Concurrency (I)

Lecture 12 (18th May 2021)

-- Liang: Web Chapter 32 --

Radboud University

What is concurrency?

- Performing operations concurrently (in parallel)
  - We can walk, talk, breathe, see, hear, smell... all at the same time;
  - Computers can do this as well: download a file, print a file, receive email, run the clock, more or less in parallel….
- How are these tasks typically accomplished?

When to apply concurrency?

Three reasons for applying concurrency:

1. Responsiveness
2. Efficiency
3. (Simulation of) naturally concurrent processes

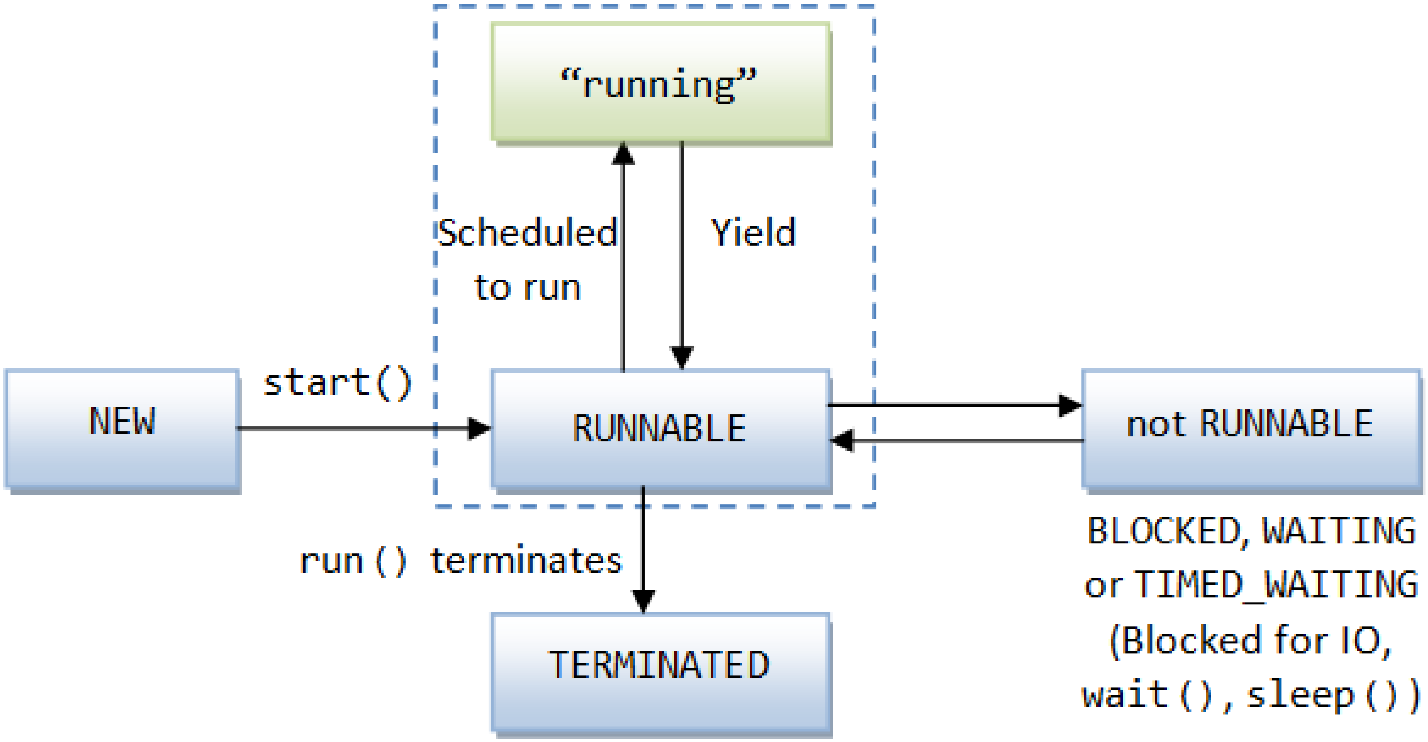
Furthermore:

When the order of actions is not relevant to the correctness of the program, concurrency enables to avoid specifying this order, resulting in simpler program code.

About threads

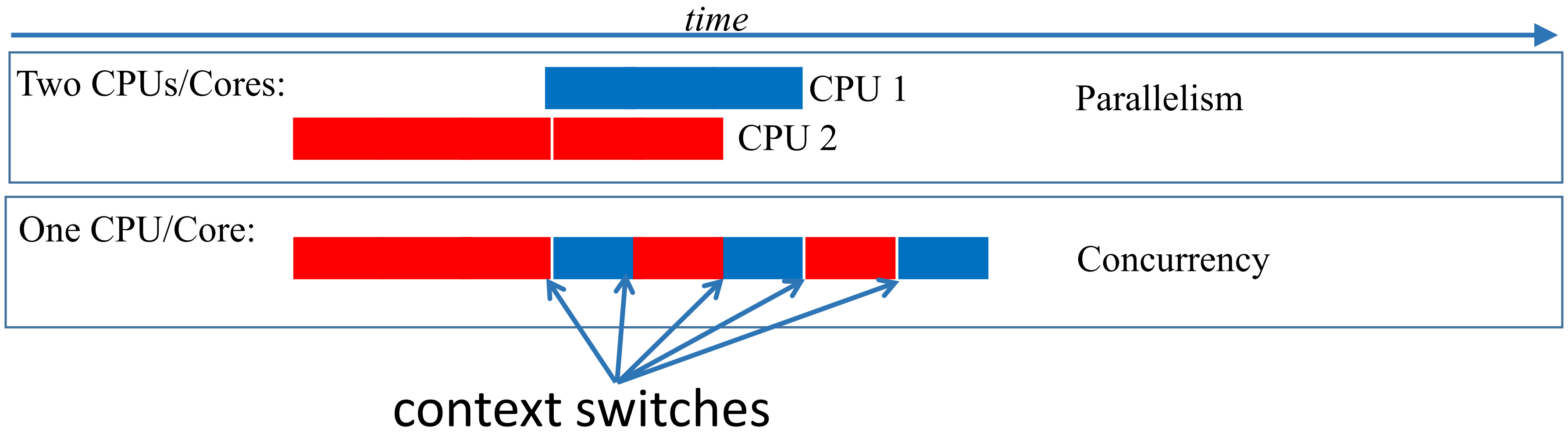
- The program *flow of control* is the order in which the computer executes the statements in a program.
- A **thread** corresponds to a *single sequential flow of control* (within a concurrent program)
- A thread executes independently of others, while at the same time sharing underlying system resources and objects constructed within the same program
- Each thread has its own call stack: threads do *not* share local variables and parameters
- Every program consists of at least one thread.

