SOFTWARE TRANSACTION MEMORY

WHAT IS THAT ABOUT

ONDŘEJ CHALOUPKA

RUNNING IN PARARELL

```
int x = 0, y = 0, z = 0;
```

```
void first {
   x = x + 1;
}
```

```
void second {
  y = y + 1;
  x = x + 1;
}
```

```
void third {
  z = z + 1;
  y = y + 1;
  x = x + 1;
}
```

RUNNING IN PARARELL

```
int x = 0, y = 0, z = 0;
```

```
void first {
    x = x + 1;
}
```

```
void second {
   y = y + 1;
   x = x + 1;
}
```

```
void third {
   z = z + 1;
   y = y + 1;
   x = x + 1;
}
```

```
x == 3, y == 2, z == 1 ???
```

```
int x = 0, y = 0, z = 0;
```

```
void first {
  synchronized(this) {
    x = x + 1;
  }
}
```

```
void second {
  synchronized(this) {
    y = y + 1;
    x = x + 1;
  }
}
```

```
void third {
  synchronized(this) {
    z = z + 1;
    y = y + 1;
    x = x + 1;
  }
}
```

```
x == 3, y == 2, z == 1 !
```

```
txn_int x = 0, y = 0, z = 0;
```

```
void first {
   atomic {
      x = x + 1;
   }
}
```

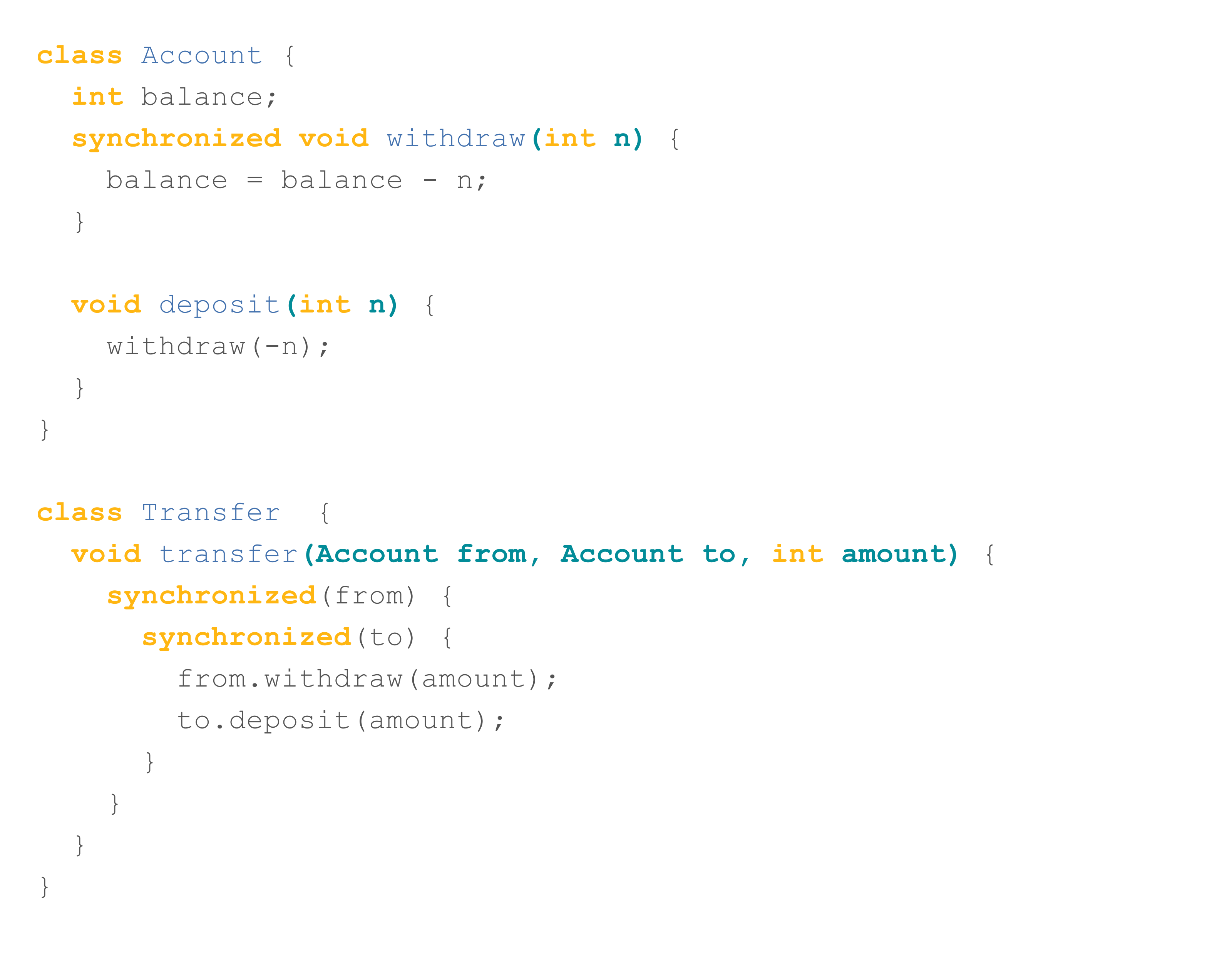
```
void second {
   atomic {
      y = y + 1;
      x = x + 1;
   }
}
```

```
void third {
   atomic {
      z = z + 1;
      y = y + 1;
      x = x + 1;
   }
}
```

