EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE

Communication Networks

Discrete Event Simulator Tutorial

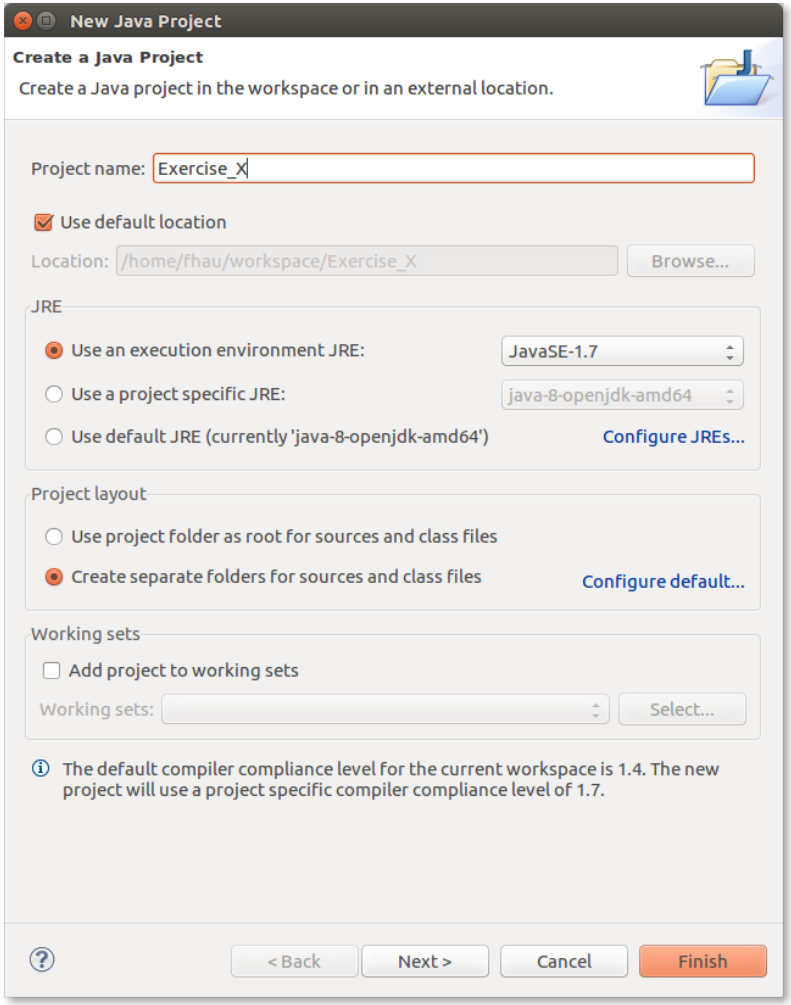
http://kn.inf.uni-tuebingen.de

Prerequisites

- As mentioned in the introduction, the discrete event simulator is written in **Jav**a. We strongly recommend using the **Eclipse IDE** or **IntelliJ** for working on the course assignments. Instructions for installing Eclipse can be found here.
- We suggest creating **separate Java projects** for each exercise. Create a new Java project in eclipse by selecting “File - New - Java Project” in the top menu.
- We suggest using drag-and-drop to put the ZIP archive’s contents into the folder *src* in the Java project as shown on the following slides

