basic Java threads programming:

- Java threads: definitions, properties
- Interference
- Locking and synchronization
- Waiting and signalling
- Running example: readers and writers

Experiences and high level structures:

- Deadlock, problems with lexical locking
- High level structures for concurrent programming (Java 5.0)
- Example: bounded buffer
Operating systems, processes, and threads

- **OS**
  - allow several processes to run on one machine via preemptive scheduling.
  - Multi-processor machines or “multi-core”-processors: processes can also run truly concurrently/in parallel

- **Thread**:
  - a process that is inexpensive for the OS to create, typically because resources are shared with the process that created the thread.
  - Resource sharing enables effective interaction between threads, but can also lead to interference between threads.

- **Java**
  - and the Java Virtual Machine (JVM): independent of the OS
  - Java threads are built into the language.
  - threads must follow certain rules.
Concrete definition of Java threads

Listing 1: java.lang.Thread

```java
public class Thread
    extends Object
    implements Runnable
{
    // constructors
    Thread() { ... }
    Thread(Runnable target) { ... }

    // methods
    // must be re-implemented by subclasses
    public void run() {}
    public void start()
    ... // much omitted
}

// a related interface
public interface Runnable {
    public void run();
}
```
Basic properties of Java threads

Each thread is controlled by a unique `Thread` object – these two can be treated as one.

Lifecycle for one thread:

- An object of class `Thread` is created
- Life (execution) begins when the `run()` is called
- The thread dies when `run()` terminates, either by returning or an exception

Threads typically communicate via shared state:

- Read/write fields in objects the threads can reference
- Local variables and method parameters: not shared between threads
define::

class Simple extends Thread {
    public void run() {
        System.out.println("Hello threaded world!");
    }
}

use:
Simple s = new Simple();
s.start();

- **start()**: prepares the object for execution.
- **when processor becomes available**: “it” calls s.run().
Example: interference and Java threads

avoid interference during concurrent execution of Java threads.

class Store {
    private int data = 0;
    public void update() { data++; }
}

... // in a method:
Store s = new Store();  // the threads below have access to s
T1 = new FooThread(s);  T1.start();
T2 = new FooThread(s);  T2.start();

Threads T1 and T2 could execute s.update() concurrently!
Interference between T1 and T2 \implies may lose updates to data.
Locks and synchronization

To avoid interference: Threads **synchronize** access to shared data

Mechanism:

1. One unique lock for each object\(^1\) \(o\).
2. **mutex:** at most one thread \(T\) can lock \(o\) at any time.
3. \(T\) makes a request to lock \(o\) in order to execute the block (statements) \(B\) as follows:
   
   ```java
   synchronized (o) {
   B
   }
   ``

   \(T\) blocks (“stops”) until the lock succeeds.
4. As soon as (if) \(T\)’s request succeeds, \(o\) is locked and \(T\) executes \(B\).
5. Several threads can compete in the locking of an object.
   Implementation-dependent order for granting of lock requests.
6. \(T\) may lock \(o\) again from \(B\) (“**reentrance**”).
7. When \(T\) finishes \(B\) or throws an exception: lock on \(o\) is released.

\(^1\)Actually: object, table, and class.
The lock on o is released when the thread leaves the synchronized block (must happen in state ‘running’).
Some conditions and thread primitives omitted.²

Example part II: no interference

Solution to earlier problem: lock the Store objects before executing problematic method:

```java
class Store {
    private int data = 0;
    public void update() {
        synchronized (this) { data++; }
    }
}
```

\dots

// inside a method:
Store s = new Store();

Now only one thread at a time can run s.update().

Short form:

```java
class Store {
    private int data = 0;
    public synchronized void update() {
        data++;
    }
}
```
Example: Sequential readers and writers

```java
class RWbasic {
    protected int data = 0; // the "database"
    public void read() {
        System.out.println("read: " + data);
    }
    public void write() {
        data++;
        System.out.println("wrote: " + data);
    }
}

class Main {
    static RWbasic RW = new RWbasic();
    public static void main(String[] arg) {
        for (int i = 0; i < 10; i++) {
            RW.read();
            RW.write();
        }
    }
}
```
Example: Parallel (unsynchronized) readers and writers

class Reader extends Thread {
    RWbasic RW;    // As before
    public Reader(RWbasic RW) {
        this.RW = RW;
    }
    public void run() {
        for (int i = 0; i < 10; i++)
            RW.read();
    }
}

class Writer ...
    // like Reader, but RW.write() instead

class Main {
    static RWbasic RW = new RWbasic();
    public static void main(String[] arg) {
        new Reader(RW).start();
        new Writer(RW).start();
    }
}
Synchronized methods give mutual exclusion.

```java
class RWexclusive extends RWbasic {
    public synchronized void read() {
        super.read();
    }
    public synchronized void write()
    {
        super.write();
    }
}
```

ie reuses code from class RWbasic.

The classes Reader, Writer and Main now use RWexclusive instead of RWbasic.

No interference, but...
Problem: At most one reader at a time.
Methods for waiting and signalling

```java
public class Object {
    // methods
    public final void wait() throws InterruptedException { ... }
    public final void notify() { ... }
    public final void notifyAll() { ... }

    ... // much omitted
}
```

A thread that calls methods on object `o` must have locked `o` beforehand, typically

