Threads in Java

A thread is a **call sequence** that **executes independently** of **others**, while at the same time possibly **sharing** system **resources**.

First, we show how this concept is realized in Java. Next, we demonstrate how different threads can be synchronized using Java language constructs.

Table of contents

1. Basics

1.1. Construction of Java threads

Every program consists of **at least one thread** – the one that runs the `main` method of the class provided as a start up argument to the Java virtual machine (JVM).

Other internal background threads may also be started during JVM initialization.

However, all **user-level threads** are **explicitly constructed** and started from the **main thread**, or from any other threads that they in turn create.

The **code** of a thread has to be realized within a **method** with name **`run`**.

```
public void run() {
    // Code will be executed into a separate thread.
}
```

A program creating threads can use the **class `Thread`** and **overwrite** its **`run`**-Method (interfaces are discussed later):

```
$ cat MyThread.java
public class MyThread extends Thread
{
        public void run() {
                System.out.println("Hello World");
        }

        public static void main(String[] args) {
                MyThread t = new MyThread();
                t.start();
        }
}
$
```

The method `start()` is defined within the class `thread` and starts a thread by executing its `run`-method.

In our example, first a thread `t` is created (`new`…), next its `start`-method is called, executing the `run`-method which prints out "`Hello world`".

When creating **more** than one **thread**, it is **not** possible to determine the **execution sequence** (as shown into the following example):

```
$ cat Loop1.java
public class Loop1 extends Thread
{
        private String myName;

        public Loop1(String name) {
                myName = name;
        }

        public void run() {
                for(int i = 1; i <= 10000; i++)
                {
                        System.out.println(myName + " (" + i + ")");
                }
        }

        public static void main(String[] args) {
                Loop1 t1 = new Loop1("Thread 1");
                Loop1 t2 = new Loop1("Thread 2");
                Loop1 t3 = new Loop1("Thread 3");
                t1.start();
                t2.start();
                t3.start();
        }
}
$
```

Identifier for the thread

creating 3 threads

starting each thread

Output of the program:

```
$ java Loop1
…
Thread 1 (7823)
Thread 2 (8886)
Thread 1 (7824)
Thread 2 (8887)
Thread 1 (7825)
Thread 2 (8888)
Thread 1 (7826)
Thread 3 (6647)
Thread 2 (8889)
Thread 3 (6648)
Thread 2 (8890)
Thread 3 (6649)
Thread 2 (8891)
Thread 3 (6650)
Thread 2 (8892)
…
```

One idea to force the execution sequence could be to let one thread sleep for a certain time (but as shown, this will **not** work):

```
$ cat Loop3.java
public class Loop3 extends Thread
{       public Loop3(String name) {
                super(name);
        }
        public void run() {
                for(int i = 1; i <= 10000; i++)
                {       System.out.println(getName() + " (" + i + ")");
                        try {
                                sleep(10);
                        }
                        catch(InterruptedException e) {
                        }
                }
        }

        public static void main(String[] args) {
                Loop3 t1 = new Loop3("Thread 1");
                Loop3 t2 = new Loop3("Thread 2");
                Loop3 t3 = new Loop3("Thread 3");
                t1.start();
                t2.start();
                t3.start();
        }
}
$
```

The method `getName` of the class `Thread` returns the thread identifier as string;

`sleep` is a static-Method of the class `Thread` letting the thread sleep for 10 millisecs.

Output:

```
java Loop3
…
Thread 1 (100)
Thread 2 (98)
Thread 3 (97)
Thread 1 (101)
Thread 3 (98)
Thread 2 (99)
Thread 1 (102)
Thread 3 (99)
Thread 2 (100)
Thread 1 (103)
…
$
```

1.2. Using threads within complex class hierarchies

If the run-method itself is part of a derived class, the class can not be derived from the class Thread (Java does **not** support **multiple** inheritance).

In this case the **interface** **`Runnable`** of the package java.lang can be used:

```
$ cat MyRunnableThread.java
public class MyRunnableThread implements Runnable
{
        public void run()
        {
                System.out.println("Hello World");
        }

        public static void main(String[] args)
        {
                MyRunnableThread runner = new MyRunnableThread();
                Thread t = new Thread(runner);
                t.start();
        }
}
$
```

The class `Thread` itself implements `Runnable`. Thus, the code runs into a `Runnable` using it as an argument to the thread constructor.

Most of the time, we will not use this method, because our examples are as easy as possible and no complex class hierarchies are necessary.

2. Synchronization of threads

2.1. Sharing resources

If several **threads share resources** they have to „**agree**" **who** is allowed to do **which** action on what **time**.

We demonstrate the possibilities of Java.

For this, the possibilities offered by java are shown demonstrating an **example** of a **banking account**.

A bank is modelled by 4 classes:

1. Class **`account`** represents a **bank account** with
   - attributes
     - `balance` to hold the actual balance
   - methods
     - `set(value)` to deposit (positive value) or withdraw (a negative value) money
     - `get()` to request the actual balance

```
class account {
        private float balance;

        public void set(float amount) {
                balance = amount;
        }

        public float get() {
                return balance;
        }
}
```

2. The class **bank** represents a bank with `accounts` on which some `booking`-operation can take place. The constructor is responsible for initializing all accounts.

```java
class bank {
	private account[] accounts;
	String bankName;

	public bank(String bankName) {
		this.bankName = bankName;

		accounts = new account[100];
		for(int i = 0; i < accounts.length; i++) {
			accounts[i] = new account();
		}
	}

	public void booking(String employee,
				int accountnr, float amount) {
		float oldBalance = accounts[accountnr].get();
		float newBalance = oldBalance + amount;
		accounts[accountnr].set(newBalance);
	}
}
```

A **booking operation** has to be done by an **employee**, who first has to read the actual balance, next to set the balance of a special account.
Until now, we do not play with threads.

3. We use **threads** to realize the class **employee**. The name of the thread will be the name of an employee. Thus **each** employee of a bank becomes a **thread**. Bookings could be

simulated by random numbers, generated in the run-method – we always use `accountnr` 1 and deposit 1000 times 1 USD.

```java
class bankEmplojee extends Thread {
        private bank bank;
        public  String name;

        public bankEmplojee(String name, bank bank) {
                super(name);
                this.bank = bank;
                this.name = name;
                start();                // thread started in the constructor
        }

        public void run() {
                for(int i = 0; i < 1000; i++) {
                        int accountnr = 1; // better random number
                        float amount = 1;  // better random number

                        bank.booking(name, accountnr, amount);
                }
        }
}
```

4. Our bank (class **`bankoperation`**) only has two employees. They start working when the objects are generated.

```
public class BankOperation {
        public static void main(String[] args) {
            bank DeutscheBank  = new bank("DeutscheBank");
            bankEmplojee eve   = new bankEmplojee("Eve", DeutscheBank);
            bankEmplojee hanna = new bankEmplojee("Hanna", DeutscheBank);
        }
}
```

This finalized the implementation of a bank.

Let us run the program and see what will happen after the bank has closed the doors?

We **extended** the program to **log each transaction** into a log file and **store account information** into a file during an **end-of-day processing**.

The situation should be: the balance of account 1 is 2000.

```
$ java BankOperation.java
$
$ head accounts.DeutscheBank
account 0: 0.0
account 1: 1114.0
account 2: 0.0
…
$
```

we have lost a lot of transactions !

This problem occurs, when more threads use common objects (here the array account) without implementing protection mechanisms.

But how can this happen?

Let's start our analysis with the following situation: Eve and Hanna are booking 1 USD onto account 1. Initially the balance is 0.

Thread Eve

```
public void booking(String employee = "Eve",          accounts[1] = 0
                    int accountnr   = 1,
                    float amount    = 1) {
       float oldBalance = accounts[accountnr].get();   oldBalance = 0
switch to thread Hanna
       float newBalance = oldBalance + amount;
       accounts[accountnr].set(newBalance);
}
```

Thread Hanna

```
public void booking(String employee = "Hanna",        accounts[1] = 0
                    int accountnr   = 1,
                    float amount    = 1) {
       float oldBalance = accounts[accountnr].get();   oldBalance  = 0
       float newBalance = oldBalance + amount;         newBalance  = 1
       accounts[accountnr].set(newBalance);            accounts[1] = 1
}
switch to thread Eve
```

Thread Eve

```
public void booking(String employee = “Eve”,
                    int accountnr   = 1,
                    float amount    = 1) {
       float oldBalance = accounts[accountnr].get();    oldBalance   = 0
continue after having switched
       float newBalance = oldBalance + amount;          newBalance   = 1
       accounts[accountnr].set(newBalance);             accounts[1] = 1
}
```

Now the deposit of Hanna is lost!

Thus, the origin of the lost booking is that **one** booking **transaction** consists of **more** than one **statement** and between these statements, the scheduler **switched** between threads.

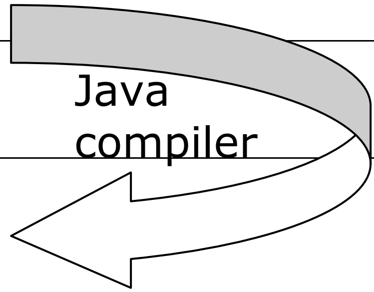
2.1.1. First try to solve the problem

Within the class bank, we realize **booking** by **one Java statement**:

```
class account {
        private float balance;

        public void booking(float amount) {
                balance += amount;
        }
}
class bank {
        private account[] accounts;
        String bankName;

        public bank(String bankName) {
                this.bankName = bankName;

                accounts = new account[100];
                for(int i = 0; i < accounts.length; i++) {
                        accounts[i] = new account();
                }
        }

        public void booking(String employee, int accountnr, float amount) {
                accounts[accountnr].booking(amount);
        }
}
```

This idea is **not** a solution, because a Java-program is compiled into byte code. The java compiler produces from the Java statement

```
    balance += amount;
```



the following code (here mnemonic code):

```
    1: LOAD(balance);
    2: ADD(amount);
    3: STORE(balance);
```

Thus, the JVM executes 3 statement and we have the same problem: switching between statements.

2.1.2. Second try

We have to find a **solution** from an **application point of view**:

> An employee may only **start booking** if **no other** employee **is booking**.

This is a solution from the employee's point of view. But how can we implement that in Java?

