Beispiel Konto-Klasse

- Die Klasse Konto ist so zu erweitern, dass man
  - ⇨ beliebige Objekte mit vorgegebenen Attributen anlegen kann und dass
  - ⇨ beim Anlegen geprüft wird, ob korrekte Werte übergeben wurden.
  - ⇨ Ebenso sind die Methoden um Prüfungen zu ergänzen.

Konstruktoren

- Klassen benötigen sogenannte **Konstruktoren**, um Objekte anlegen zu können.
  - ⇨ Jede Klasse hat immer mindestens einen **Konstruktor**.
  - ⇨ Ist kein Konstruktor explizit definiert, so wird vom Compiler ein parameterloser Konstruktor, der sogenannte **Standard-Konstruktor** (default constructor) erzeugt.
  - ⇨ Konstruktoren heißen genau wie die Klasse und unterscheiden sich durch Anzahl und Typ der Übergebenen Parameter.

```
public Konto () {
}
```

  - ⇨ Ist ein anderer Konstruktor vorhanden, so wird nicht automatisch ein Standard-Konstruktor angelegt.

Konstruktoren mit Parametern

- Eine Klasse kann theoretisch beliebig viele Konstruktoren besitzen.
- Konstruktor mit Parametern:

```
public Konto(int kontonr, String inhaber,
             double kontostand) {
    this.kontonr = kontonr;
    this.inhaber = inhaber;
    this.kontostand = kontostand;
}
```

  - ⇨ Damit man Attribut von Parameter unterscheiden kann, muss beim Attribut der Präfix **this** vor den Attributnamen geschrieben werden.
  - ⇨ **this** ist intern eine Referenz auf das aktuelle Objekt, in dem man sich befindet.

Konstruktoren gegenseitig aufrufen

- Wir können nun noch einen zweiten Konstruktor definieren, bei dem nur zwei Parameter übergeben werden.

```
public Konto(int kontonr, String inhaber) {
    this(kontonr, inhaber, 0.0);
}
```

  - ⇨ **this** bedeutet hier, dass als erste Aktion in diesem Konstruktor ein anderer Konstruktor derselben Klasse aufgerufen wird.
  - ⇨ Das hat den Vorteil, dass nur ein Konstruktor alle Initialisierungen vornehmen muss und die anderen sich darauf beziehen können.

Objekte anlegen

- Der **new-Operator** ruft automatisch den passenden Konstruktor der Klasse auf, um ein Objekt anlegen zu können.

```
// 1. Konstruktor
Konto k1 = new Konto(4711, "Meier, Hans", 1000.0);

// 2. Konstruktor
Konto k2 = new Konto(4712, "Müller, Fritz");
```


Objekte ausgeben

- Wie geht man am Besten vor, wenn ein Objekt ausgegeben werden soll?
- Dazu gibt es mehrere Lösungsmöglichkeiten:
  1. Eine Ausgabemethode in die Klasse Konto einbauen.
  2. Für alle Attribute get-Methoden schreiben und diese bei Ausgaben verwenden.
  3. Einfach mal ausprobieren, ob man das Objekt direkt ausgeben kann.

```
public class KontoTst {
    public static void main(String[] args) {
        Konto k1 = new Konto(4711, "Meier", 2000.0);
        k1.einzahlen(1000.0);
        System.out.println("Konto1: " + k1);
    }
}
```

Die toString-Methode

- Die **toString**- Methode ist eine Methode, die von der Klasse **java.lang.Object** geerbt wird (mehr dazu später).
- Damit besitzt jede Klasse automatisch eine toString-Methode.
- Aufgabe: textuelle Aufbereitung des Objektinhaltes.
- Standardimplementierung: Name der Klasse + Adresse des Objektes im Hauptspeicher.

```
Konto1: Konto@54520eb
```


Die toString-Methode

- Redefinition der **toString**-Methode in der Beispielklasse Konto:

```
public String toString() {
    return "Kontonr : " + kontonr +
           ", Inhaber: " + inhaber +
           ", Kontostand: " + kontostand;
}
```

- Ausgabe im Testprogramm:

```
Konto1: Kontonr : 4711, Inhaber: Meier, Kontostand: 3000.0
```

Prüfungen ergänzen

- Problem: Was passiert, wenn wir dem Konstruktor falsche Werte übergeben, z. B. eine negative Kontonummer oder ein Leerstring als Namen?
- Frage: Wie können wir verhindern, dass ein Objekt mit falschen Attributwerten angelegt wird?
- Ein ähnliches Problem stellt sich, wenn einer Methode ein falscher Wert übergeben wird.
  - ⇨ Wie reagiert man, wenn der Methode `einzahlen()` der Wert `-1000.0` übergeben wird?
  - ⇨ Macht es Sinn eine Fehlermeldung auf die Standardausgabe auszugeben?

Prüfungen im Konstruktor

```
public Konto(int kontonr, String inhaber,
             double kontostand){
    if (kontonr > 999 && kontonr < 10000) {
        this.kontonr = kontonr;
        this.inhaber = inhaber;
        this.kontostand = kontostand;
    } else
        System.out.println("Kontonummer nicht 4-stellig");
    }
```

- Warum ist diese Prüfung unsinnig?
- Wohin geht denn diese Ausgabe?
- Was ist mit dem Objekt, wird es trotzdem angelegt?

