Concepts of Object-Oriented Programming

Prof. Dr. Peter Müller

Chair of Programming Methodology

Exercises 10: Concurrency

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

java.util.concurrency

- Since Java 1.5
- Atomic operators
  - x.getAndAdd(2) guarantees no data races
- Concurrent Collections
  - Provides collections for use in multi-threaded programs
- Synchronizers
  - Semaphore and other common paradigms
- Executors
  - For custom thread-like systems

Atomic Operations – No “synchronized”

```
import java.util.concurrent.atomic.*;
class Even {
 AtomicInteger x;
 public Even () {
    x = new AtomicInteger ();
 }

 public void next () {
 // No need to lock x!
    x.getAndAdd (2);
 }}
```

Blocking Queue -- Producer/Consumer

```
class Producer implements Runnable {
 BlockingQueue queue;

 public Producer (BlockingQueue q) {
 this.queue = q;
 }
 public void run () {
 while (true)
    // No need for blocking.
    queue.put (new Object ());
 }}
```

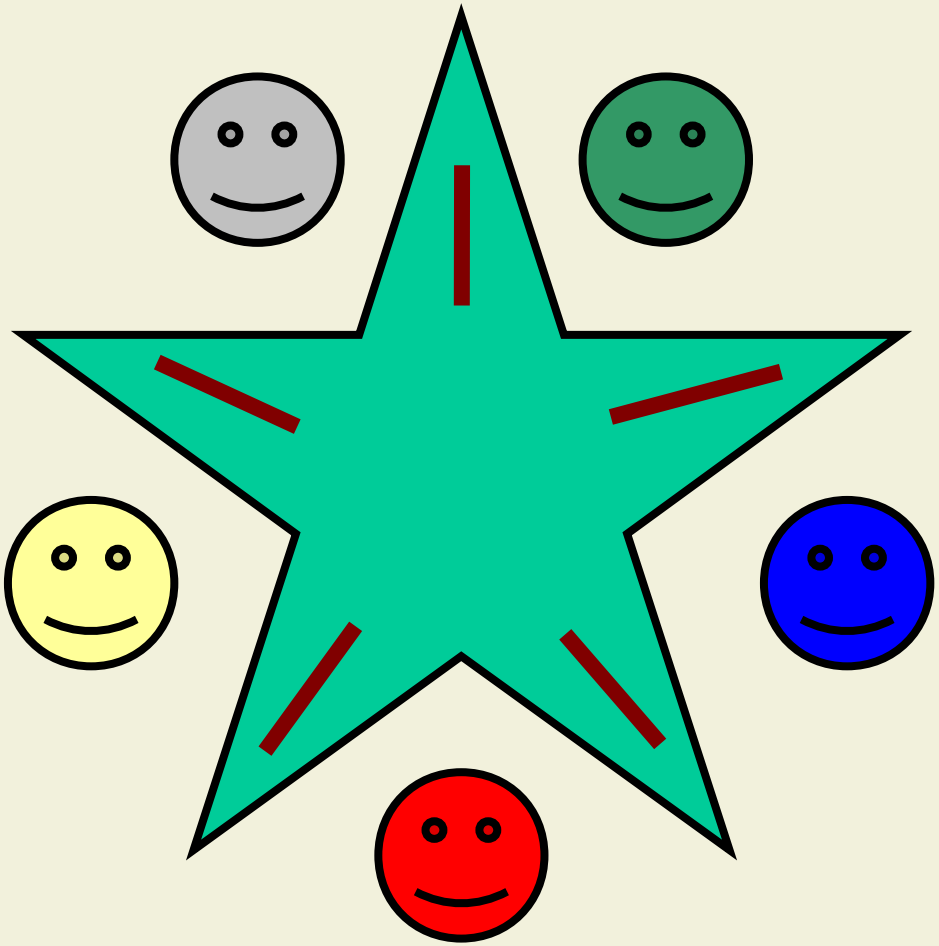
Blocking Queue -- Producer/Consumer

```
class Consumer implements Runnable {
 BlockingQueue queue;

 public Consumer (BlockingQueue q) {
 this.queue = q;
 }
 public void run () {
 while (true)
    // No need for blocking
    Object o = queue.take();
 }}
```