States and state transitions of threads



- The operating system manages the threads. A so-called *scheduler* controls the transitions. In Java this is done by the JVM (Java Virtual Machine)

Two threads



Terminology:

- **Time-slicing (aka interleaving)** : Threads *alternatingly* get a small amount of execution time on the CPU
- **Scheduler**: decides which thread, when, and for how long will get the CPU
- **Context switch**: When the scheduler gives another thread the CPU the state of the current thread is stored and the state of the other one is loaded. This is a relatively costly operation.

Nondeterminism

- Concurrent programs are nondeterministic
  - *Different executions with the same input can give different results*
  - *E.g., which thread terminates first is often unpredictable*
- Concurrent programs are complex
  - Several threads are executed simultaneously
  - In each execution, threads can get different amounts of CPU-time. Unexpected values of variables and of results can occur (and should be avoided)
    - Context switches can occur halfway between variable updates
  - Statements can be executed in 'surprising' order (different than in the sequential case, this now may become visible)
- We come back to this later (Problems with threads)

> *"… Threads discard the most essential and appealing properties of sequential computation: comprehensibility, predictability, and determinism. …"*
> *(Edward.A. Lee, The problems with Threads, 2006)*

Java: Objects and Threads

- OOP lecture 1: What are objects?
- Answer: Building blocks of software systems
  - *a program is a collection of interacting objects*
- Objects have capabilities that allow them to perform specific actions
- Thread capability
  - a thread is able to perform a task.
- A Task is an object that provides a run method.
  - If a thread is asked to perform a task it will execute the task's run method
- In Java, programs can have multiple threads

Creating a new thread(I): interface Runnable

1. Introduce a class implementing the interface `Runnable`. This class is said to be active
2. Implement the method `run` with the code for the concurrent task:

   ```
   public void run() { // code of the task goes here  }
   ```

   Note: method `run`
   - has no parameters
   - no return value,
   - cannot throw checked exceptions

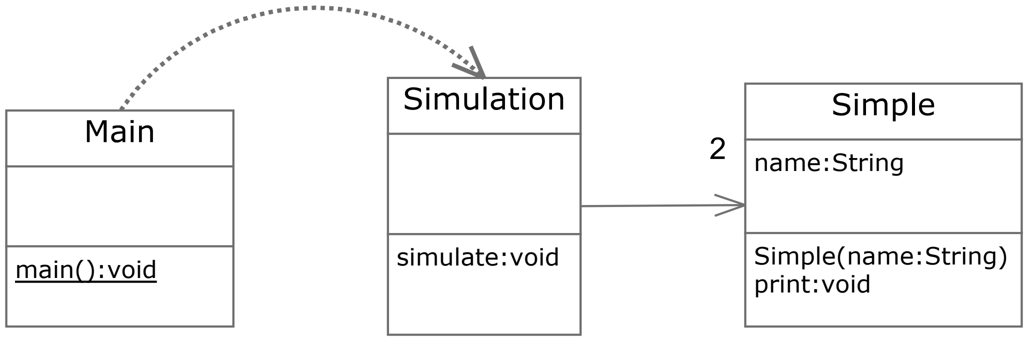
```
interface Runnable {
   void run();
}
```

3. Create an instance of class `Thread` with an active class object as argument
4. Call on this instance of Thread the method `start`; method `run` will then be executed

```
public Thread( Runnable task )
```


Radboud University

Creating a new thread(II): sequential example



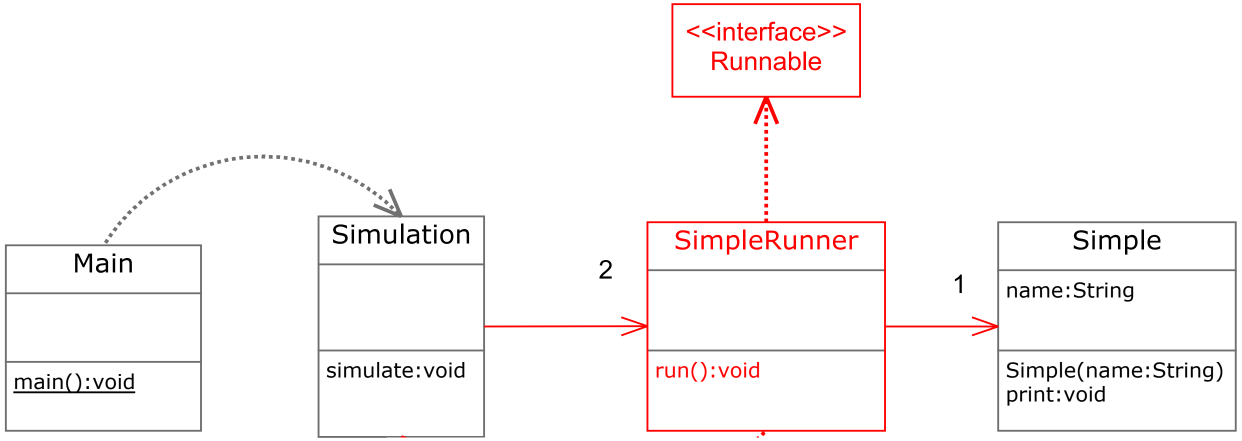

```
public class Main {

  public static void main(String[] args) {
    Simulation sim = new Simulation();
    sim.simulate();
  }
}
```

```
public class Simple {
  private String name = null;

  public Simple(String name) {
    this.name = name;
  }

  public void print() {
    System.out.println("My name =" + name);
  }

}
```

```
public class Simulation {

  private static final int N = 3;

  public void simulate() {
    Simple s1 = new Simple("One");
    Simple s2 = new Simple("Two");
    for (int i = 0; i < N; i++) {
      s1.print();
    }
    for (int i = 0; i < N; i++) {
      s2.print();
    }
  }

}
```

```
run:
My name =One
My name =One
My name =One
My name =Two
My name =Two
My name =Two
```