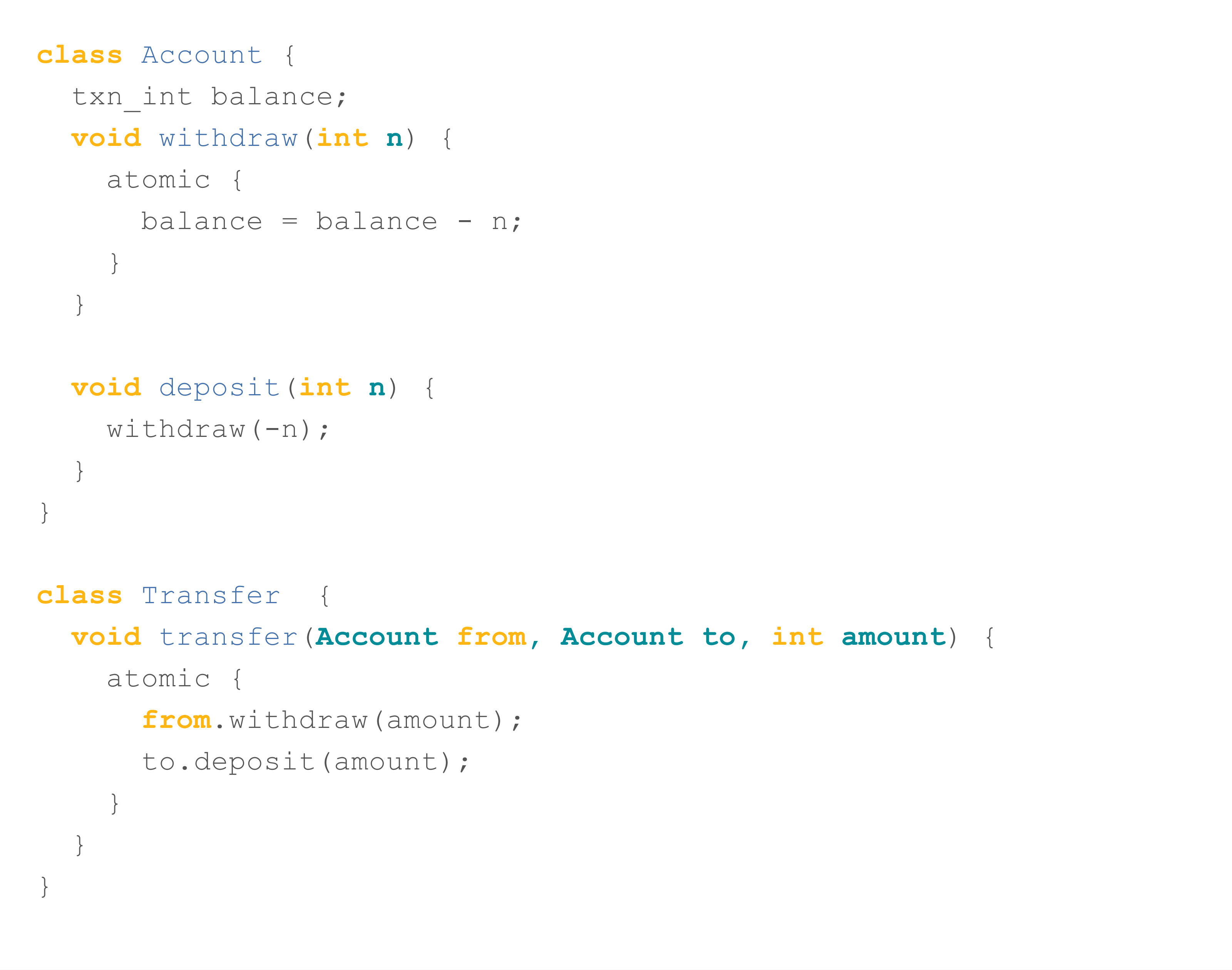
```
x == 3, y == 2, z == 1 !!!
```

LOCKS ARE NOT COMPOSABLE

```
class Account {
  int balance;
  synchronized void withdraw(int n) {
    balance = balance - n;
  }

  void deposit(int n) {
    withdraw(-n);
  }
}

class Transfer  {
  void transfer(Account from, Account to, int amount) {
    from.withdraw(amount);
    to.deposit(amount);
  }
}
```



```java
class Account {
  int balance;
  synchronized void withdraw(int n) {
    balance = balance - n;
  }

  void deposit(int n) {
    withdraw(-n);
  }
}

class Transfer  {
  void transfer(Account from, Account to, int amount) {
    synchronized(from) {
      synchronized(to) {
        from.withdraw(amount);
        to.deposit(amount);
      }
    }
  }
}
```



```
class Account {
  txn_int balance;
  void withdraw(int n) {
    atomic {
      balance = balance - n;
    }
  }

  void deposit(int n) {
    withdraw(-n);
  }
}

class Transfer  {
  void transfer(Account from, Account to, int amount) {
    atomic {
      from.withdraw(amount);
      to.deposit(amount);
    }
  }
}
```

SOFTWARE TRANSACTION MEMORY

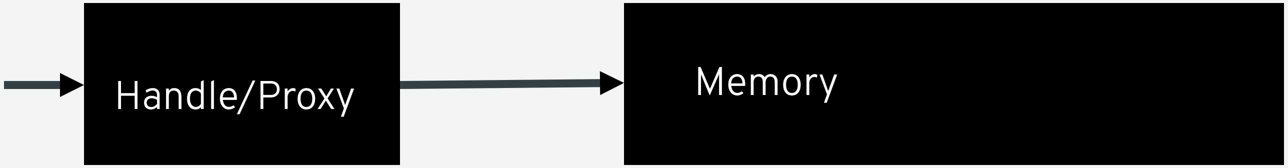
- A concurrency models which uses shared memory
- An alternative to the lock-based synchronization approach
- Grouping memory operations for them running atomically
- Simple interface for developers
- Can be implemented in various but not easy ways

WHY "TRANSACTIONAL" (STM)

- ACI(D) properties for the program
  - Atomically - either all operations are done or non of them
  - Consistency
  - Isolation - no influencing each other
    - serializable - operations appears like processing one after another
      (even system hardly thrive to process in parallel)
  - D - not usual, Narayana uses transaction log store to provide it

HOW IT WORKS

```
void second {
  atomic {
    y = y + 1;
