Selektoren

- Üblicherweise erlaubt man keinen (direkten) öffentlichen (public) Zugang zu Objektvariablen
  - mehr Kontrolle über Zugang
  - interne Details besser verborgen
- Statt dessen:
  Zugriff auf privates Feld Selector-Methode:

```
private TypeX x;

public TypeX getX() { return x; }
```

Selektoren

- **Beispiel: *read only access***
  - Kein Schreibzugriff auf Feld id von außen

```
public class Identity {
    static int counter = 1;
    private int id;
    public Identity() { id = counter++;}
    int getId() { return id; }
}
```

Selektoren: Beispiel

- **Beispiel:** Konsistenzprüfung beim Schreiben
  - Bisher kann Nutzer der Klasse Time Zuweisung wie

    ```
    t.sec=100;
    ```

    vornehmen
  - Deshalb tick() defensiv programmiert (so dass auch dann keine Sekunden verloren gehen)

Selektoren: Beispiel

```
class Time {
  int sec=0;      // seconds, 0 <= sec < 60
  int min=0;      // minutes, 0 <= min < 60
  int hrs=0;      // hours, 0 <= hrs

  void tick() {
    sec++;
    if ( sec >= 60 ) {
          sec -=  60;
          min++;
          if ( min >= 60 ) {
                min -= 60;
                hrs++;
          }
    }
  }
}
```

Selektoren: Beispiel

- Selektor setSec(byte seconds) kann aber sicherstellen, dass an sec nur natürliche Zahlen kleiner als 60 zugewiesen werden

```
class Time {

  private int sec=0;      // seconds, 0 <= sec < 60
  ...

  public setSec(byte seconds) {
     if (seconds >= 0 && seconds <= 60)
          sec = seconds;
     else
          // Fehlermeldung (s. nächster Abschnitt)
  }
}
```

Exceptions: Objekte für Ausnahmen

Objekte für Ausnahmen (exceptions)

- Spezielle Objekte für:
  - Ausnahmefall (exception)
  - Fehler (error)
- Wenn Ausnahme-/Fehlerfall eintritt, wird entsprechendes Objekt „geworfen“
- Beispiel für solche Klasse:

```
class TimeException extends Exception { ... }
```

Objekte für Ausnahmen (exceptions)

- Verwendung („werfen“):

```
public setSec(byte seconds)  throws TimeException  {
      if (seconds >= 0 && seconds <= 60)
            sec = seconds;
      else
            throw new TimeException();
}
```

Objekte für Ausnahmen (exceptions)

- Ausnahmeobjekt kann Informationen über Fehler speichern:

```
class TimeException extends Exception {
        int wrongSecNumber;
        TimeException(int seconds)  {
                wrongSecNumber = seconds;
        }
}
```

Objekte für Ausnahmen (exceptions)

- Verwendung dann:

```
public setSec(byte seconds)  throws TimeException  {
   if (seconds >= 0 && seconds <= 60)
      sec = seconds;
   else
      throw new TimeException(sec);
}
```

Objekte für Ausnahmen (exceptions)

- Ausnahmen/Fehler können „gefangen“ werden

```
try {
  setSec(s);
}
catch(TimeException te) {
  System.out.println(
     te.s + „ are too many seconds“
  )
}
```

Der Rahmen try-catch-finally

Allgemein:

```
try {
  Anweisungen
}
catch ( Exception1 e1 ) {
  Ausnahmebehandlung1
}
catch ( Exception2 e2 ) {
  Ausnahmebehandlung2
}
. . .
finally { Aufräumarbeiten }
```

Enthält (sinnvollerweise) Anweisungen der Form

```
if ( Ausnahmebedingung )
   throw new SomeException();
```

Wird immer ausgeführt, bevor der try-catch-finally-Rahmen auf irgendeine Art verlassen werden kann (Normal- und Ausnahmefall)

