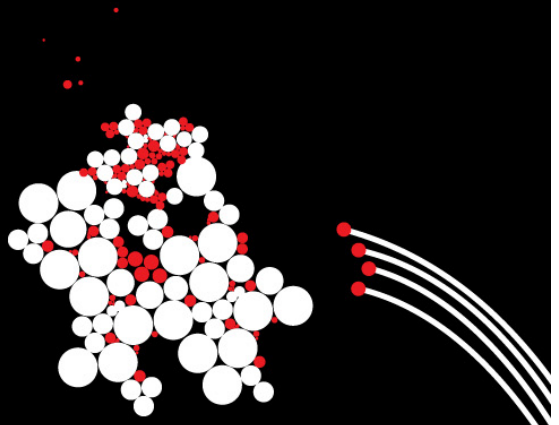
UNIVERSITY OF TWENTE.

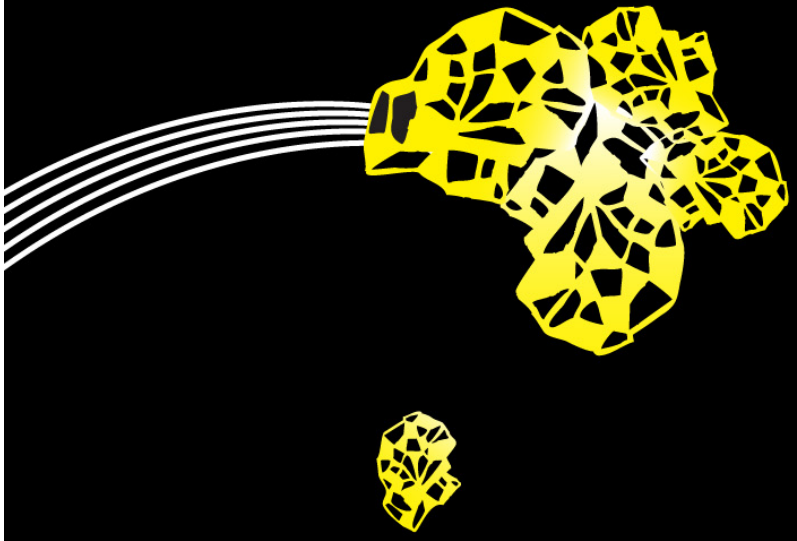


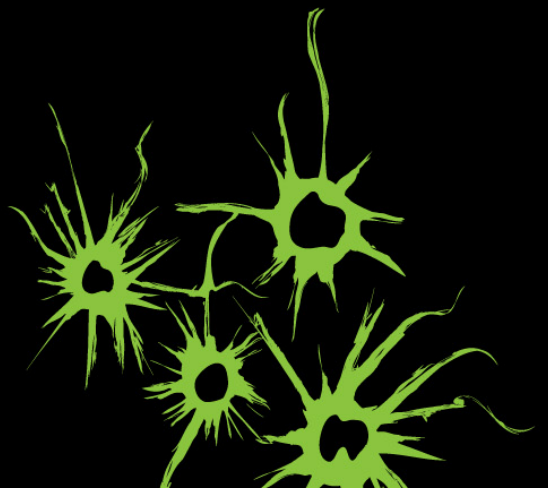
CONCURRENT PROGRAMMING – BLOCK 2

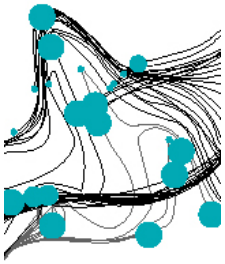
THREADS, INTERLEAVING, SHARING, SAFETY

MODULE 8: PROGRAMMING PARADIGMS

1 MAY 2018







LECTURE 1

- Overview & Organisation
- Recap: what do you remember about concurrent programming?
- Why concurrency, and why teaching it?
- Atomicity
- Thread safety
- Immutability
- Testing concurrent programs

Literature:

