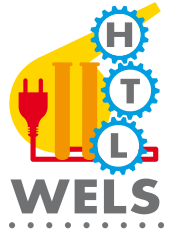
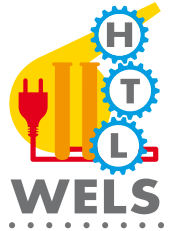
CONCURRENCY IN JAVAFX



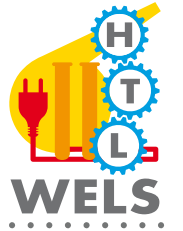
CONCURRENCY



- JavaFX Application Thread darf unter keinen Umständen blockiert werden
  - „Eingefrorene“ Anwendung
  - Ereignisbehandlung reagiert nicht mehr
- sämtliche rechenintensive Tätigkeiten müssen daher in Threads ausgelagert werden
- für JavaFX-Programme gibt es dafür eigene Klassen/Interfaces

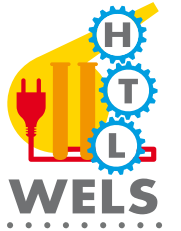
INTERFACE WORKER

CONCURRENCY - WORKER 1



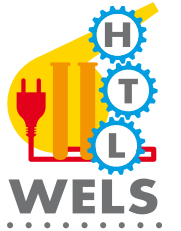
- `Interface` ***`Worker`***
  - Interface (Basis für alle Concurrent Klassen)
  - `Worker` Objekte erledigen Arbeit in Hintergrund-Thread
  - Zustände können abgefragt werden:
    - `Worker.State.READY`
    - `Worker.State.SCHEDULED`
    - `Worker.State.RUNNING`
    - `Worker.State.CANCELLED`
    - `Worker.State.SUCCEEDED`
    - `Worker.State.FAILED`

CONCURRENCY - WORKER 1



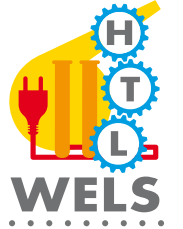
- ***Worker***
  - Zustand des Threads kann abgefragt werden:
    - `Worker.State.READY`
    - `Worker.State.SCHEDULED`
    - `Worker.State.RUNNING`
    - `Worker.State.CANCELLED`
    - `Worker.State.SUCCEEDED`
    - `Worker.State.FAILED`

CONCURRENCY – WORKER 3

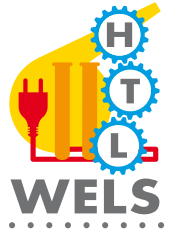


- ***Worker***
  - Fortschritt über ***Properties*** beobachtbar:
    - **totalWork**: DoubleProperty, Maximalwert für die Arbeit
    - **workDone**: DoubleProperty, Anteil an erledigter Arbeit
    - **progress**: Wert zw. 0 und 1 (prozentueller Anteil)

TASK



TASK

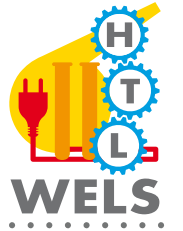


- `Klasse` ***`Task`***
  - implementiert das *`Worker`* Interface
  - Verwendung für **einmalige** Hintergrundberechnungen (abgeleitet von `FutureTask`)
  - Task implementiert `Runnable`, somit auch Start über `Executor` möglich
  - Ergebnis des Tasks mit Methode `get()`, wenn Berechnung zu Ende ist
    - Berechnung noch nicht am Ende: `get()` blockiert
  - in `call()` Methode wird Arbeit verrichtet und gegebenenfalls Properties aktualisiert

TASK

- **Fortschritt aktualisieren:**
  - in `call()` wird Methode `updateProgress` aufrufen
  - über `task.progressProperty()` kann ein Binding auf z.B. eine Progressbar realisiert werden

TASK



- **Tasks unterbrechen**
  - vgl. „interrupt“ in Threads
  - im Controller wird mit `myTask.cancel()` versucht den Task zu beenden
  - bei Tasks: in Methode „`call`“ prüfen auf **`isCancelled()`**