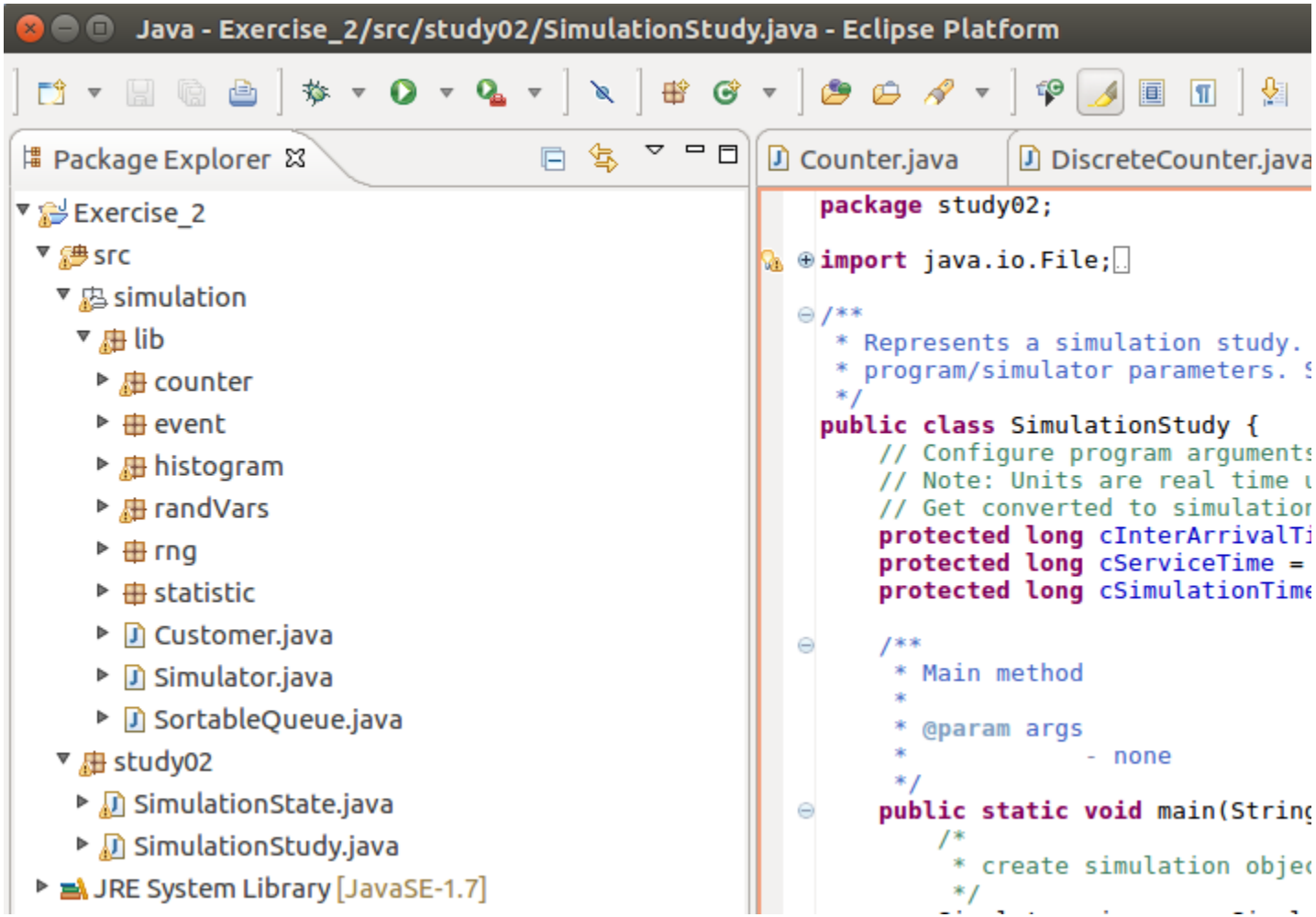
Prerequisites

- Create a new Java project in Eclipse



Prerequisites

- Use drag-and-drop to put the ZIP contents into the folder *src*

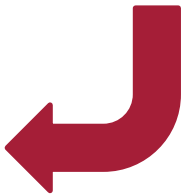


About this tutorial

- This tutorial provides first hands-on experience with the DES framework used in the course
- It is based on **source code with gaps** which comes with this document and corresponding **instructions in this presentation**
- Each step is accordingly marked with **TODOs** in the source code
- We also provide a solution to the tutorial
- We encourage you to finish this tutorial to get a better understanding of the implementation!