JCIP, Chapters 1 – 3 and 12

CONCURRENT PROGRAMMING

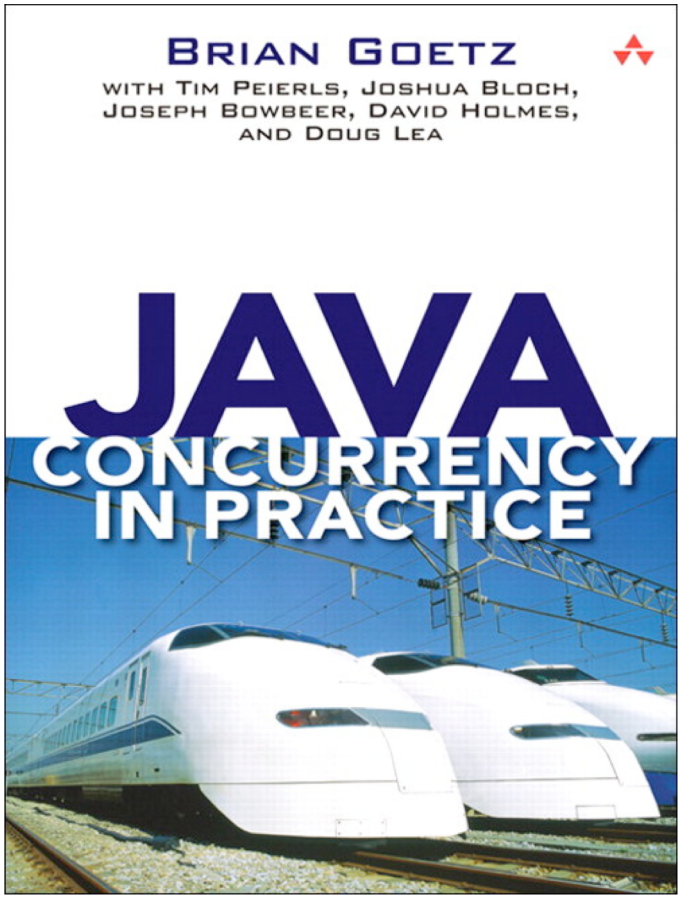
CONTENTS

2	Tue 30/4	Basic of concurrency, thread safety, testing concurrent systems	Ch. 1-3, 12 (+ SS material)
3	Tue 7/5	Synchronisation	Ch. 4, 5, 13, 14.1-14.4
4	Wed 15/5	Liveness, performance, and fairness	Ch. 10,11
5	Thu 23/5	Homogeneous threading (OpenMP, OpenCL)	Papers
6	Mon 3/6	Safe concurrency (Software Transactional Memory, Rust)	Papers
7	Fri 7/6	Fine-grained concurrency, memory models	Ch. 14.5-6, 15, 16

OBJECTIVES

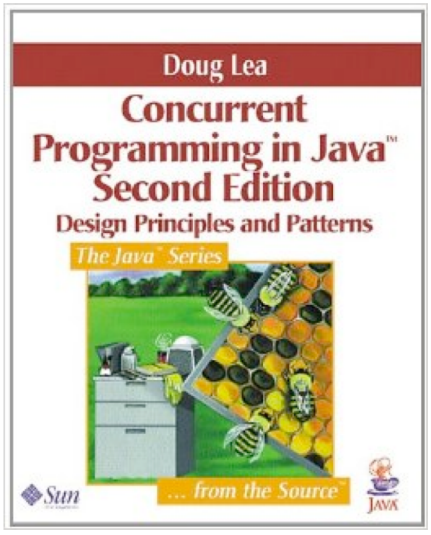
- This is not a course *Concurrent Programming in Java*
- Focus:
  - What can go wrong when you make programs concurrent
  - What are common mistakes (anti-patterns)
  - What are good examples (patterns)

MATERIAL BLOCKS 2 – 4 AND 7



Block 5 and 6: papers available from Canvas

Recommend additional reading.



LECTURER

- Prof.Dr. Marieke Huisman
  Zilverling 3055
  M.Huisman@utwente.nl

Research:

Run-time and static verification of concurrent software

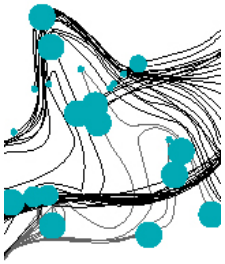
- VerCors project: verification of concurrent data structures, using permission-based separation logic
- CARP project: correct and efficient accelerator programs
- Mercedes project: maximal reliability of concurrent and distributed software

ORGANISATIONAL STRUCTURE

- Every block (5 consecutive teaching days):
  - Lecture (2 hours)
  - Lab sessions (4 hours)
  - Self-study (4 hours)
- Sign-off exercises ☞ mandatory
- Some exercises: challenging, open-ended, often more answers possible

EXAMINATION

- Written exam, on Chromebooks
- Example tests available from Canvas
- Mandatory diagnostic test: May 27
- Exam
  - Q&A: Thu, June 20
  - 1st chance: Fri, June 21 (Block 8)
  - Resit: Thu, July 4 (Block 10).



DIAGNOSTIC TEST

So let’s see what you remember about concurrency

CONSIDER QUESTION 6 SS EXAM 2015-2016

- Exercises modelling a supermarket with online ordering system
- Read the exercise text

QUESTION 1

What are the possible values printed by the main method?

a. 4 and 7
b. 10 and 13
c. 1, 4 and 7
d. 1
e. 1 and 4
f. 1 and 7

QUESTION 2

Suppose we remove the try-block

```
try {
   alice.join();
   bob.join();
   }
catch (InterruptedException e) { }
```

a. 4 and 7
b. 1 and 7
c. 1, 4 and 7
d. 1
e. 7
f. 1 and 4

What are the possible printed values?

QUESTION 3

Suppose we change the run method:

```
public void run() {
    synchronized (this) {
        my_order.placeOrder();
    }
}
```

What are the possible printed values?

a. 4 and 7
b. 1 and 7
c. 1, 4 and 7
d. 1
e. 7
f. 1 and 4

QUESTION 4

Suppose we change the run method:

```
public void run() {
    synchronized (my_order) {
        my_order.placeOrder();
    }
}
```

What are the possible printed values?

a. 4 and 7
b. 1 and 7
c. 1, 4 and 7
d. 1
e. 7
f. 1 and 4

QUESTION 5

Which values could be printed if Alice also ordered another product?

a. 4 and 7
b. 1 and 7
c. 1, 4 and 7
d. 1
e. 7
f. 1 and 4

WHAT IS CONCURRENCY?

