# Concurrency (II)

Lecture 13 (May 25th, 2021)



Radboud University

# Defining + Running a task

```
class Task implements Runnable {
   public void run() {
        doSomething();
   }
}
```

We can execute a Task by

```
new Thread( new Task () ).start();
```

Or by using an `Executer` (explained later)

```
interface Executor {
 public void execute( Runnable task );
}
```

# Sharing resources

# Thread communication

- Within the same application, threads share the same address space (memory).
- Communication can be done via shared objects
- But even simple data structures become prone to race conditions:
  - two threads may be attempting to update the same data structure at the same time and find it unexpectedly changing.
  - Bugs caused by race conditions can be very difficult to reproduce and isolate.

# Race condition example (I)

```
public class Counter{
    private int counter = 0;

    public void incr(){
        counter++;
    }

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

# Race condition example (2)

```
public class Incrementor implements Runnable {

  private Counter myCounter;
  private int myLimit;

  public Incrementor( Counter counter, int limit ) {
    this.myCounter = counter;
    this.myLimit = limit;
  }

  public void run() {
    for ( int i = 0; i < myLimit; i++ ) {
      myCounter.incr();
    }
    System.out.println( "Counter: " + myCounter.getCounter() );
  }

}
```

# Race condition example (4)

```
public class CounterThreads {

  public static void main( String[] args ) {
    Counter counter = new Counter();

    Thread t1 = new Thread( new Incrementor(counter, 60) );
    Thread t2 = new Thread( new Incrementor(counter, 60) );
    t1.start();
    t2.start();
  }
}
```

- Output?

A. Counter: 60
   Counter: 60

B. Counter: 60
   Counter: 120

C. Counter: X          ($60 \leq X \leq 120$)
   Counter: 120

D. Can be (almost) anything

Answer:

D. Counter: X ($1 \leq X \leq 120$)
   Counter: Y ($1 \leq Y \leq 120$)

# Race condition example (4)

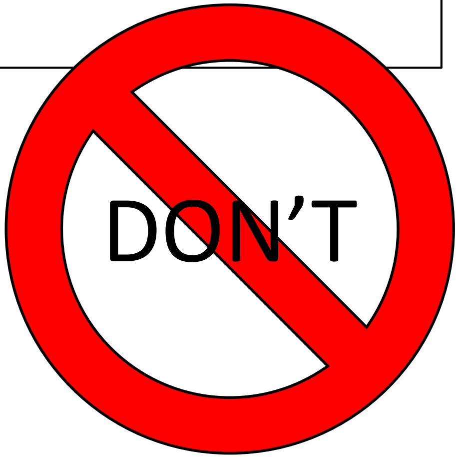
```
public class CounterThreads {

public static void main( String[] args ) {
    Counter counter = new Counter();

    Thread t1 = new Thread( new Incrementor(counter, 60) );
    Thread t2 = new Thread( new Incrementor(counter, 60) );
    t1.start();
    t2.start();
  }
}
```

```
run:
Counter: 54
Counter: 94
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Race condition example (3)

```java
public class Incrementor implements Runnable {
 private Counter myCounter;
 private int myLimit;
 public Incrementor( Counter counter, int limit ) {
  this.myCounter = counter;
  this.myLimit= limit;
 }
 public void run() {
  for ( int i = 0; i < myLimit; i++ ) {
     myCounter.incr();
  }
  System.out.println( "Counter: " + myCounter.getCounter() );
 }
}
```

```java
public class Counter{
  private int value = 0;

  public void incr(){
    value++;
  }

  public int getValue() {
    return value;
  }
}
```




```java
 public class CounterThreads {
  public static void main( String[] args ) {
   Counter counter = new Counter();
   Thread t1 = new Thread( new Incrementer( counter, 60 ) );
   Thread t2 = new Thread( new Incrementer( counter, 60 ) );
   t1.start();
   t2.start();
  }
 }
```