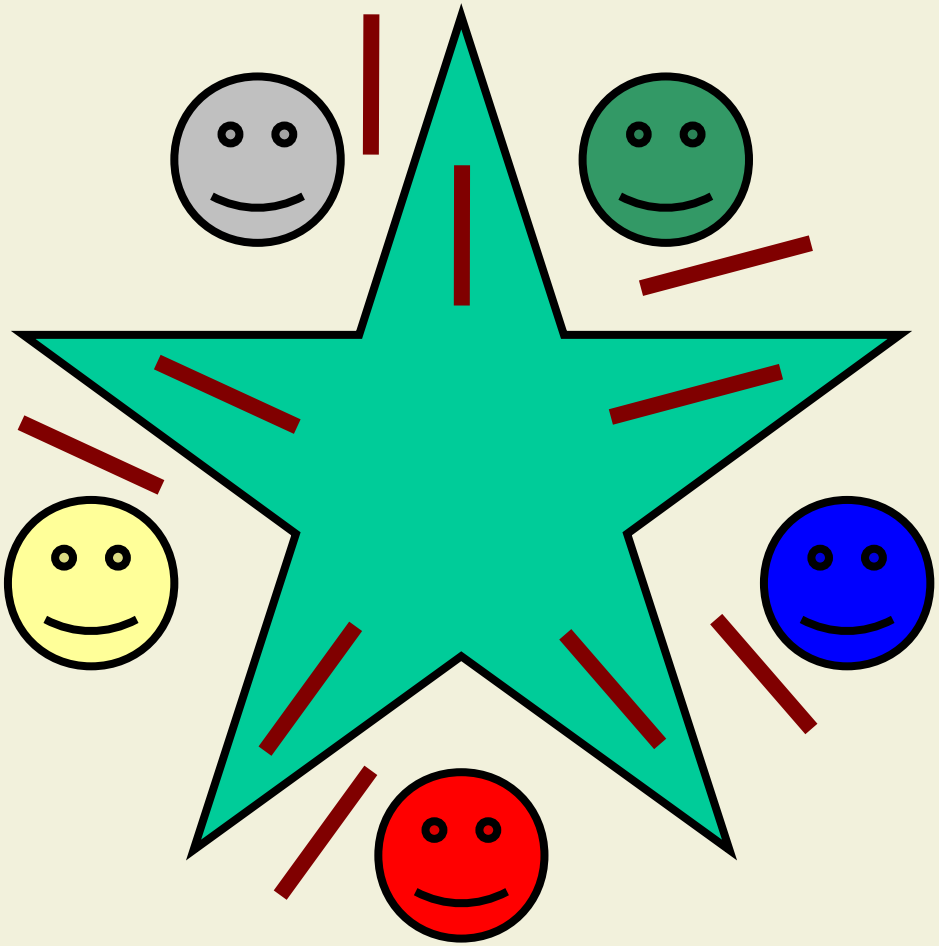
Dining Philosophers Problem

- n Philosophers
- n Chopsticks
- 2 Chopsticks needed for eating



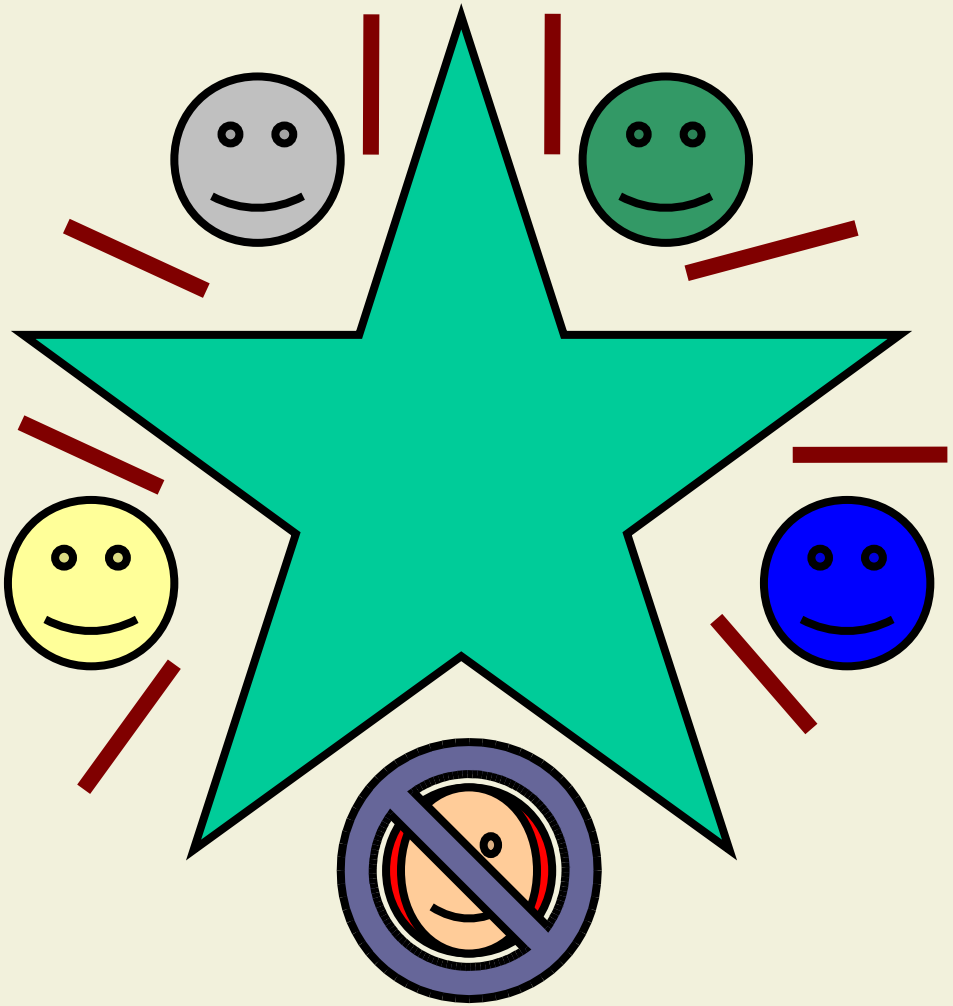
Problem: Deadlock

Everybody picks