- Sequential executions: single-threaded
- Stand-alone programs

- Multi-threaded: multiple executions in parallel
- Communication via shared data or message passing

- *In fact: graphical user interfaces are always multithreaded – interface runs in own thread to avoid blocking main application*

http://www.automatiseringgids.nl/slides/2015/q2/50-jaar-wet-van-moore

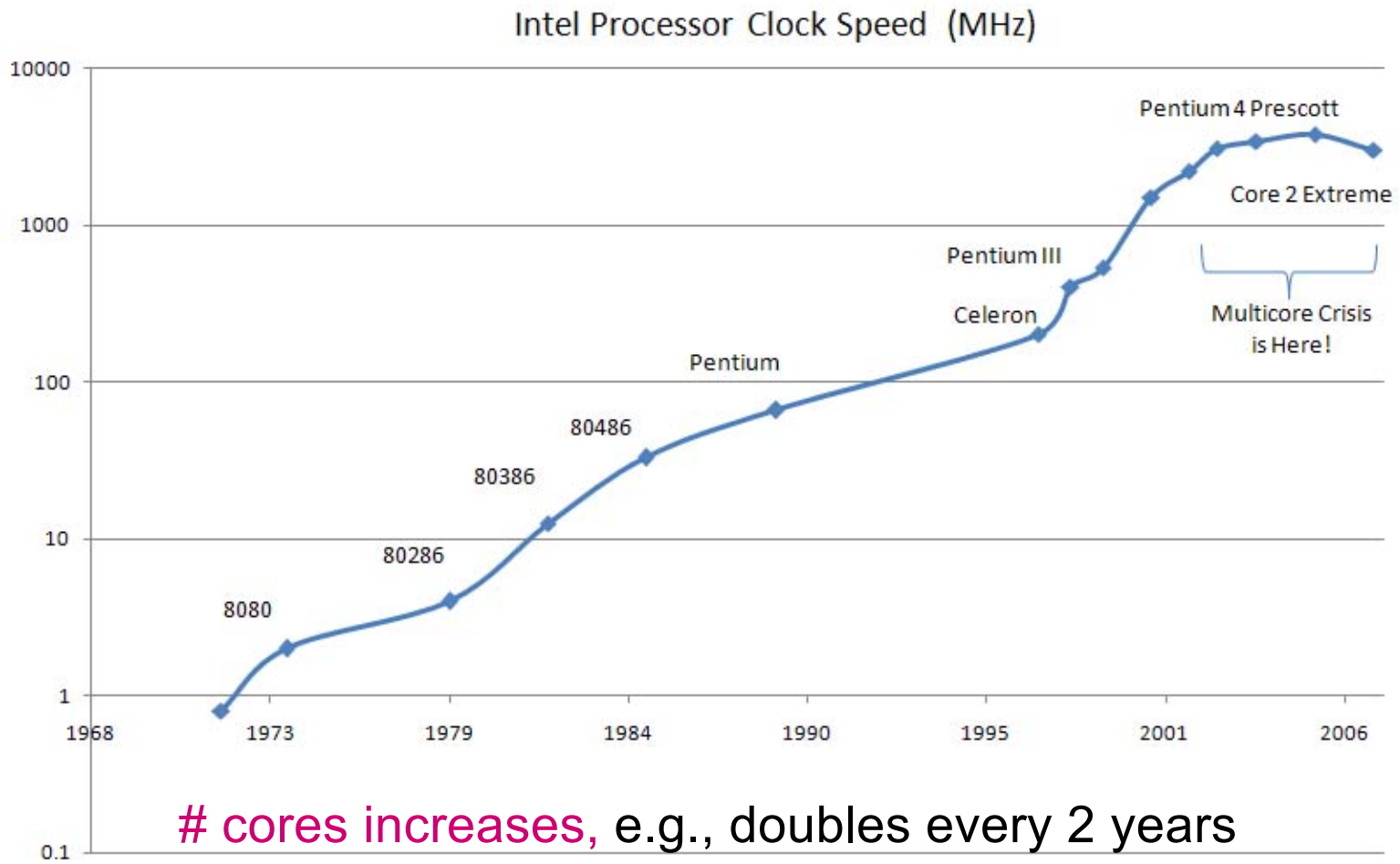
MOORE'S LAW

"Transistor count in integrated circuits doubles every two years."

Gordon E. Moore (1965)

without Moore's law

LIMITS TO CLOCK SPEED

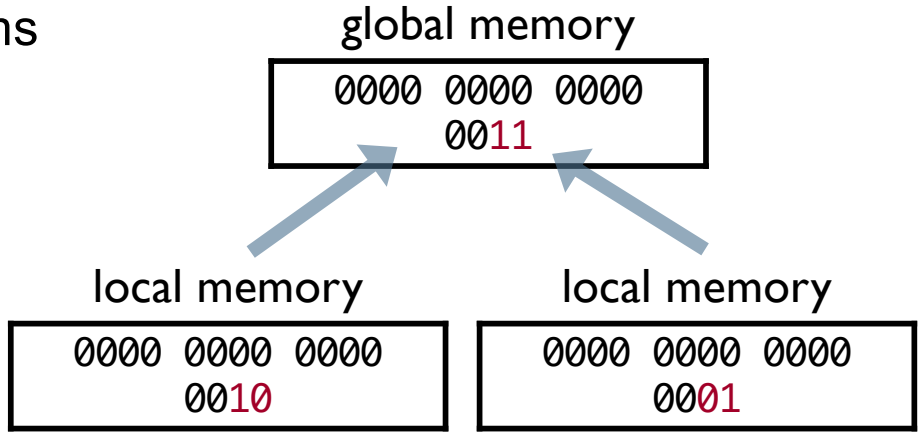


cores increases, e.g., doubles every 2 years

Software parallelism must also double every 2 years

HOW TO SUPPORT CONCURRENCY?

- Multitasking systems
  - context switch is much coarser than interleaving
- Multiprocessor (multi-core) systems
  - several CPUs share the same global memory
  - beware of atomicity
  - word-level access granularity due to hardware design
- Distributed systems
  - no global memory
  - communication through message passing (channels)



Beware of unexpected interactions!