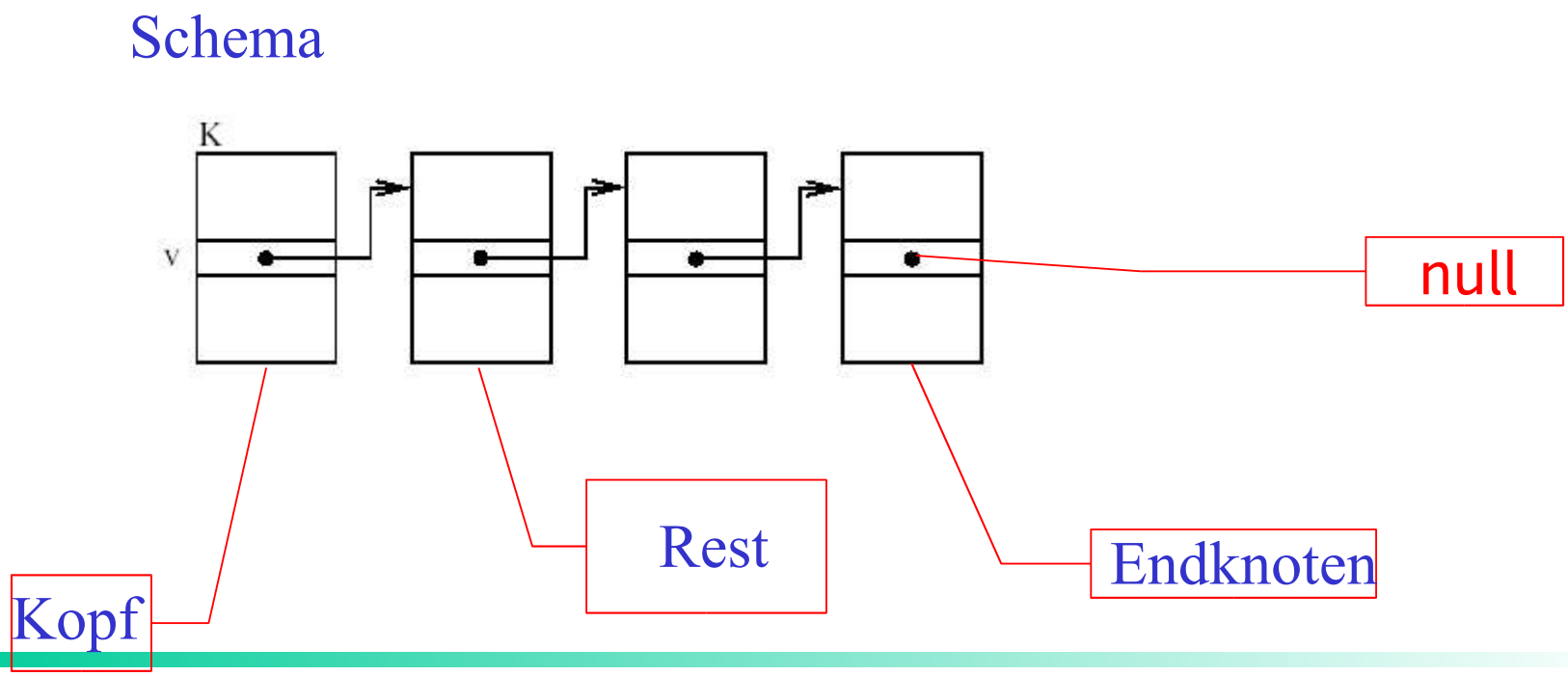
Geprüfte und ungeprüfte Ausnahmenklassen

- Unterklassen von Error oder RuntimeException
  - müssen nicht im Methodenkopf mit „throws“ deklariert werden
  - Beispiele
    - NullPointerException
    - IndexOutOfBoundsException
    - OutOfMemoryError
    - StackOverflowError

Listen

Listen (linked lists)

- Falls Objekte eines Typs K eine Variable vom selben Typ K als Feld enthalten,
  kann man K-Objekte zu einer **Liste** verknüpfen

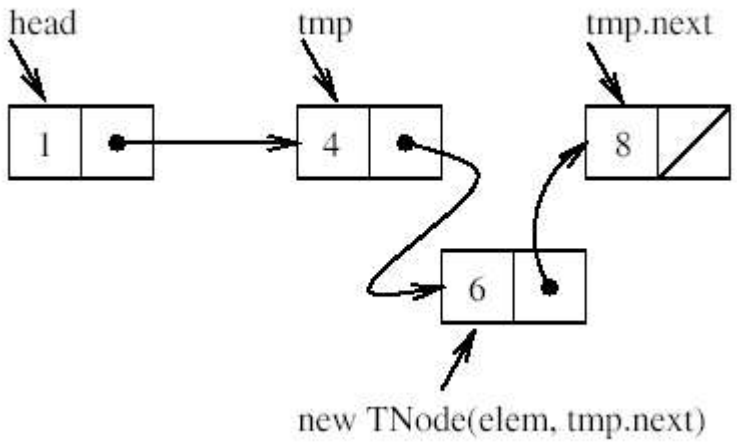


Listen (linked lists)

- „Verzeigerte“ Listen besonders geeignet zur Repräsentation von dynamisch wachsenden und schrumpfenden Mengen von Objekten
- Typische Operationen:
  - Neues Element **einflechten**, insbesondere am
    - Anfang oder
    - Ende
  - Element **löschen**
  - Zwei Listen **verknüpfen**

Typische Operationen auf Listen

- Einfügen in geordnete Liste



Container-Klassen für Listen

- Containerklassen für Listen sinnvoll
  - Enthalten Referenz auf Kopf der Liste
  - Sowie die Methoden
- Objekte der Listenklasse enthalten dann nur
  - Datenfeld
  - Referenz auf den nächsten Knoten

Listen versus Arrays

- Listen
  - Vorteile
    - Einfügen neuer Elemente leicht möglich
    - Löschen leicht möglich
  - Nachteile
    - „Durchhangeln“ durch viele Elemente der Liste, um ein spezielles zu erreichen
    - Mehr Speicherbedarf (Ein Objekt pro Listenelement)
- Arrays
  - Nachteile
    - Einfügen neuer Elemente erfordert „umkopieren“
    - Löschen von Elementen erfordert ebenfalls ein „umkopieren“
  - Vorteile
    - „Wahlfreier“ Direktzugriff
    - Speicherbedarf nur für Daten