PRAKTIKUM SW2

Multithreading

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

Overview

`Thread`:

- `Thread`-Objects represent execution thread
- defines method for start, interrupt, …
- defines `static` methods for
  - controlling current thread
  - managing thread currently active

`Runnable`:

- interface `Runnable` defined method `run()`, which represents code executed by thread

`Object`:

- class `Object` implements monitor for synchronization
- with essential methods `wait` and `notify/notifyAll`

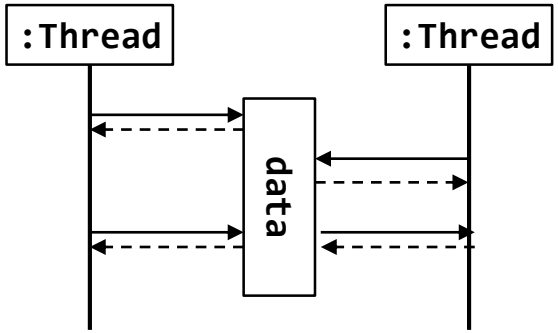
`ThreadGroup`: for building group of threads

`InterruptedException`: exception thrown when interrupted

MULTITHREADING

Threads in Java are concurrent execution threads within VM

Threads in Java run in common memory space

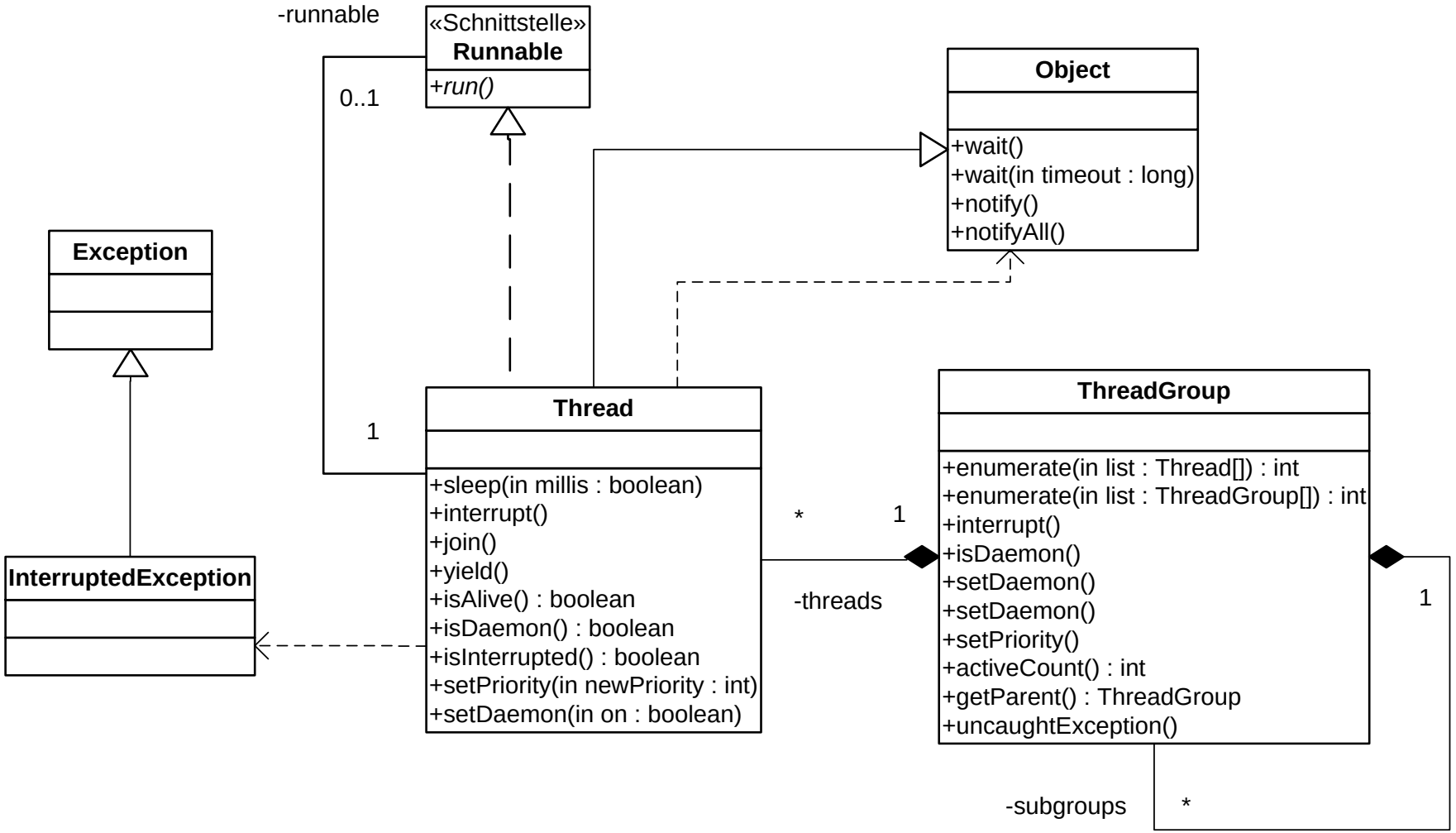


Threads cooperate based on shared objects (shared memory model).

Threads in Java for implementation of concurrent activities, like:

- application logic and user interaction in GUI applications
- server application for serving multiple customers
- Animations of multiple agents
- parallel programs
- …

Class Diagram



Create, Start and Run a Thread

Implement method run() of Runnable

Create thread object with Runnable as parameter

Start thread, which will execute run

```
class MyApp {
   public static void main(...) {
      Runnable runnable = new MyRunnable();
      Thread thread = new Thread(runnable);
      thread.start();
 }
 …

class MyRunnable implements Runnable {
   public void run() {
      try {
          …
         Thread.sleep(1000);
      }
      catch (InterruptedException e){ ... }
   }
}
```

using Lambdas of Java 8:

```
class MyApp {
  public static void main(...) {

     Thread thread = new Thread(() -> {
       try {
         …
        Thread.sleep(1000);
       } catch (InterruptedException e) {
          ...
       }
     });
…
```

Lambda implements Runnable

Thread States

new: created but not started

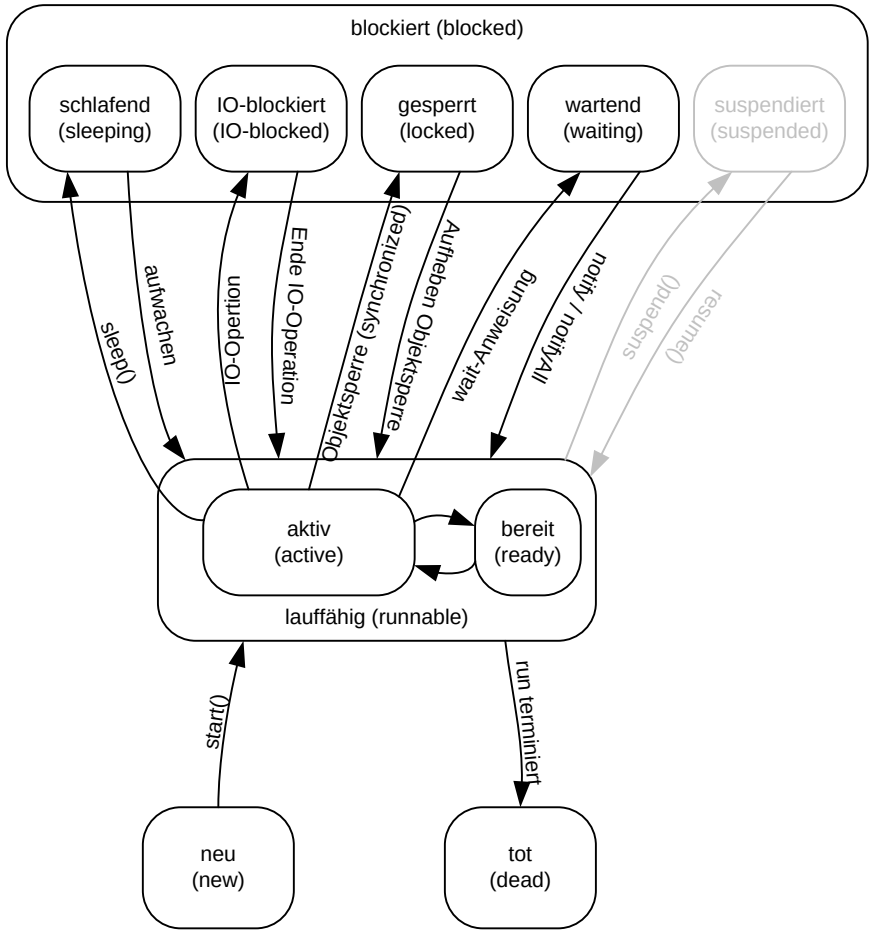
runnable:

- active: is executing
- ready: is ready to be executed

blocked:

- sleeping: sleep() was called
- IO-blocked: waiting for end of IO operation
- waiting: wait was called
- locked: waiting for object lock
- suspended: deprecated

dead: run() method terminated

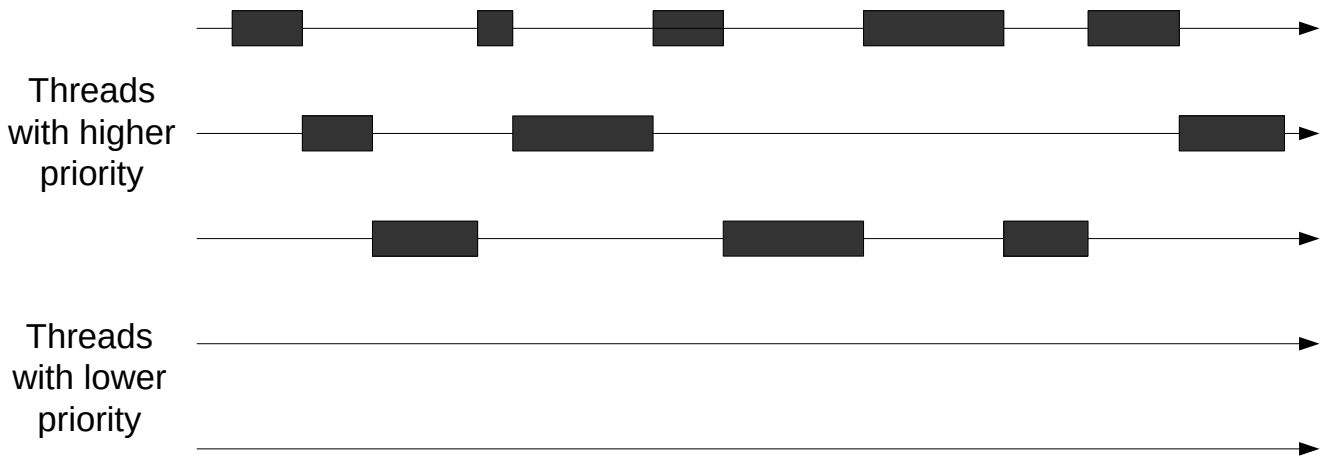


Scheduling und Priorities (1)

Runnable threads have to share the processor

Java does not define a scheduling strategy (but is dependent from operating system)

Use priorities for influencing scheduling



➔ Starvation

Scheduling and Priorities (2)

Method

```
void setPriority(int priority)
```

sets priority of thread

Values are from `MIN_PRIORITY = 0` to `MAX_PRIORITY = 10`

with `NORM_PRIORITY = 5` as default

Example:

```
   Runnable r = new MyRunnable();
   Thread thread = new Thread(r);
   thread.setPriority(priority);
   thread.start();
}
```

Method

```
static void yield()
```

allows releasing the processor so that another can be scheduled

Interrupt and Termination

- Interrupt a thread by calling

  void interrupt()

  which, when blocked currently, will throw and InterruptException

Must be called from another thread!

