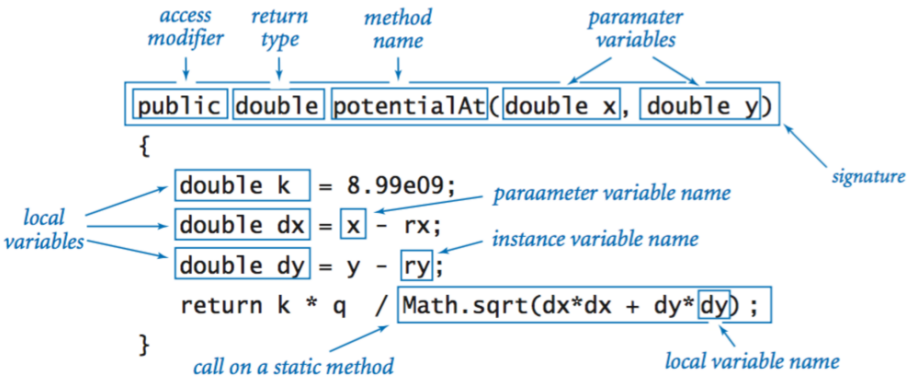


Eine Objekt besteht aus:

- Seinem Typ (Klasse, hier `String`).
- Einem Namen (hier *s*).

Hinweis: Ein Objekt wird mittels des Schlüsselworts `new` initialisiert.
Hinweis: Eis Objekt wird mittels seines Konstruktors initialisiert.
Hinweis: Mit „`Instanzname`.“ greift man auf die Methoden der Instanz zu.

Methoden

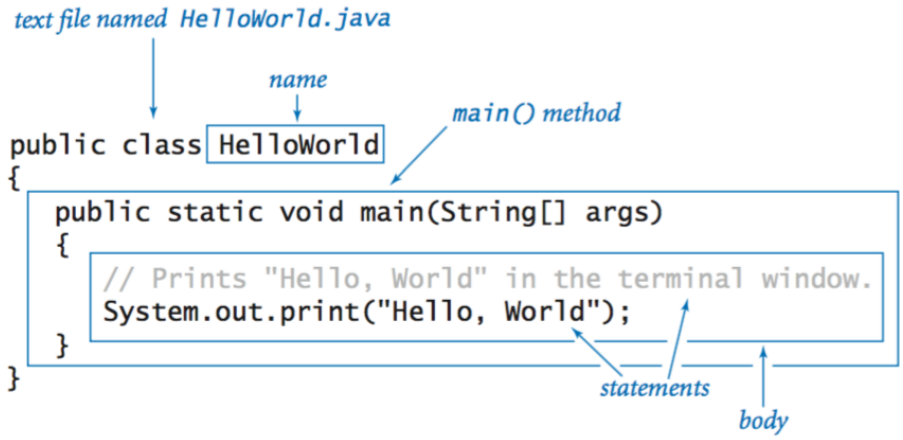


Eine Methode besteht aus:

- Zugriffsspezifikation (hier `public`).
- Rückgabetyp (wenn kein Rückgabetype, dann `void`, hier `double`).
- Methodenname (hier `potentialAt`).
- Übergabeparameter(n) (können auch leer sein ⇒ (), hier x,y).
- Methodenrumpf (durch `{` eingeleitet und mit `}` beendet).

Hinweis: Eine Methode mit Rückgabetyp erfordert immer ein `return` Statement.
Hinweis: Eine Methode wird mittels ihres Namens aufgerufen.
Hinweis: Auch wenn es keine Übergabeparamter gibt, werden die Klammern (`()`) benötigt.

