

Parallele Programmierung



9. Thread-sichere Container-Datenstrukturen

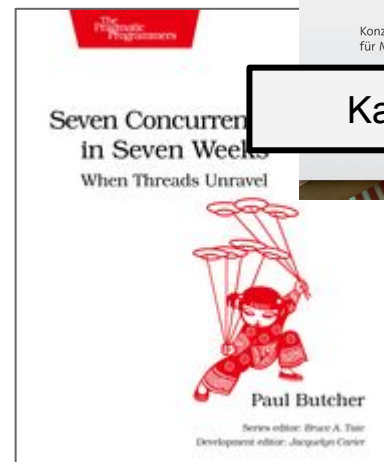
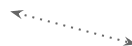
Ankündigungen

- Testergebnisse
- Pflichtaufgabe
- Lehrveranstaltungsevaluation

Überblick

Folie mit
Anmerkungen

- Thread-Sicherheit von Containern
 - Vector, Stack, HashTable und Dictionary
 - `java.util.Collection`-Implementierungen
 - `java.util.Collections.synchronizedXXX`
 - speziell: iterieren
- Experimentaldesign
- Hand-Over-Hand Locking bei verketteten Listen



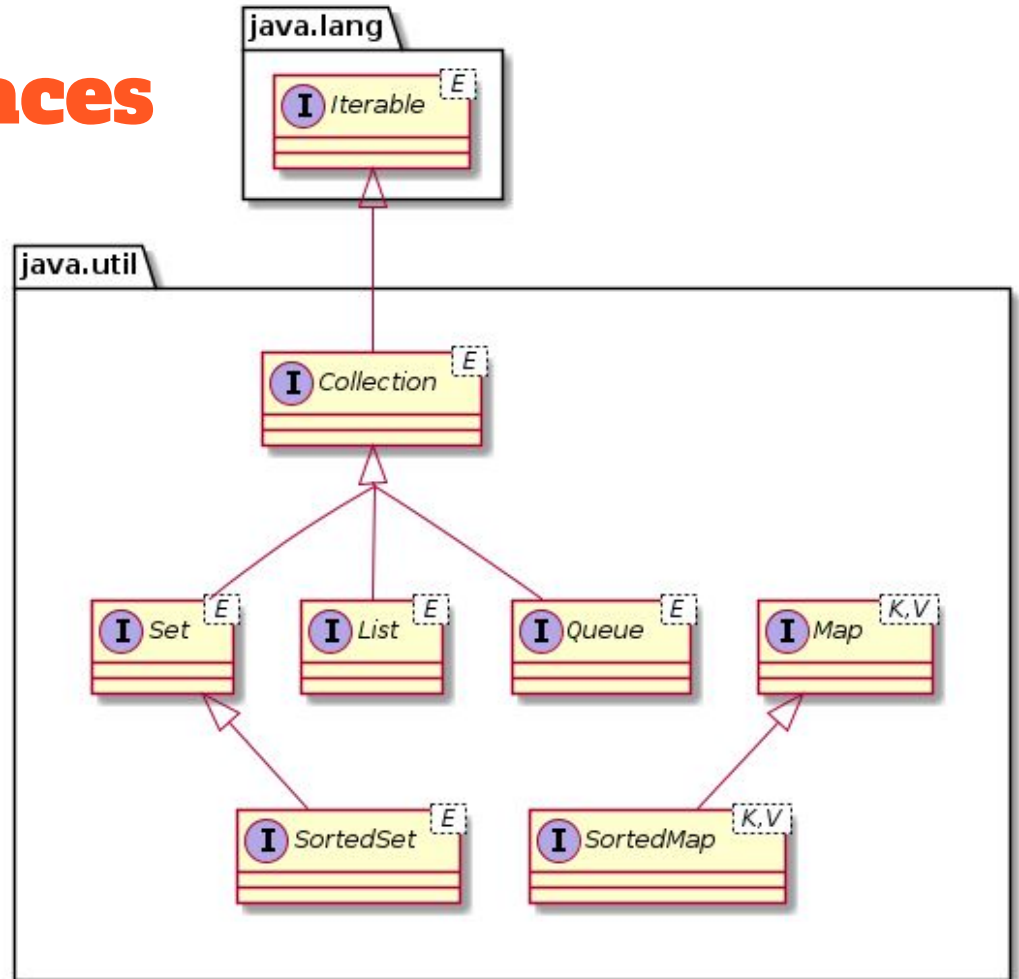
Container- Typen

Threadsicherheit von Containern

- Vector, Stack, HashTable und Dictionary
 - Alle öffentlichen Methoden sind synchronized.
 - Locking ineffizient in Single-Thread-Umgebungen
 - außerdem gibt es effizientere Methoden (ReadWriteLock)
- `java.util.Collection`-Implementierungen
 - sind deshalb nicht synchronisiert
 - Stattdessen gibt es *Wrapper-Klassen*, die die nicht Thread-sicheren `Collection`-Implementierungen kapseln:

```
List<Person> listUnsafe = new ArrayList<>();  
List<Person> listSafe = Collections.synchronizedList(listUnsafe);
```

