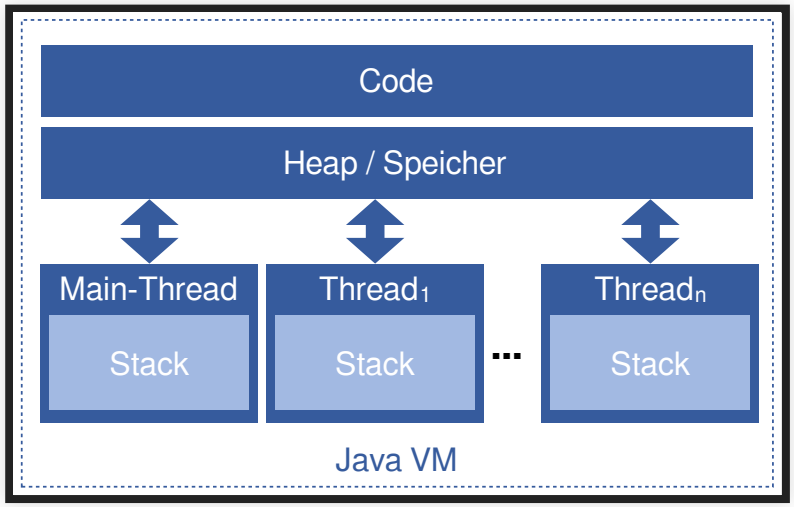
CHAPTER 11: LIGHTWEIGHT PROCESSES - THREADS

LEARNING OBJECTIVES

- Be able to explain the term thread and the applications of threads
- Be able to explain the thread concept in Java including thread creation
- Be able to explain how threads are scheduled
- Know important thread methods and their usage

11.1 PRINCIPLE

A thread is a lightweight process that runs embedded within an operating system process.



Multiple threads share the (virtual) memory of the process, with each thread having its own stack for method calls. Thus, each thread can execute its own method calls on objects independently from other threads. The Java objects allocated on the heap can be used by all threads in parallel.

Areas of application of threads include:

- non-blocking I/O
- GUI programming (parallel UI update: http://www.sorting-algorithms.com)
- Word processing (parallel spell check, saving in the background)
- Acceleration of algorithmic calculations (use of multiple CPUs)

Java provides a class `Thread` and an interface `Runnable` in the runtime library:

```
interface Runnable {
   public void run();
}
```

A **thread** can basically be created and started in two ways:

- Derivation of your own class from Thread: you define your own class which inherits from Thread. The run method of the Thread class should be overwritten; it contains the instructions that are executed after the thread has started using the specified start method.
- Creation of a thread object, thereby passing an object that implements the runnable interface (i.e. has a run method). The thread may be started by calling its start method which leads to the execution of the run method of the Runnable object.

EXERCISE: WHAT ARE THREADS

Which statements are true?

- ☐ Instructions to be executed in a thread are described using a derivation of the `Thread` class or an implementation of the `Runnable` interface
- ☐ Each thread has individual (virtual) memory
- ☐ Java uses threads to execute each statement (loops, conditions, operations) in parallel
- ☐ Threads share the same (virtual) memory

11.2 CREATION AND START OF THREADS

OPTION 1: SPECIALIZING THE THREAD CLASS

The run method is overridden, it implements the thread's main loop, which is executed when the thread starts.

```
class MyThread extends Thread {
   private int count = 0;
   public void run() {
     // what the thread does...
     System.out.println(counter++);
   }
   public static void main(String args[]) {
     Thread t = new MyThread();
     // Start thread, run is executed
     t.start();
   }
}
```

OPTION 2: PASS RUNNABLE OBJECT

This is the preferable option because it does not introduce technical inheritance. It is particularly suitable when inheritance from a class is already taking place, i.e. deriving from Thread is no longer possible. A lambda expression “r” is defined that contains the run method to be executed later. “r” is passed when the thread is created and can therefore be executed when the thread starts.

```
class Starter {
   public static void main(String args[]) throws Exception {
     Runnable r = () -> {
       int count = 0;
       while (true)
         // endlessly output the counter and count up
         System.out.println(counter++);
     };
     Thread t = new Thread(r);
     // Start thread, run is executed
     t.start();
   }
}
```

THREAD VS RUNNABLE

When to implement the `Runnable` interface and when to extend the `Thread` class?