Creating a new thread (III): changed to concurrent example



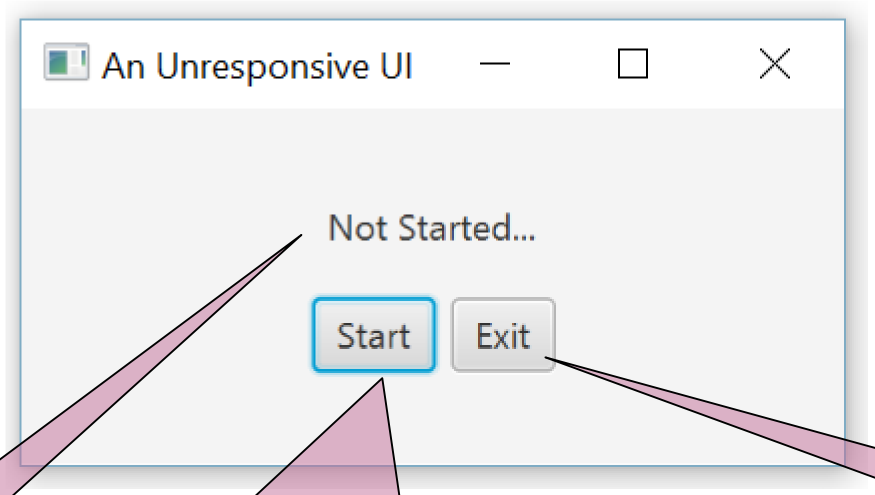

```
run:
My name =Two
My name =One
My name =Two
My name =Two
My name =One
My name =One
```

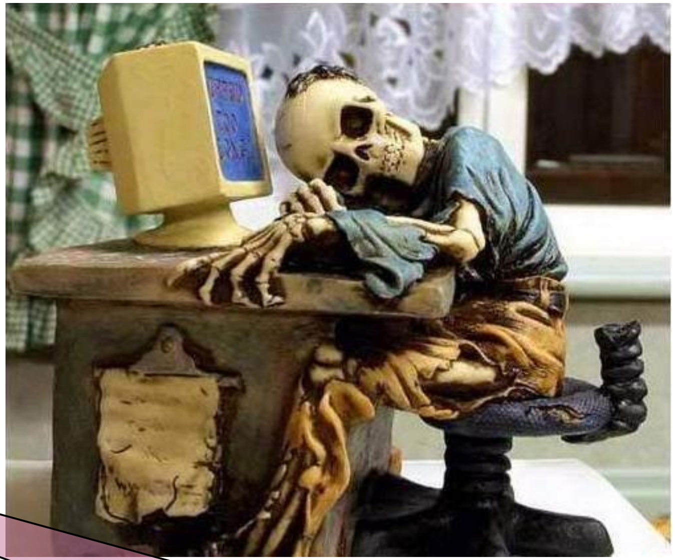
```java
public class SimpleRunner implements Runnable{
  private Simple simple = null;

  public SimpleRunner(Simple simple){
    this.simple = simple;
  }

  @Override
  public void run() {
    for (int i = 0; i < Simulation.N; i++) {
      simple.print();
    }
  }
}
```

```java
public class Simulation {

  public static final int N = 3;

  public void simulate() {
    Simple s1 = new Simple("One");
    Simple s2 = new Simple("Two");
    SimpleRunner sr1 = new SimpleRunner(s1);
    SimpleRunner sr2 = new SimpleRunner(s2);
    Thread t1 = new Thread(sr1);
    Thread t2 = new Thread(sr2);
    t1.start();
    t2.start();
  }

}
```

```java
public class Simple {
  private String name = null;

  public Simple(String name) {
    this.name = name;
  }

  public void print() {
    System.out.println("My name =" + name);
  }

}
```

Nonresponsiveness (I)





Status label

Start button: starts a task for 10 seconds

Exit button: stops the program

```
Label statusLbl = new Label( "Not Started..." );
Button startBtn = new Button( "Start" );
Button exitBtn  = new Button( "Exit" );

public void start( Stage stage ) {
   startBtn.setOnAction( e -> runTask() );
   exitBtn.setOnAction( e -> stage.close() );
```

Nonresponsiveness (II)

```
startBtn.setOnAction( e -> runTask() );
```

```
public void runTask() {
    for ( int i = 1; i <= 10; i++ ) {
        String status = "Processing " + i + " of " + 10;
        statusLbl.setText( status );
        System.out.println( status );
        takeABreak( 1000 );
    }
}
```

```
public void takeABreak( int millisec ) {
    try {
        Thread.sleep( millisec );
    } catch ( InterruptedException e ) {
        e.printStackTrace();
    }
}
```