Main

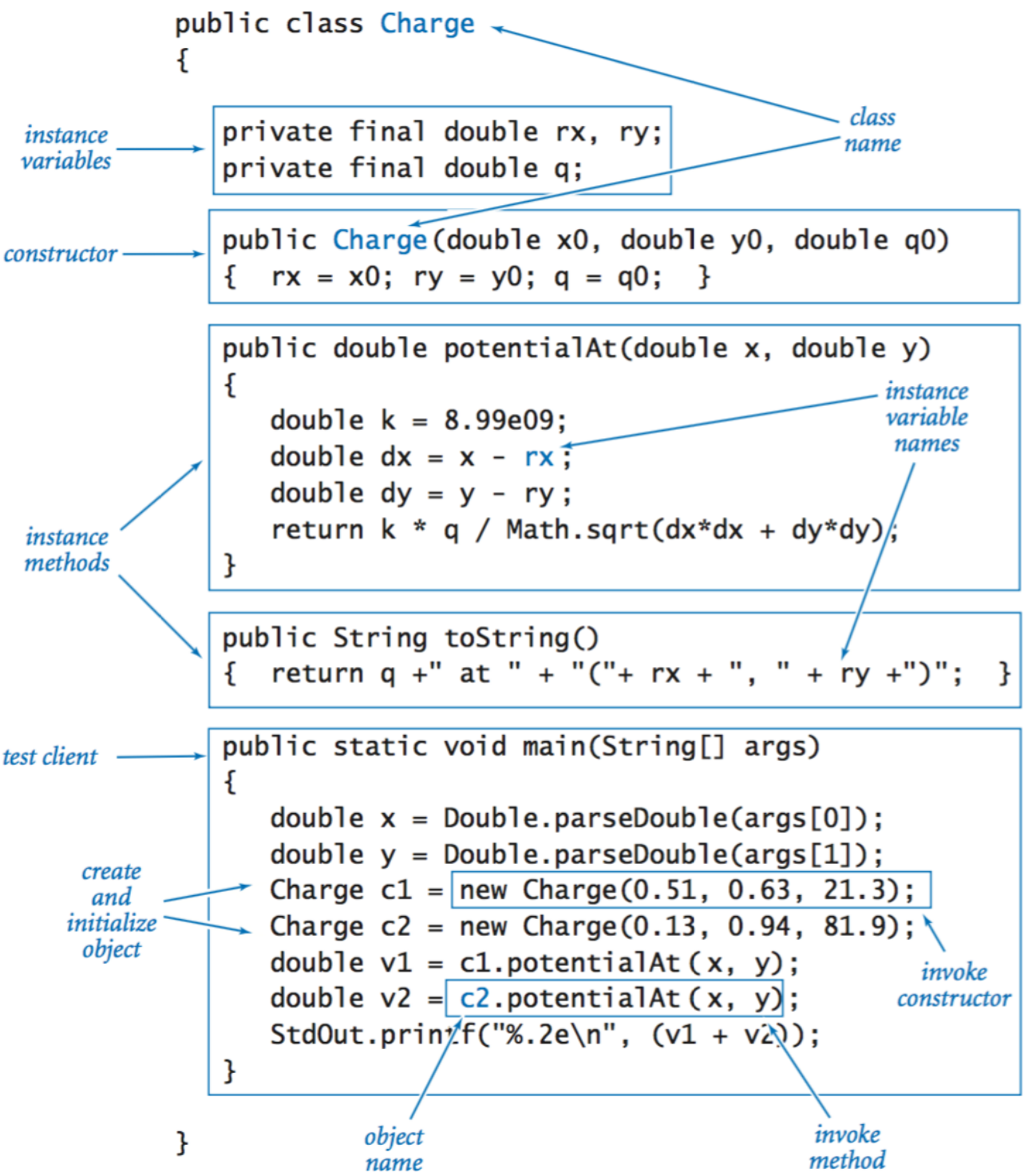


Eine Mainmethode hat den folgenden Aufbau:

- Hat als Zugriffsspezifikation `public static`.
- Der Rückgabetyp ist `void`.
- Der Methodenname ist `main`.
- Die Methode bekommt ein Array namens „args“ vom Typ `String` übergeben.

Hinweis: Die `main`-Methode wird beim Ausführen der Datei ausgeführt.

Beispielklasse



Modifikatoren

Zugriffsmodifikatoren

- `public`: Immer zugreifbar.
- `protected`: Nur aus dem Paket und seinen Erben zugreifbar.
- **Kein Modifikator (auch „package-private“ genannt)** (Default): Aus dem Paket zugreifbar.
- `private`: Nur aus der eigenen Klasse zugreifbar.