Collection Interfaces

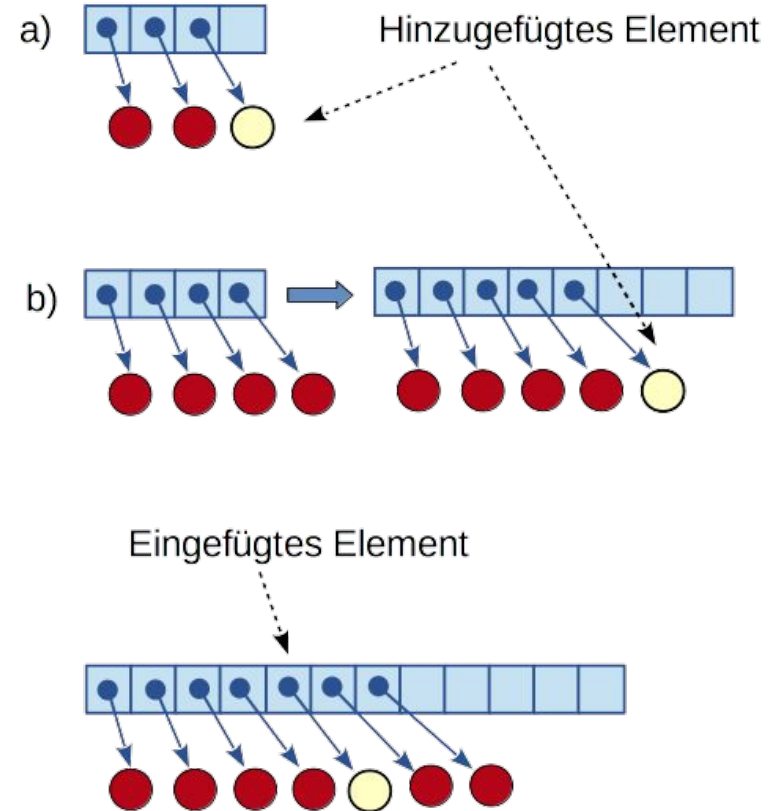


Listenimplementierungen

bei ArrayList:

am Ende einfügen sehr leicht ($O(1)$),
außer, wenn Array bereits voll ist. Dann
muss mehr Speicher alloziert werden
und die Elemente umkopiert werden
($O(N)$)

in der Mitte => alle folgenden müssen
verschoben werden ($O(n)$)



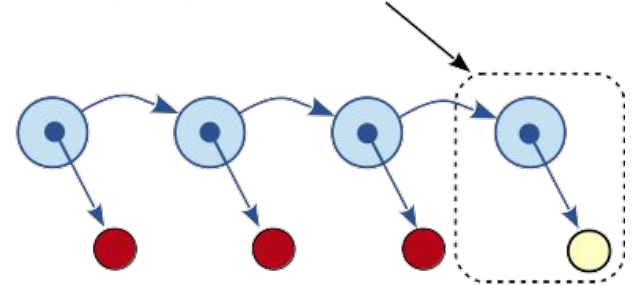
Listenimplementierungen

bei LinkedList:

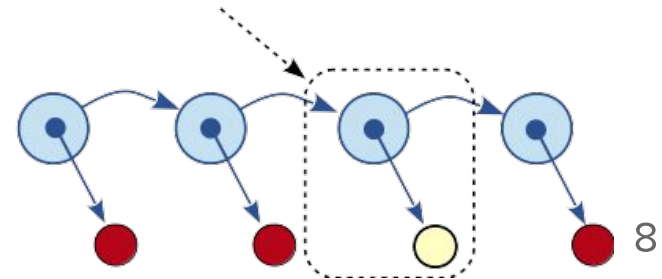
am Ende einfügen erfordert Iterieren über die ganze Liste ($O(N)$),

in der Mitte => Iterieren bis dorthin, dann einfügen ($O(n)$)

Hinzugefügtes Element



Eingefügtes Element



java.util.Collections. synchronizedXXX

Folie mit
Anmerkungen

Jeder Zugriff auf eine durch
synchronizedXXX erzeugte
Collection/Map wird durch
synchronized(this) geschützt.