the left chopstick
and then waits
forever to get
the right chopstick.



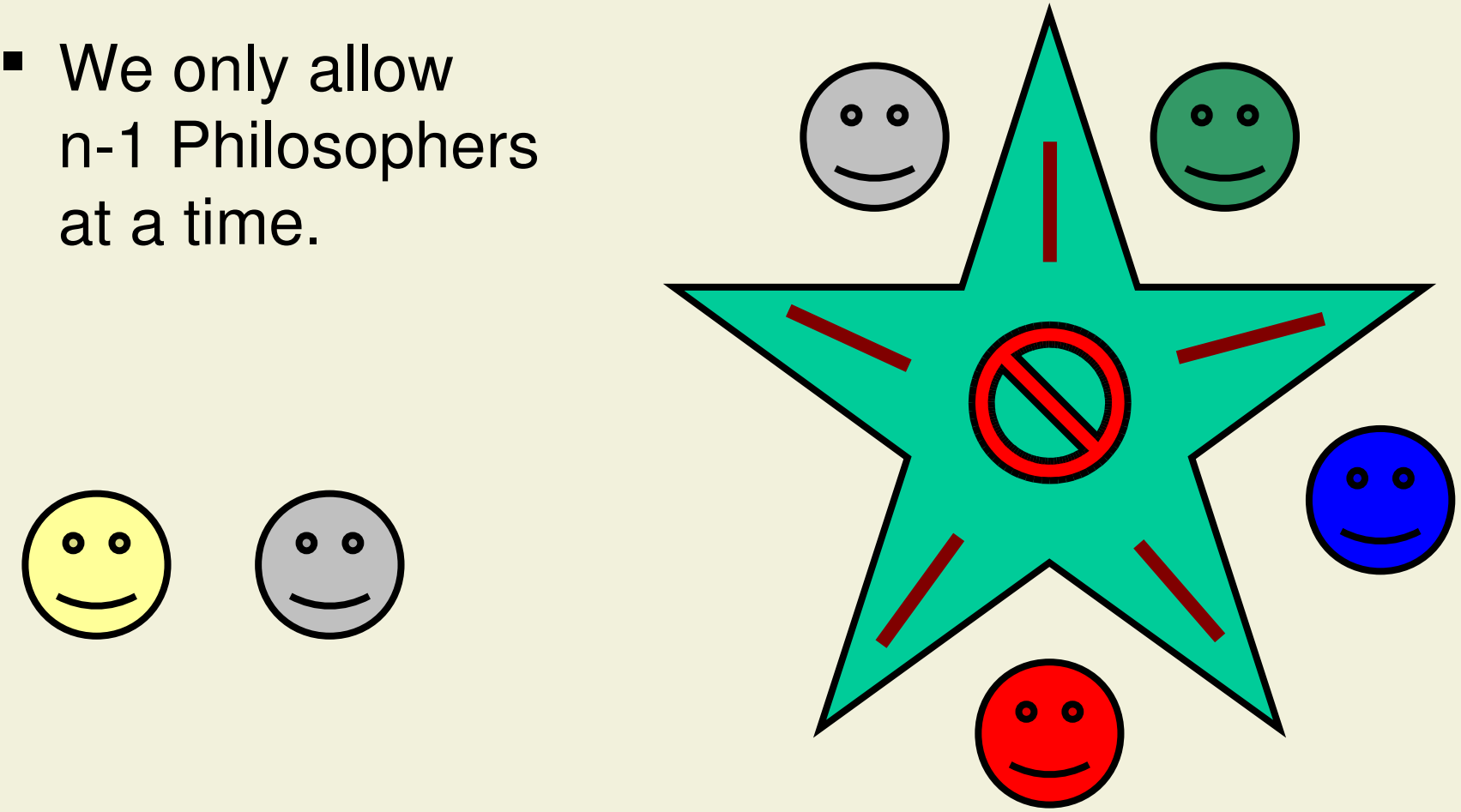
Problem: Starvation

Some philosophers Never get a chance to pick up both chopsticks and “starve”.