Nicht Zugriffsmodifikatoren

- `static`: Erzeugt Klassenmethoden/-variablen. Hierfür müssen die Klassen nicht initialisiert werden.
- `final`: Mit `final` deklarierte Methoden, Variablen, etc. lassen sich nach der Initialisierung nicht mehr verändert werden. Eine Deklaration ohne Initialisierung ist nicht möglich.
- `abstract`: Erzeugt abstrakte Methoden und Klassen, welche nicht initialisiert werden können.

Kontrollstrukturen

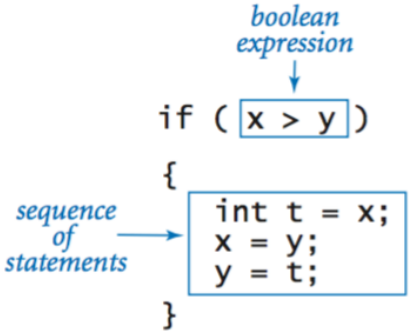
if-else Bedingung

Eine `if`-Bedingung besteht aus:

- Dem Schlüsselwort `if`.
- Eine Bedingung (hier $x > y$). Diese muss zu einem Wahrheitswert auswerten.
- Einem Rumpf (gekennzeichnet durch `{` und `}`).
- Eine `if`-Bedingung kann mit einem `else`-Statement versehen werden.
- Kann mit „Wenn $(x > y)$ dann a, ansonsten b“ übersetzt werden.

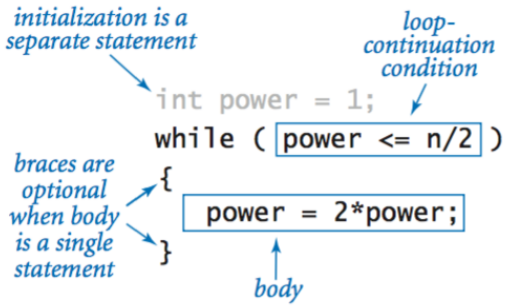
Hinweis: Führt den Rumpf aus, wenn die Bedingung erfüllt ist.

if-else Bedingung – Beispiel –




```
if(x > y) {
        //... a: wenn x>y
} else {
        //... b: sonst
}
```

while-Schleife

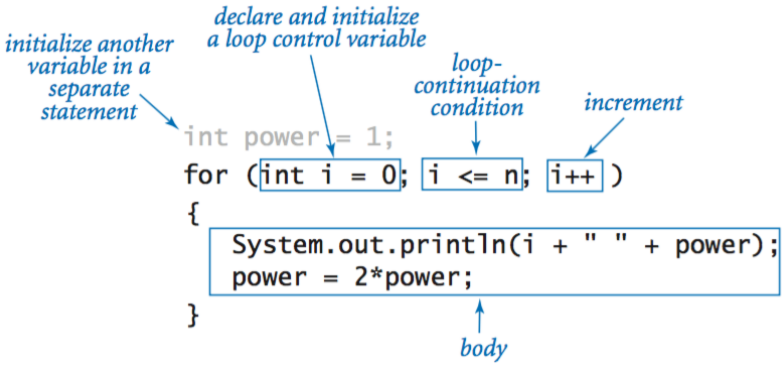


Eine `while`-Schleife besteht aus:

- Dem Schlüsselwort `while`.
- Eine Abbruchbedingung (hier $power <= n/2$). Diese muss zu einem Wahrheitswert auswerten.
- Einem Schleifenrumpf (gekennzeichnet durch { und }).

Hinweis: Wiederholt den Schleifenrumpf, bis die Bedingung nicht mehr erfüllt ist.

for-Schleife



Eine `for`-Schleife besteht aus:

- Dem Schlüsselwort `for`.
- Einer Schleifenvariable (hier i), diese kann an dieser Stelle auch deklariert und initialisiert werden.
- Eine Abbruchbedingung (hier $i <= n$). Diese muss zu einem Wahrheitswert auswerten.
- Eine Inkrementierung oder Dekrementierung der Schleifenvariable.
- Einem Schleifenrumpf (gekennzeichnet durch { und }).

foreach-Schleife

Beispiel:

```
for(int i: list){
 //...
}
```

Eine foreach-Schleife besteht aus:

- Dem Schlüsselwort `for`.
- Einer Schleifenvariable (hier i), diese muss von gleichem Typ wie die Elemente der Liste „list“ sein.
- Einem Schleifenrumpf (gekennzeichnet durch { und }).

