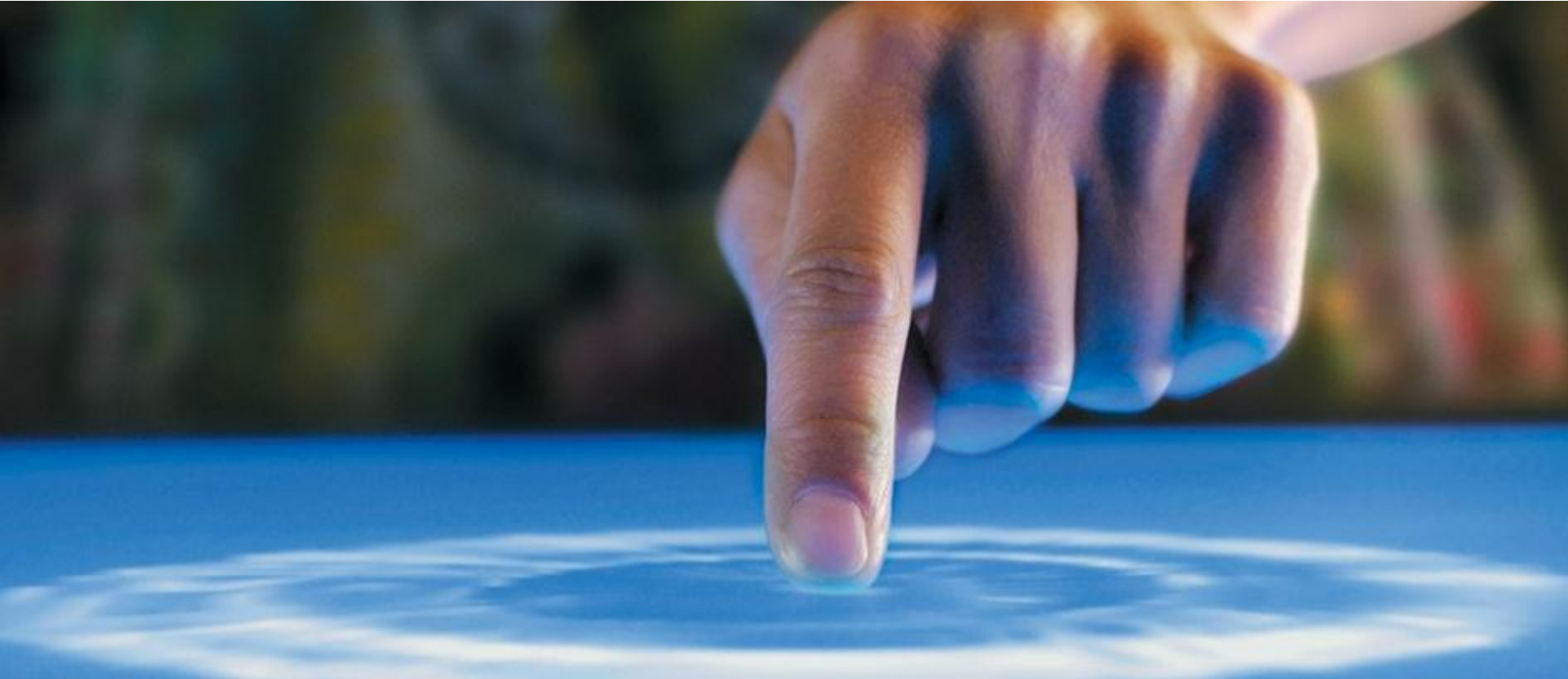




ulm university universität
uulm



Enums, Interfaces, Generics und Threads

Philipp Güttler | 18.11.2008 | Progwerkstatt
Letzte Änderung: 09.12.2009

Enum Typen

- komplexer Datentyp, der festen Menge von Konstanten enthält
- Information der Reihenfolge der Konstanten
- erbt implizit von `java.lang.Enum`

```
enum Tag {  
  
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,  
  
    FREITAG, SAMSTAG, SONNTAG  
  
}
```