Solution 1: Central Control

- We only allow n-1 Philosophers at a time.



Solution 1: The Philosopher

```
public class Philosopher extends Thread {
   Table t;
  public void run() {
    while( true ) {
      think();

      t.sit(id);
      t.chopstick[id].pick();
      t.chopstick[(id+1)%t.n].pick();

      eat();

      t.chopstick[id].put();
      t.chopstick[(id+1)%t.n].put();
      t.leave(id);
    }}}
```

Solution 1: The Chopstick

```
public class Chopstick {
  public boolean used;

  public synchronized void pick() {
    if( used ) try { wait(); }
        catch( InterruptedException e ) {};
    used = true;
  }

  public synchronized void put() {
    used = false;
    notify();
  }
}
```

Solution 1: The Table

```
public class Table {
  public Chopstick[] chopstick;
  private int phil;
  Philosopher[] phils;
  int n; //size of table

  public synchronized void sit(int id) {
    if( phil >= n ) try { wait(); }
        catch( InterruptedException e ) {}
    phil = phil + 1;
  }

  public synchronized void leave(int id) {
    phil = phil - 1;
    notify(); }
```

Solution 2: Ordered Resources

- We demand that the chopsticks are picked up in a specific order
- For one philosopher this means that he will pick up the right chopstick first
- All other philosophers pick up the left chopstick first.
- The deadlock situation of all philosophers picking up the left chopstick at the same time is prevented

Solution 2: The Philosopher

```
public void run() {
  int first, second;

  first = id < ((id+1)%t.n) ? id : (id+1)%t.n;
  second = id<((id+1)%t.n) ? (id+1)%t.n : id;

  while (true) {
     think();
     t.chopstick[first].pick();
     t.chopstick[second].pick();


     eat();
     t.chopstick[first].put();
     t.chopstick[second].put();
  }
}
```

Solution 3: Atomic Operation

- We only allow both chopsticks to be picked up together
- Needs a critical section for the atomic operation
- Careful with what objects are used for synchronization

Solution 3: The chopstick

```
public class Chopstick {
  public boolean used;

  public Chopstick() { used = false; }

  public synchronized void pick() {
      used = true;
  }

  public synchronized void put() {
      used = false;
  }
}
```

Solution 3: The Philosopher

```
public void run() {
  while( true ) {
    think();

    synchronized( t ) {
      while( t.chopstick[id].used ||
             t.chopstick[(id+1)%t.n].used ) {
        try { t.wait(); }
      catch( InterruptedException e ) {};
      }
      t.chopstick[id].pick();
      t.chopstick[(id+1)%t.n].pick();
    }
```

Solution 3: The Philosopher

```
    …

    eat();

    synchronized( t ) {
      t.chopstick[id].put();
      t.chopstick[(id+1)%t.n].put();

      t.notifyAll();
    }
  }
}
```

Entering the shower

```
class Shower {
  static int women=0, men=0;

  synchronized static void enter(
                                Person p) {
    while (women+men == 10)
      Shower.class.wait();

    if (p instanceof Man)
      men = men+1;
    else women=women+1;
  }
```

Leaving the shower

```
synchronized static void leave(Person p)
{
  if (p instanceof Man)
    men=men-1;
  else
    women=women-1;

  Shower.class.notifyAll();
}
```

Entering the shower

```
synchronized static void enter(Person p)
{
  while ( man+women == 10 ||
     (p instanceof Man && women>0) ||
     (p instanceof Woman && men>0)
        )
    Shower.class.wait();

  if (p instanceof Man)
    men = men+1;
  else women=women+1;
}
```

Entering the shower – Fairness (almost)

```
int vmen; int vwomen;
synchronized static void enter(Person p)
{
  while (... || (p instanceof Man &&
  vmen >= 20) || (p instance of Woman &&
  vwomen >= 20))
    Shower.class.wait();

  if (p instanceof Man) {
    vmen = vmen+1;
     vwomen = 0;
  else ...
}
```

Questions?