Hinweis: Ruft den Rumpf für alle Elemente innerhalb der Liste auf.

Break

```
int factor;
for (factor = 2; factor <= n/factor; factor++)
   if (n % factor == 0) break;

if (factor > n/factor)
   System.out.println(n + " is prime");
```

Ein `break`-Statement unterbricht die aktuelle Kontrollstruktur. Beispielsweise innerhalb einer Schleife wird diese durch ein `break` abgebrochen.

Switch Case

```
switch (day) {
   case 0: System.out.println("Sun"); break;
   case 1: System.out.println("Mon"); break;
   case 2: System.out.println("Tue"); break;
   case 3: System.out.println("Wed"); break;
   case 4: System.out.println("Thu"); break;
   case 5: System.out.println("Fri"); break;
   case 6: System.out.println("Sat"); break;
}
```

Eine Fallunterscheidung über eine Variable (hier *day*).

try-catch-Blöcke

Die `try-catch`-Anweisung umschließt einen Codeabschnitt und wird dafür verwendet mögliche Fehler (Exceptions) innerhalb dieses Codeabschnittes abzufangen, sodass man darauf reagieren kann. Folgend ist die generelle Syntax dargestellt:

```
try {
        // code der gesichert laeuft
} catch (ExceptionKlassenname variablenname) {
        // Fehlerbehandlung
}
```

Sie besteht aus vier Teilen. Der Block, welcher von `try` eingeschlossen wird, läuft gesichert ab. Das heißt, dass Exceptions, die in diesem Block möglicher Weise geworfen werden, vom Programm abgefangen werden. Die Fehlerbehandlung findet innerhalb des `catch`-Blockes statt. Der Exception-Klassenname beschreibt den Fehler auf den das Programm reagieren wollen. Der Variablenname benennt die Exception innerhalb des `catch`-Blockes, sodass entsprechend der Exception innerhalb des `catch`-Blockes reagiert werden kann.

Wenn die abgefangene Exception innerhalb des `try`-Blockes geworfen wird, so springt die Codeausführung direkt in den `catch`-Block. Wenn die Exception nicht geworfen wird, wird der `catch`-Block nie aufgerufen.

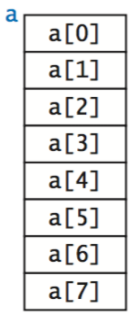
Der `finally`-Block wird einfach an das Ende der `try-catch`-Blöcke gehängt. Seine Semantik ist einfach: Der `finally`-Block wird immer ausgeführt, ganz egal ob im `try`-Block eine Exception geworfen wurde oder nicht:

```
try {
        // code der gesichert laeuft
} catch (ExceptionKlassenname variablenname) {
        // Fehlerbehandlung
} finally {
        // wird in jedem Fall ausgeführt
}
```

Der `finally`-Block ist insbesondere dann sinnvoll, wenn ein Ursprungszustand wieder hergestellt oder aufgeräumt werden soll. So eignet er sich z. B. sehr gut dazu Dateien wieder zu schließen.

Datenstrukturen

Array



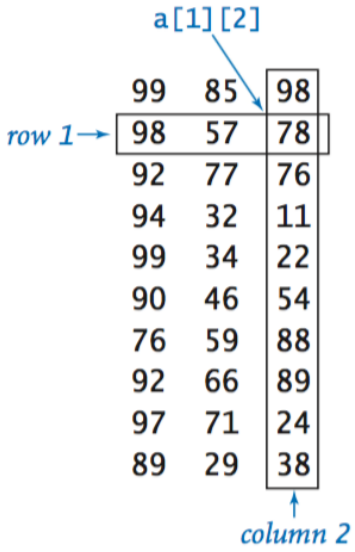
Array Initialisierung

```
String[] SUITS = { "Clubs", "Diamonds", "Hearts", "Spades" };

String[] RANKS = {
   "2", "3", "4", "5", "6", "7", "8", "9", "10",
   "Jack", "Queen", "King", "Ace"
};
```

Sammlung von Elementen eines Typs. Arrays haben eine feste Größe, welche nach der Initialisierung nicht mehr verändert werden kann.