With static method

static boolen interrupted()

the interrupt status is queried and also reset (!)

- Interrupts may be used for exceptional termination of threads

```
public static void main(String[] args)
  Thread thread = new Thread(() -> {
     int i = 0;
     while (! Thread.interrupted()) {
        try {
           …
           Thread.sleep(1000);
           …
        } catch (InterruptedException e) {
           Thread.currentThread().interrupt();
        }
     }
  });
  thread.start();
```

```
     // do somthing in main thread

     // then stop thread
     thread.interrupt();
  }

}
```

Join

Use join for waiting for other thread to terminate

```
public static void main(String[] args) {
   Thread thread = new Thread(() -> {
      for (int i = 0; i < 1000; i++) {

            …
         }
      }
   });
   thread.start();

   // wait for thread to finish
   try {
      thread.join();
   } catch (InterruptedException e) {

   }
   Out.println("finished");
}
```

End of Application and Daemon Threads

- An application terminated when all threads terminated
- Daemon threads terminate automatically when all non-daemon thread terminated
- Use

```
void setDaemon(boolean on)
```

to define a thread to be Daemon thread

```
public static void main(String[] args) {
   Thread thread = new Thread(() -> {
      for (int i = 0; i < 1000; i++) {

            …
         }
      }
   });
   thread.start();
   thread.setDaemon(true);

   // do somthing in main thread

   Out.println("finished");
}
```

daemon thread terminated when main thread terminates

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

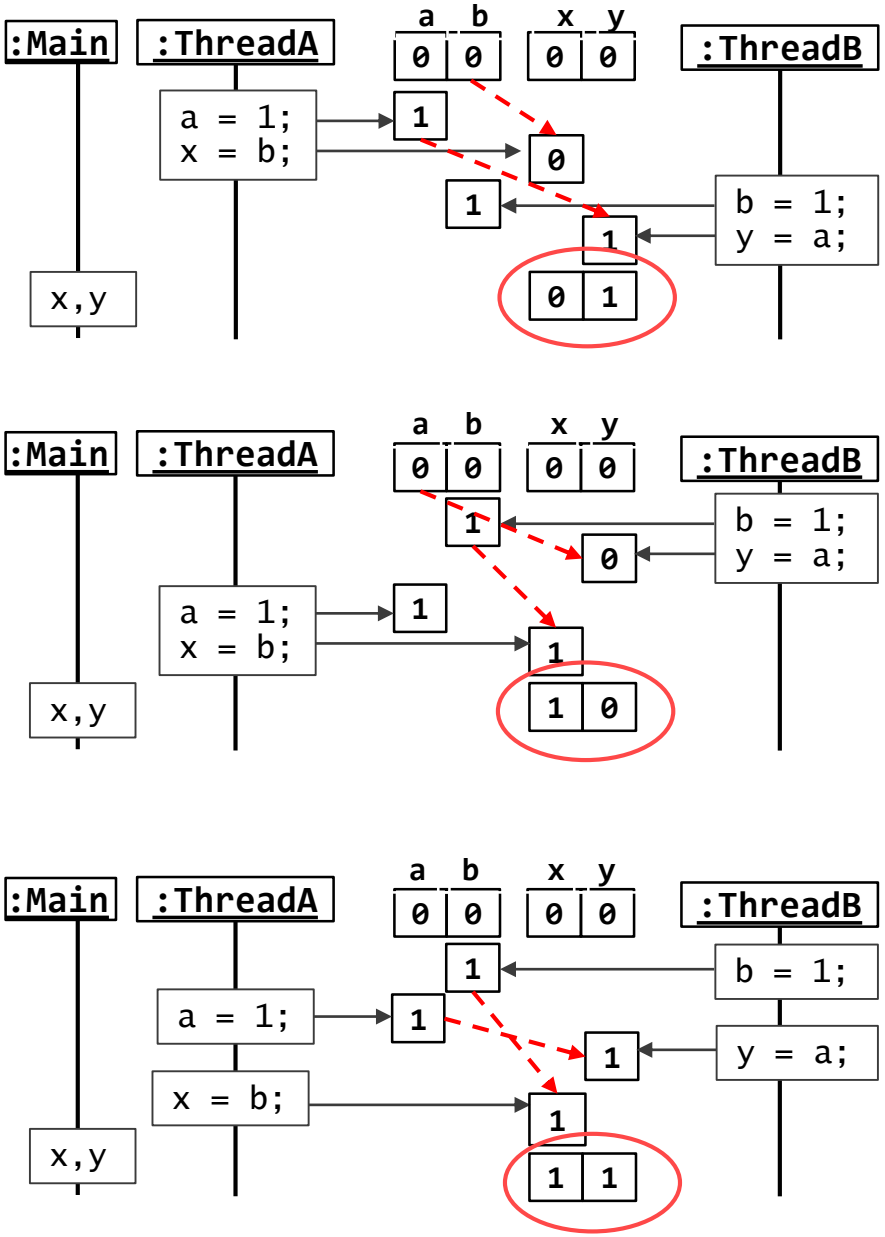
Problems

- Concurrent execution
- Order of operations
- Interrupts at arbitrary program positions
- Non-atomic operations
- Visibility of data between threads (*stale data*)

***Data Race*:**
Order of read and write operations in different threads not deterministic!

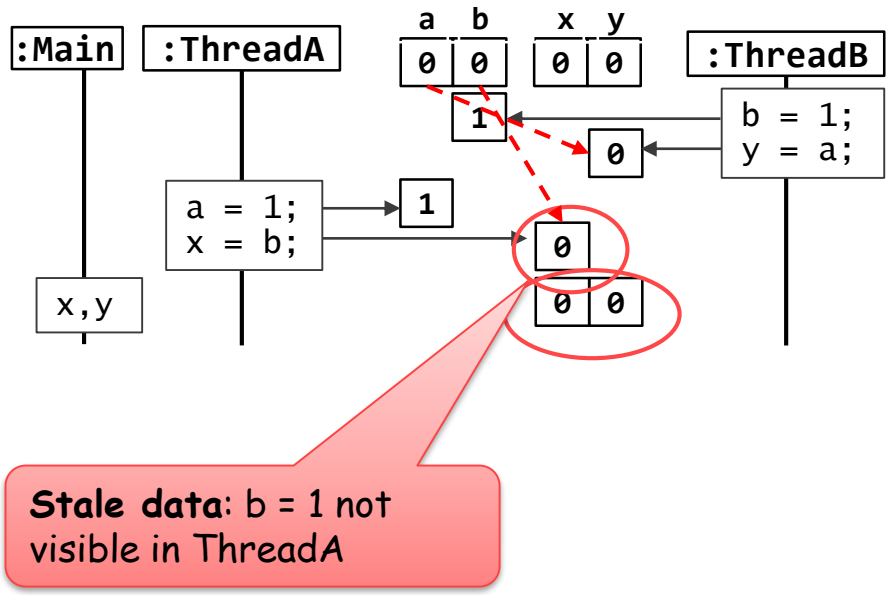
Example: Concurrent Execution

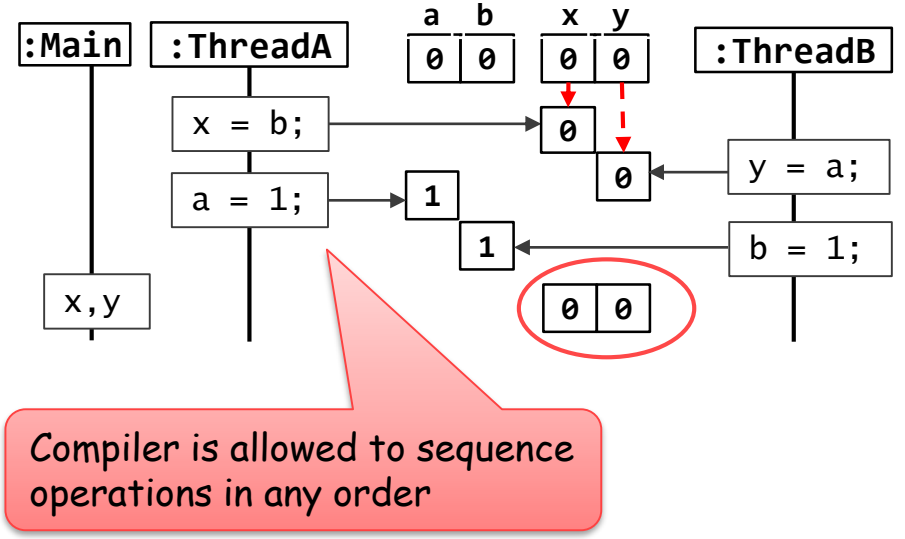
```
public class Demo0_RaceConditions {
  static int x = 0, y = 0;
  static int a = 0, b = 0;

  public static void main(String[] args)
          throws InterruptedException {
    Thread threadA = new Thread(() -> {
      @Override
      public void run() {
        a = 1;
        …
        x = b;
      }
    });

    Thread threadB = new Thread(() -> {
      @Override
      public void run() {
        b = 1;
        …
        y = a;
      }
    });

    threadA.start();
    threadB.start();

    ThreadA.join();
    threadB.join();

    System.out.format("%d, %d", x, y);
  }
}
```