- Compiler fügt automatisch Methoden hinzu, z.B. `ordinal()`

Erweitertes Beispiel

```
public enum Planet {  
  
    EARTH      (5.976e+24, 6.37814e6),  
    MARS      (6.421e+23, 3.3972e6);  
  
    private final double masse;  
    private final double radius;  
  
    Planet(double masse, double radius) {  
        this.masse = masse;  
        this.radius = radius;  
    }  
    double gravitationAufOberflaeche() {  
        return G * masse / (radius * radius);  
    }  
    double gewichtAufOberflaeche(double otherMass) {  
        return otherMass * surfaceGravity();  
    }  
}
```

Interfaces – Definition

- formale Eckpunkte, die erfüllt werden müssen
- definiert, wie zwei Objekte miteinander kommunizieren
- definiert *was* implementiert werden soll, nicht *wie*

```
public interface Vergleichbar extends Verwandt {  
    public final double umrechnungsfaktor = 2d;  
    public boolean istGroesser(Vergleichbar v);  
}
```

```
public class Rechteck implements Berechenbar, Vergleichbar {  
    ...  
    public boolean istGroesser(Vergleichbar v) {...}  
}
```

Interfaces – Definition

- formale Eckpunkte, die erfüllt werden müssen
- definiert, wie zwei Objekte miteinander kommunizieren
- definiert *was* implementiert werden soll, nicht *wie*

Bei Änderungen im Interface müssen auch alle implementierenden Klassen geändert werden!

Generics - Definition

- Typecast verhindern
- Typsicherheit beim Kompilieren überprüfen
- Typbegrenzung für allgemeine Datenstrukturen

Vorteile der generische Typen

```
public class Box {  
  
    private Object element;  
  
    public void add(Object e) {  
        element = e;  
    }  
    public Object get() {  
        return element;  
    }  
}
```

```
1 public class BoxDemo {  
2     public static void main(String[] args) {  
        ...  
100    Box integerBox = new Box();  
        ...  
250    integerBox.add("10");  
        ...  
550    Integer someInteger =  
        (Integer) integerBox.get();  
    }  
}
```

Vorteile der generische Typen

```
public class Box {  
  
    private Object element;  
  
    public void add(Object e) {  
        element = e;  
    }  
    public Object get() {  
        return element;  
    }  
}
```

```
1 public class BoxDemo {  
2     public static void main(String[] args) {  
        ...  
100     Box integerBox = new Box();  
        ...  
250     integerBox.add("10");  
        ...  
550     Integer someInteger =  
        (Integer) integerBox.get();  
    }  
}
```

- `java.lang.ClassCastException`

Vorteile der generische Typen

```
public class Box<T> {  
    private T element;  
  
    public void add(T e) {  
        element = e;  
    }  
    public T get() {  
        return element;  
    }  
}
```

```
1 public class BoxDemo {  
2     public static void main(String[] args) {  
        ...  
100         Box<Integer> integerBox = new  
            Box<Integer>();  
        ...  
250         integerBox.add("10");  
        ...  
550         Integer someInteger = integerBox.get();  
            }  
        }  
}
```

Vorteile der generische Typen

```
public class Box<T> {  
    private T element;  
  
    public void add(T e) {  
        element = e;  
    }  
    public T get() {  
        return element;  
    }  
}
```

```
1 public class BoxDemo {  
2     public static void main(String[] args) {  
        ...  
100         Box<Integer> integerBox = new  
            Box<Integer>();  
        ...  
250         integerBox.add("10");  
        ...  
550         Integer someInteger = integerBox.get();  
    }  
}
```

- Compile Error
- Kein Typecast mehr

Generische Methoden

```
public static <T> void fillBoxes(T arg, List<Box<T>> listOfBoxes) {  
    for ( Box<T> box : boxlist)  
        box.add(arg);  
}
```

```
Box.<Integer>fillBoxes(new Integer(10), listOfBoxes);
```

```
Box.fillBoxes(new Integer(10), listOfBoxes);
```

```
public static <T> void copy(List<T> dest, List<? extends T> src)
```

```
public static <T, S extends T> void copy(List<T> dest, List<S> src)
```

Unterschiedliche Typparameter

- Mehrfache Typisierung
 - `<A, B, C, ... >`
- Wildcards
 - Beliebiger Typ `<?>`
 - Upper Bounds `<? extends X>`
 - Lower Bounds `<? super X>`

Beispiel Typenhierarchie

- **interface** Rechteck **extends** GeoObjekt
- `List<Rechteck> recList, List<GeoObjekt> geoList`

- `geoList = recList;`
- `recList = geoList;`

?