2D Arrays



2D Array Initialisierung

```
double [][] a =
{
   { 99.0, 85.0, 98.0,  0.0 },
   { 98.0, 57.0, 79.0,  0.0 },
   { 92.0, 77.0, 74.0,  0.0 },
   { 94.0, 62.0, 81.0,  0.0 },
   { 99.0, 94.0, 92.0,  0.0 },
   { 80.0, 76.5, 67.0,  0.0 },
   { 76.0, 58.5, 90.5,  0.0 },
   { 92.0, 66.0, 91.0,  0.0 },
   { 97.0, 70.5, 66.5,  0.0 },
   { 89.0, 89.5, 81.0,  0.0 },
   {  0.0,  0.0,  0.0,  0.0 }
};
```

Arrays lassen sich auch bei der Initialisierung bereits zuweisen. Zweidimensionale Arrays sind da keine Ausnahme.

Lists

Eine Liste ist eine `Collection`, welche ihre Elemente indiziert. Dies erlaubt das Hinzufügen, Modifizieren und Entfernen von Elementen durch einen ganzzahligen Index. Eine Liste kann Elemente jedes komplexeren Typs speichern. Primitive Datentypen können nicht direkt verwendet werden. Hierfür existieren allerdings Wrapperklassen wie `Integer`, die anstelle von `int` verwendet werden können. Listen erlauben auch `null` und Duplikate.

Beispiel:

```
List<String> listStrings = new ArrayList<String>();
listStrings.add("One");
listStrings.add("Two");
listStrings.add("Three");
listStrings.add("Four");
```

Sets

Eine Menge, welche keine Duplikate enthalten kann. (Es gilt für alle Elementpaare der Menge, dass `e1.equals(e2)` `false` ist.)

- `add`, welches ein Wert zu dem `Set` hinzufügt.
- `contains`, welches `true` zurückgibt, wenn das übergebene Element in dem `Set` enthalten ist.
- `remove`, welches den übergebenen Wert, falls vorhanden, entfernt.
- `size`, `toArray`, `clear`, ...

Maps

Eine `Map` ist ein Objekt, welches für bestimmte Schlüssel (= Keys) einen Wert hinterlegt. Dies ist vergleichbar mit einer Tabelle, welche zwei Spalten besitzt. Eine `Map` kann keine Duplikate in den Keys enthalten. Jeder Key verweist genau auf einen Wert. Ähnlich zu `List`, kann auch `Map` nur mit komplexeren Datentypen umgehen, nicht aber mit primitiven Datentypen. Java besitzt über das `Map`-Interface die grundlegenden Funktionen:

- `put`, welches ein Key-Value Paar der `Map` hinzufügt.
- `get`, welches zu einem gegeben Key den Wert bestimmt.
- `remove`, welches ein Key-Value Paar entfernt.
- `containsKey`, `containsValue`, `size`, `empty`, ...

Beispiel:

```
public static void main(String[] args) {
        HashMap map = new HashMap();

        // Drei Objekte der Klasse Student_in werden erzeugt.
        Student_in st1 = new Student_in("Topf", "Hans", "12345");
        Student_in st2 = new Student_in("Teller", "Hannes", "12323");
        Student_in st3 = new Student_in("Besteck", "Maxi", "12345");

        // Einfügen der Objekte in die HashMap.
        // Matrikelnummer wird als Key eingetragen.
        map.put(st1.getMatrikelnummer(), st1);
        map.put(st2.getMatrikelnummer(), st2);

        // Student_in st1 wird durch st3 ersetzt, da die
        // Matrikelnummer schon als Schlüssel vergeben ist.
        map.put(st3.getMatrikelnummer(), st3);
}
```

Konventionen

Namenskonventionen

	Anfangsbuchstabe	Beispiel	Besonderheiten
Klasse	Groß	Charge	Meist Substantiv
Variable	Klein	numOfBatteries	
Methode	Klein	potentialAt(x,y)	Geginnt oft mit Verb
Konstante	Nur Großbuchstaben	`PI`	Ggf. mit Unterstrichen: `NEW_PI`
Paket	Nur Kleinbuchstaben	utility	

Hinweis: Java unterscheidet Groß- und Kleinbuchstaben.
Hinweis: Namen aus mehreren Wörtern in „CamelCase“ (jedes Wort beginnt mit Großbuchstabe, z. B. „numOfBatteries“).