Example: Concurrent Execution

```
public class Demo0_RaceConditions {
  static int x = 0, y = 0;
  static int a = 0, b = 0;

  public static void main(String[] args)
          throws InterruptedException {
    Thread threadA = new Thread(new Runnable() {
      @Override
      public void run() {
        a = 1;
        …
        x = b;
      }
    });

    Thread threadB = new Thread(new Runnable() {
      @Override
      public void run() {
        b = 1;
        …
        y = a;
      }
    });

    threadA.start();
    threadB.start();

    ThreadA.join();
    threadB.join();

    System.out.format("%d, %d", x, y);
  }
}
```





Example Problem: Class not Thread-Safe

Class invariant n >= 0

when interrupted after testing branch condition (n > 0) multiple threads can execute critical statement n--

```
class Resource {
    int n;

    public Resource(int n) {
        this.n = n;
    }

    public boolean seize() {
        if (n > 0) {
            n--;
            return true;
        } else {
            return false;
        }
    }

    public void release() {
        n++;
    }
}
```

```
for (int i = 0; i < n; i++) {
  Thread thread = new Thread(() -> {
    for (;;) {
      if (resource.seize()) {
        ..do some processing
        resource.release();
      }
    }
  });
  thread.start();
}
```

Interrupt after condition test ➔ n-- can be executed by 2 threads simultaneously

Multiple threads execute test (n > 0) concurrently

Example Problem: Compiler Optimizations

Compiler optimizes code of threads independently

- E.g., registers, cache, loops

```
class ValueStore {

    public int value = 0;

}
```

```
public class Demo {
   public static void main(String[] args) {
      ValueStore store = new ValueStore();
      new ValueReader(store).start();
      new ValueWriter(store).start();
   }
}
```

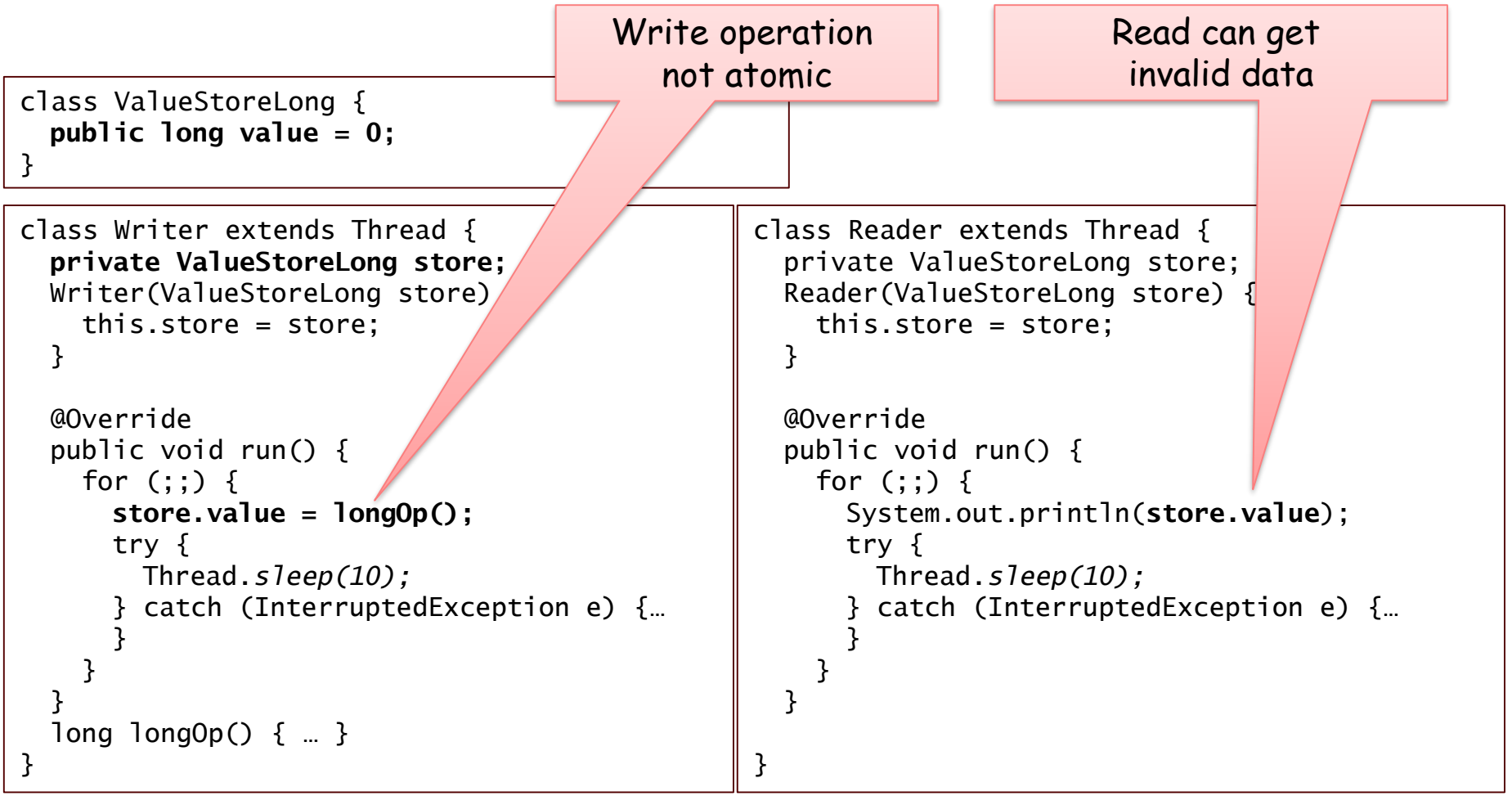
```
class Reader extends Thread {
    ValueStore store;
    Reader(ValueStore store) {
        this.store = store;
    }

    @Override
    public void run() {
        while (store.value < 10) {

            // does not chang
            // store.value
            …
        }
```

Compiler replaces while condition by true because value not changed in this tread!

```
class Writer extends Thread {
    ValueStore store;
    Writer(ValueStore store) {
        this.store = store;
    }

   @Override
   public void run() {
      for (;;) {
         store.value++;
         …

      }
   }
}
```

Example Problem: Operations not Atomic

64-Bit data types written in 2 operations with interrupt in between



Example Problem : Mutual Exclusion

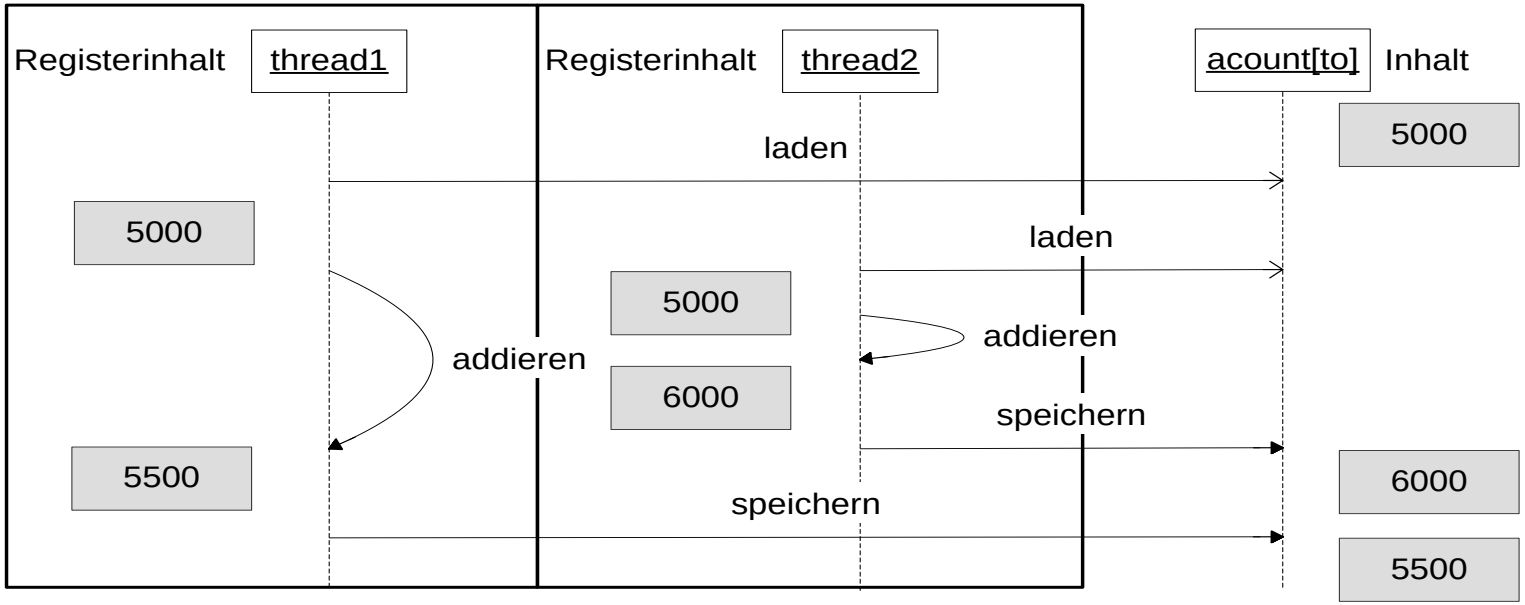
Example Bank:

- method **`transfer`** subtracts amount from first and adds it to second account
- multiple threads concurrently execute transfer operation

```
class Bank {
  int[] accounts = new int[NACCOUNTS];
  …
  void transfer(int from, int to, int amount) {
    account[from] = account[from] - amount;
    account[to] = account[to] + amount;
}
```

```
class TransferThread extends Thread {
  public void run() {
    …
    bank.transfer(from, to, amount);
  }
}
```

```
new TransferThread(..).start();
new TransferThread(..).start();
…
```



Example Problem: Singleton-Pattern

Singleton not guaranteed when simultaneously accessed by multiple threads

```
public class SingletonNoLock {

    private static SingletonNoLock instance;

    public static SingletonNoLock getInstance() {
        if (instance == null) {
            // do something
            instance = new SingletonNoLock();
            n++;
        }
        return instance;
    }

}
```

Block can be entered by multiple threads
➔ Singleton created multiple times

Example Problem: Invalid Initialization

this-pointer leaves constructor before object completely initialized

```
class EscapedThis  {

    public static List<EscapedThis> all = new ArrayList<EscapedThis>();

    public String msg;

    public EscapedThis(String msg) {
        all.add(this); // this escapes here
        …
        this.msg = msg;
    }
}
```

this is added to list before constructor terminates
➔ msg not set

```
for (EscapedThis e : EscapedThis.all.toArray(new EscapedThis[0])) {
    System.out.println(e.msg.length());
}
```

By iteration of list object may not be initialized!
➔ NullPointerException

Example Problem: Unsafe Object Publication

```
public class Holder {
    private int n;

    public Holder(int n) {
        this.n = n;
    }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError(„This is due to unsafe publication!");
    }

}
```

How can that happen?

```
public class Dmeo7_UsafePublication {
    static Holder h;

    public static void main(String[] args) {
        h = new Holder(3);
        new Thread(new Runnable() {
            public void run() {
                h.assertSanity();
        }
    }
}
```

AssertionError may be thrown!