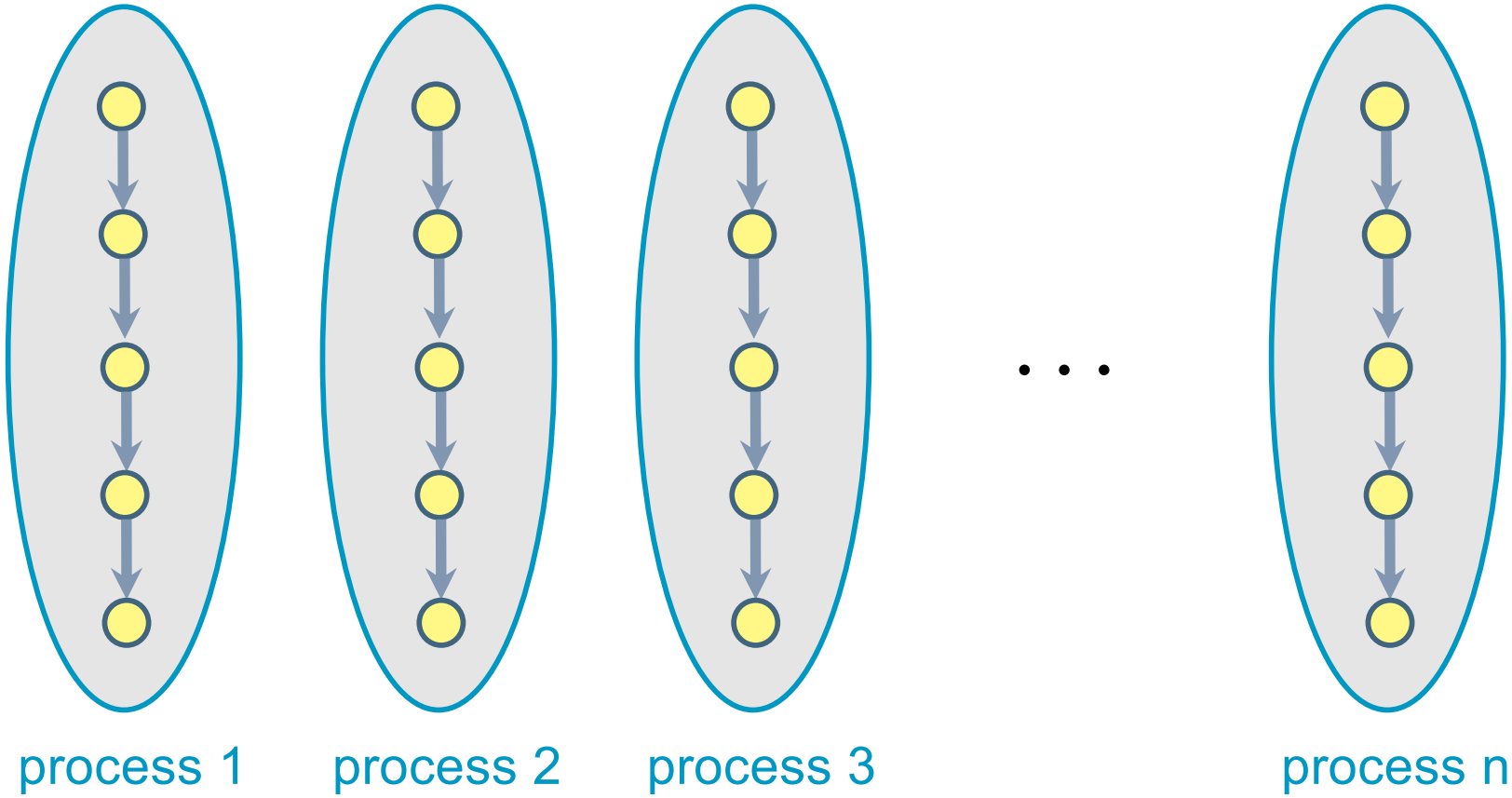
ADVANTAGES OF CONCURRENCY

- Improve performance
- For many applications closer to the real world (modelling each task as a separate task is much simpler)
- Ensure responsiveness (in particular important for blocking operations and event-based applications, i.e., that react on an event)
- Better resource usage (less idle time)
- Makes it easier to exploit multiple cores in hardware (software already indicates mapping on cores), which is becoming very common due to physical constraints