The first implementation idea is:

> We let all classes but `bank` unchanged. Into `bank`, we program a **lock mechanism** which may prevent two employees booking at the same time.

```java
class bank {
        private account[] accounts;
        String bankName;
        private boolean lock;

        public bank(String bankName) {
                this.bankName = bankName;

                accounts = new account[100];
                for(int i = 0; i < accounts.length; i++) {
                        accounts[i] = new account();
                }
                lock = false;       // initialize to "not locked"
        }

        public void booking(String employee,
                            int accountnr, float amount) {
                while(lock);       // wait unit "not locked"
                lock = true;       // lock

                float oldBalance = accounts[accountnr].get();
                float newBalance = oldBalance + amount;
                accounts[accountnr].set(newBalance);

                lock = false;      // unlock
        }
}
```

The booking statements are encapsulated by a lock mechanism: first we wait until we can't find a lock, after we can enter, we lock, perform the booking operation, unlock and return.

This implementation **seems to be correct** – but on thinking about that solution, we find following **problems**:

1. Two employees can **not work simultaneously**, but it would not cause a problem if they would book different accounts.

2. **Active waiting** (`while(lock);`) consumes CPU time by doing nothing.

3. But the principle problem is, that this implementation does **not** solve our problem: **Active waiting** is **not an indivisible operation**, the byte code looks like:

`while (lock);`	`1: LOAD(lock);` `2: JUMPTRUE 1;`
`lock = true;`	`3: LOADNUM(TRUE);` `4: STORE(lock)`

If the scheduler switches between operation 1 and 2 and the lock is not set (`lock==false`), a waiting thread can enter and perform the booking.

We now demonstrate a correct solution, using Java elements to synchronize threads.

2.2.Synchronized methods und blocks

A correct but (still inefficient) solution for the problem 2 (active waiting) and 3 (lost transaction) uses Java's `synchronized` methods. We only have to add the keyword `synchronized` to mark the `booking` method.

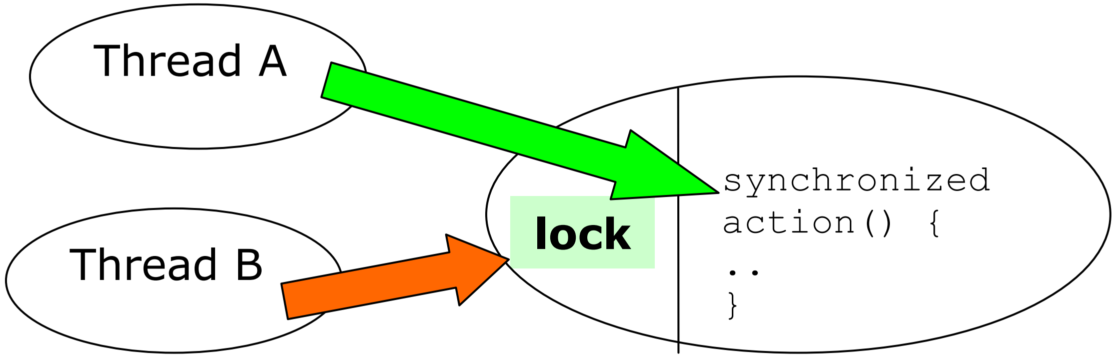
Every instance of class **`Object`** and its subclasses **possesses** a **lock**. Scalars of type `int`, `float`, etc. are no `Objects` and therefore can be locked only via their enclosing objects.

The `synchronized` keyword is **not** considered to be part of a method's signature. So the `synchronized` modifier is **not** automatically inherited when subclasses override superclass methods, and methods in `interfaces` cannot be declared as synchronized.

In addition to synchronized methods, any Java block can be marked as synchronized; in this case synchronization takes an argument of which object to lock (we will see that later).

Locking follows a build-in **acquire-release protocol**, controlled only by use of the synchronized keyword:

A **lock** is **acquired** on **entry** to a `synchronized` method or block, and **released** on **exit**, even if the exit occurs due to an exception.



Thread A is executing a synchronized method, locking thereby the object. Thread B wants to execute the same method (of the same object), it is blocked until thread A has finished the execution of the synchronized method; than the lock is released.

This mechanism is implemented within the JVM **without** active waiting; the blocked thread will not further be considered as ready by the scheduler.

A `synchronized` method or block obeys the acquire-release protocol **only** with respect to other `synchronized` methods or blocks **on the same target object**. Methods that are not `synchronized` may still execute at any time, even if a synchronized method is in progress. In other words, `synchronized` is not equivalent to atomic, but synchronization can be used to achieve atomicity.

Now we can change our coding of the bank example: `booking` becomes a synchronized method. This solves the lost transaction problem and the active waiting problem!

```
$ cat Bankoperation.java
…
class bank {
        private account[] accounts;
        String bankName;

        public bank(String bankName) {
                this.bankName = bankName;

                accounts = new account[100];
                for(int i = 0; i < accounts.length; i++) {
                        accounts[i] = new account();
                }
        }

        public synchronized void booking(String employee,
                                int accountnr, float amount) {
                float oldBalance = accounts[accountnr].get();
                float newBalance = oldBalance + amount;
                accounts[accountnr].set(newBalance);
        }
}
…
```

By this approach, the bank-Object is locked, but we only have to lock one special account. Thus, problem 1 is still open.

The solution uses the Java feature to mark a block synchronized identifying the object to lock as argument.

```
$ cat Bankoperation.java
class bank {
        private account[] accounts;
        String bankName;

        public bank(String bankName) {
                this.bankName = bankName;

                accounts = new account[100];
                for(int i = 0; i < accounts.length; i++) {
                        accounts[i] = new account();
                }
        }

        public void booking(String employee,
                            int accountnr, float amount) {

              synchronized ( accounts[accountnr] ) {
                   float oldBalance = accounts[accountnr].get();
                   float newBalance = oldBalance + amount;
                   accounts[accountnr].set(newBalance);
              }
        }
}
```

Only **one** account is locked!

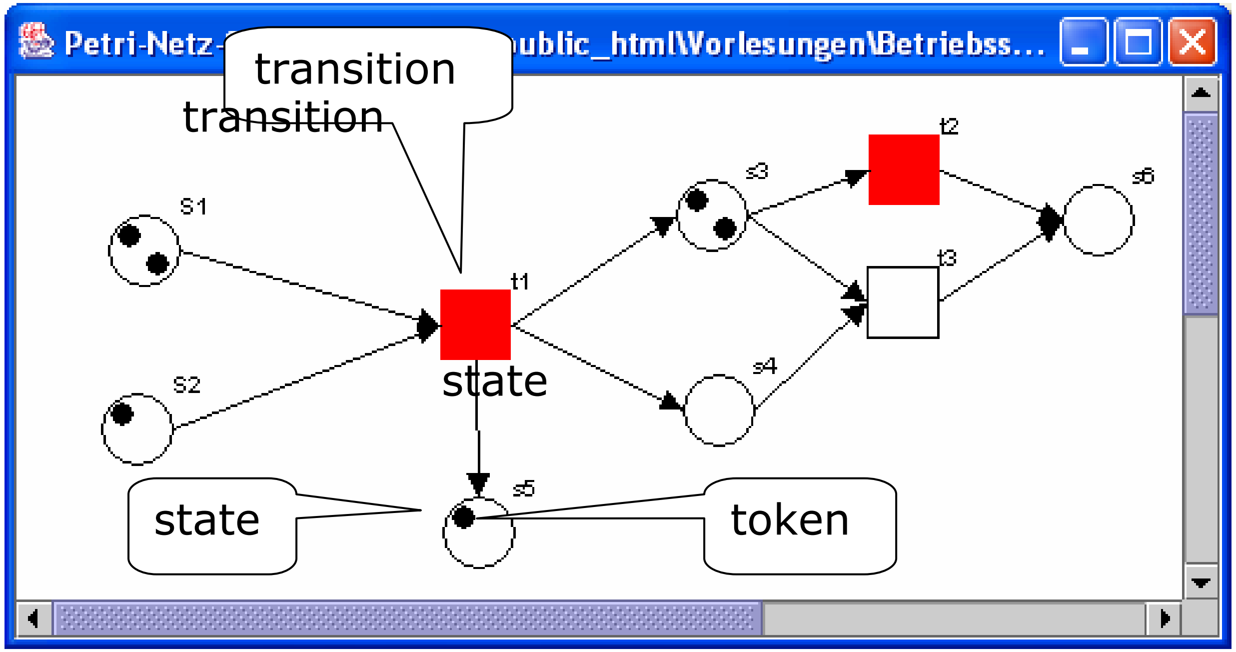
Common usage of objects can be synchronized using the keyword synchronized. However, an application scenario where all threads are only reading shared objects attributes would be not effective because of unnecessary overhead (set and unset a lock).

We can formulate the following general **rule**:

> If several threads share an object, where at **least one** thread **changes** that objects **state** (values of its attributes) than **all methods** accessing the object (no matter reading or writing) have to be **marked `synchronized`**.

To understand the behaviour of `synchronized`, we could use a **Petri net**:

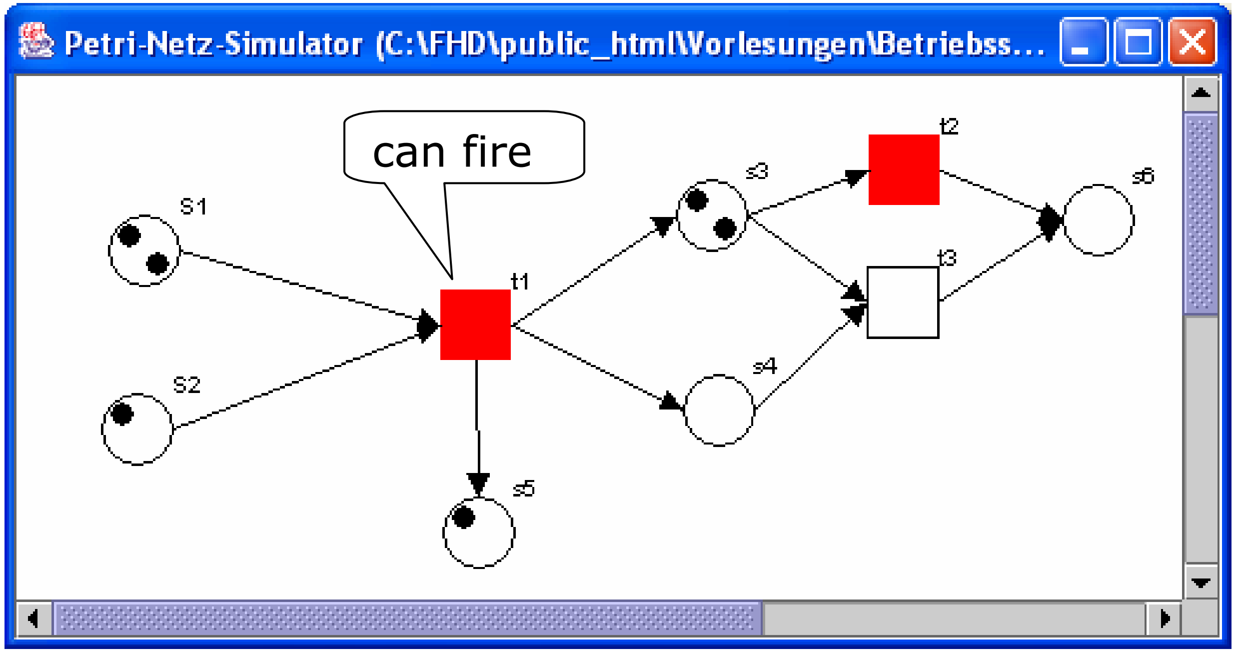
> A **Petri net** is a **model** that is used e.g. **to describe concurrent processes**.
> It consists of **links** and two types of **nodes**: **states** and **transitions**.