`Thread.sleep`

- `Thread.sleep`: static method in class `Thread`.
- A thread calling this method will be paused for the number of milliseconds given as an argument.
  - Note that it may be invoked in an ordinary (single-threaded) program to insert a pause in the single thread of that program
- It may throw a (checked) exception: `InterruptedException`
- Use:
  - To slow down programs, for, e.g., animations
  - Side effect: context switch

TimeUnit: sleep

Enum Constants
Enum Constant and Description
DAYS
HOURS
MICROSECONDS
MILLISECONDS
MINUTES
NANOSECONDS
SECONDS

```
public void takeABreak( int millisec ) {
    try {
        TimeUnit.MILLISECONDS.sleep(millisec);
    } catch ( InterruptedException e ) {
        e.printStackTrace();
    }
}
```

Analyzing the nonresponsive Program

- Two issues:
  1. Clicking Exit button has no effect until the task finishes
  2. Label is not updated
- Reason:
  - all UI event handlers in JavaFX run on a single thread:
    the JavaFX Application Thread
  - as long as one handler is executed, other handlers have to wait. In this case:
    - the handler for the exit button
    - the handler for updating the (view of the) label.

Fixing a Nonresponsive Program Using Threads

- This is how to fix the problem:

- Have the button event handler method create a new thread to execute runTask
- Once created, the new thread will be an independent process that proceeds on its own
- Now, the work of the event handler is ended, and the main thread (that executed the handler) is ready to respond to something else
- If the “Exit” button is clicked while the new thread executes the task, then the program will end

Multithreaded Program that tries to fix the Nonresponsive GUI

1. Introduce an (active) class Task implementing run which executes the runTask method.
2. Let the handler call a new method startTask which creates an instance of Task, and an instance of Thread with the former instance as argument.
3. Start the new thread, which now can run concurrently with the JavaFX Application Thread.

Step 1: active class

```java
public class Task implements Runnable {
    ResponsiveGUI gui;

    public Task ( ResponsiveGUI gui ) {
        this.gui = gui;
    }

    public void run(){
        gui.runTask();
    }

}
```

Step 2 and 3: Create and start a thread

```
startBtn.setOnAction( e -> startTask() );
```

```
public void startTask() {
    Task task = new Task( this );
    Thread taskThread = new Thread( task );
    taskThread.start();
}
```

Step 4: Run your program

After pressing the Start button you get:

```
Exception in thread "Thread-4" java.lang.IllegalStateException: Not on FX
application thread; currentThread = Thread-4
```

In principle, the solution is ok. However, when running JavaFX (or Swing) applications you have to be more careful.

We will analyse the problem, and fix the error later.

Let’s first have a look at some other issues.

Executing runnables (I)

```
public class PrintTask implements Runnable
{
   private final int sleepTime;
   private final String taskName;
   private final static Random generator = new Random();

   public PrintTask ( String name ) {
      taskName = name;
      sleepTime = generator.nextInt( 5000 );
   }

   public void run () {
      … // next slide
   }
}
```

Implement `Runnable` to define a task that can execute concurrently

Executing runnables (II)

Define task in method `run`

```
public void run() {
   System.out.printf ( "%s going to sleep for %d ms.\n",
         taskName, sleepTime );
   takeABreak( sleepTime );
   System.out.printf ( "%s done sleeping\n", taskName );
}
```

Executing runnables (III)

```
public class TaskExecutor {
   private final List<Runnable> tasks;

   public TaskExecutor(List<Runnable> runnables) {
      tasks = runnables;
   }

   public void executeConcurrently( ) {
      System.out.println("Starting Tasks");
      tasks.forEach(task -> new Thread(task).start());
      System.out.println("Tasks started");
   }
}
```

Executing runnables (IV)

```java
public class Main {
   private static final int NR_PRINT_TASKS = 3;

   public static void main(String[] args) {
      List<Runnable> tasks = IntStream.rangeClosed(1, NR_PRINT_TASKS)
              .mapToObj(taskNr -> new PrintTask("Task " + taskNr))
              .collect(Collectors.toList());
      TaskExecutor te = new TaskExecutor(tasks);
      te.executeConcurrently();
      System.out.println("main ends");
   }
}
```

```
Starting Tasks
Tasks started
Task 1 going to sleep for 100 ms.
Task 2 going to sleep for 2462 ms.
Task 3 going to sleep for 1311 ms.
main ends
Task 1 done sleeping
Task 3 done sleeping
Task 2 done sleeping
```

Executing runnables (V)

- Sometimes you want to test your program in a more predictable way: Do It Yourself

```
public class TaskExecutor {
   List<Runnable> tasks;

   public void executeSequentially(  ) {
      tasks.forEach(Runnable::run);
   }
}
```

Executing runnables (VI)

```
public class Main {
   private static final int NR_PRINT_TASKS = 3;

   public static void main(String[] args) {
      List<Runnable> tasks = IntStream.rangeClosed(1, NR_PRINT_TASKS)
              .mapToObj(taskNr -> new PrintTask("Task " + taskNr))
              .collect(Collectors.toList());
      TaskExecutor te = new TaskExecutor(tasks);
      te.executeSequentially();
      System.out.println("main ends");
   }
}
```

```
Task 1 going to sleep for 2727 ms.
Task 1 done sleeping
Task 2 going to sleep for 1890 ms.
Task 2 done sleeping
Task 3 going to sleep for 1601 ms.
Task 3 done sleeping
main ends
```

Sharing resources

Atomicity and scheduling

A (Java) statement such as `i++;` is not executed as one (indivisible = atomic) step, but consists of a sequence of smaller (assembly) steps.