CONCURRENT PROGRAM BEHAVIOUR

concurrent program = finite set of sequential processes

process = finite set of atomic statements

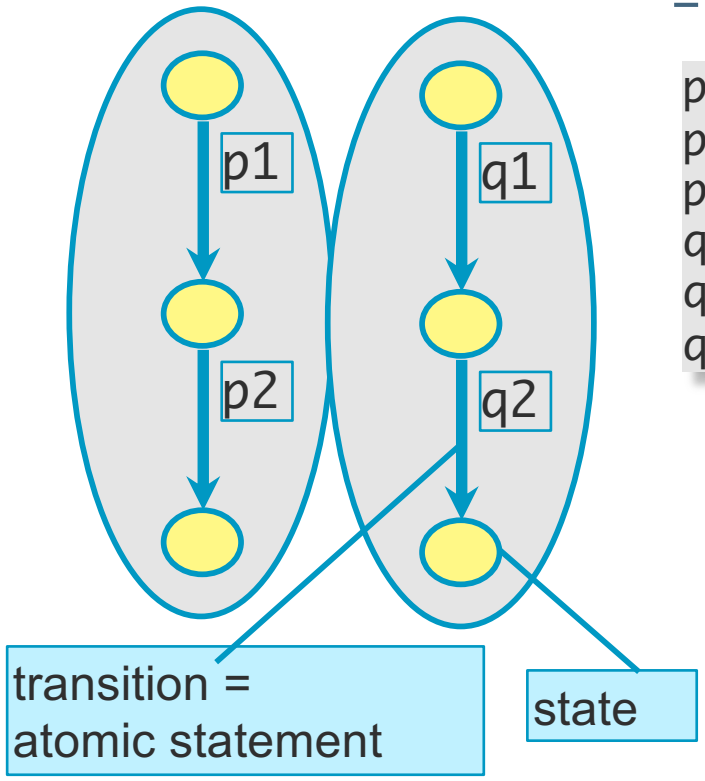


INTERLEAVING

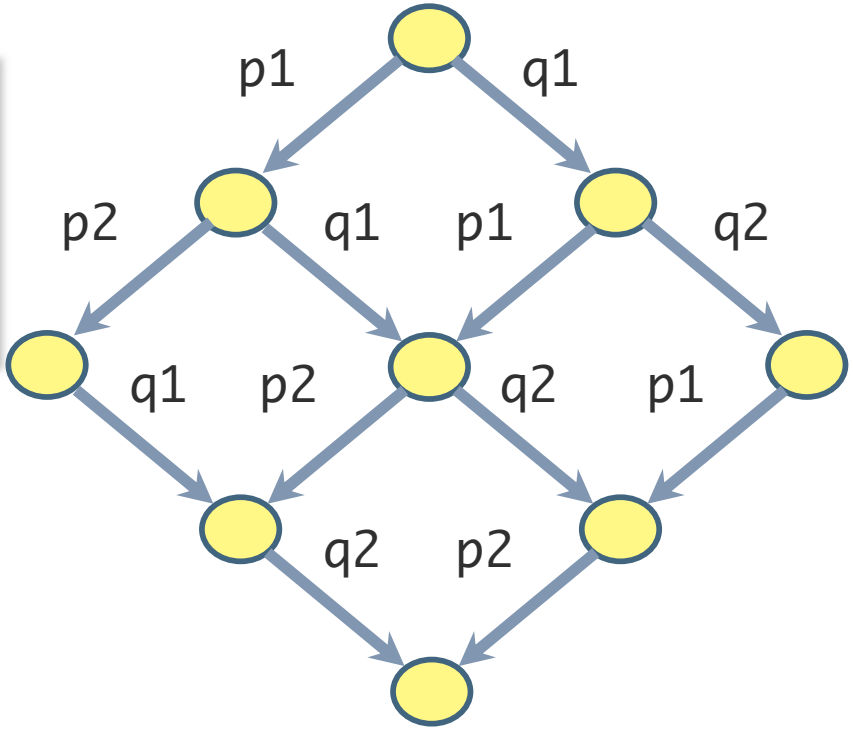
There is not just one computation: due to arbitrary interleavings, numerous different scenarios are possible.

p q Possible Scenarios State diagram

= *executions, runs*

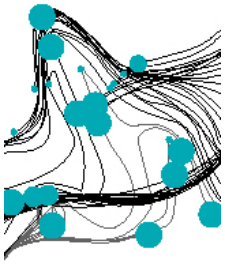



```
p1; p2; q1; q2
p1; q1; p2; q2
p1; q1; q2; p2
q1; q2; p1; p2
q1; p1; q2; p2
q1; p1; p2; q2
```



THE CHALLENGES OF CONCURRENCY

- Program complexity
- Atomicity errors
  - Race conditions
  - Data races
- Deadlock
- Livelock, starvation
- Underspecification
  - *unexpected reception of messages*
- Overspecification
  - *dead code*
- Violations of constraints
  - *buffer overruns*
  - *array bounds violations*
- Assumptions about speed
  - *logical correctness vs. real-time performance*
  - *too many context switches decline performance*



REASONING ABOUT CONCURRENCY: CONCURRENCY ABSTRACTION

- Atomic statement of a process executes
  - by itself and
  - to completion,

  before the execution of a statement of another process commences
- Execution of a concurrent program: executing a sequence of atomic statements obtained by arbitrarily interleaving of atomic statements from the processes

Are these abstractions realistic and justified?

ATOMICITY

- Atomic operation: as if executed in one step
- Basic atomic operations:
  - Read a location for non-long, non-double
  - Write a location for non-long, non-double
- Advanced atomic operations:
  - Volatile variables (atomic reads and writes for long and double)
  - Atomic classes (wrapper for volatile)
- Complex operations atomic:
  - Synchronisation (next block more about this)

Source: http://blog.regehr.org/archives/490

RACE CONDITIONS AND DATA RACES

- Race condition: flaw that occurs when the timing or ordering of events affects a program's correctness
- Data race happens when there are two memory accesses in a program where both:
  - target the same location
  - are performed concurrently by two threads
  - are not reads
  - are not synchronization operations
- Considerable overlap: many race conditions are due to data races, and many data races lead to race conditions
- But they are not the same!
- (Race condition sometimes called high-level data race)

RACE CONDITION VS DATA RACE

Does this have
- Data races
- Race conditions

```
transfer1 (amount, account_from, account_to) {
    if (account_from.balance < amount) {
        return NOPE;
    }
    account_to.balance += amount;
    account_from.balance -= amount;
    return YEP;
}
```

RACE CONDITION VS DATA RACE

Does this have
- Data races
- Race conditions

```
transfer2 (amount, account_from, account_to) {
  atomic {
    bal = account_from.balance;
  }
  if (bal < amount) return NOPE;
  atomic {
    account_to.balance += amount;
  }
  atomic {
    account_from.balance -= amount;
  }
  return YEP;
}
```

Atomic: some construct to make sure that code is executed atomically

Problem: it allows other threads to see intermediate state where key invariant: "preservation of money" is broken