Links always **connect** a **state** and a **transition**, which is possible in both directions. So it's not possible to connect two states or two transitions with each other. Further, every **node** can have a **name** that illustrates the meaning it has in the context of the Petri net. Every state has also a certain number of **tokens** and every transition has a certain priority.

When you designed a Petri net, you probably might want to simulate it. **Simulating** a Petri net works as following:

> States can have a number of tokens, that are taken or given away when certain transitions 'fire'. A transition is able to fire (or is enabled), if for every of it's incoming links there is at least one token in the connected state. So when the transition fires (in the Simulator this happens, in fact, when you click on it), it takes one token of every state from which a link goes to it and gives one token to every state it has an outgoing link to.




```
($ petrinet.sh Netz_bsp01)
```

For not every transition must have the same number of incoming and outgoing links, the amount of tokens in the whole Petri net after firing a transition has not to be the same as before!

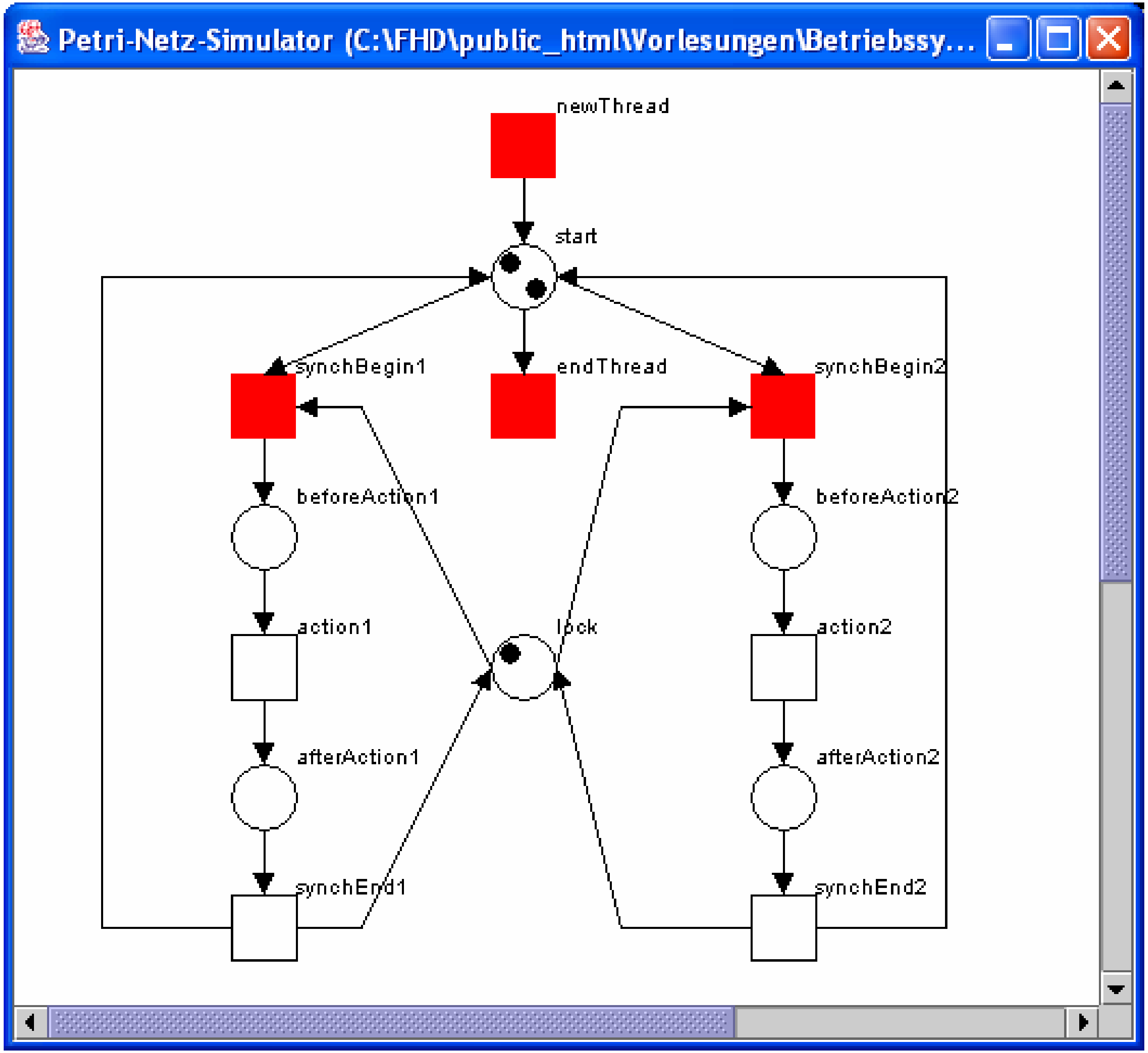
The possibility of firing enabled transitions can also be limited by the priorities of transitions: If transitions that have common incoming states are enabled, in fact only those with the highest priority really can fire.

Now we can consider a **Java program** and its **Petri net** to see the locking mechanism.

In that program all threads share one object.

```
class K {
        public synchronized void m1() {
                action1();
        }
        public synchronized void m2() {
                action2();
        }
        private void action1() { … }
        private void action2() { … }
}

class T extends Thread {
        private K einK;
        public T(K k) {
                einK = k;
        }

        public void run() {
                while (…) {
                        switch (…) {
                                case …: einK.m1(); break;
                                case …: einK.m2(); break;
                        }
                }
        }
}
```

($ petrinet.ksh)



A **state** represents the **position between** two Java **statements**.

A **transition** represents a Java **statement**.

A **token** represents a **thread**, i.e. the lock for the shared K-Object.

To start a thread (transition „synchBegin“) we need a token on states “start” **and** “lock”.

If a transaction fires, the token will be taken away from “lock”; thus, no other thread is able to start until a token becomes available on “lock”. This only happens, when the active thread terminates.

Hence, only one thread is able to enter the critical region at a time.

Classroom exercise

2.3.Termination of Threads

A thread **terminates** when its **`run` method** has **terminated**, or in case of the “master” thread, when the main method has terminated (daemons are considered separately).

The class `Tread` has a method `isAlive` which can be used to check whether a thread is still living. Using this method, you could implement active waiting in the following way (but there is **never** a need to do it):

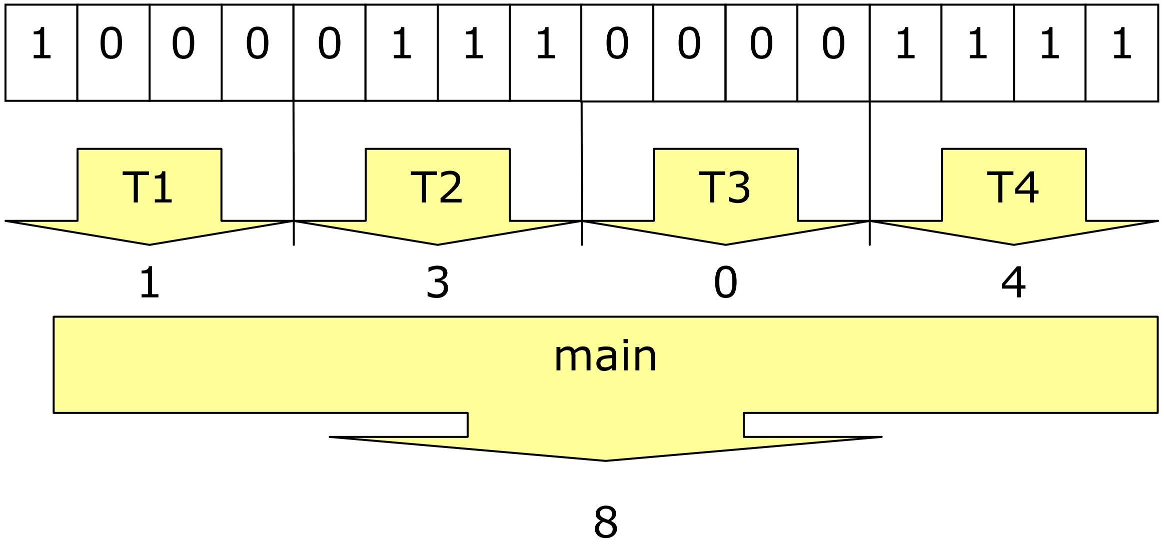
```
// MyThread is assumed to be a subclass of Thread
MyThread t = new myThread();
t.start();
while (t.isAlive())
    ;
// here we have: t.isAlive == false, thread t has terminated
```

In some **application** scenarios, it is necessary to **wait** until a **thread** has **terminated** its activities (because its results are needed to do further computations). For this purpose, the method **`join`** (of class Thread) can be used. A call of `join` terminates, if the corresponding thread has finalized its activities.

```
// MyThread is assumed to be a subclass of Thread
MyThread t = new myThread();
t.start();
t.join();  // block until t has terminated.
    ;
// here we have: t.isAlive == false, thread t has terminated
```

2.3.1. Using `join` to get results of thread computations

We consider an example where an array of Boolean is analyzed by several threads; each thread is responsible for a special range of the array.