Example for `i++;` (depends on processor type):

```
LOAD @i, r0;   load the value of 'i' into a register from memory
ADD r0, 1;     increment the value in the register
STORE r0, @i;  write the updated value back to memory
```

Scheduling is done on the level of assembly code (or even lower) and hence during the execution of `i++;` a context switch can occur

Two possible scenarios

Threads T1 and T2 update a common variable i

Scenario as you might expect

T1	T2	R/W	i
			0
Read		←	0
+ 1			0
Write		→	1
	Read	←	1
	+ 1		1
	Write	→	2

Scenario with unexpected result

T1	T2	R/W	i
			0
Read		←	0
	Read	←	0
+ 1			0
	+ 1		0
Write		→	1
	Write	→	1

- When `i++` is executed by two different threads, the end result is that `i` can be both 2 and 1 larger than at the start
- NB: this is irrespective of whether the threads are executed on one or two processors

Unsynchronized Data Sharing Demo (I)

```
public class Buffer // CAUTION: NOT THREAD SAFE!
{
    private final int[] bufferArray;
    private int writeIndex = 0;

    public static final int PAUSE = 50;


    public Buffer( int size ) {
        bufferArray = new int[ size ];
    }

    public void add( int value ) {
        … // next slide
    }
}
```


Radboud University

Unsynchronized Data Sharing Demo (II)

```
public void add( int value ) {
    int position = writeIndex;

    takeABreak();

    bufferArray[ position ] = value;

    takeABreak();

    writeIndex = position + 1;
}
```

Unsynchronized Data Sharing Demo (III)

```
public class BufferWriterTask implements Runnable
{
   private final Buffer myBuffer;
   private final int startValue, nrOfItems;

   public BufferWriterTask( int start, int nr_of_items, Buffer array ) {
      startValue = start;
      nrOfItems  = nr_of_items;
      myBuffer   = array;
   }

   public void run() {
      for ( int i = 0; i < nrOfItems; i++ ) {
         myBuffer.add( startValue + i );
      }
   }
}
```

Unsynchronized Data Sharing Demo (IV)

```
private static final int BSIZE = 10;

public static void main( String[] arg ) {
   Buffer buffer = new Buffer( BSIZE );

   BufferWriterTask writer1 = new BufferWriterTask( 1, BSIZE/2 , buffer );
   BufferWriterTask writer2 = new BufferWriterTask( 14, BSIZE/2 , buffer );

   Thread writer_thread1 = new Thread ( writer1 );
   Thread writer_thread2 = new Thread ( writer2 );

   writer_thread1.start();
   writer_thread2.start();

   System.out.println( buffer );
}
```

Unsynchronized Data Sharing Demo (V)

You can use the `join` method to force one thread to wait for another thread to finish.

```
public static void main( String[] arg ) {

    <....>
    // wait for the completion of the other threads
    try {
      writer_thread1.join();
      writer_thread2.join();
    } catch ( InterruptedException ex ) {
       ex.printStackTrace();
    }

    // display the contents of the shared array
    System.out.println( buffer );
}
```

Race conditions

- A race condition: possibly incorrect result caused by the update of a common variable/object by several threads.

```
public void add( int value ) {
    int position = writeIndex;

    takeABreak();

    bufferArray[ position ] = value;

    takeABreak();

    writeIndex = position + 1;
}
```

Non-atomic Operations

- Writing

```
public void add( int value ) {
    buffer[ writeIndex++ ] = value;
}
```

won't work: array[writeIndex++] = value; is not atomic (single unit of work, which cannot be interrupted in the middle).

- Without special measures, atomicity in Java is hard to obtain: almost any Java statement consists of several non-atomic operations.

Thread Synchronization (I)

- Coordinates access to shared data by multiple concurrent threads
  - Indeterminate results may occur unless access to a shared object is managed properly
  - *Code* (typically the body of a method) that manipulates shared data is called a critical section.

- Basic idea: Give only one thread at a time exclusive access to a critical section.
- Other threads have to wait
- When the thread with exclusive access leaves the critical section (i.e. finishes manipulating the shared object), one of the threads that was waiting is allowed to proceed

Thread Synchronization (II)

- Exclusive access to such a critical section (also called mutual exclusion) is obtained through locking.
- The basic form of locking in Java is by means of so-called
  1. *synchronized methods*
  2. *synchronized blocks*

- Any Java object can act as a lock.
  - when calling a synchronized method, the current object (`this`) is used for locking.
  - in a synchronized block the lock object is specified explicitly.
- When a thread calls a synchronized method (or enters a synchronized block) the object's lock, when it is free, will become occupied (by the executing thread). When the lock is not free, the calling/entering thread has to wait.