- It may be necessary to pass additional data and store them in the thread object during construction. In that case you could consider deriving from the `Thread` class.
- If you only want to outsource the instructions that can be executed in a thread, for example, we recommend implementing the `Runnable` interface.

EXERCISE: THREAD CREATION (1)

The following code is given. Copy it and run the example.

```
public class Test {
     public static void main(String args[]) {
          Runnable r = () -> {
               for (int i = 0; i < 100; i++) {
                    System.out.println(i);
               }
          };
          new Thread(r).start();
          new Thread(r).start();
     }
}
```

Which statements are true?

- [ ] Console output is continuous from 0 to 100
- [ ] Each number appears twice in the console
- [ ] There are always two identical numbers one after the other in the console
- [ ] There are two sequences, both ascending but issued at different speeds

EXERCISE: THREAD CREATION (2)

The following code is given. Copy it and run the example.

```
public class Test {
     private static int i;
     public static void main(String args[]) {
          Runnable r = () -> {
               for (i = 0; i < 100; i++) {
                    System.out.println(i);
               }
          };
          new Thread(r).start();
          new Thread(r).start();
     }
}
```

Which statements are true?

- [ ] Each number appears twice in the console
- [ ] There are always two identical numbers one after the other in the console
- [ ] There are two sequences, both ascending but output at different speeds
- [ ] Console output is continuous from 0 to 100
- [ ] The sequence contains numbers from 0 to 100, but is not continuous and may contain duplicates

EXERCISE: THREAD CREATION (3)

The following code is given. Copy it and run the example.

```
public class Test extends Thread {
     private int i;

     public void run() {
          for (i = 0; i < 100; i++) {
               System.out.println(i);
          }
     }

     public static void main(String args[]) {
          new Test().start();
          new Test().start();
     }
}
```

Which statements are true?

- [ ] The console output is (usually) continuous from 0 to 100
- [ ] There are always two identical numbers one after the other in the console
- [ ] There are two sequences, both ascending but issued at different speeds
- [ ] Each number appears twice in the console

11.3 TERMINATING THREADS

*Threads automatically terminate at the end of the execution of their **run** method.*

A common method of terminating a thread is the following construction (here with a thread derivation):

```
class MyThread extends Thread {
   private boolean terminated = false; // the thread is active
   public void run() {
     while (!terminated) {
       // Execution of thread activity
     }
   }
   public void terminate() {
     terminated = true; // the thread is inactive, see run()
   }
   public static void main(String args[]) {
     Thread t = new MyThread();
     t.start(); // Start thread
     Thread.sleep(1000); // wait 1000 ms
     t.terminate(); // end the created thread
   }
}
```

The `terminated` attribute controls the execution of the spawned thread. As long as it has the value `false`, the thread is active.

THREAD AND THE MAIN THREAD

If a new thread is started, it is not automatically terminated when the process in which the Main method is executed is terminated.

In order to terminate processes cleanly, it is therefore important to implement strategies that terminate all spawned threads in a controlled manner.

EXERCISE: TERMINATING THREADS

Which statements are true?

- [ ] The point in time at which thread terminates depends on the implementation
- [ ] The Java VM runs until the last thread of the application has ended
- [ ] Java ensures that all threads are terminated when the main method ends
- [ ] Threads are never terminated
- [ ] Threads run until they have processed all statements of the run method

11.4 IMPORTANT THREAD OPERATIONS

STATIC METHODS

- `currentThread()`
  returns reference to the current thread
- `sleep(long)`
  pauses the calling thread for the number of milliseconds given