TASK

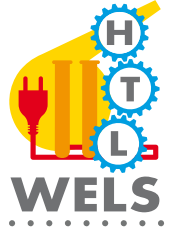
- Aktionen nach Beendigung des Tasks:

```
task.setOnSucceeded((WorkerStateEvent event) -> {
    Object value = task.getValue();
    // do anything with the result
    updateTheUI(value);
});
// setOnFailed
// setOnScheduled
// setOnCanceled
// setOnRunning
```

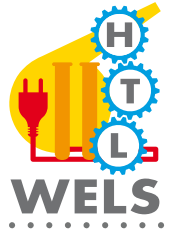
TASK

- Aufgabe: **Simple_Task**

SERVICE

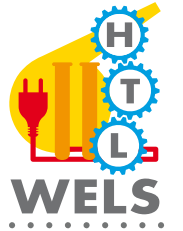


CONCURRENCY – SERVICE 1



- Klasse **`Service`**
  - verwaltet einen `Task`
  - Tasks können über Service mehrfach ausgeführt werden
  - Task ohne Service kann nur 1x ausgeführt werden!
- Klasse **`ScheduledService`**
  - führt Tasks in vorbestimmten Intervallen

CONCURRENCY – SERVICE 2



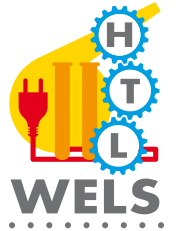
- Klasse **`Service`** - wichtige Methoden:
  - `start()` - startet den Service
  - `reset()` - resettet den Service, funktioniert aber nur, wenn Thread in finished Status ist (*SUCCEEDED*, *FAILED*, *CANCELLED*, *READY*)
  - `restart()` - laufender Thread wird gecancelt und dann neu gestartet
  - `cancel()` - canceled laufenden Thread

CONCURRENCY – SERVICE 3

- Bsp. Service: Task definieren

```
public CounterTask extends Task<Integer>{
    public CounterTask(int max) {
      this.max = max;
      updateMessage("Ready to count...");
    }
    @Override protected Integer call() throws Exception {
      updateMessage("Counting...");
      for (int i = 0; i < max; i++) {
        Thread.sleep(10);
        updateProgress(i, max);
      }
      updateMessage("READY");
      return max;
    }
}
```

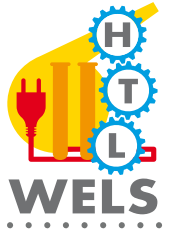
CONCURRENCY - SERVICE 4



- Service definieren:

```
public class CounterService extends Service<Integer>{
    private final int max;
    public CounterService(int max) {
      this.max = max;
    }
    @Override
    protected Task<Integer> createTask() {
      return new CounterTask(max);
    }
}
```

CONCURRENCY – TASK 5



- Aufgabe **4207_SimpleService**

SCHEDULEDSERVICE

CONCURRENCY – SCHEDULEDSERVICE 1

- **`ScheduledService`**
  - führt Tasks in vorbestimmten Intervallen wieder aus
  - Ändere im vorigen Beispiel (4207) folgende Zeile und es wird der Task immer wieder ausgeführt:

```
public class CounterService extends
ScheduledService
```

CONCURRENCY - SCHEDULEDSERVICE 2

- Verzögerung des Restarts um 2 Sekunden

```
public CounterService(int max) {
  super();
  setPeriod(Duration.seconds(2));
  this.max = max;
}
```

CONCURRENCY – SCHEDULEDSERVICE 3

- Bei `Task/Service` Wert (z.B. in GUI) an `valueProperty` binden - somit ist Wert immer aktuell

```
Label label = new Label();
label.textProperty().bind(Bindings.concat("Value: ",
    counterService.valueProperty()));
```