# Race condition solution

```
public class Counter {
 private int value;
 public synchronized void incr() {
   value++;
 }
 public synchronized int getValue() {
   return value;
 }
}
```

or equivalently:

```
public class Counter {
  private int value;
  public void incr() {
    synchronized ( this ) {
      value++;
    }
  }
  public int getValue() {
    synchronized ( this ) {
       return value;
    }
  }
}
```

# Race condition solution: run main

```
public class CounterThreads {
  public static void main( String[] args ) {
    Counter counter = new Counter();
    Thread t1 = new Thread( new Incrementer( counter, 60 ) );
    Thread t2 = new Thread( new Incrementer( counter, 60 ) );
    t1.start();
    t2.start();
  }
}
```

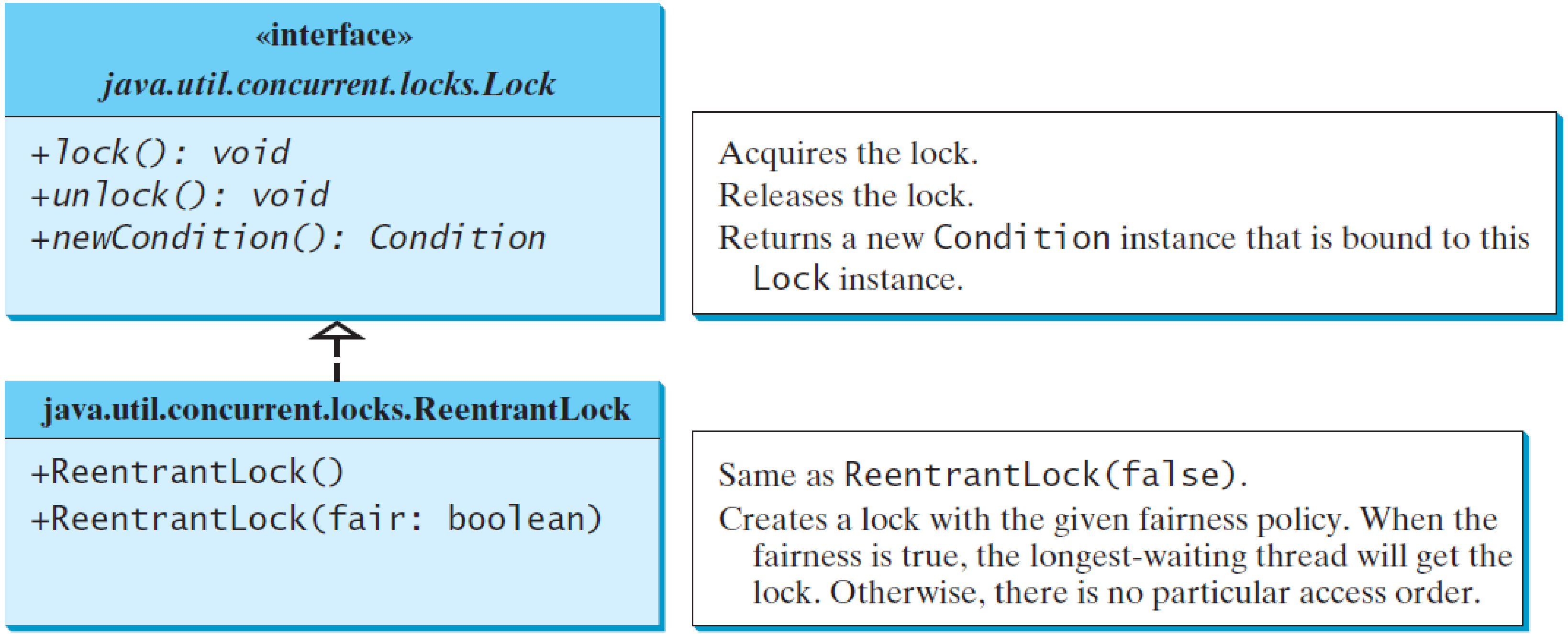
- Output?
  A. `Counter: 60`
     `Counter: 60`
  B. `Counter: 60`
     `Counter: 120`
  C. `Counter: X      (60 ≤ X ≤ 120)`
     `Counter: 120`
  D. Something else