Reason: Stale Data
Visibility of object reference and fields of object not at same time !

Solutions

Stateless methods and immutable data

Immutable data objects

- are thread-safe
- final variables avoid stale data

Single-Thread Rule: everything executed in one thread

- e.g., Swing
- but usually one synchronization point

Thread-safe implementations with

- atomic access operations
- mutual exclusion
- using thread-safe building blocks

Thread-Safe Implementations

Definition: thread-safe

"A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on part of the calling code."

From: B. Goetz, Java Concurrency in Practice, Addison-Wesley, 2006

„Class variants" must be guaranteed also with concurrent execution

without client being forced to make special provisions

Example Problem: Thread-safe Classes

Class invariant: n >= 0

```
public class Resource {
    int n;

    public Resource(int n) {
        this.n = n;
    }

    public boolean seize() {
        if (n > 0) {
            n--;
            return true;
        } else {
            return false;
        }
    }

    public void release() {
        n++;
    }
}
```

Klasse is not thread-safe because invariant >= 0 violtated!

Interrupt after condition test ➔ n-- can be executed by 2 threads simultaneously

Java Memory Model (JMM): Order of Operations and Visibility of Data

Java Memory Model (JMM)

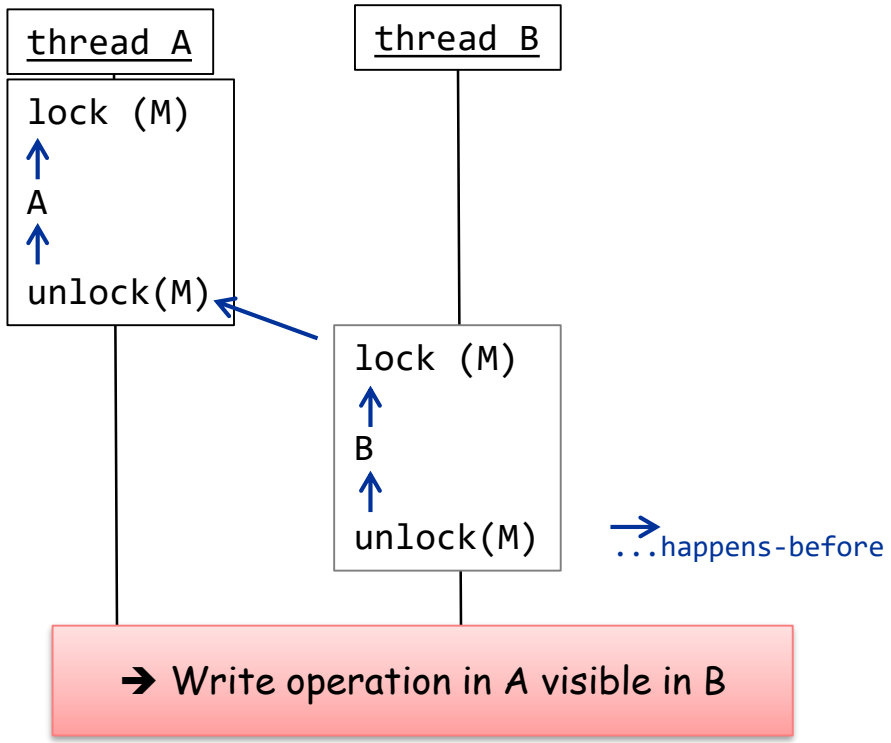
- defines conditions on sequence of operations and visibility of shared data!

„happens-before Relation“

- partial order of synchronization points

Synchronization guarantee visibility of shared data

```
Object M = new Object();

Thread threadA = new Thread(new Runnable() {
  @Override
  public void run() {
    synchronized (M) {
      A;
    }
  }
});

Thread threadB = new Thread(new Runnable() {
  @Override
  public void run() {
    synchronized(M) {
      B;
    }
  }
});

threadA.start();
threadB.start();
```



JMM: HAPPENS-BEFORE

JMM guarantees the following *happens-before* relations between synchronization points

- Monitor: Unlock of monitors *happens-before* following lock of same monitor
- Volatile: Write operation of volatile variable *happens-before* following read operation
- Thread start: Start of thread *happens-before* all operations of thread
- Thread termination: All operations in thread *happens-before* another thread is noticed that thread terminated
- Interrupt: Call of interrupt of a thread *happens-before* thread reacts to interrupt
- Finalizer: End of constructor *happens-before* call to finalizer
- Sequential order: Every synchronization point *happens-before* of all following synchronization points of same thread

FINAL VARIABLES IN JMM

JMM treats final variables specially:

- for final variables there is a value freeze at end of constructor
- The visibility of final fields of all threads guaranteed!

Example: Object publication

```
public class Holder {
    private final int n;

    public Holder(int n) {
        this.n = n;
    }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError(„This is due to unsafe publication!");
    }

}
```

Value freeze for **n** ➔ Wert visible in all threads

Condition cannot be true!

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

VOLATILE

Marking variables as `volatile`

Effect of `volatile`:

- Atomic access also for `double` and `long`
- Visibility of writes in all threads (no stale data)

```
class ValueStore implements Runnable {
  public volatile long value = 0;
}
```

volatile forces atomic write operations

```
class Reader extends Thread {
    ValueStore store;
    …

    @Override
    public void run() {
        while (store.value < 10) {
            …
        }
    }
}
```

value always valid

```
class Writer extends Thread {
    ValueStore store;
    Writer(ValueStore store) {
        this.store = store;
    }

   @Override
   public void run() {
      for (;;) {
         store.value++;
         …

      }
   }
}
```

Atomic Values

Wrapper classes for basic data types and references

- with atomic access operations
- combined operations (test and act)
- visibility in all threads

```
public class Demo2_AtomicIntegerTest {

   static AtomicInteger value = new AtomicInteger(10);

   static class ValueWriter implements Runnable {

      @Override
      public void run() {
         for (;;) {
            if (! value.compareAndSet(0, 10)) {
               System.out.println(value.getAndDecrement());
            }
         }
      }

   }


   public static void main(String[] args) {
      new Thread(new ValueWriter()).start();
      new Thread(new ValueWriter()).start();
      new Thread(new ValueWriter()).start();
      new Thread(new ValueWriter()).start();
   }
```

atomic:
if (value == 0) {
  value = 10;
  return true;
} else {
  return false;
}

atomic:
int prev = value
value = value - 1
return prev

Monitor

Object implements monitor for mutual exclusion

Monitor has lock and maintains queue

- threads can request lock
- threads are queued as waiting for lock
- when available, lock is assigned to a waiting thread

Use method

synchronized

to request lock

Synchronized Methods

- When method is declared as synchronized, method cannot only be executed with lock of this object
- Static methods are locked on Class object

```
class Bank {

    int[] accounts = new int[NACCOUNTS];
     ...
    synchronized void transfer(int from, int to, int amount) {
        accounts[from] -= amount;
        accounts[to] += amount;
    }
    ...
}
```

mutual exclusion on this

Synchronized Block

Blocks can be synchronized on arbitrary object

```
class Bank {
   private Object accountLock = new Object();
   private Object customerLock = new Object();
   ...

   void transfer(int from, int to, int amount) {
         synchronized (accountLock) {
            account[from] -= amount;
            account[to] += amount;
         }
   }

   void addCustomer(Customer customer) {
         synchronized (customerLock) {
            customers.add(customer);
         }
   }
}
```

mutual exclusion on accountLock

mutual exclusion on customerLock

WAIT AND NOTIFY

Thread can “wait” on lock until awoken by “notify”

Object methods

- wait() thread is blocked as waiting for notify on object; lock of object is released
- wait(long timeout) wait with timeout; interrupt occurs after timeout milliseconds
- notify() one thread waiting is waked up
- notifyAll() all threads waiting are waked up

Ex

```
synchronized void transfer(int from, int to, int amount) throws InterruptedException {
    while (accounts[from] < amount) {
            wait();
    }
    accounts[from] -= amount;
    accounts[to] += amount;
    notifyAll();
}
```

Example: BankAccounts (1)

Transfer money between accounts in multiple threads

```
public class SynchBankTest {

   public static final int NACCOUNTS = 10;
   public static final int INITIAL_BALANCE = 10000;

   public static void main(String[] args) {
         Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
         int i;
         for (i = 0; i < NACCOUNTS; i++) {
               TransferThread t = new TransferThread(b, i, INITIAL_BALANCE);
               t.setPriority(Thread.NORM_PRIORITY + i % 2);
               t.start();
         }
      }

}
```

Example: BankAccounts (2)

```
class Bank {

   private final int[] accounts;
   private long ntransacts;

   public Bank(int n, int initialBalance) {
      accounts = new int[n];
      for (int i = 0; i < accounts.length; i++) {
         accounts[i] = initialBalance;
      }
      ntransacts = 0;
   }

   public synchronized void transfer(int from, int to, int amount)
            throws InterruptedException {
      while (accounts[from] < amount) {
         wait();
      }
      accounts[from] -= amount;
      accounts[to] += amount;
      ntransacts++;
      notifyAll();
   }

   ...
}
```

Example: BankAccounts (3)

```
class TransferThread extends Thread {

   private final Bank bank;
   private final int fromAccount;
   private final int maxAmount;

   public TransferThread(Bank b, int from, int max) {
         bank = b;
         fromAccount = from;
         maxAmount = max;
   }

   public void run() {
         try {
            while (!interrupted()) {
               int toAccount = (int) (bank.size() * Math.random());
               int amount = (int) (maxAmount * Math.random());
               bank.transfer(fromAccount, toAccount, amount);
               sleep(1);
            }
         } catch (InterruptedException e) {
         }
   }

}
```

Example: Producer – Consumer (1)

- Producer produces elemente and writes them to buffer
- Consumer takes elements from buffer and consumes them

```
public class ProducerConsumerApp {

   public static void main(String[] args) {

      Buffer buffer = new Buffer();

      Producer p = new Producer(buffer);
      Consumer c = new Consumer(buffer);

      p.start(); // Start Producer
      c.start(); // Start Consumer

      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
          // nothing to do
      } finally {
         p.interrupt();
         c.interrupt();
      }

   }

}
```

```
public class Buffer {

   private Object obj = null;

   synchronized public void put(Object o) {
      obj = o;
   }

   synchronized public Object retrieve() {
      Object o = obj;
      obj = null;
      return o;
   }

    synchronized public boolean isEmpty() {
      return obj == null;
   }

}
```