RACE CONDITION VS DATA RACE

Does this have
- Data races
- Race conditions

```
transfer3 (amount, account_from, account_to) {
  atomic {
    if (account_from.balance < amount) {
       return NOPE;
       }
    account_to.balance += amount;
    account_from.balance -= amount;
    return YEP;
  }
}
```

RACE CONDITION VS DATA RACE

Does this have
- Data races
- Race conditions

```
transfer4 (amount, account_from, account_to) {
  account_from.activity = true;
  account_to.activity = true;
  atomic {
    if (account_from.balance < amount) {
       return NOPE;
       }
    account_to.balance += amount;
    account_from.balance -= amount;
    return YEP;
  }
}
```

WHAT WILL THIS PROGRAM DO

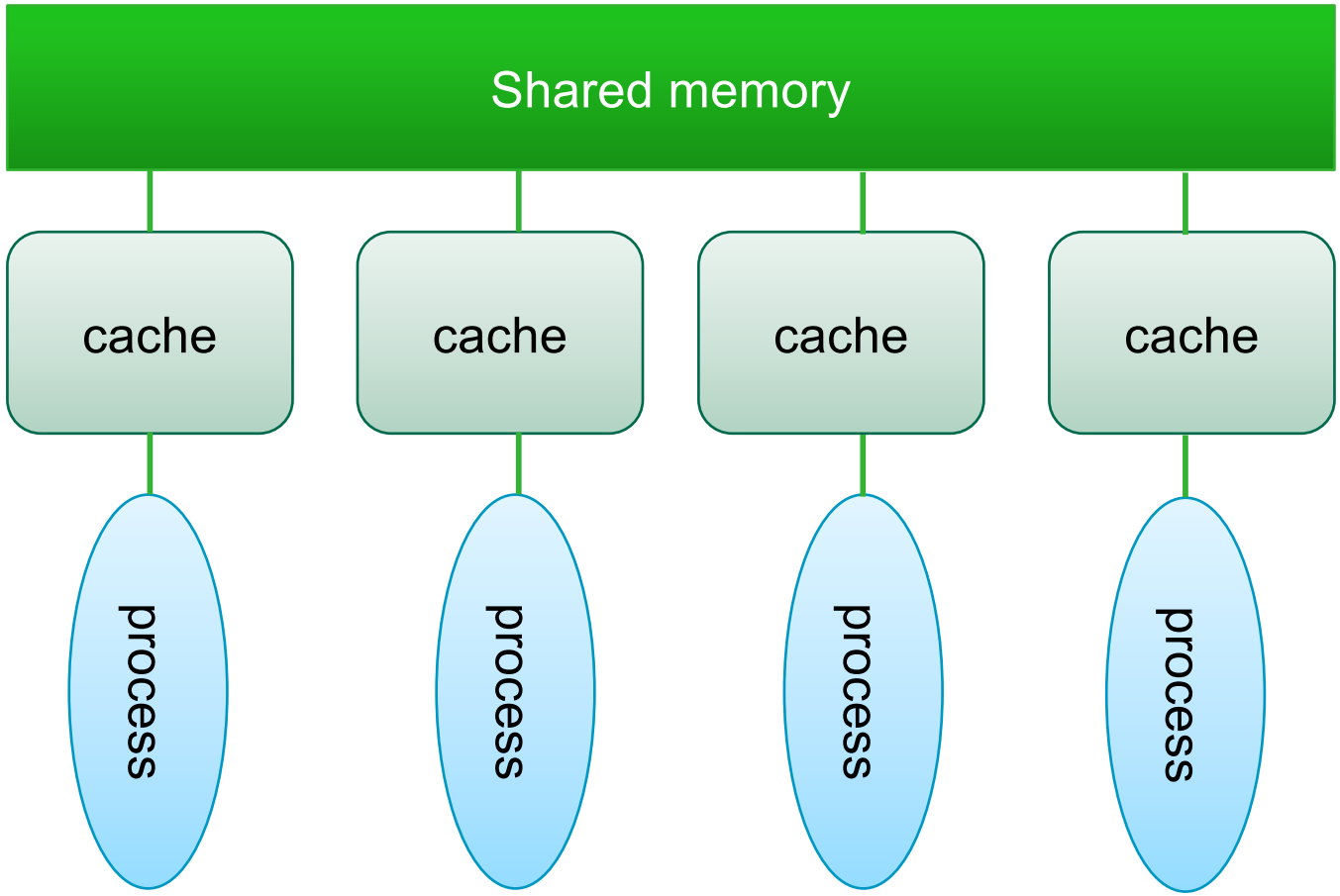
```
public class C {
    private static boolean ready;
    private static int number = 0;

    private static class ReaderThread
                          extends Thread {
        public void run() {
            while (!ready);
            System.out.println(number);
        }
}
```

```
public static void main (String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

An important source of problems:
Stale values

HARDWARE EXPLANATION

Flush the cache:
- Synchronisation
- Atomics



ATOMICS AND VOLATILE

- Volatile variables
  - Atomic read and writes also for long and double
  - Wrapper classes: AtomicInteger etc
    - Provide atomic get, set, and compare-and-set operation
  - Any volatile update is immediately visible to other threads
  - Typical usage: status flags

Beware, this does not solve all synchronisation problems!!

What is the problem?
volatile int x;
x = x + 1;

Very common pattern:
- Fetch-and-store
- Test-and-set
- Compare-and-swap

First read, inspect and then write
Developer needs to ensure atomicity

THREAD SAFETY

- Thread-safe code fragment:
  - shared data structures safely executed by multiple threads at same time
  - Parallel execution should not introduce errors
- Ensuring thread-safety: coordinating access to shared, mutable state

Programmer's responsibility!

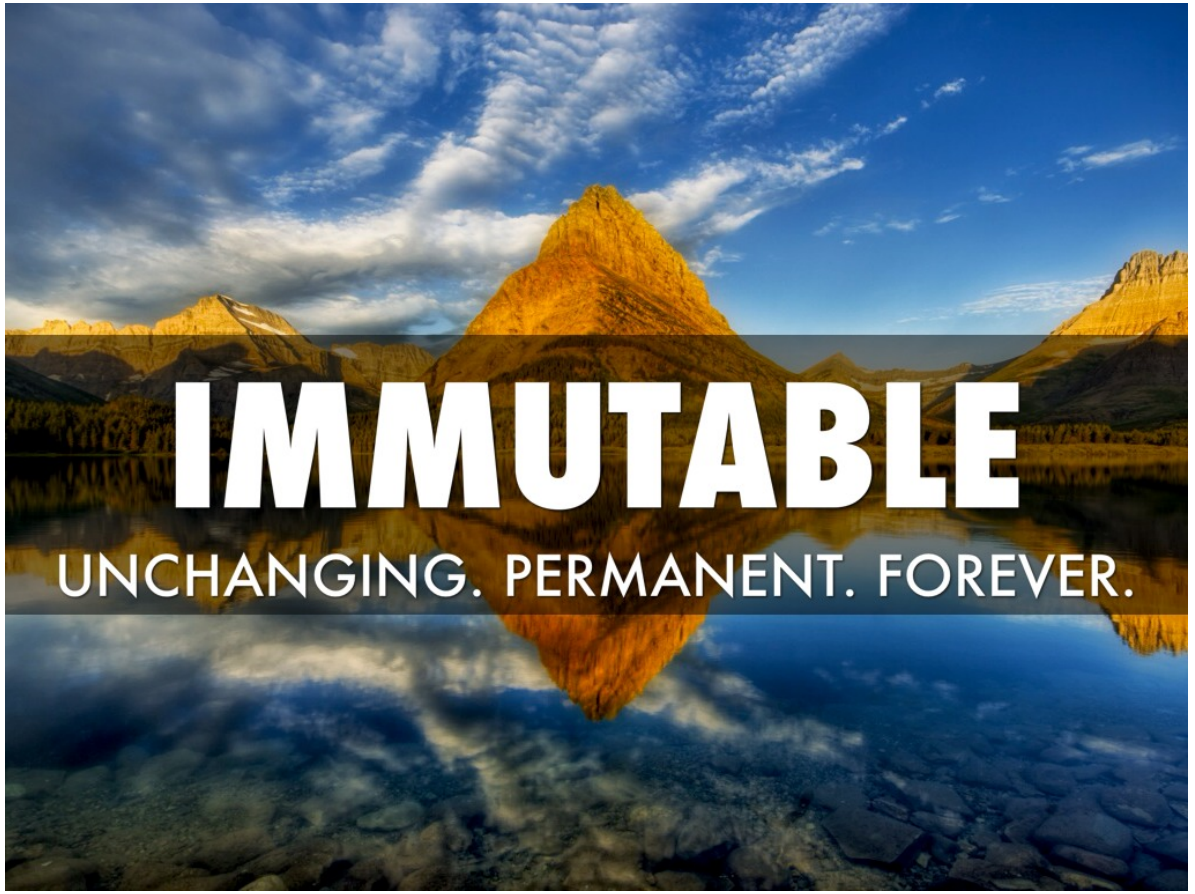
COORDINATING ACCESS TO SHARED STATE

- Thread confinement: thread local data
- Restricted volatile: single thread read-modify-writes, all others just read
- Stack confinement: local variables are local, and are not leaked

  API class: ThreadLocal (conceptually: Map<Thread, Value>)
- Immutable objects
- Safe publication: store reference in a place protected by a synchroniser
  - from static initialiser into volatile or final field
  - into field guarded by a lock

@guardedby annotation

IMMUTABILITY



MAKING A CLASS IMMUTABLE

- No "setter" methods that modify fields or objects referred to by fields
- All fields final and private
- Don't allow subclasses to override methods
  - Declare class final
  - Or: make constructor private and construct instances in factory methods
- If instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify mutable objects
  - Don't share references to mutable objects
  - Never store references to external, mutable objects passed to the constructor
  - If necessary, create copies, and store references to copies

SYNCHRONIZED RGB