Each thread's task is to count the true (1) values within its part of the array.

The main method accumulates the results.

```
$ cat AsynchRequest.java
class Service implements Runnable
{
        private boolean[] array;
        private int start;
        private int end;
        private int result;

        public Service(boolean[] array, int start, int end) {
                this.array = array;
                this.start = start;
                this.end = end;
        }

        public int getResult() {
                return result;
        }

        public void run() {
                for(int i = start; i <= end; i++) {    // count true values
                        if(array[i])
                                result++;
                }
        }
}
```

```java
public class AsynchRequest
{
        private static final int ARRAY_SIZE = 100000;
        private static final int NUMBER_OF_SERVERS = 100;

        public static void main(String[] args) {
                // start time
                long startTime = System.currentTimeMillis();

                // array creation, init with random boolean values
                boolean[] array = new boolean[ARRAY_SIZE];
                for(int i = 0; i < ARRAY_SIZE; i++) {
                        if(Math.random() < 0.1) array[i] = true;
                        else array[i] = false;
                }

                // creation of array for service objects and threads
                Service[] service = new Service[NUMBER_OF_SERVERS];
                Thread[] serverThread = new Thread[NUMBER_OF_SERVERS];

                int start = 0;
                int end;
                int howMany = ARRAY_SIZE / NUMBER_OF_SERVERS;
```

```java
// creation of services and threads
for(int i = 0; i < NUMBER_OF_SERVERS; i++) {
        end = start + howMany - 1;
        service[i] = new Service(array, start, end);
        serverThread[i] = new Thread(service[i]);
        serverThread[i].start(); // start thread i
        start = end + 1;
}

// wait for termination of each service (thread)
try {
        for(int i = 0; i < NUMBER_OF_SERVERS; i++)
                serverThread[i].join();
} catch(InterruptedException e) {
}

// accumulate service results
int result = 0;
for(int i = 0; i < NUMBER_OF_SERVERS; i++) {
        result += service[i].getResult();
}

// end time
long endTime = System.currentTimeMillis();
float time = (endTime-startTime) / 1000.0f;
System.out.println("computation time: " + time);
```

wait for termination of service thread no i.

```
                // print result
                System.out.println("result: " + result);
        }
}
$
```

Execution:

```
$ java AsynchRequest
computation time: 0.11
result: 9942
$ java AsynchRequest
computation time: 0.11
result: 9923
$ java AsynchRequest
computation time: 0.121
result: 10092
$
```

2.3.2. Termination of Threads stop()

The class `Thread` has a build-in method `stop()`. **`Thread.stop`** causes a thread to abruptly throw a `ThreadDeath` **exception** regardless of what it is doing.

Like `interrupt`, `stop` does not abort waits for locks or IO. But unlike `interrupt`, it is not strictly guaranteed to wait, `sleep` or `join`.

The usage of `stop` can be **dangerous**. Because `stop` generates asynchronous signals, activities can be terminated while they are in the middle of operations or code segments that absolutely must roll back or roll forward for consistency reasons.

This behaviour has been the reason for mark it being ***deprecated***.

Example:

```
class C {
	private int v;                  // invariant: v >= 0
	synchronized void f() {
		v = -1;                     // temporarily set to illegal value as flag
		compute();                  // possible stop point (*)
		v = 1;
	}

	synchronized void g() {
		while (v!= 0) {
			--v;
			something();
		}
	}
}
```

If a `Thread.stop` happens to cause termination at line (*), then the object will be broken: upon thread termination, it will remain in an inconsistent state because variable v is set to an illegal (negative) value.

Any call on the object from other threads might make it perform undesired or dangerous action: for example, here the loop in method g will spin "infinite" (`2*Integer.MAX_VALUE`) times as v wraps around the negatives.

Classroom exercise

2.4. wait und notify

`synchronized` methods can be used to **guarantee consistent states** of objects, even if a lot of threads share the object.

There exists application scenarios where consistency is **not** sufficient; in addition application specific conditions have to be fulfilled.

We demonstrate this implementing a **parking garage**. The actual **state** of a parking garage is defined by the **number** of **free** parking **places**. **Cars** are modelled by **thread** whereby a car can **`enter`** or **`leave`** the parking garage; each of these methods changes the actual state of the garage:

- When a car enters, the number of free places is decremented; leaving implies incrementing the free places.
- The number of free places can not be decremented, if the parking garage has become full (free places == 0)
- A parking garage can simultaneously be used by more than one car (each changing the state), therefore methods `enter()` and `leave()` have to be marked as `synchronized`.

First, we develop two **not satisfying** realization for our problem “free places”, **after** that, we show a **correct** solution.

2.4.1. Erroneous experiments

```
$ cat ParkingGarage1.java
class ParkingGarage {
   private int places;

   public ParkingGarage(int places) {
      if(places < 0)
         places = 0;
      this.places = places;
   }

   // enter parking garage
   public synchronized void enter() {
      while (places == 0);          // active wait
      places--;
   }

   // leave parking garage
   public synchronized void leave() {
      places++;
   }
}
$
```

This approach has two problems:

1. Active waiting -> performance!
2. The program is not working as it is desired when the parking garage has become full (places==0):

   a new car C1 enters and is in the while loop (waiting for a place); no other car is able lo leave, because the **lock** held be car C1 trying to enter will **never** be released.

The **origin** of the problem is (active) **waiting** for a free place **within** a **`synchronized`** **method** (`enter`). Thus, we try to **modify** the approach **waiting outside** a **synchronized** method.

```
$ cat ParkingGarage2.java
class ParkingGarage2 {
   private int places;

   public ParkingGarage2(int places) {
        if(places < 0)
             places = 0;
        this.places = places;
   }

   private synchronized boolean isFull() {
      return (places == 0);
   }
   private synchronized void reducePlaces() {
      places--;
   }

   // enter parking garage
   public void enter() {
      while (isFull() ); // active wait
      reducePlaces();
   }
   // leave parking garage
   public synchronized void leave() {
      places++;
   }
}
$
```

Method `enter` is now not `synchronized`, that means we do not wait within a `synchronized` method.

But this approach has other problems:

1. We still use active waiting
2. The shared object (`places`) is managed by **two** `synchronized` methods (`isFull`, `reducePlaces`). That cause the some problem we had in the bank example: a car can enter, if the scheduler switches the threads just after the while loop in enter, before `reducePlaces` is executed.

2.4.2. Correct solution without active waiting

As we saw, waiting for a free place is neither correct within a locked state (of the object `Park-ingGarage`) nor within an unlocked state.

Java offers methods of the class `Object` for **waiting** and **notification**.

`wait()`, `notify()` and `notifyAll` are methods of class `Object`:

```
public class Object {
    …
    public final void wait() throws InterruptedException {…}
    public final void notify() { …}
    public final void notifyll() { …}
}
```

All these methods **may be invoked only** when the synchronization **lock is held** on their targets. This, of course cannot be verified at compile time. Failure to comply causes these operations to throw an `IllegalMonitorStateException` at run time.

A **`wait`** invocation results in the following actions:

- If the current thread has been interrupted, then wait exits immediately, throwing an `InterruptedException`. Otherwise, (normal case) the current **thread** is **blocked**.
- The JVM places the **thread** in the internal and otherwise inaccessible **wait set** associated with the target object. (It is really a wait set, not a waiting queue).

- The synchronization **lock** for the target object is **released**, but all other locks held by the tread are retained.

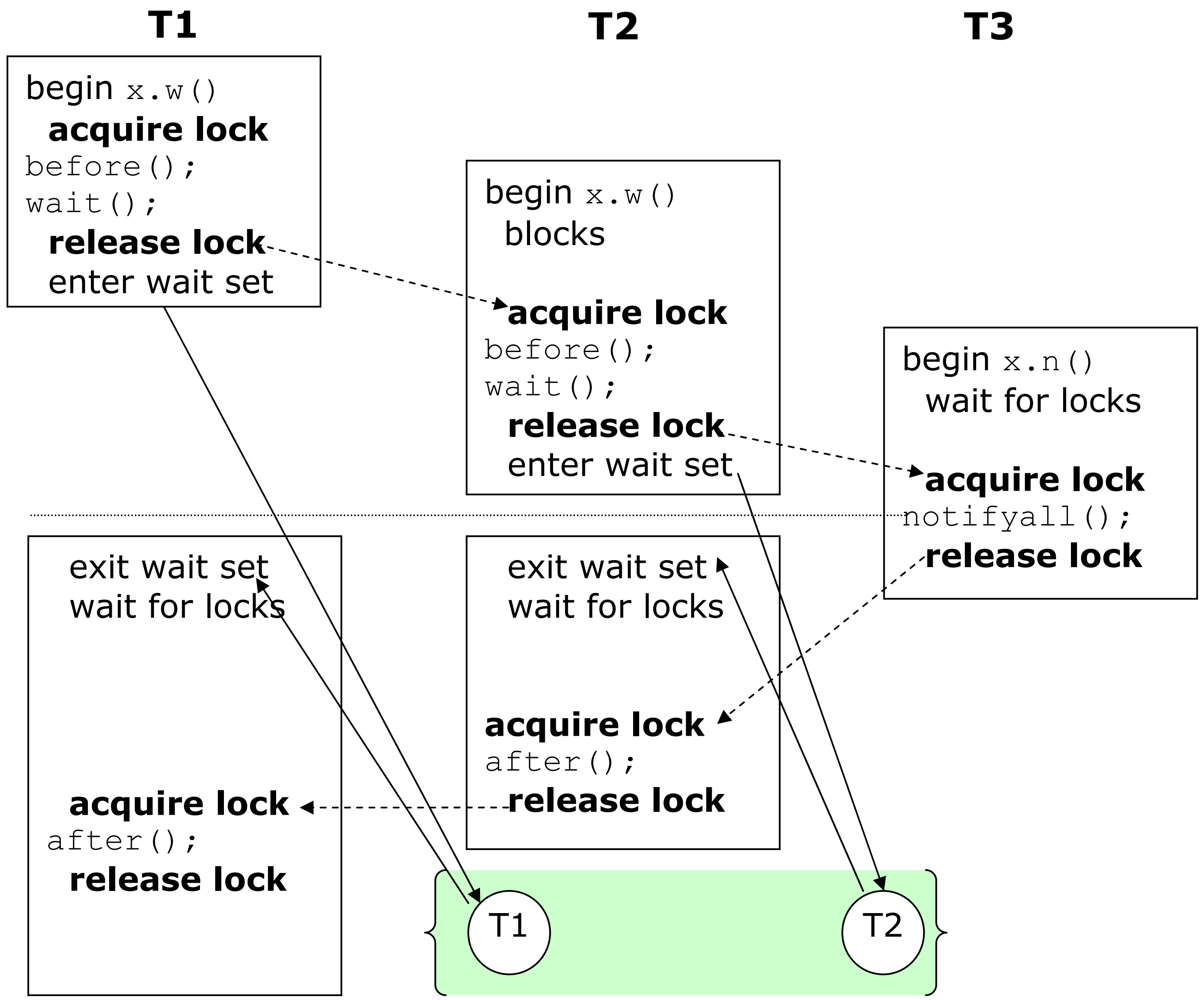
A **`notify`** invocation results in the following actions:

- If one exists, an **arbitrarily** chosen **thread**, say T, is **removed** by the JVM from the internal **wait set** associated with the target object. There is no guarantee about which waiting thread will be selected when the wait set contains more than one thread.
- T must re-obtain the synchronization lock for the target object, which will always cause it to block al least until the tread calling notify releases the lock. It will continue to block if some other thread obtain the lock first.
- T is then resumed from the point of its wait.