Concurrency/Nebenläufigkeit erklärt:

"Nebenläufigkeit liegt vor, wenn mehrere Ereignisse in keiner kausalen Beziehung zueinander stehen, sich also nicht beeinflussen." wikipedia.de

- *Simultane* (gleichzeitige) Abläufe bei Web-Videoplayern
 - Quelle runterladen
 - Fragmente/Teile dekodieren
 - Video/Audio abspielen
 - Anzeige aktualisieren
 - Benutzereingaben berücksichtigen

Prozesse

- meist synonym zu Anwendung, Programm
- werden in einer "Ausführungsumgebung" ausgeführt
- enthält Informationen, wie Programmcode, Speicher und Betriebsmittel des OS

Threads (Aktivitätenträger)

- haben eigene Ausführungsumgebung innerhalb des Prozesses
- verwenden Betriebsmittel des Prozesses
- Concurrency in Java mit Threads realisiert
- ab Java5: `java.util.concurrent` (Sermaphoren, ThreadPools, ...)

Java-Programme und Threads

- Java-Programm beginnt mit einem *Main-Thread*
- keine *User-Threads* » Programm beendet
- *Daemon-Threads* laufen im Hintergrund
- erst wenn kein User-Thread mehr läuft, werden die Daemon-Threads beendet
- Threads können nur einmal gestartet werden!

Erzeugung durch Vererbung

1. Klasse **extends** `java.lang.Thread`
2. Methode `run()` überschreiben
3. Klasse instanziiieren
4. Methode `start()` des Threads aufrufen

+ einfache Verwendung

+ Überschreiben von Methoden der Klasse Thread möglich

Erzeugung durch Implementierung

1. Klasse **implements** `java.lang.Runnable`
2. Methode `run()` implementieren
3. Klasse instanziiieren
4. **new** `Thread(Instanz)` erzeugen
5. Methode `start()` des Threads aufrufen

6. + allgemein nutzbar, flexibler
7. + Vererbung weiterhin nutzbar
8. + Thread wird vom eigentlichen Objekt gekapselt

Synchronisierung von Aktivitäten

Warum müssen gleichzeitige Zugriffe unterbunden werden?

- Threads kommunizieren durch Variablenzugriffe und Objektreferenzen
- evtl. Überschneidung bei Abarbeitung von mehreren Prozessschritten

```
class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
}
```