Example: Producer – Consumer (2)

```
class Producer extends Thread {
   private final Buffer buffer;

   public Producer(Buffer buffer) { this.buffer = buffer; }

   public void run() {
      int i = 0;

      while (!interrupted()) {
         try {
            synchronized (buffer) {               // locking buffer
               while (!buffer.isEmpty()) {
                  buffer.wait();                  // wait for buffer being empty
               }
               Object o = new Integer(i++);
               buffer.put(o);                     // produce element
               System.out.println(„Produzent erzeugte " + o);
               buffer.notifyAll();                // notify waiting consumer
            }
            Thread.sleep((int) (100 * Math.random())); // sleep
         } catch (InterruptedException e) {
            interrupt();
         }
      }
   }
}
```

Example: Producer – Consumer (3)

```
class Consumer extends Thread {
   private final Buffer buffer;

   public Consumer(Buffer buffer) {
         this.buffer = buffer;
   }

   public void run() {
      while (!interrupted()) {
         try {
            synchronized (buffer) {                          // locking buffer
               while (buffer.isEmpty())  {
                  buffer.wait();                             // wait for buffer being empty
               }
               Object o  = buffer.retrieve();                // consume element
               System.out.println("Konsument fand " + o);
               buffer.notifyAll();
            }
            Thread.sleep((int) (100 * Math.random()));       // sleep
         } catch (InterruptedException e) {
            interrupt();
         }
      }
   }
}
```

Collections: unsynchronized Collections

Collections are NOT thread-safe

Result in ConcurrentModificationException

```
static List<Integer> list = new ArrayList<Integer>();

public static void main(String[] args) {
    Thread a = new Thread(() -> {
        for (int i = 1; i < 1000; i++) {
            …
            list.add(i);
        }
    });
    a.start();

    Thread b = new Thread(() -> {
        for (int i = 1000; i < 2000; i++) {
            …
            list.add(i);
        }
    });
    b.start();

    for (int i : list) {
        …
        System.out.println(i);
    }
}
```

add not thread-safe

ConcurrentModificationException

Synchronized Collections

- **Synchronized wrappers for collections proved thread-safe reads and writes**

```
static List<Integer> list = Collections.synchronizedList(new ArrayList<Integer>());

public static void main(String[] args) {
    Thread a = new Thread(() -> {
        for (int i = 1; i < 1000; i++) {
            …
            list.add(i);
        }
    });
    a.start();

    Thread b = new Thread(() -> {
        for (int i = 1000; i < 2000; i++) {
            …
            list.add(i);
        }
    });
    b.start();

    for (int i : list) {
        …
        System.out.println(i);
    }
}
```

synchronized Wrapper

add is thread-safe!

Iteration NOT thread-safe!

But: Iteration NOT thread-safe!

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

Swing

Swing is not thread-safe

Swing is built based on *single thread rule*

- all calls to Swing components in single thread

AWT Thread

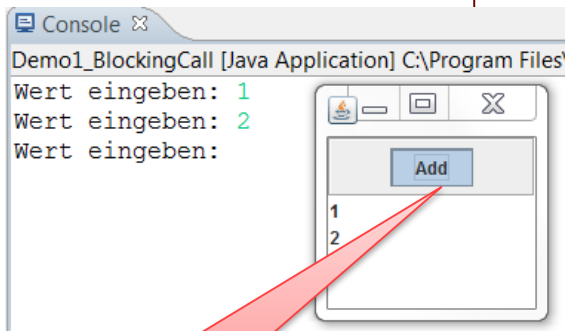
- executes events in event queue:
  - Paint operations
  - Input events, e.g., `actionPerformed`, `mouseClicked`, …

Blocking UI

No long lasting and blocking calls in event methods

➔ **blocking UI**

```
public class Demo1_BlockingCall {
    private static JFrame frame;
    private static JList list;
    private static DefaultListModel listModel;
    private static JButton addBtn;

    public static void main(String[] args) throws InterruptedException {
        frame = new JFrame("AWT Thread Test");
        listModel = new DefaultListModel();
        list = new JList(listModel);
        frame.getContentPane().add(list, BorderLayout.CENTER);
        …
        addBtn = new JButton("Add");
        addBtn.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                Out.print("Wert eingeben: ");
                int x = In.readInt();
                listModel.addElement(x);
            }
        });
    …
```



Blocks UI

Application Thread

Execution of long lasting and blocking operations in separate application thread

➔ Updates not in AWT Thread

➔ Violates **Single-Thread** rule of Swing!

```
...
add1Btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                Out.print("Wert eingeben: ");
                x = In.readInt();
                listModel.addElement(x);
            }
        }).start();
    }
});
```

Executed in AWT Thread

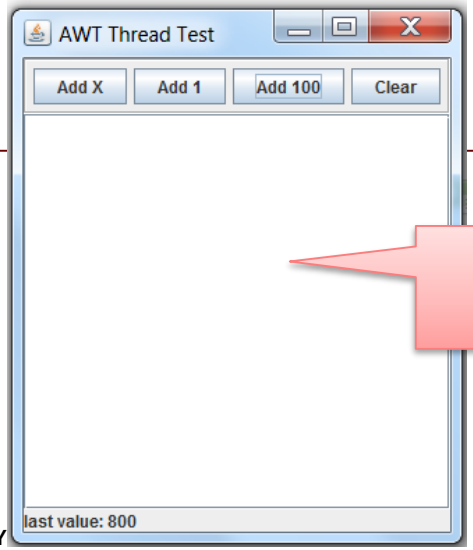
Executed in user thread

Application Thread

Results in failure of Swing

```
private static int y = 0;
public static void main(String[] args) throws InterruptedException {
    …
    add100Btn = new JButton("Add 100");
    add100Btn.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
            new Thread(new Runnable() {

                @Override
                public void run() {
                    for (int i = 0; i < 100; i++) {
                        listModel.addElement(y++);
                    }
                }
            }).start();
        }
    });
```

Adding a 100 elements in ListModel



JList fails!

invokeLater and invokeAndWait

`EventQueue` allows adding tasks into AWT event queue

- `invokeLater`: Adding as future task (asynchronous)
- `invokeAndWait`: Add and then wait until executed in Swing

Note: All operations which update UI elements should be added to Swing event queue by `invokeLater` or `invokeAndWait`!

```
private static int y = 0;
public static void main(String[] args) throws InterruptedException {
    …
    add1Btn.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    Out.print("Wert eingeben: ");
                    x = In.readInt();
                    EventQueue.invokeLater(new Runnable() {
                        public void run() {
                            listModel.addElement(x);
                        }
                    });
                }
            }).start();
        }
    });
```

Adding the updates into AWT EventQueue

invokeLater and invokeAndWait

Beispiel `invokeAndWait`: Adding 100 elements

```
private static int y = 0;
public static void main(String[] args) throws InterruptedException {
    …
    add100Btn.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 100; i++) {
                        y++;
                        try {
                            EventQueue.invokeAndWait(new Runnable() {
                                public void run() {
                                    listModel.addElement(y);
                                }
                            });
                        } catch (InterruptedException e) {
                        } catch (InvocationTargetException e) {
                        }
                    }
                }
            }).start();
        }
    });
```

Blocks until AWT thread has executed task

invokeLater and invokeAndWait

Example `invokeLater`:

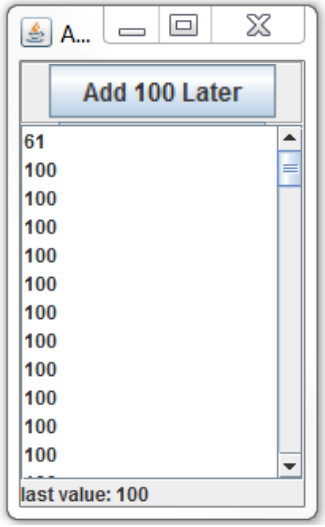
- With `invokeLater` tasks are submitted
- AWT Thread executes tasks later

➔ **Submitting and execution at different points in time**

```
add100LaterBtn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    y++;
                    EventQueue.invokeLater(new Runnable() {
                        public void run() {
                            listModel.addElement(y);
                        }
                    });
                    try { Thread.sleep(1); } catch (InterruptedException e) {}
                }
            }
        }).start();
    }
});
```

Submitting task

When executed by AWT Thread
y already has value 100



Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

Executors

Executors support thread management

- Threads are not created and managed by application
- Thread are reused and managed by Executors
- Overhead can be reduced, in particular, threads creation is limited
- different Executor implementations provide different strategies for thread management

Interface Executor

```
public interface Executor {
   void execute(Runnable command)
}
```

Application

```
Executor executor = …;
Runnable runnable = …;
executor.execute(runnable);
```

instead of working with threads directly:

```
Runnable runnable = …;
(new Thread(runnable)).start();
```

Interface ExecutorService

Advanced interface ExecutorService provides

- submit and invoke methods for submitting tasks
- Lifecycle-management for tasks und executor itself

```
public interface ExecutorService extends Executor {
   void shutdown()
   List<Runnable> shutdownNow()
   boolean isShutdown()
   boolean isTerminated()
   boolean awaitTermination(long timeout, TimeUnit unit)
           throws InterruptedException
   <T> Future<T> submit(Callable<T> task)
   <T> Future<T> submit(Runnable task, T result)
   Future<?> submit(Runnable task)
   <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
           throws InterruptedException
   <T> T invokeAny(Collection<? extends Callable<T>> tasks)
           throws InterruptedException, ExecutionException
}
```

Interface ExecutorService

submit (and invokes) allow Callable<V> and return Future<V>

```
public interface ExecutorService extends Executor {
   …
   <T> Future<T> submit(Callable<T> task)
}
```

- Callable analogous to Runnable but with return value of type V

```
public interface Callable<V> {
   V call() throws Exception
}
```

- Future then allow retrieving the result later

```
public interface Future<V> {
   public boolean isCancelled()
   public boolean isDone()
   public boolean cancel(boolean mayInterruptIfRunning)
   public V get() throws InterruptedException, ExecutionException
   public V get(long timeout, TimeUnit unit)
}
```

Application

- Submit task
- execute task asynchronously
- retrieve result later

```
ExecutorService executorService = …;

Future<String> futureResult = executorService.submit(() -> {
        String result;
        result = … do a long lasting operation …
        return result;
     });

… do something else in main thread until result needed …

String result = futureResult.get();
```

Callable executed asynchronously!

Blocks until result available!