A **`notifyAll`** works in the same way as notify expect that the steps occur (in effect, simultaneously) for all threads in the wait set for the object. However, because they must acquire the lock, threads continue at a time.

The following picture illustrates some of the underlying mechanics, using the useless class X.

```
class X {
  synchronized void w() {
    before();// some actons
    wait();  // Thread.wait
    after(); // some actions
  }
  synchronized void n() {
    notifyall();// Thread.notify
  }
  void before {}
  void before {}
}
```



Using these concepts, we are able to find a solution for he parking garage problem:

- method `enter` uses `Thread.wait` instead of active waiting and
- method leave performs `Thread.notify` in order to let cars enter the parking garage.

```
$ cat ParkingGarageOperation.java
class ParkingGarage {
    private int places;

    public ParkingGarage(int places) {
        if (places < 0)
            places = 0;
        this.places = places;
    }

    public synchronized void enter() { // enter parking garage
        while (places == 0)          {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        places--;
    }

    public synchronized void leave() { // leave parking garage
        places++;
        notify();
    }
}
```

A car is a thread, where we let it drive (using `sleep()`) before entering the parking garage. We also use `sleep` to simulate the pause within the garage.

```java
class Car extends Thread {
    private ParkingGarage parkingGarage;

    public Car(String name, ParkingGarage p) {
        super(name);
        this.parkingGarage = p;
        start();
    }

    public void run() {
        while (true) {
            try {
                sleep((int)(Math.random() * 10000)); // drive before parking
            } catch (InterruptedException e) {}
            parkingGarage.enter();
            System.out.println(getName()+": entered");
            try {
                sleep((int)(Math.random() * 20000)); // stay within the parking garage
            } catch (InterruptedException e) {}
            parkingGarage.leave();
            System.out.println(getName()+": left");
        }
    }
}
```

Letting a parking garage become operational, we create a garage with 10 places and let 40 cars drive around parking and continue driving around.

```
public class ParkingGarageOperation {
    public static void main(String[] args){
        ParkingGarage parkingGarage = new ParkingGarage(10);
        for (int i=1; i<= 40; i++) {
            Car c = new Car("Car "+i, parkingGarage);
        }
    }
}
$
```

The operational garage in action is:

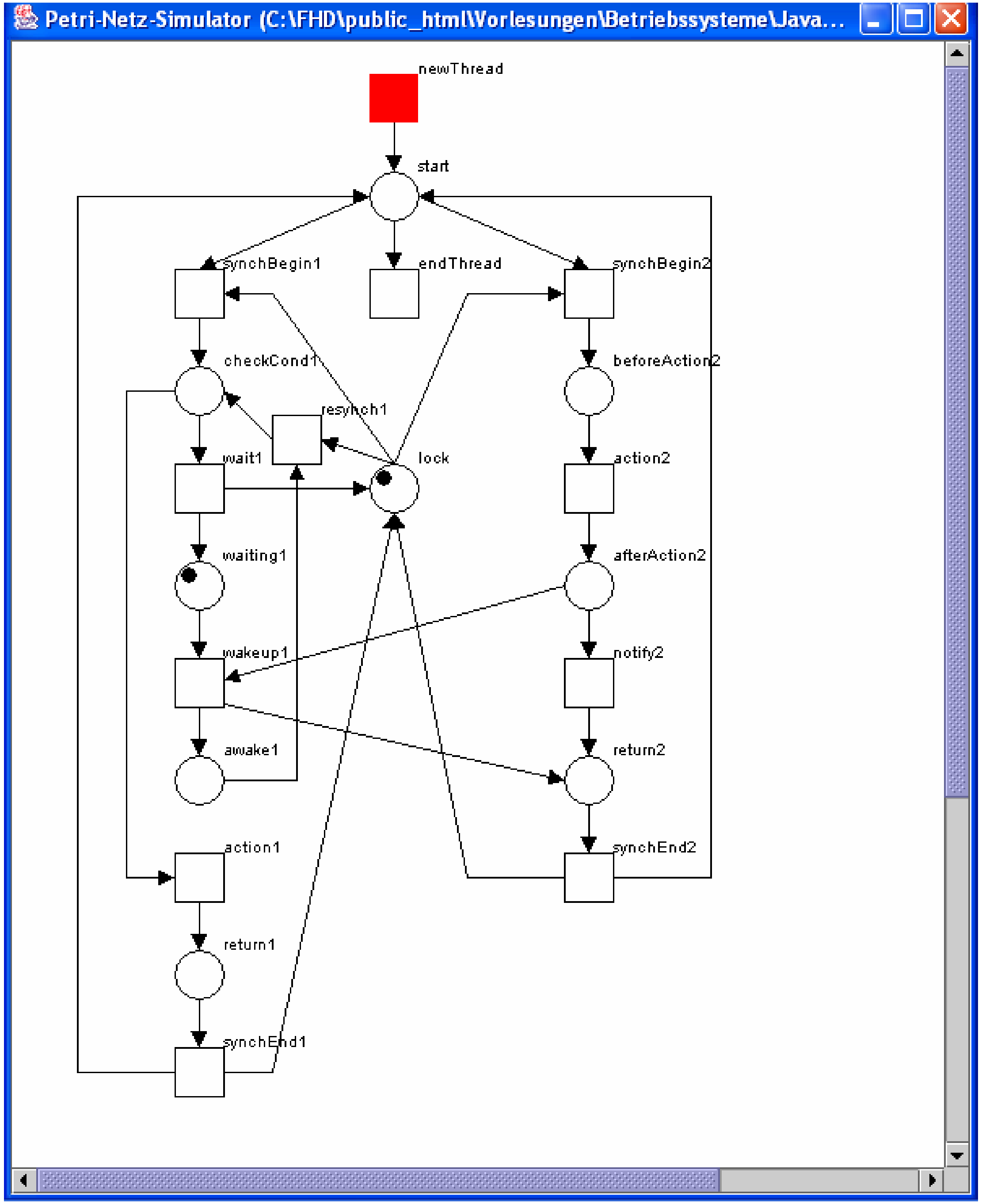
```
$ java ParkingGarageOperation
Car 38: entered
Car 21: entered
Car 12: entered
Car 22: entered
Car 23: left
Car 5: entered
Car 32: entered
Car 28: entered
Car 18: entered
Car 5: left
Car 37: entered
Car 22: left
Car 35: entered
….
```

2.4.3. wait and notify with Petri nets

To demonstrate the behaviour of wait and notify, we use a Java class and show the corresponding Petri net.

<pre>class K {
  public synchronized void **m1**() {
    while (…) {
      try {
        wait();
      } catch (
        InterruptedException e
) {}
    }
    action1();
  }
 	
  public synchronized void **m2**() {
    action2();
    notify();
  }
  private void action1() { … }
  private void action2() { … }
}</pre>
	<pre>class T extends **Thread** {
	    private K myK;
	    public T(K k) {
	       myK = k;
	    }
	
	    public void **run**() {
	       while (…) {
	          switch (…) {
	             case …: myK.m1(); break;
	             case …: myK.m2(); break;
	          }
	       }
	    }
	}</pre>

The Petri net for `m1` and `m2` looks like:



(`$ petrinet.ksh`)

State *checkCond1* correspond with the check of the wait condition. `wait` is modelled by *wait1*. The number of tokens corresponds with the number of waiting threads. Firing of transition *wait1* can be seen as releasing a lock, which means that further threads can call method `m1`.

The Petri net seems to be complex, but it is the semantics of notify that at least one thread may waked up. If there is no thread waiting then the invocation of notify rests without any effect. This semantics let the Petri net become a little bit complex.

Use the Petri net emulator, to see how it works!

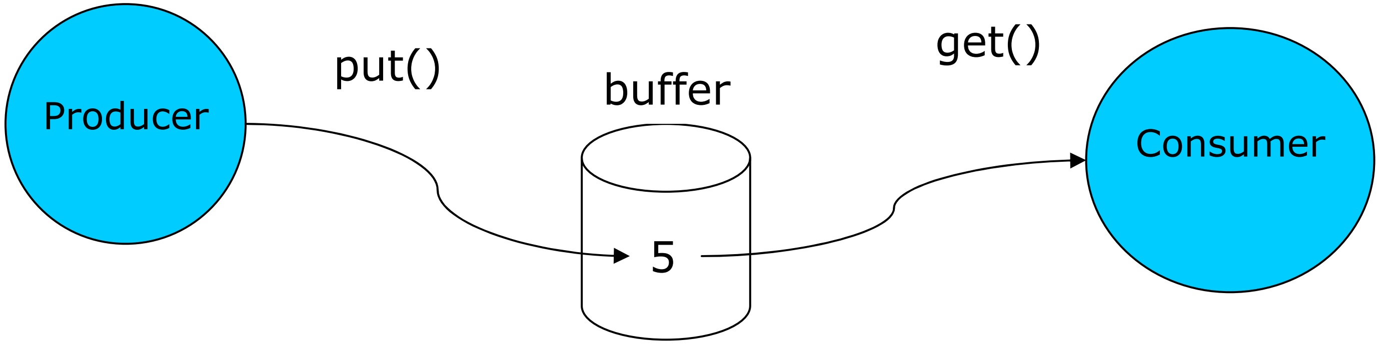
2.5. wait and notifyAll

While implementing concurrent applications using wait and notify, problems can occur when several threads are within the wait set and the **wrong thread** is **selected** by notify.