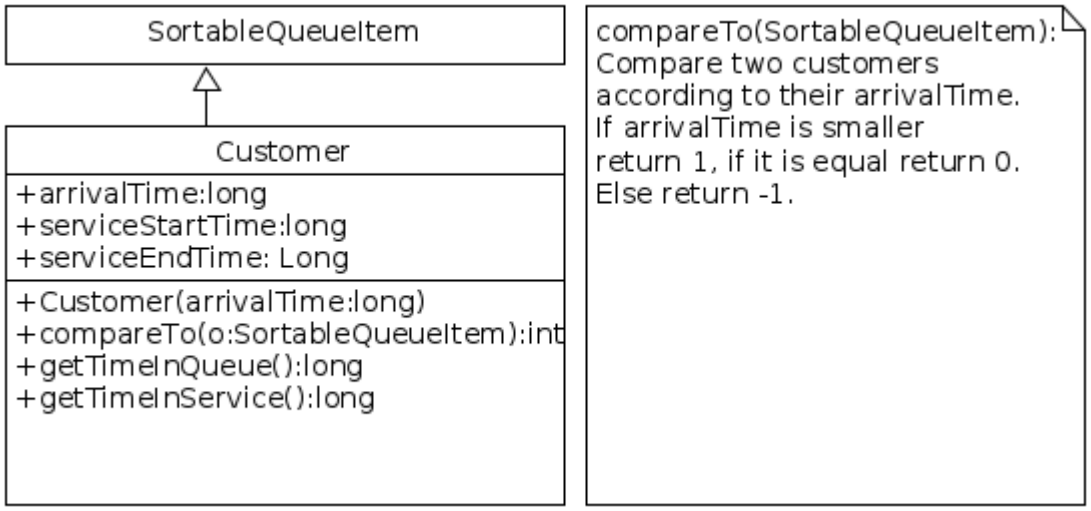
```
@Override
public void process() {
    fireUpdateQueueOccupancyNotification();
    /*
     * TODO Step 2.1 - create new Customer and fire event notification
     * Implement the steps below:
     */

    /*
     * TODO Step 2.1.1 create new Customer with the eventTime as arrivalTime:
     */
    long eventTime = this.getTime();
    // Customer customer = ...

    /*
     * TODO Step 2.1.2 fire a event notification that there is a new customer arrival event
     * Hint: firePushNewEventNotification expects a Class object (CustomerArrivalEvent, ServiceCompletionEvent or Si
     */
    //firePushNewEventNotification(...);
```

This is what it looks like



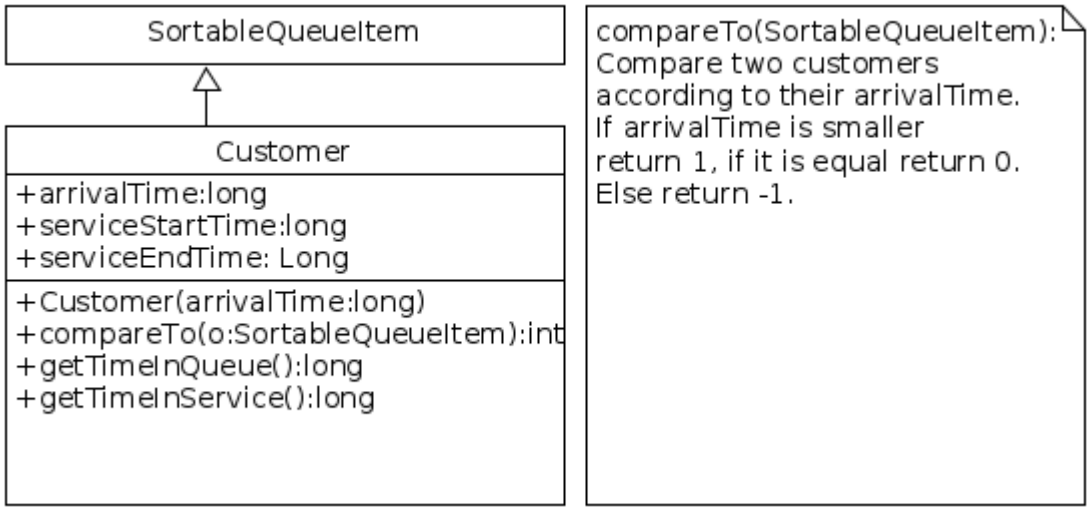
Step 1

- To make a statement about waiting times, we introduce a queue (in simulation state) and customer objects that are inserted into the queue at their **arrival instants**.
- The customer is removed from the queue at its **service initiation time** and it is deleted at it's *service completion time*.
- The time of **arrival**, **service initiation** and **service completion** are stored with the customer object to easily evaluate its waiting and service time in the system.