```
// Values must be between 0 and 255
//@ private invariant 0 <= red && red <= 255;
//@ private invariant 0 <= green && green <= 255;
//@ private invariant 0 <= blue && blue <= 255;
```

```
public class SynchronizedRGB {
    private int red; private int green; private int blue; private String name;

    public SynchronizedRGB(int red, int green, int blue, String name) { //body }
    public void set(int red, int green, int blue, String name) {//body
        check(red, green, blue); // valid range
        synchronized (this) {
            this.red = red; this.green = green; this.blue = blue; this.name = name;
        } }
    public synchronized String getName() { return name; }
    public synchronized void invert() {
        red = 255 – red; green = 255 – green; blue = 255 – blue;
        name = "Inverse of " + name; }
}
```

How to make this immutable?

IMMUTABLE RGB

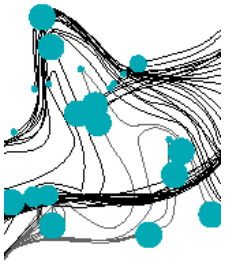
```
final public class ImmutableRGB {
    final private int red; final private int green;
    final private int blue; final private String name;

    public ImmutableRGB(int red, int green, int blue, String name) {// body }
    public String getName() { return name; }
    public ImmutableRGB invert() {
        return new ImmutableRGB(255 – red, 255 – green, 255 - blue,
                                "Inverse of " + name);
    }
}
```

STATELESS ≠ IMMUTABLE