We consider this situation using the classical **producer consumer relationship**:

> A **producer** generates information and sends it to a **consumer**. Both, consumer and producer are realized as **threads**. As **communication device**, both partner use a shared **buffer**.

The buffer will be realized by an integer. The method `put` will be used by the producer to store a value into the buffer, consuming will be done by reading the content of the buffer by method `get`. This scenario is illustrated next:



The **implementation** has to **ensure** that **no value can be lost** (`put` before `get` has be done). Further, a value may **not be received twice** while consuming.

We try to realize the behaviour by wait and notify: after having written a value into the buffer, the producer waits until the consumer has notified and verse visa, the consumer waits until a value is available into the buffer.

2.5.1. First erroneous try

Class `Buffer` has private attributes `data` (for the value) and `available` (flag indicating availability of a value).

```
$ cat ProduceConsume.java
class Buffer {
        private boolean available = false;
        private int data;

        public synchronized void put(int x) {
                while(available) {   // wait until buffer is empty
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }
                data = x;
                available = true;
                notify();
        }

        public synchronized int get() {
                while(!available) { // wait until data available
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }
                available = false;
                notify();
                return data;
        }
}
```

Both, producer and consumer are implemented as threads. The constructor of each class has as parameter a reference to the shared object (Buffer). 100 values are transferred.

```
$ cat ProduceConsume.java
…
class Producer extends Thread {
        private Buffer buffer;
        private int start;

        public Producer(Buffer b, int s) {
                buffer = b;
                start = s;
        }

        public void run() {
                for(int i = start; i < start + 100; i++) {
                        buffer.put(i);
                }
        }
}
```

```
class Consumer extends Thread {
        private Buffer buffer;

        public Consumer(Buffer b) {
                buffer = b;
        }

        public void run() {
                for(int i = 0; i < 100; i++) {
                        int x = buffer.get();
                        System.out.println("got " + x);
                }
        }
}
```

Within the `main` method of the application class, one buffer and one producer and one consumer are created.

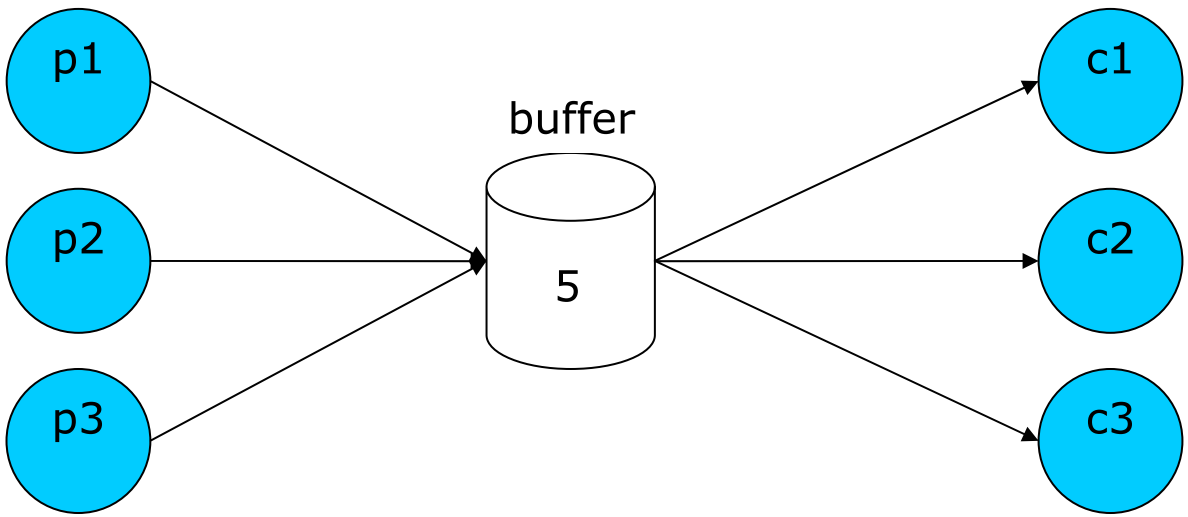
```
public class ProduceConsume {
        public static void main(String[] args) {
                Buffer   b = new Buffer();
                Consumer c = new Consumer(b);
                Producer p = new Producer(b, 1);
                c.start();
                p.start();
        }
}
$
```

Starting the program let us have the output:

```
$ java ProduceConsume
got 1
got 2
got 3
got 4
got 5
…
got 100
$
```

As we can see, **we have the desired behaviour**.

In real life situations, we find not only one producer and consumer. Rather, we have a scenario like the following:



To implement it, we just modify the main method of our last program by adding 2 consumers and 2 producers:

```
$ cat ProduceConsume2.java
…
public class ProduceConsume2
{
        public static void main(String[] args)
        {
                Buffer b = new Buffer();
                Consumer c1 = new Consumer(b);
                Consumer c2 = new Consumer(b);
                Consumer c3 = new Consumer(b);
                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 101);
                Producer p3 = new Producer(b, 201);
                c1.start();
                c2.start();
                c3.start();
                p1.start();
                p2.start();
                p3.start();
        }
}
$
```

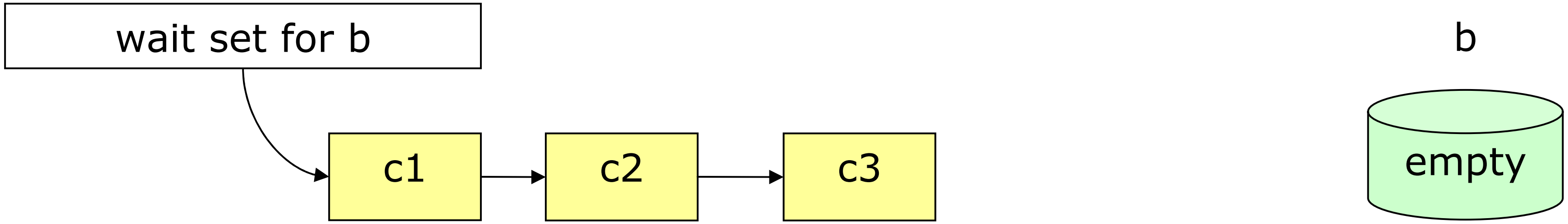
Starting the program produces the following output (depending on operating system and Java version)

```
$ java ProduceConsume2
got 1
got 101
got 2
got 102
got 103
got 201
got 3
got 104
got 202
…
got 230
got 231
got 33
got 8
got 232
```

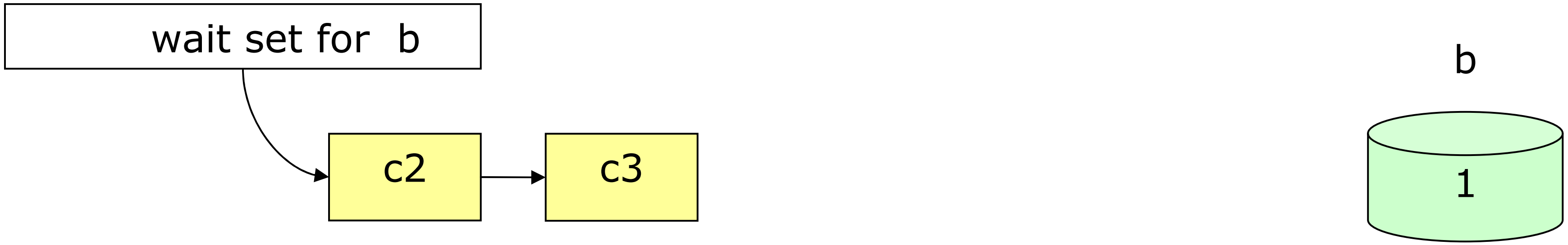
The program hangs, no further output is written, but the program in fact has **not** yet terminated!

The origin of this behaviour is the scheduling mechanism: a "**wrong**" thread has been chosen out of the wait set, as illustrated next.

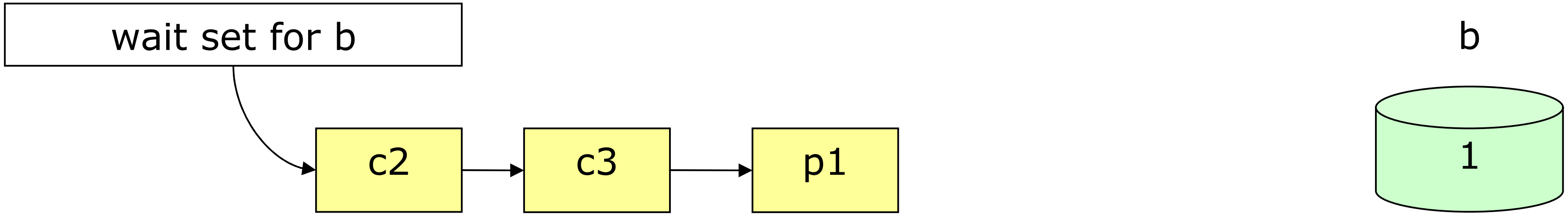
1. When the program starts, all consumer `c1`, `c2` and `c3` are able to run. Initially, `buffer` is empty. Therefore all consumers are blocked by `wait`.