Step 1

- Implement the **Customer** class according to the following class diagram.
- This includes following methods:
  - compareTo(SortableQueueItem o)
  - getTimeInQueue()
  - getTimeInService()



Step 2

- Implement the **process()** function in the **CustomerArrivalEvent** class.

```
@Override
public void process() {
    fireUpdateQueueOccupancyNotification();
    /*
     * TODO Step 2.1 create new Customer with the eventTime as arrivalTime:
     */
    long eventTime = this.getTime();
    // Customer customer = ...

    /*
     * TODO Step 2.1 push a new event notification that there is a new customer arrival event
     * Hint: firePushNewEventNotification expects a Class object (CustomerArrivalEvent, ServiceCompletionEvent or SimulationTerminationEvent)
     */
    //firePushNewEventNotification(...);


    if (state.serverBusy == true) {
        /*
         * TODO Step 2.2 - the server is busy
         * Push the customer to the state.waitingCustomers list
         * And update state.queueSize
         */
    } else {
        /*
         * The server is not busy
         * TODO Step 2.3 - start the service of the new customer
         * Set state.customerInService to the newly created customer
         * Set customer.serviceStartTime to the eventTime
         */
        // state.customerInService = ...
        // customer.serviceStartTime = ...

        /*
         * TODO Step 2.3 - push a new service completion event
         * Push a service completion event to the event queue
         * Hint: firePushNewEventNotification expects a Class object (CustomerArrivalEvent, ServiceCompletionEvent or SimulationTerminationEvent)
         */
        //firePushNewEventNotification(...);

        /*
         * TODO Step 2.3 - set the serverBusy flag
         */
    }

    /*
     * TODO Step 2.4 - inform the simulator that it has to update its statistic objects
     * Hint: Use null as argument for the fireUpdateStatisticsNotification() method
     */
}
```

Step 2

- *Step 2.1:* Each arrival event first generates a new customer object and then pushes a new arrival event to the simulator queue
  - Use **firePushNewEventNotification()** for this

- *Step 2.2:* If the server is busy, add the customer to the queue of waiting customers (in the simulation state) by using the **pushNewElement()** function. Afterwards, update **queueSize**.

- *Step 2.3:* If the server was not busy, set the created customer as **customerInService**, assign a **service starting time** to the customer and push **a service completion event** to the Simulator's event queue. Then, set the **serverBusy** flag to true.

- *Step 2.4:* Last, use **fireUpdateStatisticsNotification()** to inform the Simulator that there are statistic objects to be updated.

Step 3

- Implement the **process()** function in the **ServiceCompletionEvent** class.
- Each service completion event first extract the **current customer** in service. Afterwards, it checks the **queueSize** to determine if there are waiting customers.

```
@Override
public void process() {
    // Get the current customer in service
    Customer currentCustomer = state.customerInService;

    // Check queue size
    if (state.queueSize > 0) {
        /*
         * TODO Step 3.1 - the queue contains customers
         * Set the next customer as state.customerInService (Hint: use state.waitingCustomer.popNextElement() and cast it to Customer)
         * Set the serviceStartTime of nextCustomer to the eventTime (Hint: use this.getTime())
         */
        // Customer nextCustomer = ...

        //state.customerInService = ...

        /*
         * TODO Step 3.1 - push a new service completion event to the event chain
         * Hint: firePushNewEventNotification expects a Class object (CustomerArrivalEvent, ServiceCompletionEvent or SimulationTerminationEvent)
         */
        //firePushNewEventNotification(...);

        /*
         * TODO Step 3.1 - update state.queueSize
         * One customer was removed from the queue and is now in service
         */
    } else {
        /*
         * TODO Step 3.2 - the queue is empty
         * Set the state.serverBusy flag to false
         * Set the state.customerInService to null
         */
    }

    /*
     * TODO Step 3.3 - update statistics
     * Increase sample counter with state.increaseNumSamplesByOne()
     * Call fireUpdateStatisticsNotification(Object arg) (Hint: use currentCustomer as argument)
     */
}
```

Step 3

- *Step 3.1:* If the queue size is greater than zero, set the next customer as **customerInService** and set its **serviceStartTime**. Afterwards, push a new service completion event to the simulator's event chain (**firePushNewEventNotification()**) and update the **queue size of waiting customers**.