Konzept der Zusicherungen

- Die Java Version 1.4 hat das Konzept der **Zusicherungen** (Assertions) als Sprachelement aufgenommen.
- Eine **Zusicherung** ist eine Aussage über das laufende Programm, die der Programmierer für erfüllt hält.
- Ist diese Aussage wider Erwarten nicht erfüllt, wird eine sogenannte **Ausnahme** ausgelöst.

Die assert-Anweisung

Syntax:

```
assert Boolescher-Ausdruck : [Ausdruck];
```

Wirkungsweise:

- Der boolesche Ausdruck wird ausgewertet
- Ist er `true`, so wird mit der nächsten Anweisung fortgefahren
- Ist er `false`, so wird die Ausnahme `AssertionError` ausgelöst. Der Programmablauf wird abgebrochen, es sei denn es gibt eine sogenannte Ausnahmebehandlung (dazu später)

Anwendung im Konstruktor

```
public Konto(int kontonr, String inhaber,
             double kontostand) {
    assert kontonr > 999 && kontonr < 10000
           : "Kontonr muss 4-stellig sein";
    this.kontonr = kontonr;
    this.inhaber = inhaber;
    this.kontostand = kontostand;
}
```

- Ablauf bei einer gültigen Parameterübergabe:
  - ⇨ das Programm läuft nach assert normal weiter
- Ablauf bei einer ungültigen Parameterübergabe, z.B. einer 3-stelligen Zahl:
  - ⇨ assert löst eine Ausnahme **AssertionError** aus mit der angegebenen Fehlermeldung.
  - ⇨ folgende Anweisungen werden übersprungen.
  - ⇨ Das bereits allokierte Objekt wird wieder freigegeben.

Alternative zu assert

- `assert` ist leider für Fehlerbehandlung zu unflexibel. Deshalb werden wir Zusicherungen direkt mit Hilfe der Ausnahmebehandlung programmieren
- Statt

```
assert kontonr > 999 && kontonr < 10000
       : "Kontonr muss 4-stellig sein";
```

heißt es dann

```
if (kontonr < 1000 || kontonr >= 10000)
    throw new IllegalArgumentException(
              "Kontonr muss 4-stellig sein");
```

Auswerfen einer Ausnahme

- `throw` *Ausnahmeobjekt*
  - ⇨ Auswerfen einer Ausnahme.
  - ⇨ Die Programmsteuerung wird an die nächste zuständige Ausnahmebehandlung übergeben, sofern eine vorhanden ist.
  - ⇨ Ist keine Ausnahmebehandlung vorhanden, so wird das Programm abgebrochen.
- `IllegalArgumentException`
  - ⇨ Spezielle Exceptionklasse:
    "*Thrown to indicate that a method has been passed an illegal or inappropriate argument.*"

Anwendung im Konstruktor

```
public Konto(int kontonr, String inhaber,
             double kontostand) {
   if (kontonr < 1000 || kontonr >= 10000)
       throw new IllegalArgumentException
                 ("Kontonr muss 4-stellig sein");
   if (inhaber == null || inhaber.isEmpty())
       throw new IllegalArgumentException
                 ("Inhaber darf nicht leer sein");
   if (kontostand < 0)
       throw new IllegalArgumentException
                 ("Kontostand muss >= 0.0 sein");
   this.kontonr = kontonr;
   this.inhaber = inhaber;
   this.kontostand = kontostand;
}
```

Ist das nicht ein wenig umständlich?

check-Methode

```
public static void check(boolean bedingung, String msg) {
    if (!bedingung)
        throw new IllegalArgumentException(msg);
}
```

- Übergebe eine Bedingung und werfe eine **`IllegalArgumentException`**, wenn die Bedingung nicht erfüllt ist.
- Der Zusatz **`static`** bedeutet, dass die Methode eine sogenannte Klassenmethode ist.

Anwendung im Konstruktor

```
public Konto(int kontonr, String inhaber,
             double kontostand) {
    check(kontonr >= 1000 && kontonr <= 10000,
          "Kontonr muss 4-stellig sein");
    check(inhaber != null && !inhaber.isEmpty(),
          "Inhaber darf nicht leer sein");
    check(kontostand >= 0,
          "Kontostand muss >= 0.0 sein");
    this.kontonr = kontonr;
    this.inhaber = inhaber;
    this.kontostand = kontostand;
}
```

Und was ist mit der Methode `setInhaber`?
Müsste da nicht auch geprüft werden?

Verbesserte Anwendung im Konstruktor

```
public Konto(int kontonr, String inhaber,
             double kontostand) {
    check(kontonr >= 1000 && kontonr <= 10000,
          "Kontonr muss 4-stellig sein");
    check(kontostand >= 0,
          "Kontostand muss >= 0.0 sein");
    this.kontonr = kontonr;
    setInhaber(inhaber);
    this.kontostand = kontostand;
}

...

public void setInhaber(String inhaber) {
    check (inhaber != null && !inhaber.isEmpty(),
           "Inhaber darf nicht leer sein");
    this.inhaber = inhaber;
}
```


Methoden

- Es gibt **(Objekt-)Methoden**, also Methoden, die über die this-Referenz auf Objektattribute zugreifen können wie z. B. *einzahlen* und *abheben* in der Klasse Konto.
  - ⇨ Aufruf
    `Objektname.Methodenname (Par1, Par2, ...)`
- Es gibt **Klassen-Methoden**, zu erkennen an dem Zusatz **static**.
  - ⇨ Kein Zugriff auf Objektattribute
  - ⇨ Nur Zugriff auf Klassenattribute (Zusatz static).
  - ⇨ Aufruf
    `Klassenname.Methodenname (Par1, Par2, ...)`