CONCURRENCY - SCHEDULEDSERVICE 4

- bei `ScheduledService` wird `valueProperty` regelmäßig `null` sein, da der `Service` immer wieder neu gestartet wird (und somit der Wert zurückgesetzt wird)
  - daher gibt es die Property `lastValue`

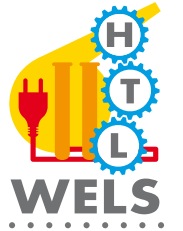
```
label.textProperty().bind(Bindings.concat("Value: ",
counterService.lastValueProperty()));
```

CONCURRENCY - SCHEDULEDSERVICE 5

- Was passiert im Fehlerfall? Server nicht erreichbar, …

```
protected Integer call() throws Exception {
  updateMessage("Counting...");
  for (int i = 0; i < max; i++) {
    Thread.sleep(10);
    updateProgress(i, max);
  }
  if (max>=3) throw new Exception("Das ist zu
kompliziert!");
   updateMessage("READY");
   return max;
  }
```

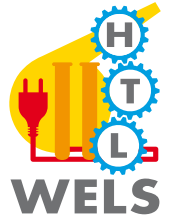
CONCURRENCY – SCHEDULEDSERVICE 6



- Abbruch bei Misserfolg
  - Service muss manuell wieder gestartet werden!

```
counterService.setRestartOnFailure(false);
counterService.start();
```

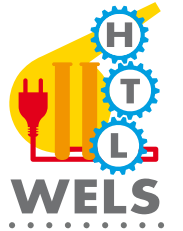
CONCURRENCY – SCHEDULEDSERVICE 7



- Festlegen, wie oft es der Service im Fehlerfall versuchen soll:

```
counterService.setRestartOnFailure(true);
counterService.setMaximumFailureCount(3);
counterService.start();
```

CONCURRENCY – SCHEDULEDSERVICE 8



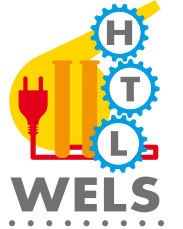
- Nach Fehler ist es meist nicht sinnvoll es sofort neu zu versuchen
- Daher unterschiedliche Strategien:
  - LOGARITHMIC_BACKOFF_STRATEGY
  - EXPONENTIAL_BACKOFF_STRATEGY
  - LINEAR_BACKOFF_STRATEGY

```
counterService.setRestartOnFailure(true);
counterService.setMaximumFailureCount(3);
counterService.setBackoffStrategy(
        ScheduledService.EXPONENTIAL_BACKOFF_STRATEGY );
counterService.start();
```

CONCURRENCY - PROGRESS ANZEIGEN

```
Task task = new Task<Void>() {
    @Override public Void call() {
        static final int max = 1000000;
        for (int i=1; i<=max; i++) {
            if (isCancelled()) {
                break;
            }
            updateProgress(i, max);
        }
        return null;
    }
};
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```

PLATFORM.RUNLATER



PLATFORM.RUNLATER

- Soll eine GUI Komponente von einem Nicht-GUI-Thread heraus modifiziert werden, so kann `Platform.runLater` verwendet werden

```
public static void runLater(Runnable runnable)
```

  - die Aufgabe wird in den GUI Thread eingereiht und frühest möglich abgearbeitet
  - kleinere Aufgaben können ebenso mit `Platform.runlater()` realisiert werden
  - größere/rechenintensivere Aufgaben mittels Threads!

PLATFORM.RUNLATER

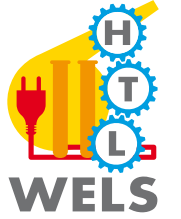
- Annahme: Eine ListView wird über ein Property an einen Service „gebindet“

```
...
listView.itemsProperty().bind(myListService.resultProperty());
...
```

  - in dem Fall muss eine Änderung der ListView über runlater realisiert werden:

```
...
Platform.runLater(() -> result.add("Element " + finalI));
...
```

JAVA DOCUMENTATION



- Java Dokumentation für Concurrency in JavaFX:
  - https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm#JFXIP546
-