```
class Immutable {
  final String testString;

  Immutable(String testString) {
    this.testString = testString;
  }

  void test() {
    System.out.println(testString);
  }
}
```

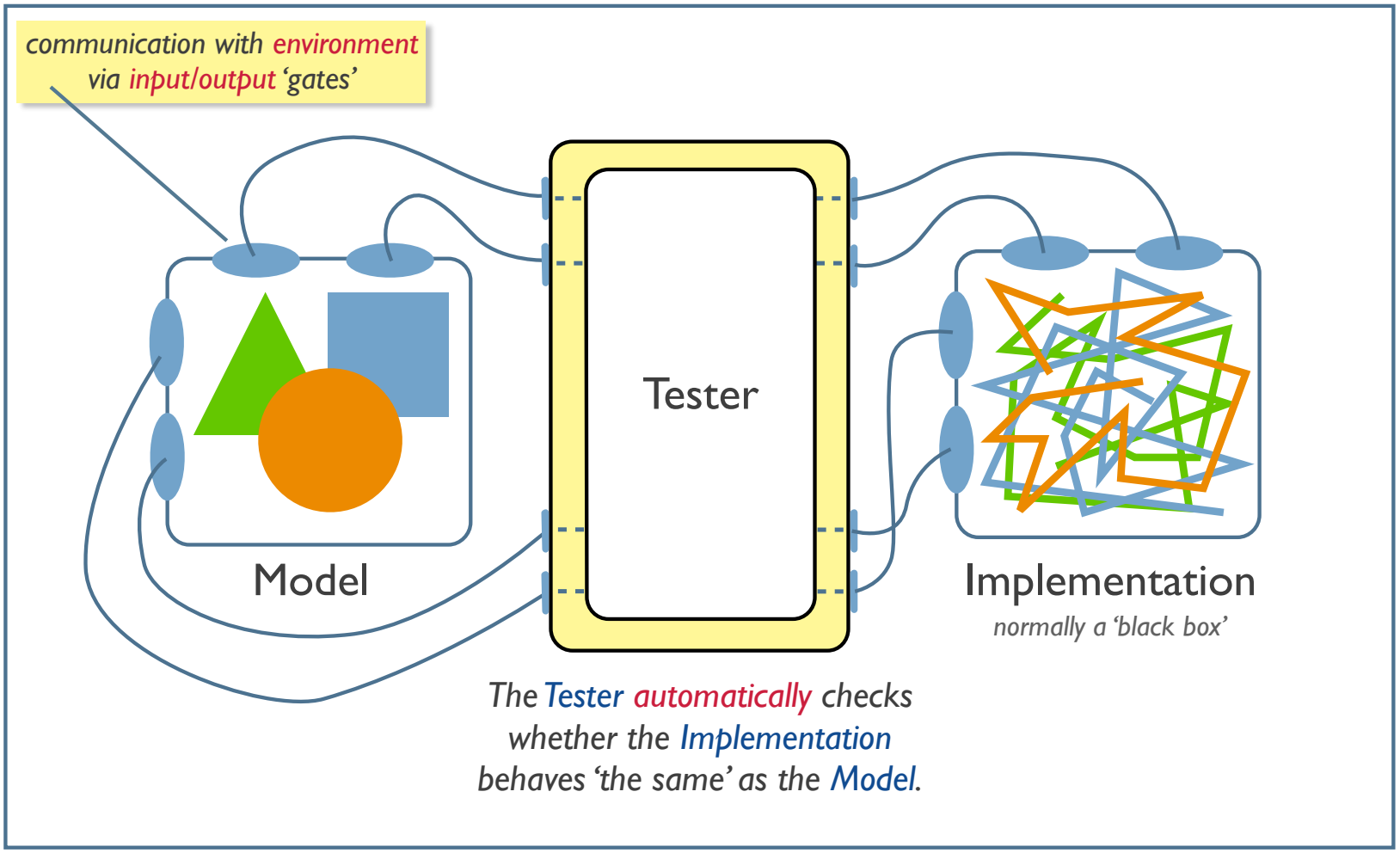
```
class Stateless {
  void test() {
    System.out.println("Test!");
  }
}
```

```
class Stateless {
  final String TEST = "Test!";
  void test() {
    System.out.println(TEST);
  }
}
```



TESTING CONCURRENT PROGRAMS

CHAPTER 12

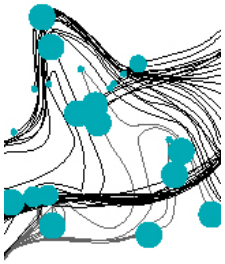


TEST CHALLENGES

MUCH HARDER THAN SEQUENTIAL CASE

Heisenbug: *bug that disappears when you test for it*

- detect (very) rare problems
  - *some errors only show under extreme load of the system*
- test should not change (synchronisation) behaviour
  - *be careful with synchronisation within test classes*
- other threads can influence assertion evaluation
  - *assertions are not atomic*
- not only safety, but also liveness
- … and how to test performance?



IMPORTANT POINTS

- Multithreading: multiple threads execute in parallel
- Thread behaviour: interleaving of atomic steps
- Many advantages, but also many challenges
- Unordered memory accesses can cause problems
- Threads share data, but this should be done with care
- Testing of concurrent programs should be done with care