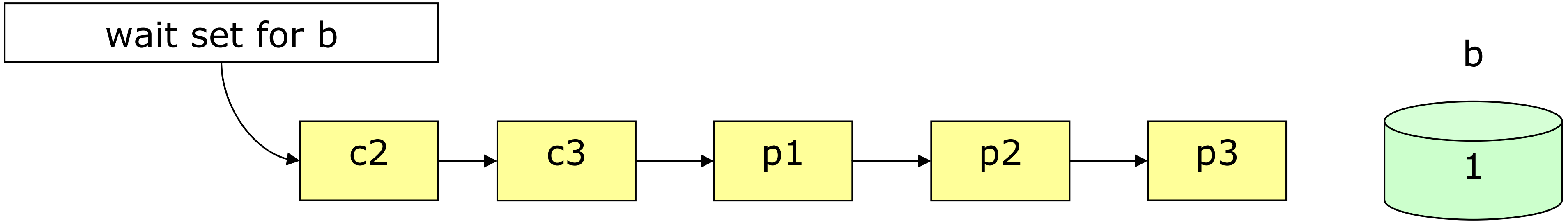
2. Now, thread `p1` puts a value into the buffer `b`, notifying a consumer; let's assume `c1`. The result is:



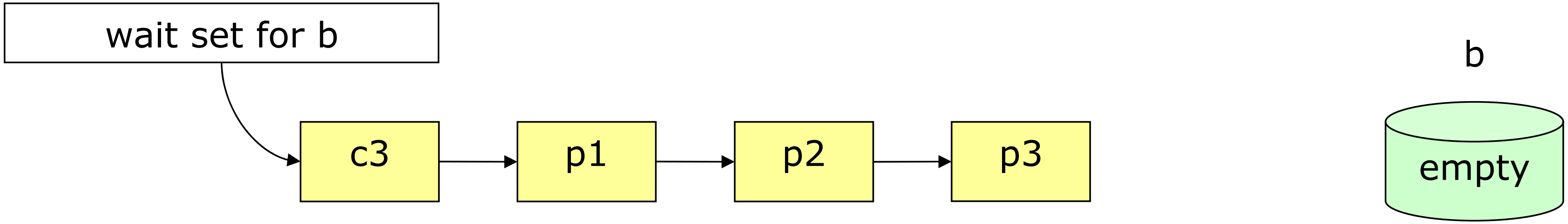
3. `p1` now tries to put an additional value into `b`, but the buffer is not empty. This let `p1` become blocked and it will be inserted into the wait set.



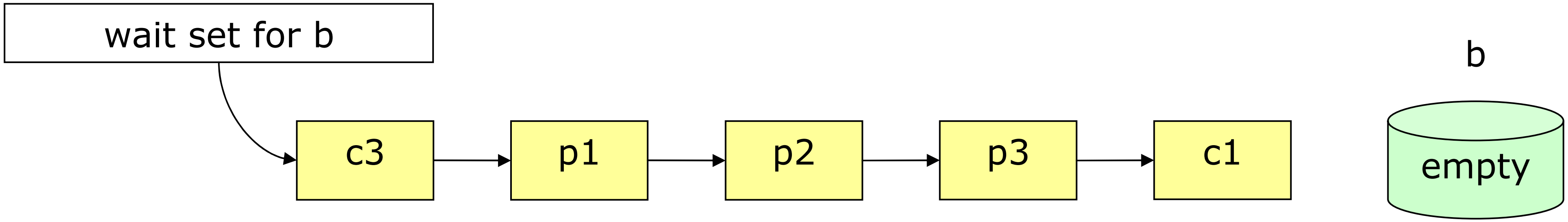
4. Let's assume, JVM switches to producer `p2`. Because buffer b is still not empty, `p2` will be inserted into the wait set. The same procedure happens for `p3`.



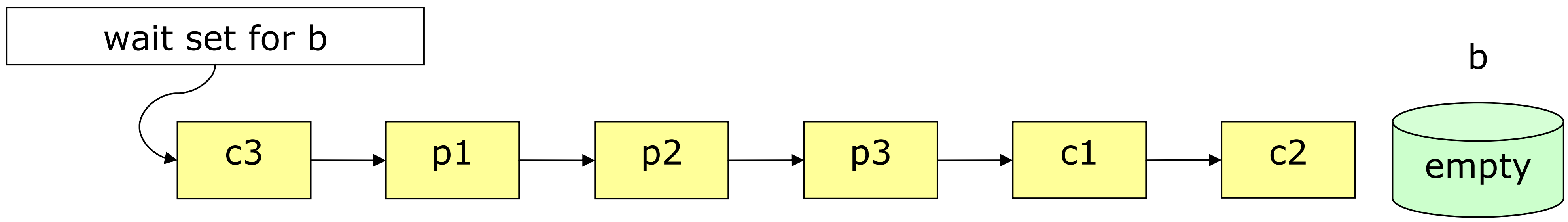
5. There is only one thread, which can do any action: `c1`. It consumes a value from buffer. Now, **exactly one** of the elements within the waiting set will wake up, let's assume `c2`.



6. `c1` continues, trying to get a value out of the buffer; `b` is empty, thus `c1` blocks and is inserted into the waiting set. Note b is still empty!



7. The only not blocked thread now is `c2`, a consumer. `c2` tries to get a value and blocks, the waiting set now holds all threads:



Now, we have the **hanging** situation: There is no tread able to run because each one is blocked (element of the waiting set).

Step 5 was responsible for the misleading situation: **consumer `c1` has waked up another consumer (`c2`)**. This was the ”wrong” thread.

The solution could be that not one thread wakes up exactly one other thread, instead one thread should be able to wake up all threads. This will work, because all threads are waked up, but only one is chosen and each thread tests in a while loop, whether it can continue to work.

2.5.2. Correct solution with `wait` and `notifyAll`

As we saw, the class `Object` has another method to wake up threads: `notifyAll` wakes up all threads within the waiting set of an object.

Thus, we replace `notify` by `notifyAll`:

```
$ cat ProduceConsume3.java
…
class Buffer {
        private boolean available = false;
        private int data;
        public synchronized void put(int x) {
                while(available) {
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }
                data = x;
                available = true;
                notifyAll();
        }
        public synchronized int get() {
                while(!available) {
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }
                available = false;
                notifyAll();
                return data;
        }
}
...
```

Basically, you can always use `notifyAll` instead of `notify`, but as we saw, the reversal is not always possible.

The following rule explains, when using which of the two notify methods:

Method `notifyAll` has to be used if at least one of the following situations take place:

1. Within the wait set, you find threads belonging to different wait conditions (for example buffer empty and buffer full). Using `notify` is dangerous, because the wrong thread could be waked up.

2. Modifying the state of an object implies that more threads are able to continue working (for example buffer changes from empty to full ⇒ all waiting threads can continue).

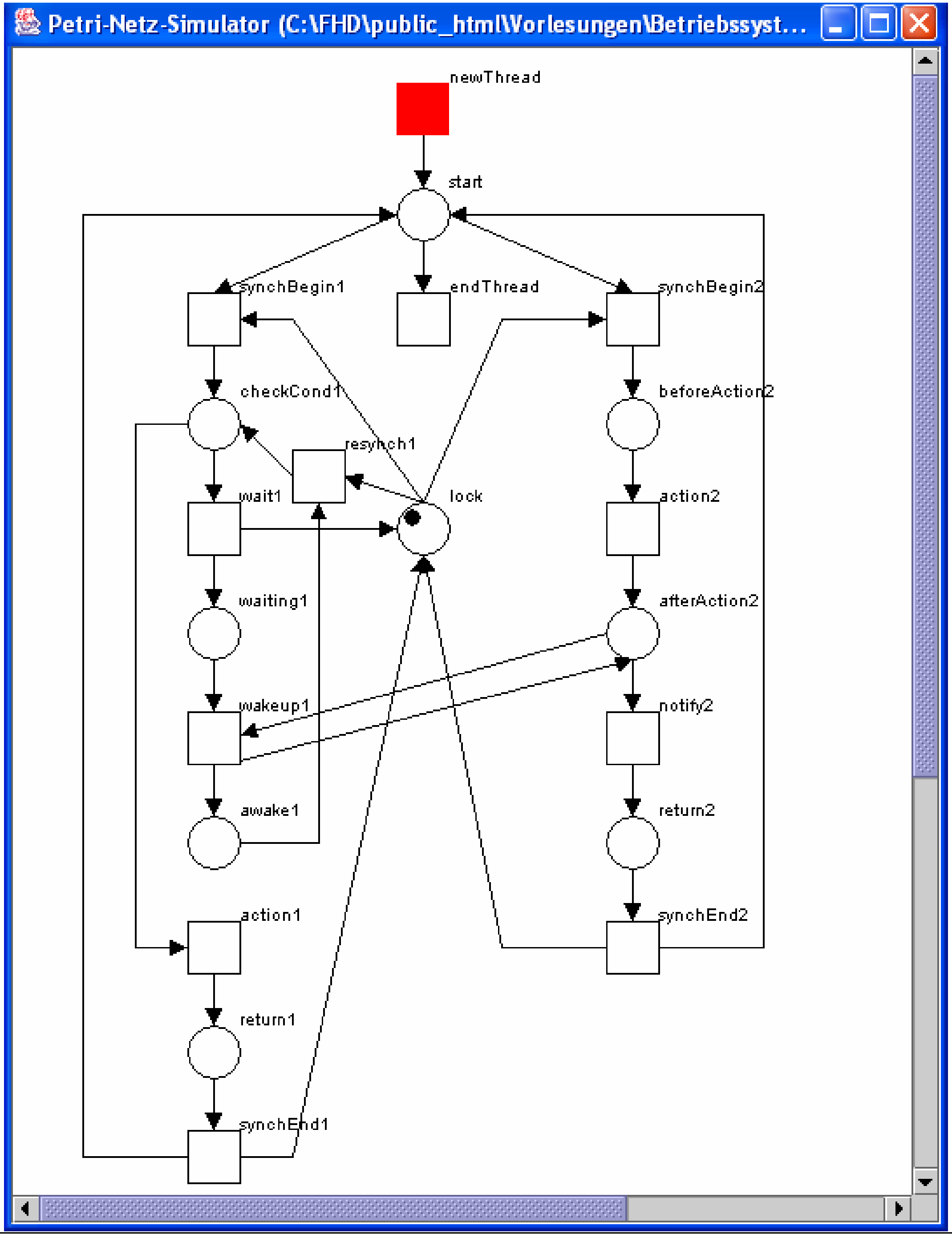
2.5.3. `wait` and `notifyAll` whith Petri nets

One again, we use a short Java program to illustrate, how `wait` and `notifyAll` can be simulated by a Petri net:

```
class K {
    public synchronized void m1() {
        while (…) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        action1();
    }
    public synchronized void m2() {
        action2();
        notifyAll();
    }
    private void action1() { … }
    private void action2() { … }
}
```