- *Step 3.2:* If the queue is empty, set the **busy flag** to **false** and set the **customer in service** to **null**.

- *Step 3.3:* Last, increase the **sample counter** (using **increaseNumSamplesByOne()**) and use **fireUpdateStatisticsNotification()** to inform the Simulator that there are statistic objects to be updated. Note that the notification will use the **current customer** as function argument.

Step 4

- Complete methods in **Simulator** class.

```java
public long realTimeToSimTime(double realTime) throws NumberFormatException {
    /*
     * TODO Step 4.1 - convert real time to simulation time
     * Hint: multipy with this.simTimeInRealTime as conversion factor
     * Round up conversion result (Use Math.ceil(...))
     * Check if conversion result is greater than Long.MAX_VALUE and throw a new NumberFormatException if that's the case
     */
    return 0;
}

/**
 * Converts sim time to real time
 * @param simTime units in sim time
 * @return units in real time
 */
public double simTimeToRealTime(long simTime) {
    /*
     * TODO Step 4.1 - convert simulation time to real time
     * Hint: Again, use this.simTimeInRealTime as conversion factor
     * Cast result to double
     */
    return 0;
}

/**
 * Starts the simulation
 * @throws Exception is thrown when event order is invalid
 */
private void run() {
    while(!stop) {
        Event e = (Event) ec.popNextElement();
        if(e != null) {
            //check if event time is in the past
            if(e.getTime() < now) {
                throw new RuntimeException("Event time " + e.getTime()
                        + " smaller than current time " + now);
            }
            /*
             * @TODO Step 4.2 - set the simulation time
             * Hint: use e.getTime() to retrieve the current time
             */
            //this.now = ...

            /*
             * @TODO Step 4.2 - register the simulator as observer to get event notifications
             * Hint: use e.register(IEventObserver obs)
             */

            /*
             * @TODO Step 4.2 - process event
             * Hint: use e.process()
             */
```

Step 4

- *Step 4.1:* Complete **realTimeToSimTime()** and **simTimeToRealTime()**.
  - In **realTimeToSimTime:**
    - make sure the associated simulation time doesn't exceed the range of a **long** (throw a **NumberFormatException** then).
    - Round the result up (**Math.ceil()**)

- *Step 4.2:* Complete the **run()** function in the simulator class.
  - Set the **simulation time**, (assign the event time)
  - Register the simulator as **observer**
  - **process the event**
  - **unregister** the simulator.

Step 4

- *Step 4.3:* Complete **pushNewEventHandler()**
  - Check if the argument is type of **customer arrival event** or **service completion event**
  - Then create new events with appropriate event times (use the **current time**, use **getSimTime()** and add the time stored in **interArrivalTime** or **serviceTime**).

- *Step 4.4:* Set the customer's **SeviceEndTime** in **updateStatsSCE()**.

Step 5

- Configure the Simulation in **SimulationStudy** class.

- Each Event has an event time
  - **CustomerArrival**: use a constant inter-arrival time of 10 seconds
  - **ServiceCompletion**: use a constant service time of {9,11} seconds
  - **SimulationTermination**: use a constant simulation time of $10^{\{4,5\}}$ seconds

Step 6

- Congratulations! You are now able to run your simulation.
- Perform the 4 simulation runs according to the parameters given (all parameter pairs) and explain the observed results for the **minimum** and the **maximum queue occupation**.
- How would you **classify** this simulation (dynamic/static, deterministic/stochastic, continuous/discrete)?