- or check if task is done; if not, e.g. cancel operation

```
… do something else in main thread until result needed …

if (futureResult.isDone()) {
   result = futureResult.get();
} else {
    futureResult.cancel();
}
```

Executor Implementations

- Created by static methods of class `Executors`
- Different implementations with different management strategies

```
public class Executors {
   public static ExecutorService newFixedThreadPool(int nThreads)
   public static ExecutorService newCachedThreadPool()
   public static ExecutorService newSingleThreadExecutor()
   public static ExecutorService newWorkStealingPool()
   ...
```

Application:

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
…
executorService.submit(() -> …);
…
executorService.shutdown();
```

Types of ExecutorServices

FixedThreadPool

```
static ExecutorService newFixedThreadPool(int nThreads)
```

- fix number of worker thread
- tasks must wait for worker thread and are queued
- use especially for long-running, possibly blocking task (e.g. with IO), and when system should not be overloaded by too many threads

CachedThreadPool

```
static ExecutorService newCachedThreadPool()
```

- threads are created on demand but reused
- especially for many short tasks

WorkStealingThreadPool

```
static ExecutorService newWorkStealingPool(int parallelism)
```

- for parallel execution without blocking

ScheduledExecutorService

```
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
```

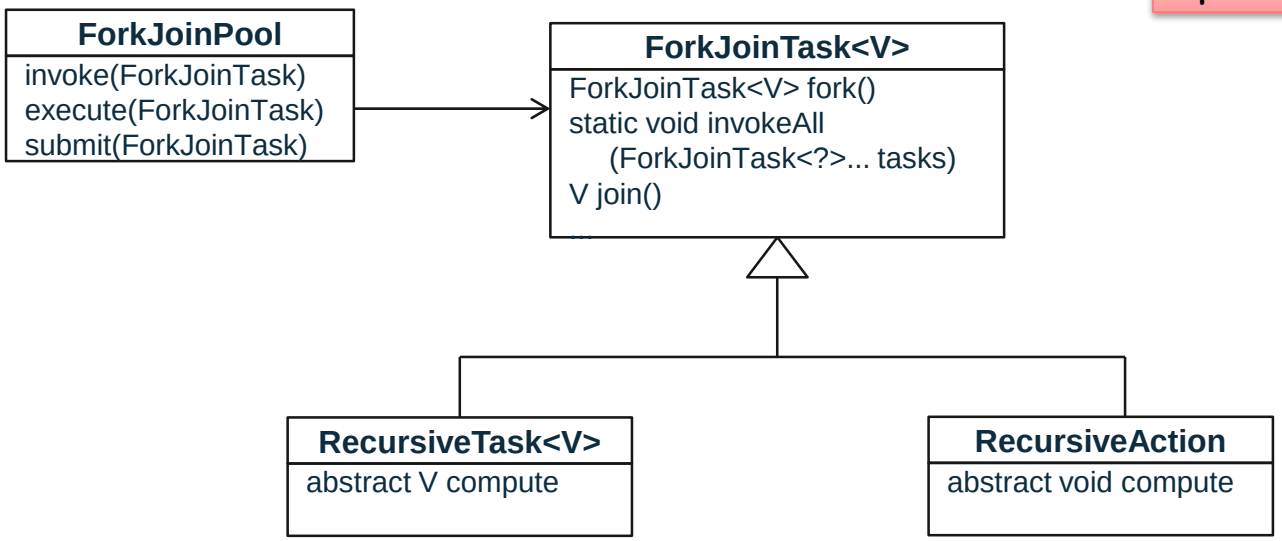
- supports scheduling of tasks in time (see later)

Fork-Join Pool

Fork-Join API

- ForkJoinPool is thread-pool with „work stealing“
- Tasks are executed in ForkJoinPool and can create further tasks
- In ForkJoinPool work stealing will distribute tasks within work threads, i.e., as soon as task is blocked, the task is freed so that it can be used for other tasks
- RecursiveTask and RecursiveAction are base classes for recursive and hierarchical task structures
- Especially suitable for parallel execution on multicore systems

e.g. also used in parallel streams!



Execution by Fork-Join

Fork-Join works according to the Divide&Conquer approach

- **RecursiveTask**s recursively create subtasks
- which are then again executed in Fork-Join thread pool

Approach of RecursiveTasks

```
public class MyRecursiveTask<T> extends RecursiveTask<T> {

  @Override
  protected T compute() {
    if (problem small) {
      result = solve problem sequentially
      return result;
    }

    split sub-problem and
    create subtask for sub-problem
    send task to fork-join pool
    ... possibley more splits

    join with subtasks for partial solutions

    result = combine partial solutions
    return result;
  }
```

is problem small, then solve it sequentially I

Split problem in subproblems and send thtos for execution to Fork-Join Thread-Pool

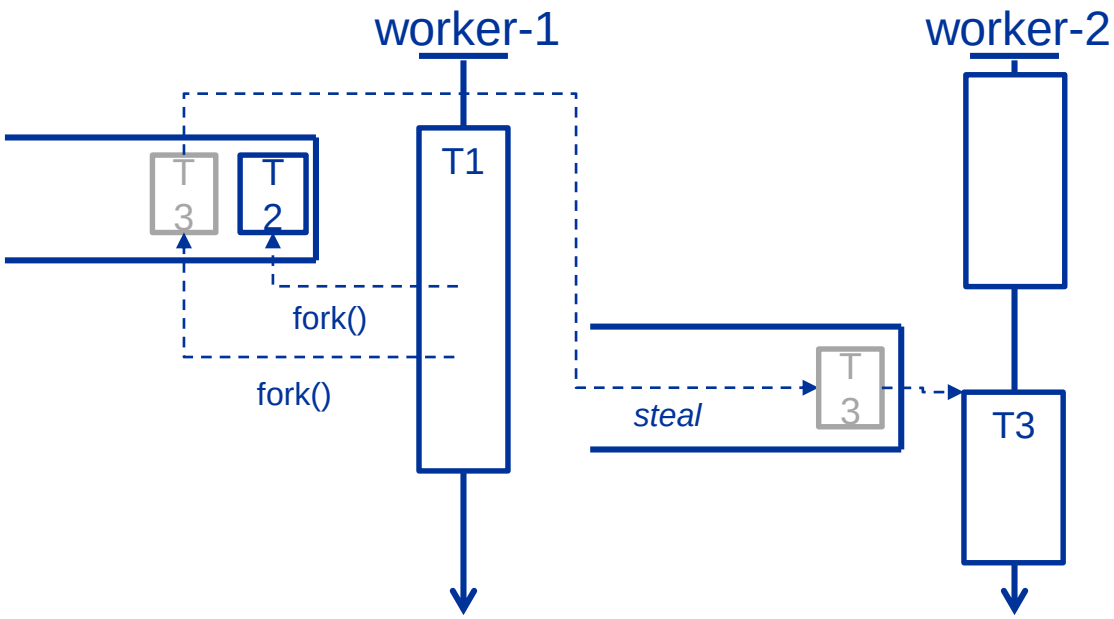
Wait for result of sub-problems

Combine sub-solutions to solution and return it

Fork-Join Pool: Work Stealing

Fork-Join Pool

- works with fix number of threads (usually dependent on number of CPU cores, e.g. 3)
- each thread maintains a queue of tasks
  - **fork()** creates new task and puts it into queue of current thread
- work stealing
  - idle threads then “steal” tasks from queue of other threads



- This results in balanced utilization of threads

Fork-Join Pool: Example Image Blur (1/2)

Image Blurring parallel according to Divide-and-Conquer approach

```
public class ForkBlurAction extends RecursiveAction {
    private int[] mSource;
    private int mStart;
    private int mLength;
    private int[] mDestination;

    public ForkBlurAction(int[] src, int start, int length, int[] dst) {
      …
    }

    protected void computeDirectly() {
        int sidePixels = (mBlurWidth - 1) / 2;
        for (int index = mStart; index < mStart + mLength; index++) {
            … Calculate average.
         }
    }
    protected static int sThreshold = 1000;

    protected void compute() {
        if (mLength < sThreshold) {
            computeDirectly();
            return;
        }
        int split = mLength / 2;
        invokeAll(new ForkBlurAction(mSource, mStart, split, mDestination),
                  new ForkBlurAction(mSource, mStart + split, mLength - split,
                                     mDestination));
    }
}
```

Sequential

if problem small, compute it

Split problem in 2 parts

Submit 2 parts to Fork-Join-Pool

Fork-Join Pool: Example Image Blur (2/2)

```
public class ForkBlurMain {

   static int N = 100_000_000;

   public static void main(String[] args) {

      Random r = new Random();

      int[] src = new int[N];
      for (int i = 0; i < src.length; i++) {
         src[i] = r.nextInt();
      }
      int[] dst = new int[N];

      ForkBlurAction fb = new ForkBlurAction(src, 0, src.length, dst);

      ForkJoinPool pool = new ForkJoinPool();

      pool.invoke(fb);
   }

}
```

Create initial Action object

Create ForkJoinPool

Submit initial action to pool

ScheduledExecutorService

ScheduledExecutorService allow

- delayed task execution
- periodic task execution

```
public interface ScheduledExecutorService extends ExecutorService {
   <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)
   ScheduledFuture<?> schedule(Runnable cmd, long delay, TimeUnit unit)
   ScheduledFuture<?> scheduleAtFixedRate(Runnable cmd, long initDelay, long period, TimeUnit unit)
   ...
}
```

Example: periodic output of time

```
ScheduledExecutorService e = Executors.newScheduledThreadPool(1);

e.scheduleAtFixedRate(
      () -> {
         LocalTime now = LocalTime.now();
         System.out.format("%2d:%02d:%02d%n", now.getHour(), now.getMinute(), now.getSecond());
      },
      0,
      1,
      TimeUnit.SECONDS);
```

```
18:39:03
18:39:04
18:39:05
18:39:06
18:39:07
18:39:08
```

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

Synchronizers

Lock und Conditions

- efficient and flexible variant of monitor; replacement of monitor of Object

CountDownLatch

- Initialized with initial value
- `await()`: wait until counter is 0
- `countDown()`: decrement counter

Semaphore

- access to a restricted resource with n authorizations
- using `acquire()` and `release()`

CyclicBarrier

- synchronization of multiple threads

Exchanger

- data exchange

SynchronousQueue

- synchronous exchange of values

Lock and Condition

Lock und Condition as efficient and more flexible variant of monitor