    x = x + 1;
  }
}
```

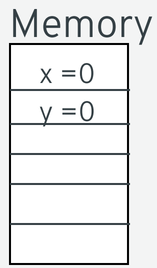


Handle/Proxy

Memory

HOW IT WORKS

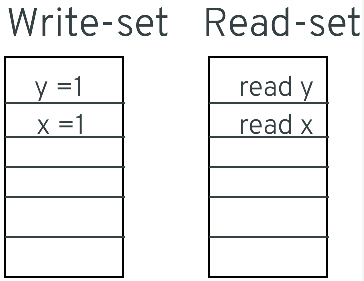
```
void second {
  atomic {
    y = y + 1;
    x = x + 1;
  }
}
```



Memory

x =0

y =0



Write-set

y =1

x =1

Read-set

read y

read x

```
void third {
  atomic {
    z = z + 1;
    y = y + 1;
    x = x + 1;
  }
}
```

NARAYANA STM

```
public class Container<T> {
  public enum TYPE { RECOVERABLE, PERSISTENT };
  public enum MODEL { SHARED, EXCLUSIVE };
  public Container ();
  public synchronized T create (T member);
  public static final Container<?>
          getContainer (Object proxy);
}
```

NARAYANA STM

```
@Transactional
public interface StockLevel {
  int get () throws Exception;
  void set (int value) throws Exception;
  void decr (int value) throws Exception;
}
```

NARAYANA STM

```java
Container<StockLevel> container = new Container<>();
StockLevelImpl stock = new StockLevelImpl();

StockLevel stockWrapped = container.create(stock);

// update the STM object inside a transaction
// or use annotations to define transaction boundaries
AtomicAction a = new AtomicAction();

a.begin();
stockWrapped.set(1234);
a.commit();
```

NARAYANA STM

```
// Implementations of interface
// are container managed
@Transactional

// Container will create
// a new transaction for each method
@Nested & @NestedTopLevel

@Optimistic & @Pessimistic

@ReadLock & @WriteLock

@State & @NotState

@TransactionFree
```

RESOURCES

- Transactional actors with Eclipse Vert.x
- http://jbossts.blogspot.com/2011/06/stm-arjuna.html
- Narayana quickstarts and documentation
- A (brief) retrospective on transactional memory
- Software Transactional Memory in Haskel
- Beautiful concurrency
- Software Transaction Memory and Clojure
- Maurice Herlihy — Transactional Memory and Beyond (part1, part2)