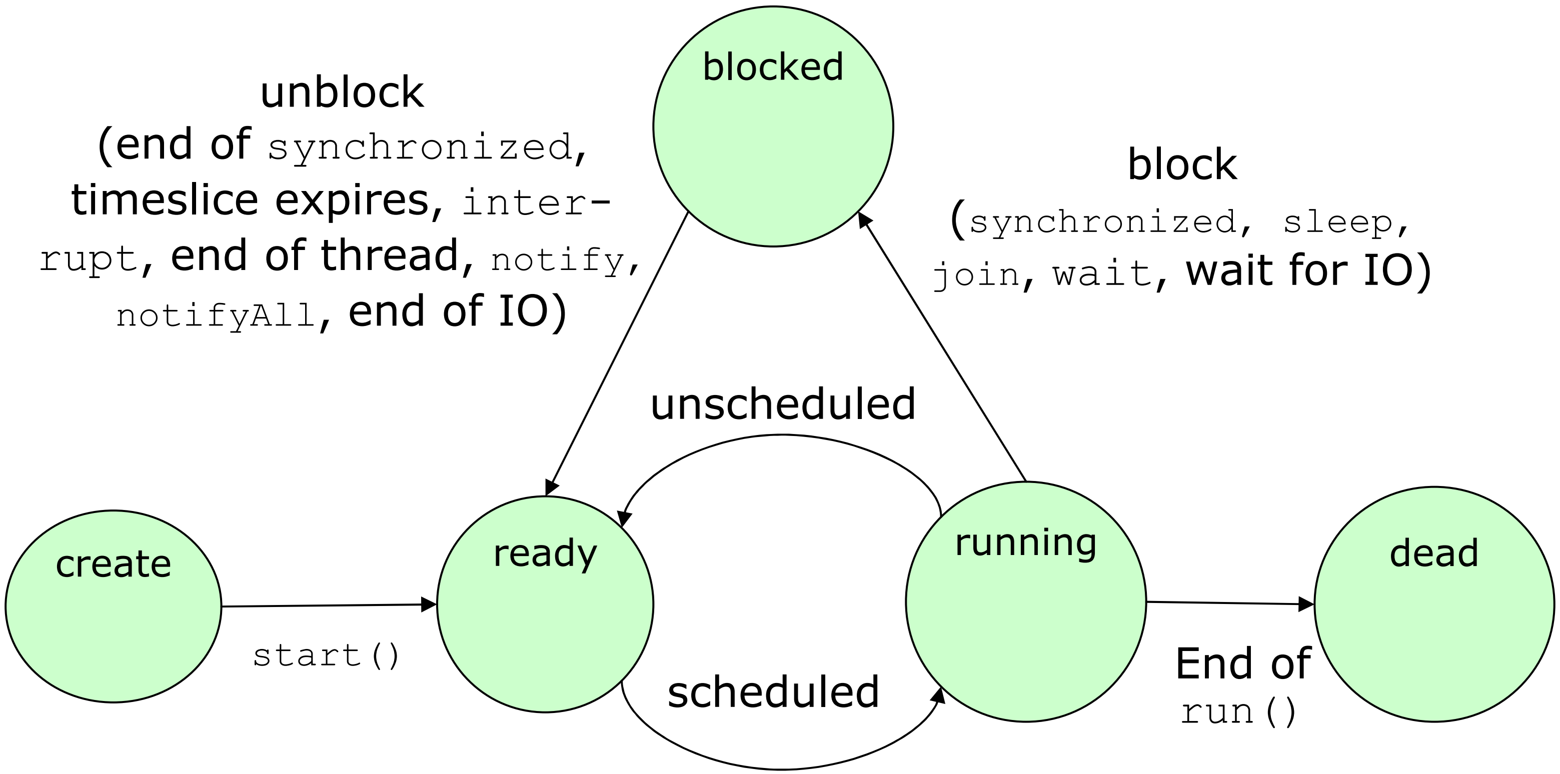
```
class T extends Thread {
    private K einK;
    public T(K k) {
        einK = k;
    }

    public void run() {
        while (…) {
            switch (…) {
                case …: einK.m1(); break;
                case …: einK.m2(); break;
            }
        }
    }
}
```

The following Petri net can be used to simulate invocations of method `m1` and `m2`.


Petri-Netz-Simulator (C:\FHD\public_html\Vorlesungen\Betriebssyst...
newThread
start
synchBegin1
endThread
synchBegin2
checkCond1
beforeAction2
resynch1
wait1
lock
action2
waiting1
afterAction2
wakeup1
notify2
awake1
return2
action1
synchEnd2
return1
synchEnd1

3. Scheduling

Java has build in construct, allowing a thread to capture one the following states:



The **JVM** together with the **scheduler** of the underlying operating system **are responsible** for the **transition** from one state to another.

After creation, a thread is not immediately executed; by `start`, it becomes ready for execution.

The scheduler switches threads from ready to running, assigning CPU. A thread leaves this state when

- the run method has terminated, the thread will not longer live;
- the time slice has expired, the thread become ready;

- a `sleep`, `wait`, `join` or IO operation or a `synchronized` method or block let the thread become blocked, the scheduler do not further consider that thread;

If the reason for blocking is not longer valid, the thread becomes ready.

3.1. Thread priorities

To make it possible to implement the JVM across diverse platforms, the **Java language** makes **no promise** about **scheduling** or **fairness**.

But threads do support priority methods that influence schedulers:

- Each **thread** has a **priority**, ranging between `MIN_PRIORITY` and `MAX_PRIORITY` (defined **1** and **10** respectively)
- By default, each **new thread** has the **same priority** as the **thread** that **created it**. The initial thread associated with `main` by default has priority `Thread.NORM_PRIORITY` (**5**).
- The current priority of any thread can be accessed via method `getPrioritya()`.
- The priority of any thread can be dynamically changed via method `setPriority()`. The maximum allowed priority for a thread is bounded by its `TheadGroup` (we see that into the next part).

When there are **more runnable** (ready) threads **than available CPUs**, a scheduler generally **prefers** threads with **higher priorities**. The exact policy may and does vary across platforms.

For example, some JVM implementations always select the thread with the highest current priority; some JVM implementations group priorities to classes and chose a thread belonging to the highest class.

Priorities have **no impact** on **semantics** or **correctness**. In particularly, priority manipulations **cannot** be used as a substitute for locking.

Priority should only be used to express the relative importance of different threads.

Example:

We construct an example, where 3 Threads run with different priorities, each performing only some outputs operations.

```
$ java ThreadPriority
                                                                  Thread Three:  Iteration = 0
                                                                  Thread Three:  Iteration = 1
                                                                  Thread Three:  Iteration = 2
                                                                  Thread Three:  Iteration = 3
                                                                  Thread Three:  Iteration = 4
Thread One:  Iteration = 0
Thread One:  Iteration = 1
Thread One:  Iteration = 2
Thread One:  Iteration = 3
Thread One:  Iteration = 4
                                Thread Two:  Iteration = 0
                                Thread Two:  Iteration = 1
                                Thread Two:  Iteration = 2
                                Thread Two:  Iteration = 3
                                Thread Two:  Iteration = 4
$
```

```
$ cat  ThreadPriority.java
import java.util.*;
import java.io.*;

class ThreadPriority {
  public static void main(String [] args )     {
      ThreadPriority t = new ThreadPriority();
      t.doIt();
    }

  public void doIt()     {
     MyThread t1 = new MyThread("Thread One: ");
     t1.setPriority(t1.getPriority() -1);   //Default Priority is 5
     t1.start();
     MyThread t2 = new MyThread("Thread Two: ");
     t2.setPriority(t2.getPriority() -2);
     t2.start();
     MyThread t3 = new MyThread("Thread Three: ");
     t3.setPriority(10);
     t3.start();
    }
}
```

```
class MyThread extends Thread  {
  static String spacerString ="";
  public String filler;

   public MyThread(String ThreadNameIn)     {
      filler = spacerString;
      spacerString = spacerString + "                                ";
      setName(ThreadNameIn);
    }

  public  void run() {
     for (int k=0; k < 5; k++) {
        System.out.println(filler + Thread.currentThread().getName()
                                  + " Iteration = " + Integer.toString(k) ) ;
       }
    }
 }
```

3.2. Thread interruption

Each thread has an associated `boolean` **interruption status**. Invoking `t.interrupt` for some `Threrad t` sets t's interruption status to `true`, unless t is engaged in `Object.wait`, `Thread.sleep`, or `Thread.join`; in this case interrupt causes these action (in t) to throw `InterruptedException`, but t's interruption status is set to `false`.

Example (sleep throws the exception):

```
public class TimeControl {
    Timer timer;

    TimeControl() {
        timer = new Timer();
        timer.start();
        try {
            Thread.sleep(10); // enough to init Timer
        } catch (Exception e) {}
        timer.interrupt();
    }

    public static void main(String args[]) {
        new TimeControl();
    }
}

class Timer extends Thread {
    public void run() {
        try {
            sleep(10000); // sleeps for a long time and catches InterruptedException
        } catch (InterruptedException e) {e.printStackTrace();}
    }
}
$
```

interrupt thread, created by starting timer.

catches InterruptedException

```
$ java TimeControl
java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at Timer.run(TimeControl.java:26)
$
```

The interruption status of any thread can be inspected using method `isInterrupted`.

```
public class TimeControl2 {
    Timer timer;
    TimeControl2() {
        timer = new Timer();
        timer.start();
        try {
            Thread.sleep(10); // enough to init Timer
        } catch (Exception e) {}
        System.out.println("TimeControl2 " + timer.isInterrupted());
        timer.interrupt();
    }
    public static void main(String args[]) {
        new TimeControl2();
    }
}

class Timer extends Thread {
    public void run() {
        try {
            sleep(10000);    // sleeps for a long time
        } catch (Exception e) {}
    }
}
```

```
$ java TimeControl2
TimeControl2 false
$
```

Classroom exercise

4. Background Threads

Until now, we only considered **foreground** threads. There are some activities, which are always performed within the **background**, for example the **garbage collection**.

We have to distinguish two kinds of threads:

- ☐ User threads (running in the foreground)
- ☐ Daemon Threads (running in the background).

A Java program has **terminated**, if all **user threads** are terminated.

Within the class `Thread`, we find two methods to influence the way a thread is running this way:

```
public class Thread {
    public final void setDaemon(Boolean on) {…}
    public final Boolean isDaemon() {…}
}
```

If we call the `setDaemon` method with `true` as argument, the thread will become a daemon; `false` as argument let the thread be a user thread (the default).

The **status** of a thread (user thread or daemon) does **not influence** the **scheduler**; for that, we can user priorities.

The status can only be set (by `setDaemon()`) after the thread has been created and **before** it has been started.

The status of the **main** thread can not be changed; it is always a **user thread**.