Klassenaufbau

Eine Klasse ist wie folgt aufgebaut:
1. Anfangskommentar zur Klasse.
2. `package`- und `import`-Statements.
3. Klassendeklaration/-kopf (= Definition der Klasse!).
4. Variablen und Konstruktoren (Statische Variablen, Instanzvariablen, Konstruktoren).
5. Methoden (gruppiert nach Funktionalität).

Deklarationen

Bei Deklarationen ist Folgendes zu beachten:
- Eine Deklaration pro Zeile.
- Wenn möglich initialisiere Variablen auch dort, wo sie deklariert werden.
- Versuche Deklarationen immer in Blöcken zu machen (z. B. am Anfang einer Methode).
- Methoden werden durch eine Leerzeile getrennt.

Statements

Bei Statements/Befehlen ist Folgendes zu beachten:
- Eine Statement pro Zeile (versuche max. ein Semikolon in einer Zeile zu haben außer in Ausnahmen wie z. B. bei `for`-Schleifen).
- Die `return`-Statements haben nur Klammern, wenn diese der Operatorpräzedenz dienen.
- Kontrollstrukturen haben immer einen Rumpf mit Klammern, die öffnende Klammer ist in der gleichen Zeile wie die Definition.

Beispiel:
```
public int fakultaet(int k) {
    int f = 1; // Jedes Statement in eigener Zeile.

    if(k > 0){ // { In der gleichen Zeile wie die if-Bedingung.
        f = f * k;
        k--; // Nicht mehr als ein Statement, wie "f = f * k; k--;".
    }

    return f; // Keine unnötigen Klammern, wie (f).
}
```

Kommentare

Block Kommentare

```
/*
 * Ist ein Kommentar.
 */
```

Single-Line Kommentare

```
...
/* Ist ein Kommentar. */
...
```

End-of-Line Kommentare

```
... // Ist ein Kommentar.
```

JavaDoc

- Dokumentationskommentare werden in HTML geschrieben.
- Sie beginnen mit /**.
- Sie bestehen aus zwei Teilen.
  - Beschreibender Text.
  - Tags.
- Jede Klasse und Methode sollte einen Javadoc-Kommentar besitzen.
- Häufig genutzte Tags:
  - @param, @author, @return, @see, @throws, @since, @deprecated, @version, @date, ...

Dokumentation (Doc) Kommentar Beispiel

```
/**
* Returns an Image object that can then be painted on the screen.
* The url argument must specify an absolute {@link URL}. The name
* argument is a specifier that is relative to the url argument.
* <p>
* This method always returns immediately, whether or not the
* image exists. When this applet attempts to draw the image on
* the screen, the data will be loaded. The graphics primitives
* that draw the image will incrementally paint on the screen.
*
* @param  url  an absolute URL giving the base location of the image
* @param  name the location of the image, relative to the url argument
* @return      the image at the specified URL
* @see         Image
*/
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Vererbung

Casting

- Casting eines primitiven Types verändert den Typ selbst und verändert den Wert irreversible.
- Casting bei komplexeren Typen/Objekten ändert den Typ bzw. das Objekt selbst nicht.
- Casting verändert die Referenz auf das Objekt, sprich das Objekt wird sozusagen nur neu gelabelt.
- Nicht jeder Cast ist möglich (Siehe `ClassCastException`)!

instanceof Operator

```
// Beispiel:
if (animal instanceof Cat) {
    ((Cat) animal).meow();
}
```

- Der `instanceof` Operator wertet zu einem Wahrheitswert aus.
- Überprüft den Typ eines Objekts gegen den übergebenen Typ.

ClassCastException

```
// Beispiel:
Animal animal;
String s = (String) animal;
```

- Bei fehlerhaften Casts wird eine `ClassCastException` geworfen,

Upcasting

Beispiel:

```
public class Animal {
    public void eat() {
        // ...
    }
}

public class Cat extends Animal {
    public void eat() {
        // ...
    }

    public void meow() {
        // ...
    }
}

Cat cat = new Cat();
Animal animal = cat;
```

Eigenschaften:

- Casting von einer Subklasse zu einer Superklasse.
- Verallgemeinert die Klasse.
- Subklassen spezifische Methoden sind nicht verwendbar.
- Wird eher selten gebraucht.

Downcasting

Beispiel:

```
public class Animal {
    public void eat() {
        // ...
    }
}

public class Cat extends Animal {
    public void eat() {
        // ...
    }

        public void meow() {
        // ...
    }
}

Animal animal = new Cat();

if(animal instanceof Cat)
    ((Cat) animal).meow();
```

Eigenschaften:

- Casting von einer Superklasse zu einer Subklasse.
- Spezifiziert die Klasse.
- Erweitert die Superklasse um spezifische Methoden.
- Zum sicheren Downcasten, wird vorher `instanceof` verwendet.

Vererbung

Allgemeine Informationen:

- Bei der Vererbung wird die Basis oder Elternklasse auch als Superklasse oder Oberklasse bezeichnet.
- Kinderklassen, sprich solche die von anderen erben, werden als Subklasse oder Unterklasse der Elternklasse bezeichnet.
- Kinder erben alle Informationen, sowie Methoden aus der Elternklasse.
- Jede Klasse erbt standardmäßig von der `Object`-Klasse.
- Mit dem Schlüsselwort `super` kann man auf die Elternklasse zugreifen.
- `final` Methoden lassen sich nicht überschreiben.

Beispiel:

```
class Tier {

    public Tier() {
    }
    void bewegtSich() {
        System.out.println("Schwimmen, laufen, kriechen, hüpfen oder fliegen?");
    }
}
class Vogel extends Tier {

    public Vogel(){
        super();
    }
    @Override
    void bewegtSich() {
        System.out.println("Ich fliege");
    }
    void zwitscher() {
        System.out.println("tirilli");
    }
    void frisst() {
        System.out.println(getClass().toString() + " frisst");
    }
}
class Wurm extends Tier {
    //...

    @Override
    void bewegtSich() {
        System.out.println("Ich krieche");
    }
    void frisst() {
        System.out.println(getClass().toString() + " frisst");
    }
}
```

Es ist zu beachten, dass:

- *Vogel* und *Wurm* von *Tier* erben (Schlagwort `extends`).
- Alle drei Klassen die Methode `bewegtSich()` implementieren.
- *Vogel* und *Wurm* überschreiben die Methode der Superklasse in diesem Fall.
- Überschriebene Methoden sollte man mit der Annotation `@Override` kennzeichnen. Der Compiler erfährt hierdurch, dass eine Methode der Elternklasse spezifiziert wird.
- Mit `super()` wird auf die Elternklasse zugegriffen.

Abstrakte Methoden/Klassen

- Eine abstrakte Methode ist eine nur deklarierte, aber nicht definierte (implementierte) Methode.
- Sie wird mit dem Modifikator `abstract` gekennzeichnet und mit einem Semikolon nach der Deklaration abgeschlossen.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst mit `abstract` gekennzeichnet werden.
- Eine solche Klasse kann nicht instantiiert werden, d.h. es gibt keinen Konstruktor und keine Instanzierung mit `new`.
- Jede Unterklasse einer abstrakten Klasse, die nicht alle abstrakten Methoden implementiert (durch Überschreiben), muss selbst abstrakt sein.
- Abstrakte Methoden können nicht `static`, `private` oder `final` sein.
- Auch eine Klasse ohne abstrakte Methoden kann mit `abstract` deklariert werden (kann dann nicht instantiiert werden!).

Interfaces/Schnittstelle

- Das Schlüsselwort `interface` muss (an Stelle von `class`) vor der Definition stehen.
- Alle Methoden eines Interfaces sind (implizit) `public`, d. h. `private` und `protected` sind verboten.
- Alle Methoden eines Interfaces sind (implizit) `abstract`.
- Ein Interface darf nur Attribute haben, die `static` und `final` sind.
- Ein Interface ist nicht instantiierbar und darf keinen Konstruktor haben.
- Wenn eine Klasse ein Interface implementiert, wird dies durch das Schlüsselwort `implements` angezeigt.
- Die Klasse muss alle Methoden implementieren oder abstrakt sein.

Sonstiges

Ausgabe

Die einfachste Ausgabe geschieht über die Kommandozeile/Konsole durch einen der folgenden Befehle:

```
System.out.print("Eine Ausgabe");
System.out.println("Eine Ausgabe mit anschließendem Zeilenumbruch");
```

this

Mit dem Schlüsselwort `this` kann auf die eigene Instanz zugegriffen werden. Somit können wir über `this` auch auf die Instanzvariablen zurückgreifen.

Beispiel:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Tests

JUnit

- Um JUnit verwenden zu können, muss junit.jar als Bibliothek zur Compilezeit verfügbar sein.
- **Begriffe:**
  - **Als Testklasse** wird eine Klasse bezeichnet, die Methoden zum Testen von Code enthält. Die einzige Bedingung besteht darin, dass sie durch einen öffentlichen Default-Konstruktor instantiierbar ist. In der Testklasse sind mit Hilfe der JUnit-Annotationen Testmethoden gekennzeichnet.
  - **Testmethoden** bezeichnen Methoden, die durch Annotationen als solche gekennzeichnet sind. Jede Methode, die das Sichtbarkeitsattribute `public` besitzt, keine Parameter verlangt und als Rückgabetyp `void` liefert, darf als Testmethode gekennzeichnet werden.
  - **Ein Testfall** ist zunächst ein bestimme Vorgehensweise, wie ein Teil einer Software getestet werden soll, zu ihm gehören Testwerte und das erwartete Ergebnis. Abhängig von der Komplexität der zu testenden Funktionalität können Testfälle der selben Art zu einer Testmethode zusammen gefasst werden.
- **Verwendung:** Testmethoden werden durch die JUnit-Annotation `@Test` gekennzeichnet. Zum Überprüfen liefert das Testframework die Klasse `org.junit.Assert`. Tritt eine Abweichung auf, wird ein `java.lang.AssertionError` oder ein davon abgeleiteter Fehler geworfen.
- **Startzustand definieren:** Häufig kommt es vor, dass viele oder sogar alle Testfälle einer Testklasse eine identische Umgebung benötigen. Um diese Vorbereitungen nicht in jeder Testmethode erneut hinschreiben zu müssen, bietet JUnit die Annotation `@Before`.
- **Annotations:**

Annotation	Beschreibung
`@Test`	Kennzeichnung als Testfall.
`@Test(expected = Exeption.class)`	Kennzeichnung als Testfall mit der Festlegung, dass der Test nur erfolgreich ist, wenn die geforderte Exception auftritt.
`@Test(timeout = ...ms)`	Kennzeichnung als Testfall mit der Festlegung, dass der Test erfolgreich ist, wenn die geforderte Zeit in ms nicht überschritten wird.
`@Before`	Ausführung vor jedem Aufruf einer Testmethode zum Aufbau einer definierten Testumgebung.
`@After`	Ausführung nach jedem Aufruf einer Testmethode zur Erledigung von Aufräumarbeiten.
`@BeforeClass`	Eine derartig gekennzeichnete Methode muss statisch definiert sein. Diese Annotation dient der Kennzeichnung zur einmaligen Ausführung vor dem Aufruf aller anderen Testmethoden.
`@AfterClass`	Eine derartig gekennzeichnete Methode muss statisch definiert sein. Diese Annotation dient der Kennzeichnung zur einmaligen Ausführung nach dem Aufruf aller anderen Testmethoden.
`@Ignore(Kommentar)`	Methode wird temporär nicht ausgeführt. Es sollte unbedingt ein Kommentar mit übergeben werden, dieser wird im Protokoll des Testlaufs ausgegeben.

Beispiel:

```
public class CollectionTest {
    @Test
    public void sortedSet() {
        Set<String> s = new TreeSet<String>();
        s.add("Bär");
        s.add("Aal");
        Iterator<String> iter = s.iterator();
        assertEquals("Aal", iter.next());
        assertEquals("Bär", iter.next());
    }
}
```