INSTANCE METHODS

- `getName()`
  returns the name of the current thread (Thread-0, Thread-1, ...)
- `setDaemon(boolean)`
  marks a thread to automatically terminate when the application exits
- `join()`
  Caller thread waits for the called thread to complete, example:

```
Thread t = new Thread(() -> {
   for (int i = 0; i < 1000; ++i)
     System.out.println(i);
};
t.start();
t.join(); // block until the thread terminates
```

11.5 EXERCISE

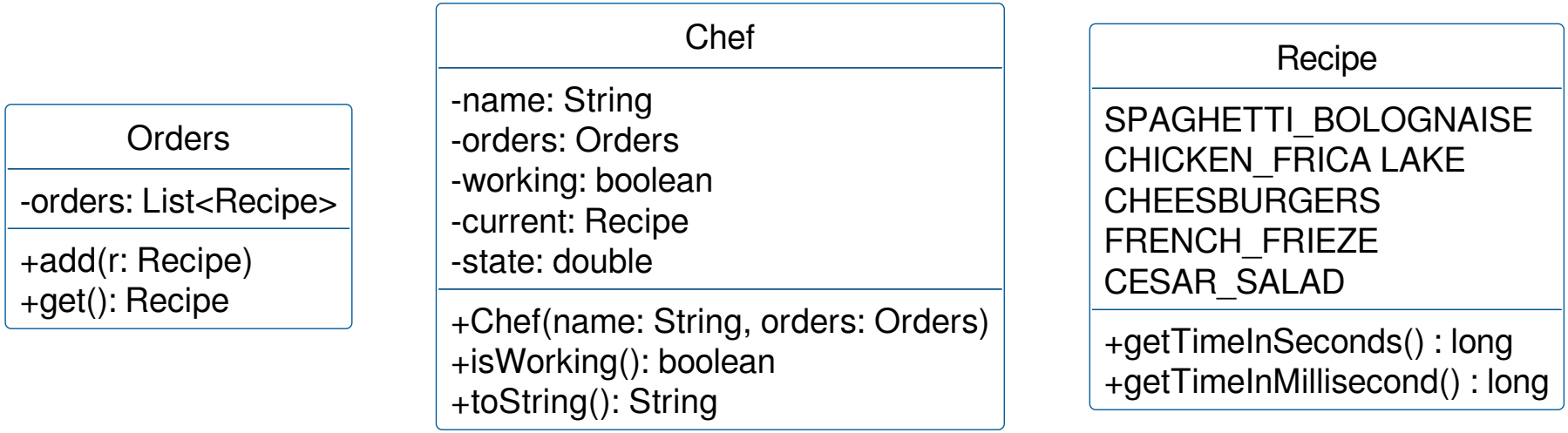
COOKING SIMULATION

In the following simulation, several orders must be processed in parallel. There is a globally sorted order list and several chefs who process the orders.

```
Starte mit 13 Rezepten
```