```
class BankWithLock {
   private double[] accounts;
   private Lock bankLock;
   private Condition sufficientFunds;
   …
   public BankWithLock(int n, double initialBalance) {
      accounts = new double[n];
      bankLock = new ReentrantLock();
      sufficientFunds = bankLock.newCondition();
      for (int i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
   }

   public void transfer(int from, int to, double amount) throws InterruptedException {
      bankLock.lock();
      try {
         while (accounts[from] < amount)
            sufficientFunds.await();
         }
         accounts[from] -= …
         sufficientFunds.signalAll();
      } finally {
         bankLock.unlock();
      }
   }
   …
```

Lock and Condition

Locking each account

```
class BankWithMultipleLocks {
   private double[] accounts;
   private Lock[] accountLocks;
   private Condition[] sufficientFunds;

   public BankWithLock(int n, double initialBalance) {
      accounts = new double[n];
      accountLocks = new ReentrantLock[n];
      sufficientFunds = new Condition[n];
      for (int i = 0; i < accounts.length; i++) {
         accountLocks[i] = new ReentrantLock();
         sufficientFunds[i] = accountLocks[i].newCondition();
         accounts[i] = initialBalance;
      }
   }
   public void transfer(int thread, int from, int to, double amount)throws … {
      accountLocks[from].lock();
      accountLocks[to].lock();
      try {
         while (accounts[from] < amount)
            sufficientFunds[from].await();
         accounts[from] -= amount;
         accounts[to] += amount;
         sufficientFunds[to].signalAll();
      } finally {
         accountLocks[from].unlock();
         accountLocks[to].unlock();
      }
   }
}
```

Deadlock!

Lock: tryLock

- tryLock with timeout avoids deadlocks

```
…
   public void transfer(int thread, int from, int to, double amount) throws … {
      if (accountLocks[from].tryLock(1000, TimeUnit.MILLISECONDS)) {
         try {
            if (accountLocks[to].tryLock(1000, TimeUnit.MILLISECONDS)) {
               try {
                  while (accounts[from] < amount) sufficientFunds[from].await();
                  accounts[from] -= amount;
                  accounts[to] += amount;
                  sufficientFunds[to].signalAll();
               } finally {
                  accountLocks[to].unlock();
               }
            } else {
               System.out.println(thread + ":failed to get lock for " + to);
            }
         } finally {
            accountLocks[from].unlock();
         }
      } else {
         System.out.println(thread + ":failed to get lock for " + from);
      }
   }
…
```

CountDownLatch

Example: synchronization of start and stop of multiple threas

```
public class CountDownLatchDemo {
    private CountDownLatch startGate;
    private CountDownLatch endGate;

    public void startTasks(Task[] tasks) {
        try {
            startGate = new CountDownLatch(1);
            endGate = new CountDownLatch(tasks.length);
            for (Task task : tasks) {
                new Thread(task).start();
            }
            System.out.println("startet");
            long startTime = System.nanoTime();
            startGate.countDown();
            endGate.await();
            System.out.format("finished with run time %d", System.nanoTime() - startTime);
        } catch (InterruptedException ex) {  … }
    }

    class Task implements Runnable {
        @Override
            public void run() {
                try {
                    startGate.await();
                    // do something …..
                } finally {
                    endGate.countDown();
                }
            }
    }
```

wait that all tasks are finished

wait that all tasks are started

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

Concurrent Collections

Package java.util.concurrent

- **Optimized for concurrent access from multiple threads**
- **Do not throw `ConcurrentModificationExceptions`**
- **Extended atomic access operations like putIfAbsent, replace**

Interface		Concurrent Implemenation
Map	➔	ConcurrentHashMap
Set	➔	CopyOnWriteArraySet
SortedMap	➔	ConcurrentSkipListMap
SortedSet	➔	ConcurrentSkipListSet
List	➔	CopyOnWriteArrayList
BlockingQueue	➔	ArrayBlockingQueue LinkedBlockingQueue

The Iterator is a "weakly consistent" iterator that will never throw ConcurrentModificationException, and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction.

Concurrent Collections

- **Concurrent collections are thread-safe**

```
static Queue<Integer> list = new ConcurrentLinkedQueue<Integer>();

public static void main(String[] args) {
    Thread a = new Thread(() -> {
        for (int i = 1; i < 1000; i++) {
            …
            list.add(i);
        }
    });

    a.start();

    Thread b = new Thread(() -> {
        for (int i = 1000; i < 2000; i++) {
            …
            list.add(i);
        }
    });
    b.start();

    for (int i : list) {
        …
        System.out.println(i);
    }
```

concurrent collection

add is thread-safe!

Iteration is thread-safe!

BlockingQueue

Provide different methods which differ in reactions when access condition is not fulfilled (e.g. queue empty or full):

- **add**, **remove**, **element** throw exceptions
- **put** and **take** block until condition fulfilled
- **offer**, **poll** and **peek** return values showing success or failure
- **offer** and **poll** with time out block for a specific time then return with value

	Throw exceptions	***Return special value***	***Block***	***Time out***
Insert	add(e)	offer(e)	put(e)	offer(e, timeout, timeunit)
Remove	remove()	poll()	take()	poll(timeout, timeunit)
Examine	element()	peek()	*NA*	*NA*

Block for specified time

Example: Producer-Consumer with BlockingQueue

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
class Producer extends Thread {
    private BlockingQueue<Integer> buffer;
    public Producer(BlockingQueue buffer) {
        this.buffer = buffer;
    }
    public void run() {
        int i = 0;
        while (!interrupted()) {
            try {
                Integer o = new Integer(i++);
                buffer.put(o);
                System.out.println("Produziert " + o);
            } catch (InterruptedException ex) {
            }
        }
    }
}
class Consumer extends Thread {
    private BlockingQueue buffer;
    public Consumer(BlockingQueue buffer) {
        this.buffer = buffer;
    }
    public void run() {
        while (!interrupted()) {
            try {
                Object o = buffer.take();
                System.out.println("Konsumiert " + o);
            } catch (InterruptedException ex) {
            }
        }
    }
}
```

Blocks when buffer full

Blocks when buffer empty

```
public class ProducerConsumerApp {
    private static final int CAPACITY = 3;
    public static void main(String[] args) {
        BlockingQueue<Integer> buffer =
            new SynchronousQueue<Integer>
                    (CAPACITY, true);
        Producer p1 = new Producer(buffer);
        Producer p2 = new Producer(buffer);
        Consumer c1 = new Consumer(buffer);
        Consumer c2 = new Consumer(buffer);
        p1.start();
        p2.start();
        c1.start();
        c2.start();
        …
    }
}
```

This class supports an optional fairness policy for ordering waiting producer and consumer threads. By default, this ordering is not guaranteed. However, a queue constructed with fairness set to true grants threads access in FIFO order. Fairness generally decreases throughput but reduces variability and avoids starvation.

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

Example Total File Size

aus: V. Subramaniam: Programming Concurrency on the JVM

In the following different variants of determining the total size of files in directory tree

Sequential approach:

```
public class SequentialTotalFileSize {

    private long getTotalSizeOfFilesInDir(final File file) {
        if (file.isFile())
            return file.length();
        final File[] children = file.listFiles();
        long total = 0;
        if (children != null)
            for (final File child : children)
                total += getTotalSizeOfFilesInDir(child);
        return total;
    }

    public static void main(final String[] args) {
        final long start = System.nanoTime();
        final long total = new TotalFileSizeSequential().getTotalSizeOfFilesInDir(new File(args[0]));
        final long end = System.nanoTime();
        System.out.println("TotalFileSizeSequential");
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start) / 1.0e9);
    }
}
```

Results strongly dependent of computer and operating system!

```
> java SequentialTotalFileSize C:\Users\hp
SequentialTotalFileSize
Total Size: 83623093266
Time taken: 34.442712065
```

```
> java SequentialTotalFileSize C:\Users\hp
SequentialTotalFileSize
Total Size: 83689700425
Time taken: 17.180551995
```

Example Total File Size: Variant 1 (1/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

Total File Size with Callables and Futures

```
public class NaivelyConcurrentTotalFileSize {

    private long getTotalSizeOfFilesInDir(final ExecutorService service, final File file)
            throws InterruptedException, ExecutionException, TimeoutException {
        if (file.isFile())
            return file.length();
        long total = 0;
        final File[] children = file.listFiles();
        if (children != null) {
            final List<Future<Long>> partialTotalFutures = new ArrayList<Future<Long>>();
            for (final File child : children) {
                partialTotalFutures.add(service.submit(new Callable<Long>() {
                    public Long call() throws InterruptedException, ExecutionException, TimeoutException {
                        return getTotalSizeOfFilesInDir(service, child);
                    }
                }));
            }
            for (final Future<Long> partialTotalFuture : partialTotalFutures)
                total += partialTotalFuture.get(100, TimeUnit.SECONDS);
        }
        return total;
    }


    private long getTotalSizeOfFile(String fileName)
                throws InterruptedException, ExecutionException, TimeoutException {
        final ExecutorService service = Executors.newFixedThreadPool(100);
        try {
            return getTotalSizeOfFilesInDir(service, new File(fileName));
        } finally {
            service.shutdown();
        }
    }
    ...
```

get is blocking call
➔ Task is blocked and also thread is occupied

Example Total File Size: Variant 2 (2/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

```
    ...
    public static void main(final String[] args) throws InterruptedException,
            ExecutionException, TimeoutException {
        final long start = System.nanoTime();
        final long total = new NaivelyConcurrentTotalFileSize().getTotalSizeOfFile(args[0]);
        final long end = System.nanoTime();
        System.out.println("NaivelyConcurrentTotalFileSize");
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start) / 1.0e9);
    }
}
```

```
> java NaivelyConcurrentTotalFileSize C:\Users\hp
Exception in thread "main" java.util.concurrent.TimeoutException
  at java.util.concurrent.FutureTask.get(FutureTask.java:201)
  at pcj.NaivelyConcurrentTotalFileSize.getTotalSizeOfFilesInDir(NaivelyConcurrentTotalFileSize.java:44)
  at pcj.NaivelyConcurrentTotalFileSize.main(NaivelyConcurrentTotalFileSize.java:64)
```

Program terminates with timeout!

Analysis:

- After submitting callables task is blocked in Future.get
- Thread will not be freed
- System runs out of threads in ThreadPool
- Result is a so-called „pool induced deadlock“

➔ Working with futures generally problematic because get blocks!