Thread Synchronization JAVA

- **synchronized** statement
  - Enforces mutual exclusion on a block of code
  - ```
    synchronized ( object ) {
        statements
    } // end synchronized statement
    ```
    where *object* is the object whose lock will be acquired (normally **this**)
- A **synchronized** method is equivalent to a **synchronized** statement that encloses the entire body of a method, using the lock of **this**

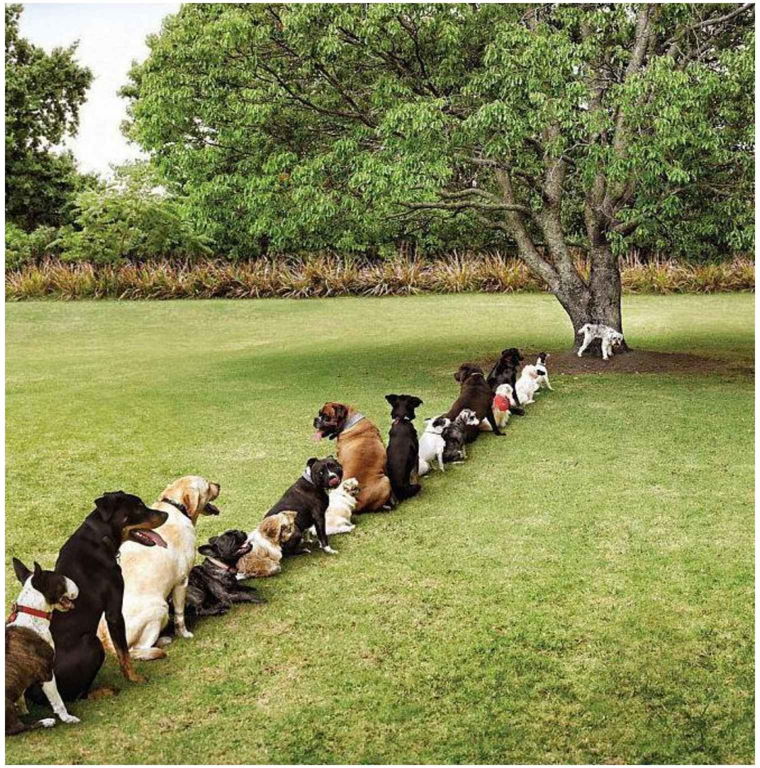

    ```

Synchronized Data Sharing—Making Operations Atomic (III)

```
public synchronized void add( int value ) {
    int position = writeIndex;

    bufferArray[ position ] = value;

    writeIndex = position + 1;
}
```

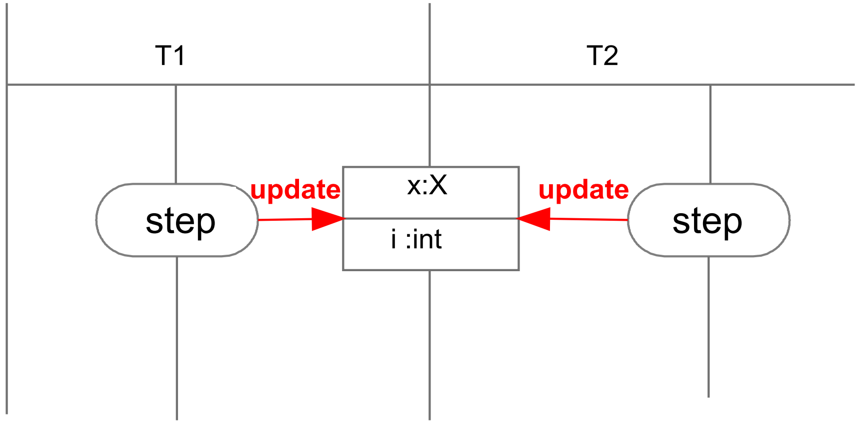


Using synchronized prevents more than one thread at a time from calling this method

Visualization of race conditions (I)

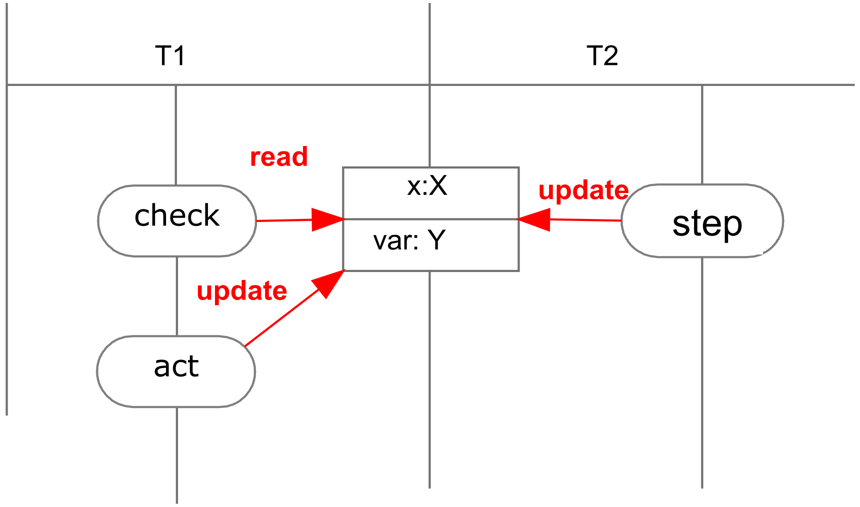
- The cause of a race condition are *updates* (state changes) of an object that is *shared* between several threads
- Visualization: Use an UML activity diagram in which the shared object is explicitly indicated
- In case of accessing the shared object distinguish between read and write/update
- In the shared object *explicitly* indicate the *attribute* that is accessed

Visualization of race conditions (II)