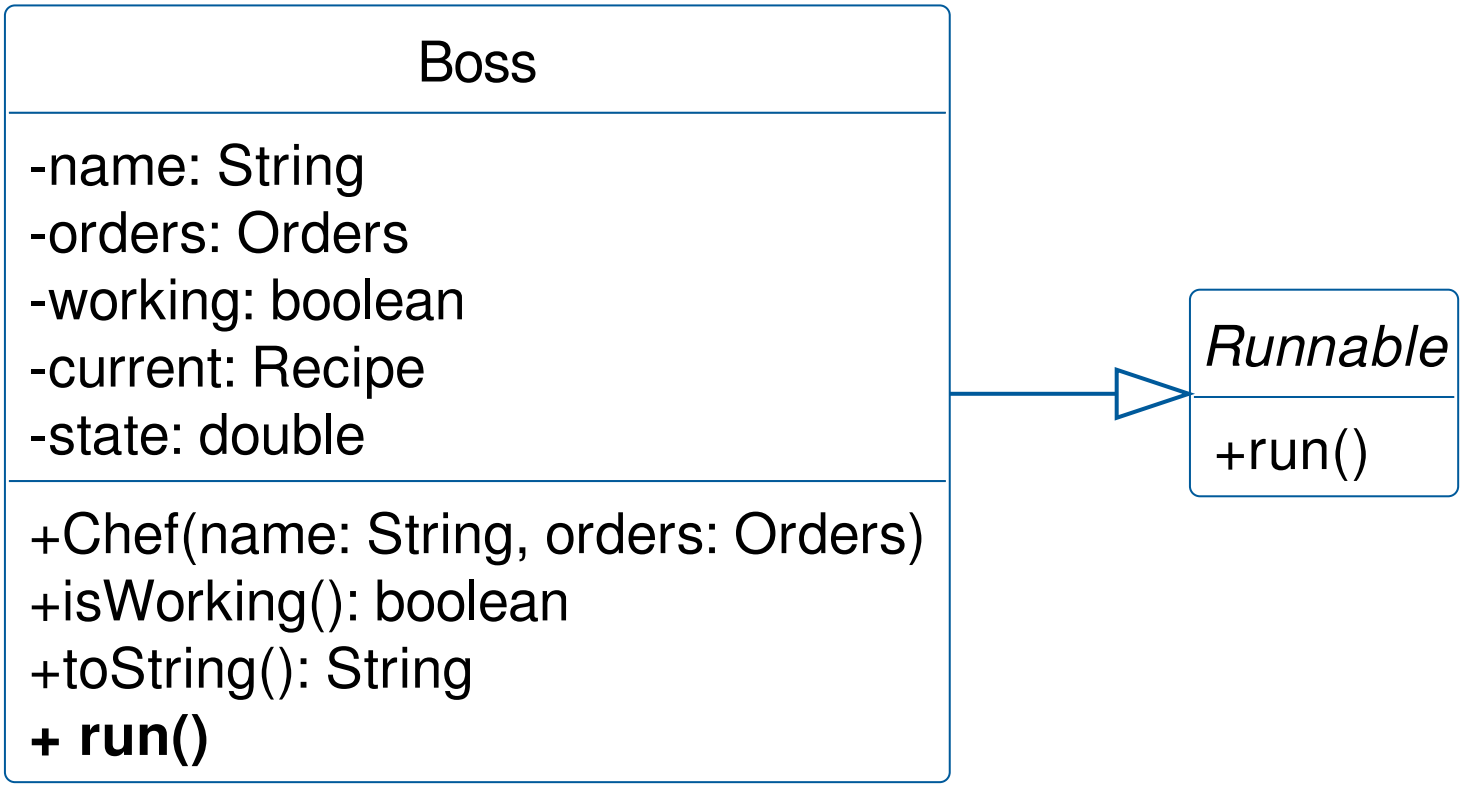
Unzip cooking-threads.zip and import the folder as "Existing Maven Project".

Given a number of chefs based on class `Chef`, an order list based on class `Orders` and possible recipes in the enumeration type `Recipe`.



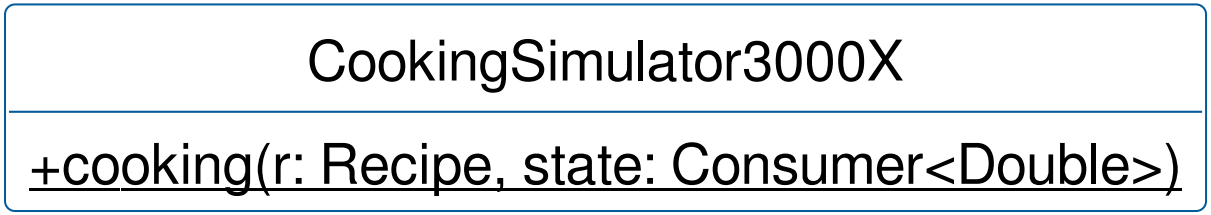
STEP 1: PARALLEL COOKS

Extend the Chef class to allow it to edit recipes in its own thread. To do this, implement the `Runnable` interface and provide a `run` method.