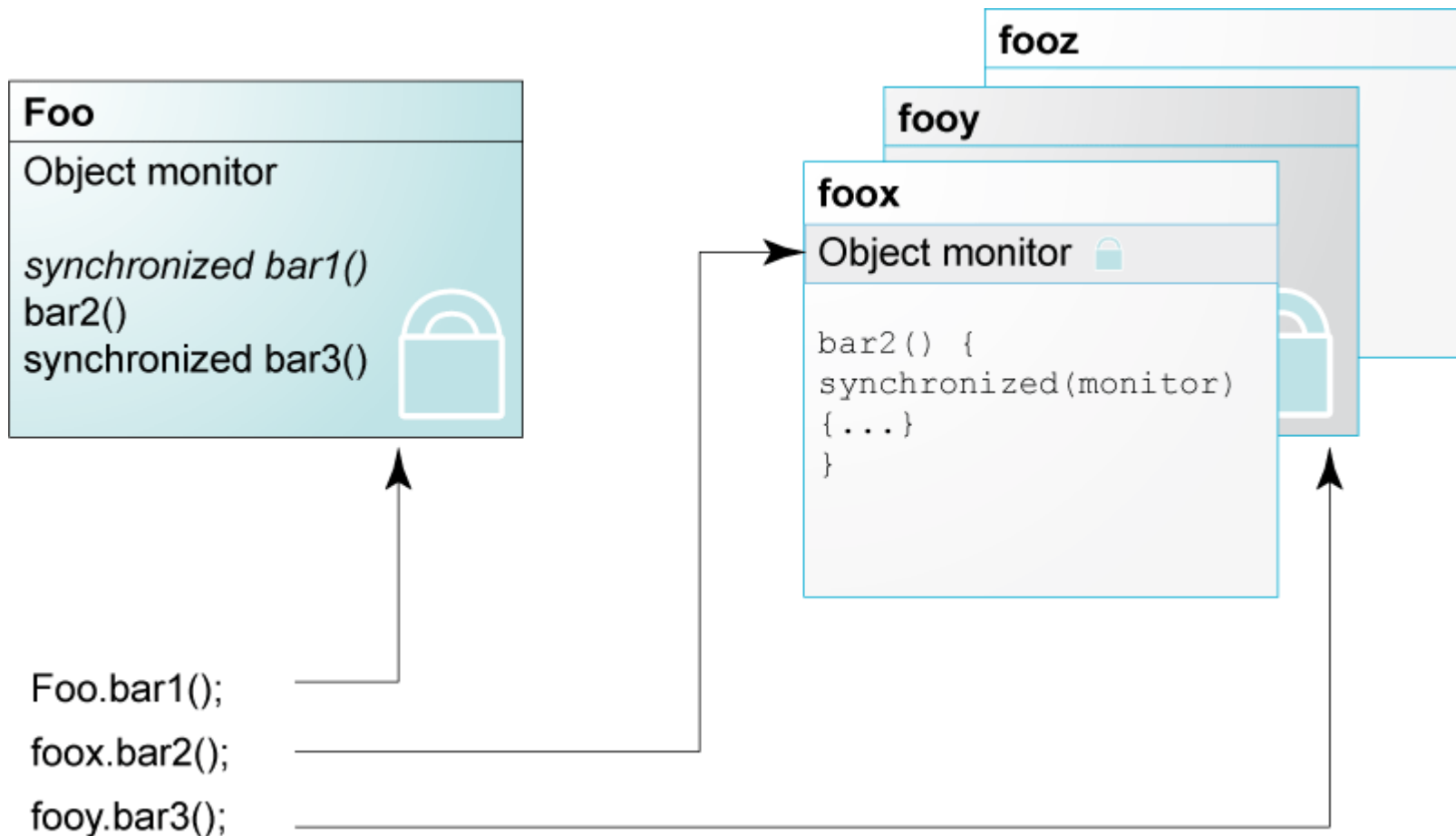
Synchronisierung von Aktivitäten

Wie kann eine Ressource vor gleichzeitigem Zugriff geschützt werden?

- **synchronized**
 - sinnvoll für Abfolge von Operationen
 - nicht auf Konstruktoren anwendbar
 - Sperren von Bereichen mit Monitorobjekt
- **volatile**
- `java.util.concurrent.atomic`

Synchronisierung von Aktivitäten

Was passiert beim Zugriff auf einen **synchronized**-Bereich?



Koordinierung von Aktivitäten

Wie kann der Zugriff Threads aktiv zugeteilt werden?

- `java.lang.Object.wait([[ms], [ns]])`
- `java.lang.Object.notify()`
- `java.lang.Object.notifyAll()`

- `java.lang.Thread.sleep([[ms], [ns]])`
- `java.lang.Thread.join()`
- `java.lang.Thread.interrupt();`

Beispiel: Guarded Blocks

```
public synchronized guardedExample() {  
    while(!condition) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    ...  
}
```

```
public synchronized notifyCondition() {  
    condition = true;  
    notifyAll();  
}
```

Probleme der Koordinierung

- *Deadlocks*
 - zwei Threads warten auf die Freigabe einer Sperre, die von dem jeweils Anderen gehalten wird
- *Verhungernde Threads*
 - ein Thread kommt nicht zur Ausführung, weil andere Threads sehr lange bei Verarbeitung brauchen
- *Livelocks*
 - Threads sind voneinander abhängig, rufen sich gegenseitig auf, ohne bei der Verarbeitung voran zu kommen

Ausblick `java.util.concurrent`

- *Executors*
 - Kapselung von Thread-Erzeugung und Anwendung
- *Thread Pools*
 - performante Erzeugung und Verwaltung von Threads
 - Fixed Thread Pool: bestimmte Anzahl von Threads (Bsp. Webserver)
 - Cached Thread Pool: für kurzlebige Threads
 - Single Thread Pool: nur jeweils ein Thread pro Pool
- *Semaphore*
 - Verwaltung von bestimmter Anzahl an Ressourcenzugriffen
- *ConcurrentCollections*
 - BlockingQueue, ConcurrentMap, ...