```
void inc() {
   i++;
}
```

Two threads use a shared object x:X with attribute i and try to update attribute i



Check then act
(or read-modify-write)

Two threads use a shared object x with attribute var

Avoiding race conditions

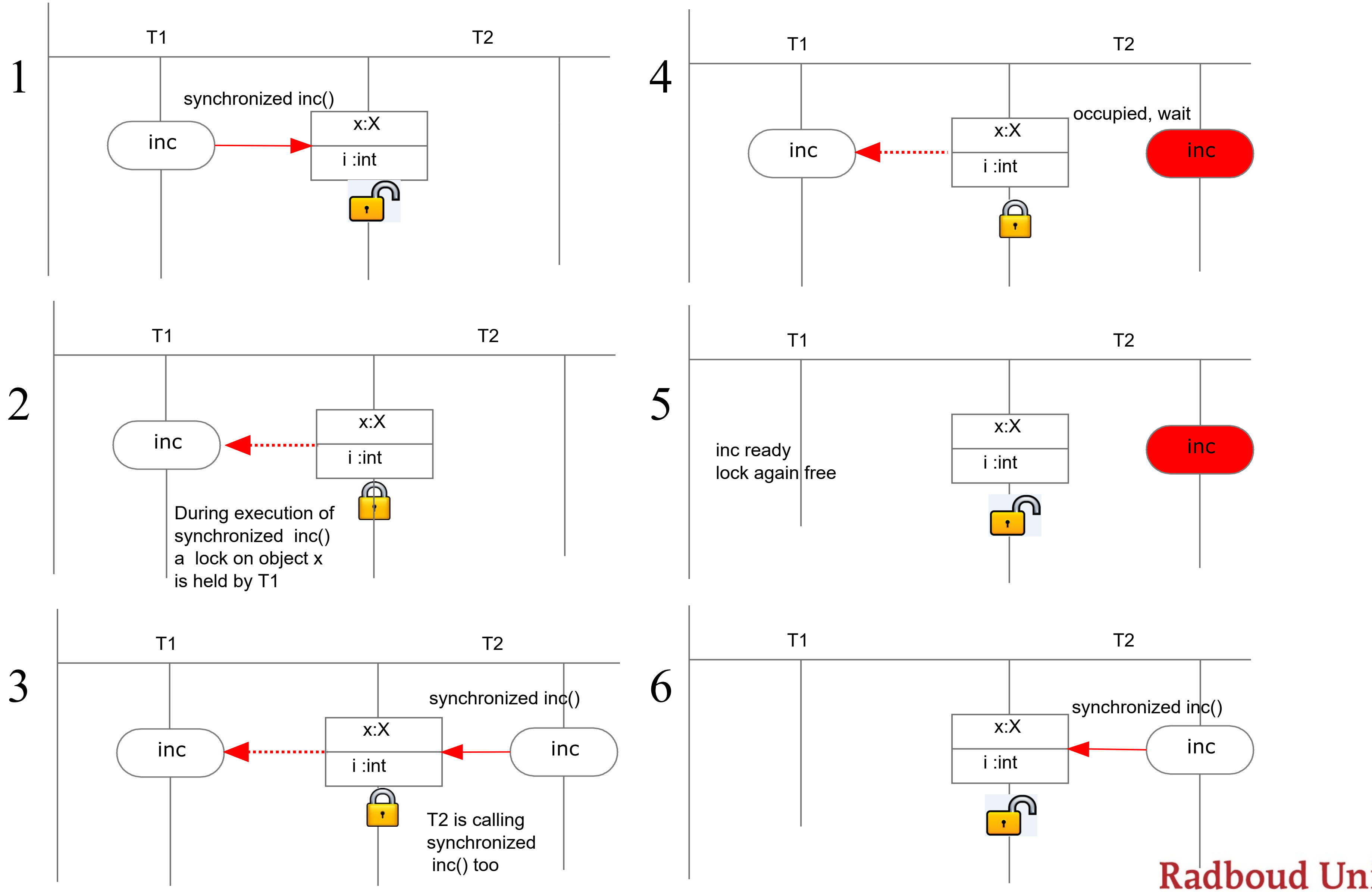
```
public synchronized void inc() {
  i++;
}
```

… or (equivalently) …

```
public void inc() {
  synchronized (this) {
    i++;
  }
}
```

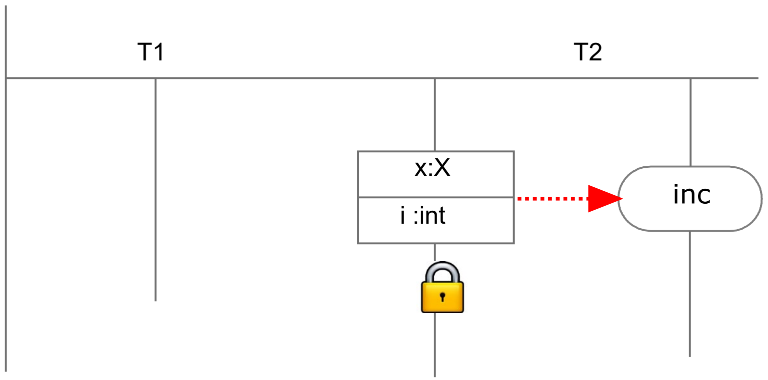
- Method inc now can only be executed by one thread at the time.
- The use of synchronized will decrease performance, because the JVM has to do additional bookkeeping, so be frugal in the use of synchronized methods.

Visualizing synchronized methods (I)

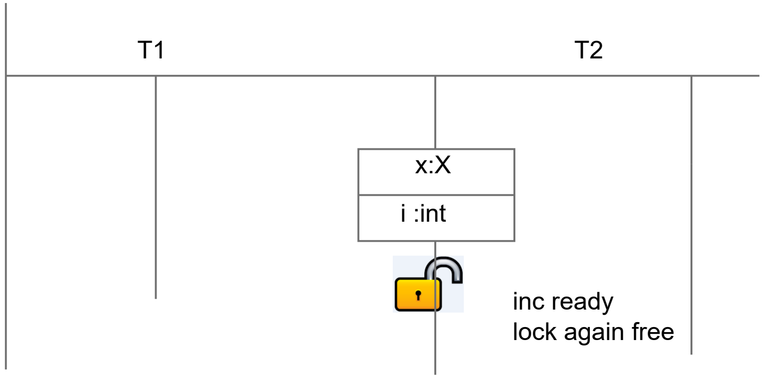


Visualizing synchronized methods (II)

7



8



These figures are called a Lock Allocation Diagrams or LADs

Check-Then-Act

- Check-then-act: if a context switch occurs *after* checking a condition but *before* the subsequent action is performed, the condition may have changed. This is dangerous if that action relies on the condition.

CTE-example (I)

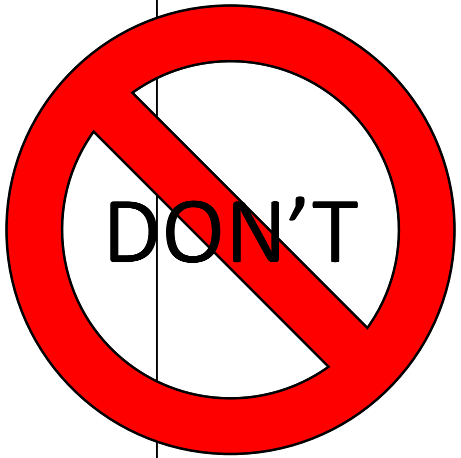
```
public class Counter {
    private int counter;

