

Aufgabe 1 (12 Punkte)

Wenn ein Faden mehrere Methodenaufrufe für ein Objekt durchführen und dabei verhindern will, dass andere Fäden ihm dabei in die Quere kommen, sind Sperroperationen wie die im folgenden Beispiel von Nutzen:

```
Locking.lock(queue);
queue.append(x);
queue.append(y);
Locking.unlock(queue);
```

Damit wäre gesichert, dass x und y in der Schlange direkt nebeneinander stehen.

a) (4 P.) Entwickeln Sie eine Klasse, die diese Methoden bereitstellt:

```
class Locking {
    .....
    static void lock(Object x) throws DoubleLockException {...}
    static void unlock(Object x) throws NotOwnerException {...}
}
```

Überlegen Sie, ob Sie dafür `java.util.concurrent.locks.LockSupport` einsetzen wollen, und begründen Sie Ihre Entscheidung!

b) (8 P.) Entwickeln Sie verallgemeinerte Sperroperationen, die *beliebig viele* Objekte als Argumente akzeptieren, z.B.

```
Locking.lock(x, y, z); .....
Locking.unlock(x); .....
Locking.unlock(y, z);
```

Machen Sie dabei von Parameterlisten variabler Länge Gebrauch (neu in Java 1.5)! Erlauben Sie nur *konservative Sperren* (vgl. 3.1.6, Seite 10/11): wenn ein Faden Objekte gesperrt hat, darf er keine zusätzlichen Objekte sperren (Ausnahme melden!).

Achten Sie darauf, dass Ihre Implementierung „angenehme Fairness-Eigenschaften“ hat (das kann FCFS sein, muss aber nicht)! Erläutern und bewerten Sie die gewählte Ablaufsteuerung! [Zur Erinnerung: wenn ein Faden in `lock(x, y)` hängt, dürfen ihn nicht andere Fäden durch abwechselndes Sperren von x und y beliebig lange darin festhalten.]

Aufgabe 2 (12 Punkte)

Die *Simulation* realer Systeme ist ein typisches Anwendungsgebiet der nichtsequentiellen Programmierung, insbesondere in der *Militärinformatik*; dem mag man neutral oder nicht neutral gegenüberstehen – es ist jedenfalls instruktiv, sich mit Beispielen aus diesem Bereich zu befassen. Wir betrachten die folgenden Fragmente eines Simulationsprogramms:

```
class Vector3D {                                // 3-dimensional vector
    double x,y,z; .....
}
class BaseObject {                              // object in space
    .....
    void setPosition(Vector3D pos) { ..... }
    Vector3D getPosition() { ..... }
}
class MovingObject extends BaseObject { // autonomously moving object
    .....
    void setVelocity(Vector3D dir) { ..... }
}
abstract class SteeredObject extends MovingObject { //... self-steered
    .....
    private Vector3D calculateNewVelocity() { ..... }
}
abstract class LimitedFuel extends SteeredObject { //... limited fuel
    LimitedFuel(int initialFuel, int fuelConsumption) { ..... }
    private void fuelEmpty() {
        System.out.println(this + ": no fuel left");
    }
    .....
}
class Fighter extends LimitedFuel {           // fires missiles
                                                // towards other fighters
    Fighter(int missiles, ..... ) { ..... }
    private void notifyOfMissileImpact() {
        System.out.println("BOOM! "+myName+" downed. ");
    }
    void notifyOfEnemy(Fighter enemy) { ..... }
    .....
}
class Missile extends LimitedFuel {           // explodes next to target
    Missile(Fighter target, ..... ) { ..... }
    .....
}
```

Vervollständigen Sie dieses Programm unter Einsatz von JAC und führen Sie einfache Tests durch, z.B. so:

```

/** @jac.controlled */
Fighter jet1 = new Fighter(
    new Vector3D(0.0, 0.0, 0.0),
    new Vector3D(1.0, 1.0, 1.0),
    100, 1, 20, "Tomcat");
/** @jac.controlled */
Fighter jet2 = new Fighter(
    new Vector3D(10.0, 10.0, 0.0),
    new Vector3D(0.0, 0.0, 1.0),
    100, 1, 20, "IceMan");

jet1.notifyOfEnemy(jet2);

/** @jac.controlled */
Missile mis1 = new Missile(           // ground-air missile
    new Vector3D(20.0, 20.0, 20.0),
    new Vector3D(-3.0, -3.0, -3.0),
    100, 5, jet1);
/** @jac.controlled */
Missile mis2 = new Missile(           // ground-air missile
    new Vector3D(10.0, 10.0, 20.0),
    new Vector3D(0.0, 0.0, -1.0),
    100, 5, jet2);

```