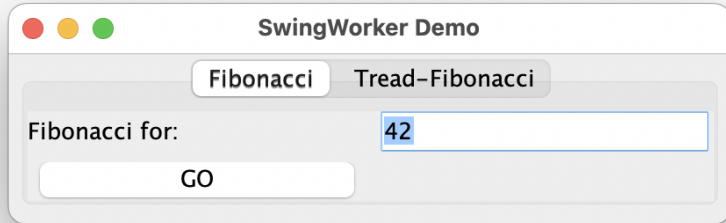


Einführung in die nebenläufige Programmierung mit Threads

Carsten Gips (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.



Traditionelle Programmierung

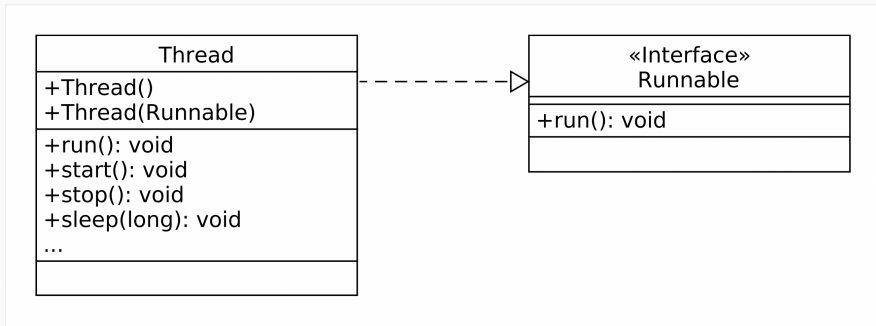
```
public class Traditional {  
    public static void main(String... args) {  
        Traditional x = new Traditional();  
  
        System.out.println("main(): vor run()");  
        x.run();  
        System.out.println("main(): nach run()");  
    }  
  
    public void run() {  
        IntStream.range(0, 10).mapToObj(i -> "in run()").forEach(System.out::println);  
    }  
}
```

Nebenläufige Programmierung

```
public class Threaded extends Thread {  
    public static void main(String... args) {  
        Threaded x = new Threaded();  
  
        System.out.println("main(): vor run()");  
        x.start();  
        System.out.println("main(): nach run()");  
    }  
  
    @Override  
    public void run() {  
        IntStream.range(0, 10).mapToObj(i -> "in run()").forEach(System.out::println);  
    }  
}
```

Erzeugen von Threads

- Ableiten von `Thread` oder Implementierung von `Runnable`



- Methode `run()` implementieren, aber nicht aufrufen
- Methode `start()` aufrufen, aber (i.d.R.) nicht implementieren

Zustandsmodell von Threads (vereinfacht)

Threads können wie normale Objekte kommunizieren

- Zugriff auf (`public`) Attribute
- Aufruf von Methoden

Threads können noch mehr

- Eine Zeitlang schlafen: `Thread.sleep(<duration_ms>)`
- Prozessor abgeben und hinten in Warteschlange einreihen: `yield()`
- Andere Threads stören: `otherThreadObj.interrupt()`
- Warten auf das Ende anderer Threads: `otherThreadObj.join()`

Threads sind weitere Kontrollflussfäden, von Java-VM (oder (selten) von OS) verwaltet

- Ableiten von `Thread` oder implementieren von `Runnable`
- Methode `run` enthält den auszuführenden Code
- Starten des Threads mit `start` (nie mit `run`!)



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.