    public synchronized void incr() {
        counter++;
    }

    public synchronized int getCounter() {
        return counter;
    }
}
```

CTE-example (II)

```java
public class Incrementer implements Runnable {
    private Counter counter;
    private int limit;

    public Incrementer( Counter counter, int limit ) {
        this.counter = counter;
        this.limit   = limit;
    }

    @Override
    public void run() {
        while ( counter.getValue() != limit ) {
            counter.incr();
        }
    }
}
```



CTE-example (III)

```
public class CounterTest {
    public static void main( String[] args ) {
        Counter counter       = new Counter();
        Incrementer  incr1    = new Incrementer( counter, 50 );
        Incrementer  incr2    = new Incrementer( counter, 50 );

        Thread incr_thread1  = new Thread( incr1 );
        Thread incr_thread2  = new Thread( incr2 );

        incr_thread1.start();
        incr_thread2.start();

        try {
            incr_thread1.join();
            incr_thread2.join();
        } catch ( Exception e ) {
            e.printStackTrace();
        }
        System.out.println("Final counter: "+ counter.getValue());
    }
}
```

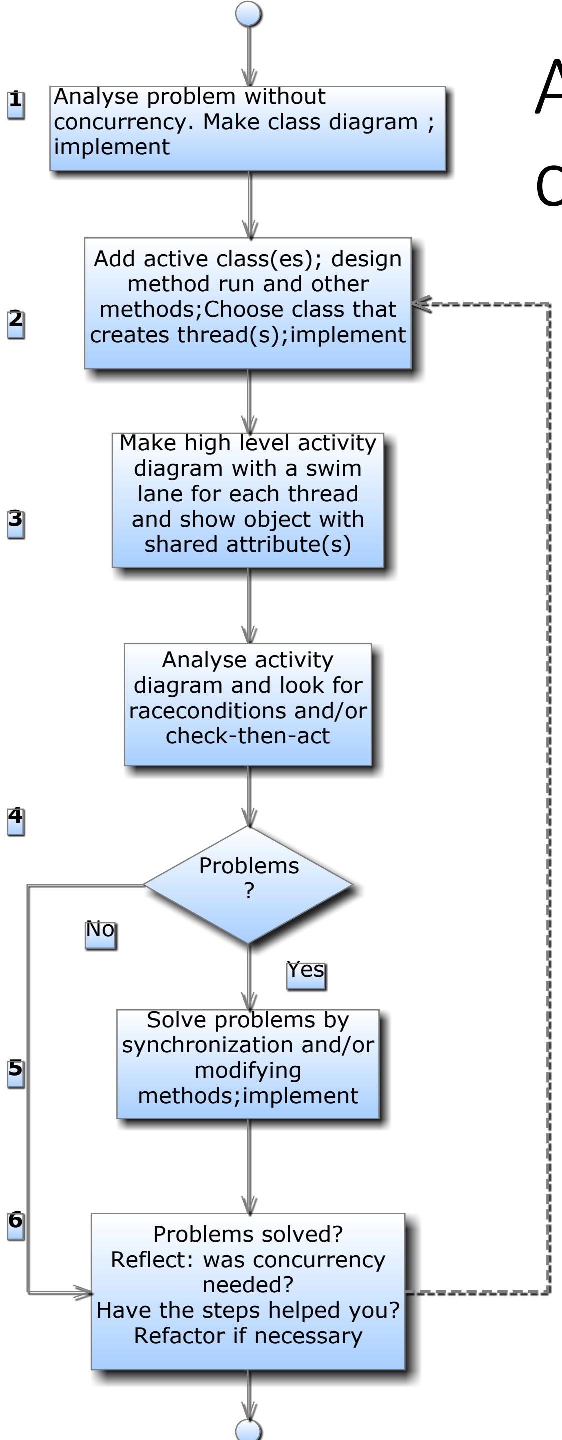
CTE-example (IV)

- Check-then-act problem.
- Synchronizing `getValue` and `incr` is insufficient
- Solution: put `getValue` and `incr` in a synchronized block

```
public class Incrementer implements Runnable {
    private Counter counter;
    private int limit;

    public Incrementer( Counter counter, int limit ) {
        this.counter = counter;
        this.limit   = limit;
    }

    @Override
    public void run() {
        boolean ready = false;
        while ( ! ready ) {
            synchronized ( counter ) {
                if ( counter.getValue() != limit ) {
                    counter.incr();
                } else {
                    ready = true;
                }
            }
        }
    }
}
```


Radboud University

Approach for designing a concurrent program



- This approach consists of 6 steps and guides you to solve a problem by using concurrency, and to discover and handle possible issues caused by the use of threads.
- Step1 and 2 involve creating a concurrent program;
- Step 3, 4 and 5 are necessary to solve the possible (synchronization) problems that are introduced by using threads
- Step 6 is a final check