Example Total File Size: Variant 2 (1/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

Total File Size with collection of partial results of a directory without recursive calls

```
public class ConcurrentTotalFileSize {

    private long getTotalSizeOfFilesInDir(final File file)
            throws InterruptedException, ExecutionException, TimeoutEx
        final ExecutorService service =
                    Executors.newFixedThreadPool(100);
        try {
            long total = 0;
            final List<File> directories = new ArrayList<File>();
            directories.add(file);
            while (!directories.isEmpty()) {
                final List<Future<SubDirsAndSize>> partialResults = new ArrayList<Future<SubDirsAndSize>>();
                for (final File directory : directories) {
                    partialResults.add(service.submit(new Callable<SubDirsAndSize>() {
                                public SubDirsAndSize call() {
                                    return getTotalAndSubDirs(directory);
                                }
                            }));
                }
                directories.clear();
                for (final Future<SubDirsAndSize> partialResultFuture : partialResults) {
                    final SubDirsAndSize subDirectoriesAndSize = partialResultFuture.get(100, TimeUnit.SECONDS);
                    directories.addAll(subDirectoriesAndSize.subDirs);
                    total += subDirectoriesAndSize.size;
                }
            }
            return total;
        } finally {
            service.shutdown();
        }
    }
    ...
```

```
class SubDirsAndSize {
   final public long size;
   final public List<File> subDirs;

   public SubDirsAndSize(long ts, List<File> sd)
     size = ts;
     subDirs = sd;
   }
}
```

while not all subdirectories handled

submit task for file

get results

Add found result of subdirectories to list of directories still to handle

Example Total File Size: Variant 2 (2/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

```
    ...
    private SubDirectoriesAndSize getTotalAndSubDirs(final File file) {
        long total = 0;
        final List<File> subDirectories = new ArrayList<File>();
        if (file.isDirectory()) {
            final File[] children = file.listFiles();
            if (children != null)
                for (final File child : children) {
                    if (child.isFile())
                        total += child.length();
                    else
                        subDirectories.add(child);
                }
        }
        return new SubDirectoriesAndSize(total, subDirectories);
    }

    public static void main(final String[] args) throws InterruptedException,
            ExecutionException, TimeoutException {
        final long start = System.nanoTime();
        final long total = new ConcurrentTotalFileSize().getTotalSizeOfFilesInDir(new File(args[0]));
        final long end = System.nanoTime();
        System.out.println("ConcurrentTotalFileSize");
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start) / 1.0e9);
    }
}
```

```
> java ConcurrentTotalFileSize C:\Users\hp
ConcurrentTotalFileSize
Total Size: 5696929234
Time taken: 13.404754691
```

```
> java ConcurrentTotalFileSize C:\Users\hp
ConcurrentTotalFileSize
Total Size: 5695150268
Time taken: 11.58708715
```

Example Total File Size: Variant 3 (1/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

- Total File Size with partial results of tasks summed up in AtomicLong
- and CountDownLatch for synchronization

```
public class ConcurrentTotalFileSizeWLatch {

    private ExecutorService service;
    final private AtomicLong pendingTasks = new AtomicLong();
    final private AtomicLong totalSize = new AtomicLong();
    final private CountDownLatch latch = new CountDownLatch(1);

    private void updateTotalSizeOfFilesInDir(final File file) {
        long fileSize = 0;
        if (file.isFile())
            fileSize = file.length();
        else {
            final File[] children = file.listFiles();
            if (children != null) {
                for (final File child : children) {
                    if (child.isFile())
                        fileSize += child.length();
                    else {
                        pendingTasks.incrementAndGet();
                        service.execute(new Runnable() {
                            public void run() {
                                updateTotalSizeOfFilesInDir(child);
                            }
                        });
                    }
                }
            }
        }
        totalSize.addAndGet(fileSize);
        if (pendingTasks.decrementAndGet() == 0)
            latch.countDown();
    }
```

thread-safe objects

Submit task for file
Increment counter of pending tasks

Update fileSize ➔ thread-safe

Update pending tasks + CountDownLatch ➔ thread-safe

Example Total File Size: Variant 3 (2/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

```
    ...
    private long getTotalSizeOfFile(final String fileName)
            throws InterruptedException {
        service = Executors.newFixedThreadPool(100);
        pendingFileVisits.incrementAndGet();
        try {
            updateTotalSizeOfFilesInDir(new File(fileName));
            latch.await(100, TimeUnit.SECONDS);
            return totalSize.longValue();
        } finally {
            service.shutdown();
        }
    }

    public static void main(final String[] args) throws InterruptedException,
            ExecutionException, TimeoutException {
        final long start = System.nanoTime();
        final long total = new ConcurrentTotalFileSizeWLatch().getTotalSizeOfFile(args[0]);
        final long end = System.nanoTime();
        System.out.println("ConcurrentTotalFileSizeWLatch");
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start) / 1.0e9);
    }
}
```

Start with root directory

Wait all tasks completed

```
> java ConcurrentTotalFileSizeWLatch C:\Users\hp
ConcurrentTotalFileSizeWLatch
Total Size: 83625673020
Time taken: 17.499908567
```

```
> java ConcurrentTotalFileSizeWLatch C:\Users\hp
ConcurrentTotalFileSizeWLatch
Total Size: 83663127072
Time taken: 12.961295139
```

Example Total File Size: Variant 4 (1/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

- TotalFile Size with Queue of partial results

```
public class ConcurrentTotalFileSizeWQueue {
    private ExecutorService service;
    final private BlockingQueue<Long> fileSizes = new ArrayBlockingQueue<Long>(500);
    final AtomicLong pendingTasks = new AtomicLong();

    private void startExploreDir(final File file) {
        pendingTasks.incrementAndGet();
        service.execute(new Runnable() {
            public void run() {
                exploreDir(file);
            }
        });
    }
    private void exploreDir(final File file) {
        long fileSize = 0;
        if (file.isFile())
            fileSize = file.length();
        else {
            final File[] children = file.listFiles();
            if (children != null)
                for (final File child : children) {
                    if (child.isFile())
                        fileSize += child.length();
                    else
                        startExploreDir(child);
                }
        }
        try {
            fileSizes.put(fileSize);
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
         pendingTasks.decrementAndGet();
    }
```

thread-safe objects

Submit task for file
Increment counter of pending tasks

Add partial result to queue
➔ thread-safe

Example Total File Size: Variant 4 (2/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

```
    ...
    private long getTotalSizeOfFile(final String fileName)
            throws InterruptedException {
        service = Executors.newFixedThreadPool(100);
        try {
            startExploreDir(new File(fileName));
            long totalSize = 0;
            while (pendingTasks.get() > 0 || fileSizes.size() > 0) {
                final Long size = fileSizes.poll(10, TimeUnit.SECONDS);
                totalSize += size;
            }
            return totalSize;
        } finally {
            service.shutdown();
        }
    }

    public static void main(final String[] args) throws InterruptedException {
        final long start = System.nanoTime();
        final long total = new ConcurrentTotalFileSizeWQueue().getTotalSizeOfFile(args[0]);
        final long end = System.nanoTime();
        System.out.println("ConcurrentTotalFileSizeWQueue");
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start) / 1.0e9);
    }
}
```

Processing reults from queue!

```
> java ConcurrentTotalFileSizeWQueue C:\Users\hp
ConcurrentTotalFileSizeWQueue
Total Size: 83661507904
Time taken: 13.3983663
```

```
> java ConcurrentTotalFileSizeWQueue C:\Users\hp
ConcurrentTotalFileSizeWQueue
Total Size: 83691338377
Time taken: 10.754997324
```

Example Total File Size: Variant 5 (1/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

- **TotalFile Size with Fork-Join API**

```
public class ForkJoinTotalFileSize {
    private final static ForkJoinPool forkJoinPool = new ForkJoinPool();

    private static class FileSizeFinder extends RecursiveTask<Long> {
        final File file;
        public FileSizeFinder(final File theFile) {
            file = theFile;
        }

        @Override
        public Long compute() {
            long size = 0;
            if (file.isFile()) {
                size = file.length();
            } else {
                final File[] children = file.listFiles();
                if (children != null) {
                    List<ForkJoinTask<Long>> tasks = new ArrayList<ForkJoinTask<Long>>();
                    for (final File child : children) {
                        if (child.isFile()) {
                            size += child.length();
                        } else {
                            tasks.add(new FileSizeFinder(child));
                        }
                    }
                    for (final ForkJoinTask<Long> task : invokeAll(tasks)) {
                        size += task.join();
                    }
                }
            }
            return size;
        }
    }
    ...
```

Thread-Pool for execution of tasks

Task by deriving from RecursiveTask<T>

Overwrite compute of RecursiveTask<T>

Create subtasks

Submit subtask for File: recursive !!

blocking wait for retrieving results

Example Total File Size: Variant 5 (2/2)

aus: V. Subramaniam: Programming Concurrency on the JVM

```
    ...
    public static void main(final String[] args) {
        final long start = System.nanoTime();
        final long total = forkJoinPool.invoke(new FileSizeFinder(new File(args[0])));
        final long end = System.nanoTime();
        System.out.println("ForkJoinTotalFileSize");
        System.out.println("Total Size: " + total);
        System.out.println("Time taken: " + (end - start) / 1.0e9);
    }
}
```

Execution of root task by thread-pool

```
> java ForkJoinTotalFileSize C:\Users\hp
ForkJoinTotalFileSize
Total Size: 83626897049
Time taken: 13.98460454
```

```
> java ForkJoinTotalFileSize C:\Users\hp
ForkJoinTotalFileSize
Total Size: 83663143456
Time taken: 9.15555707
```

Program similar to variant 1: recursive task structure analogous to recursive directory structure

Tasks are queue at threads in thread-pool

„Work-Stealing“: as soon as thread is free, it will get tasks from queue of other threads

Multithreading

Introduction and Basics

- Multithreading Basics
- Problems with Multithreading

Classical Model

- Synchronization
- Threads und Swing

New Model

- Executors and Futures
- Synchronization
- Concurrent Collections
- Case Example

Summary

Summary

Thread-safe = Program which are safe to be executed by multiple threads concurrently and in parallel

Problem is shared mutable state

Provisions

- no (or strongly limited) shared mutable state
- execution of parts by a single thread (single thread rule)
- synchronized access operations

Since Java version 1.5 new threading concepts

- Executors and thread pools
- Futures und Callables
- new synchronization mechanisms (Locks, Latch, ...)
- Concurrent Collections

Literature

- Doug Lea: ***Concurrent Programming in Java****: Design Principles and Pattern* (2nd Edition) . A comprehensive work by a leading expert, who's also the architect of the Java platform's concurrency framework.
- Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea: ***Java Concurrency in Practice***. A practical guide designed to be accessible to the novice.
- Jeff Magee and Jeff Kramer: ***Concurrency: State Models & Java Programs*** (2nd Edition). An introduction to concurrent programming through a combination of modeling and practical examples.
- Venkat Subramaniam: ***Programming Concurrency on the JVM***. The Pragmatic Bookshelf, 2011. Introduction to concurrency, software transactional memory and actors in Java.