```java
synchronized (o) {
    ...
    o.wait();
    ...
}
```
Explanation of wait-related methods

The following is defined for an arbitrary Java object \( o \):

- \( o.wait() \): release lock on \( o \), enter \( o \)'s wait queue and wait
- \( o.notify() \): wake up one thread in \( o \)'s wait queue
- \( o.notifyAll() \): wake up all threads in \( o \)'s wait queue

Objects that wait on \( o \) (\( o.wait() \)) sit in a queue. The ordering of the wait queue is implementation-dependent. The methods \( notify() \)/\( notifyAll() \) have “Signal and Continue” semantics.

Note: A thread which wakes from \( o.wait() \) must lock \( o \) again before it continues. An awakened thread has no advantage in the competition for the lock to \( o \).
There are two independent places where a thread can block, per object o:
- by locking o:  
  \[
  \text{synchronized (o) \{ B \}}
  \]
- by waiting on o:  
  \[
  o.wait()
  \]
Example: Real readers and writers

```java
class ReadersWriters extends RWbasic {
    private int nr = 0;
    private synchronized void startRead() { nr++; }
    private synchronized void endRead() {
        nr--;
        if (nr==0) notify(); // awaken waiting Wrs.
    }
    public void read() { // not synchronized
        startRead(); super.read(); endRead();
    }
    public synchronized void write() {
        while (nr>0)
            try { wait(); }
            catch (InterruptedException ex) {
                return;
            }
        super.write();
        notify(); // awaken another waiting W.
    }
}
```

The classes Reader, Writer and Main now use ReadersWriters instead of RWbasic.

Problem: The solution **favors readers** ("starvation").
Deadlock

\[ T_1: \]
\[
\text{synchronized (a) \{ 
\quad \ldots 
\quad \text{synchronized (b) \{ 
\quad \ldots 
\quad \} 
\quad \ldots 
\} 
\ldots 
\} }
\]

\[ T_2: \]
\[
\text{synchronized (b) \{ 
\quad \ldots 
\quad \text{synchronized (a) \{ 
\quad \ldots 
\quad \} 
\quad \ldots 
\} }
\]

Deadlock if \( T_1 \) locks \( a \) and then \( T_2 \) locks \( b \).
Lexical (blockwise) locking

Locking using `synchronized`:

**Advantages:**
- Automatic unlocking at the end of the `synchronized` block
- Transparent, easy to see when a lock is held

**Disadvantages:**
- When several objects are locked: locks must be released in reverse order
- Locks must be released in the same lexical scope that they were locked in
Java (5.0) supports concurrent programming at a high level of abstraction in the packages `java.util.concurrent*`. Motivation for high-level support:

- More **flexible** and/or abstract than `synchronized`
- Efficient implementations, do not use `synchronized` directly
- Correct implementations of synchronization primitives, e.g. semaphores (frequently a non-trivial task!)
Flexible locks with conditions

Listing 2: Interface java.util.concurrent.locks.Lock

```java
public interface Lock {
    // methods
    public void lock();
    public void unlock();
    public boolean trylock();
    public Condition newCondition();

    ... // omitted
}
```

Listing 3: Interface java.util.concurrent.locks.Condition

```java
public interface Condition {
    public void await() throws InterruptedException;
    public void signal();
    public void signalAll();

    ... // omitted
}
```
Standard implementation and semantics

Offered as standard:

Listing 4: Class java.util.concurrent.locks.ReentrantReadWriteLock

```java
public class ReentrantReadWriteLock
extends Object
implements ReadWriteLock, ...
{
    ... }
```

Semantics of flexible locks

- The same number of calls must be made to `lock()` and `unlock()` for each lock
- Methods in `Condition` have "signal and continue" behaviour
Use of flexible locks

```java
Lock l = ...;
l.lock();
try {
    ... // accesses the resource protected by the lock
} finally {
    l.unlock();
}
```

Code executed when the lock is held should be protected by `try-finally` or similar to ensure that the lock is released in case of interruption.

A **condition** associated with a lock is created as follows:

```java
Lock l = ...;
Condition c = l.newCondition();
```

The lock must be held when the condition is used.
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] buf = new Object[100];
    int rear, front, count; // int defaults to 0

    public void deposit(Object x) throws InterruptedException { ... }

    public Object fetch() throws InterruptedException { ... }
}
class BoundedBuffer {
    // initialization
    public void deposit(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == buf.length) notFull.await();
            buf[rear] = x;
            rear = (rear + 1) % buf.length;
            count++;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}

....
class BoundedBuffer {
    // initialization

    public Object fetch() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0) notEmpty.await();
            Object x = buf[front];
            front = (front + 1) % buf.length;
            count--;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}
Eg.: bounded buffer using \texttt{Condition}

```java
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] buf = new Object[100];
    int rear, front, count; // int defaults to 0

    public void deposit(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == buf.length) notFull.await();
            buf[rear] = x;
            rear = (rear + 1) % buf.length;
            count++;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public Object fetch() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0) notEmpty.await();
            Object x = buf[front];
            front = (front + 1) % buf.length;
            count--;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```
Concurrent programming is difficult...

Method `stop()` in class `Thread`:
- Stops arbitrary threads
- Stopping can break invariants in locked objects
Eventually declared deprecated.

For a long time, there were severe errors in Java’s memory model:
- “Double-checked locking” — a common idiom in concurrent programming – was invalid in Java
- Errors and ambiguities in the definition of low-level behaviour of Java threads
The language JR is Java extended with *language* support for both concurrent and distributed programming. JR represents an alternative approach, inspired by Andrews’ language SR which is used in the textbook.

A JR implementation is freely available for various operating systems.

*The JR Concurrent Programming Language*,
http://www.cs.ucdavis.edu/~olsson/research/jr/