- nicht sehr effizient
- immer die ganze Collection
(oft nicht nötig)
- keine Unterscheidung zwischen
Lesen und Ändern
- nur einzelne Methode ist
synchronized, Problem bei
Transaktion (z.B. Iterieren über
Collection)

java.util



«Factory»
Collections

-----Factory Methoden für synchronizedXXX-----

synchronizedCollection(c: Collection<T>): Collection<T>

synchronizedList(l: List<T>): List<T>

synchronizedSet(s: Set<T>): Set<T>

synchronizedSortedSet(s: SortedSet<T>): SortedSet<T>

synchronizedMap(m: Map<K, V>): Map<K, V>

synchronizedSortedMap(m: SortedMap<T>): SortedMap<T>

-----weitere Methoden ...-----

...

Laborübung (15 Minuten)

pp.09.02.synchronizedWrapper

Thread-sicheres Iterieren

Iterieren

```
for (var i = 0; i < listSafe.size(); i++) { /*...*/ }
```

- nebenläufig Element aus `listSafe` entfernen => `IndexOutOfBoundsException`
- nebenläufig Element zu `listSafe` hinzufügen => keine Iteration
- nebenläufig Element aus `listSafe` ändern => funktioniert

```
for (var p: listSafe) { /*...*/ }
```

... identisch mit...

```
var it = listSafe.iterator();
```

```
while (it.hasNext()) {
```

```
    var p = it.next();
```

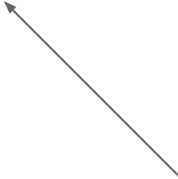
```
    // ...
```

```
}
```

- nebenläufig Element aus `syncList` ändern/entfernen/zu `listSafe` hinzufügen => `ConcurrentModificationException`

Iterieren

```
var listUnsafe = new ArrayList<Type>();  
var listSafe = Collections.synchronizedCollection(listUnsafe);  
synchronized (listSafe) {  
    for (var e : listSafe) {  
        Main.foo(e);  
    }  
}
```



ebenso:
synchronizedList,
synchronizedSet,
synchronizedMap,
synchronizedSortedMap,
synchronizedSortedSet

Durch `synchronizedCollection` ist nur jeder einzelne isolierte Zugriff geschützt. Beim Iterieren muss (i.A.) ein äußerer Lock benutzt werden.

Über HashMap iterieren

```
var mapUnsafe = new HashMap<KeyType, ValType>();  
var mapSafe = Collections.synchronizedMap(mapUnsafe);  
var setOfKeys = mapSafe.keySet();
```

```
synchronized (mapSafe) {  
    for (var k : setOfKeys) {  
        Main.foo(k);  
    }  
}
```

Beim Iterieren über eine Map muss die gesamte Datenstruktur gesperrt werden, da sonst während des Iterierens eine `ConcurrentModificationException` geworfen werden kann, wenn aus einem anderen Thread heraus die Map z.B. mit `put` modifiziert wird. Ohne `synchronized (mapSafe)` wäre nur jeder einzelne Zugriff auf `mapSafe` Thread-sicher, nicht die Gesamttransaktion (Iterieren über alle Elemente).

Experimental- design

Experimente/Experimental-design

Folie mit
Anmerkungen

Am Anfang steht eine Hypothese über die Wirkzusammenhänge. Man postuliert also eine Abhängigkeit zwischen Elementen einer Situation.

unabhängige Variable (UV): Einflussgröße der Situation, die mehrere Ausprägungen hat.

abhängige Variable (AV): Messgröße, die man beobachten kann und von der man Änderungen erwartet, wenn UV variiert werden.

Experiment: planmäßige Variation der UV bei gleichzeitiger Beobachtung der Auswirkung auf die AV.

Aus den protokollierten Beobachtungen werden Rückschlüsse auf zu prüfende Hypothese gezogen.

Laborübung (30 Minuten)

pp.09.01.Collections

nicht prüfungsrelevant!

Hand-over-Hand Locking

Hand over Hand Lock

Bei einer verketteten Liste muss man bei einigen Operationen durch die Liste iterieren. Dafür muss man auf die Elemente der Liste lesend zugreifen.

Es ist nicht erforderlich, dass die gesamte Liste dafür gelockt wird.

1. Stattdessen wird immer nur das eine Element gelockt, das gerade untersucht wird (1).
2. Beim Schritt zum nächsten Element wird kurzfristig das nächste Element mitgelockt (1,2),
3. dann kann der vorige Lock gelöst werden (2),
4. Beim Schritt zum nächsten Element wird kurzfristig das nächste Element mitgelockt (2,3),
5. dann kann der vorige Lock gelöst werden (3),
6. Beim Schritt zum nächsten Element wird kurzfristig das nächste Element mitgelockt (3,4). Nun ist die richtige Position in der Liste erreicht und das neue Element kann zwischen (3,4) eingefügt werden.