- Implement the `Runnable` interface
- Implementation for `run` method in `Chef` class
  - Set the `working` attribute to `true`
  - Get a new order from the order list in the `orders` attribute using `get`
  - Check whether the order is non-zero
  - Set the `current` attribute to the corresponding recipe
  - Start cooking, use the `CookingSimulator3000X` and call the static method `cooking`

Note about CookingSimulator3000X

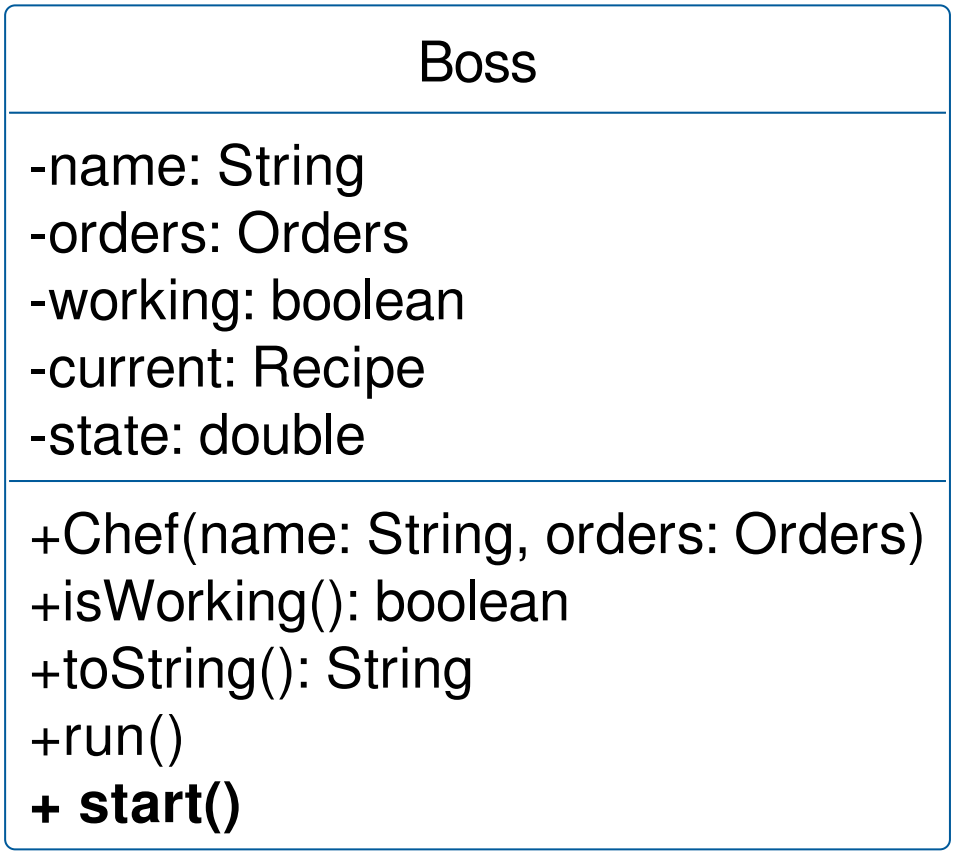


In addition to the recipe parameter, you also need to pass a lambda expression of type `Consume<Double>`. You can use this to get the current status of the simulator.

To do this, formulate a lambda expression to assign the current state of the preparation to the `state` attribute in the `chef` class.

STEP 2: START THREADS

Extend the `Chef` class with a `start` method that creates and starts a new thread.
Your own instance should be used as a runnable for the constructor of Thread.



STEP 3: LET CHEFS COOK

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Orders orders = new Orders();
        orders.add(Recipe.CESAR_SALAD);
        // ...

        Chef chef1 = new Chef("Max", orders);
        Chef chef2 = new Chef("Erika", orders);
        Chef chef3 = new Chef("Inge", orders);

        chef1.start();
        chef2.start();
        chef3.start();

        waitUntilFinished(orders, new Chef[] {chef1, chef2, chef3});
    }
}
```

In of class `Main`, an order list and three cooks are prepared.

Start all three chefs by uncommenting the corresponding lines.