Answer:

D. `Counter: X (60 ≤ X ≤ 120)`
   `Counter: 120`
   `or`
   `Counter: 120`
   `Counter: X (60 ≤ X ≤ 120)`

# Synchronization Using `Locks`

- A synchronized instance method implicitly acquires a lock on the instance.
- JDK 1.5 enables you to use locks *explicitly*.
- These locks are flexible and give you more control for coordinating threads.
- A lock is an instance of (a class implementing) the `Lock` interface.
- A lock may also use the `newCondition()` *factory method* to create any number of `Condition` objects, which can be used for more elaborate thread communication.



# Race condition solution using Locks

```
public class Counter {
 private int myValue = 0;
 private Lock myLock = new ReentrantLock();

 public Counter( int initial_value ) {
  this.myValue = initial_value;
 }

 public int getValue() {
  myLock.lock();
  int value = myValue;
  myLock.unlock();
  return value;
 }

 public void incr (  ) {
  myLock.lock();
  myValue = myValue + 1;
  myLock.unlock();
 }

}
```

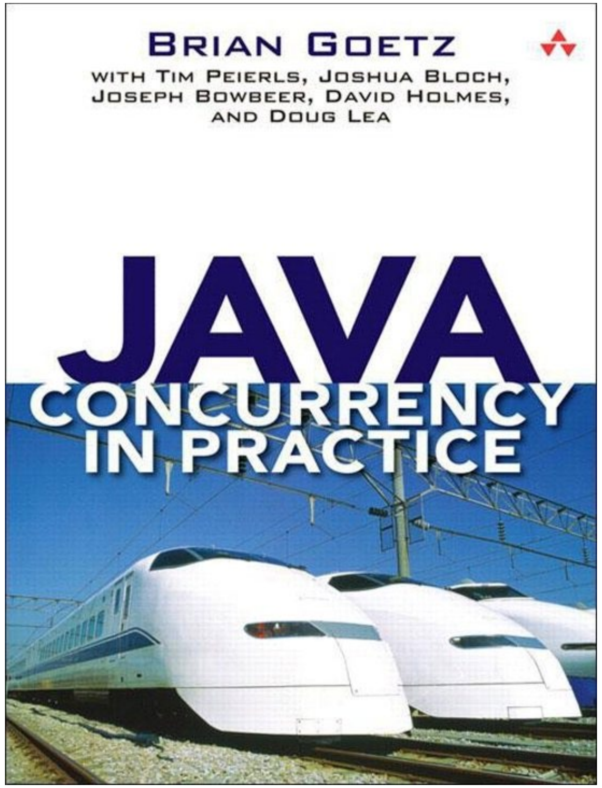

This is not the recommended way to use locks; see slide 24 and further.

# Should getters/accessors be synchronized?

… It is a common mistake to assume that synchronization needs to be used only when writing to shared variables; this is simply not true.

For each mutable state variable that may be accessed by more than one thread, **all accesses** to that variable must be performed with the same lock held. In this case, we say that the variable is guarded by that lock. …

*Java Concurrency in Practice*




- Answer: Yes!
- ItJP (Liang): ???

# Synchronization: two locks

```
public class CounterThreads {

public static void main( String[] args ) {
  Counter counter1 = new Counter();
  Counter counter2 = new Counter();

  Thread t1 = new Thread( new Incrementor( counter1, 50 ) );
  Thread t2 = new Thread( new Incrementor( counter1, 50 ) );
  Thread t3 = new Thread( new Incrementor( counter2, 100 ) );
  Thread t4 = new Thread( new Incrementor( counter2, 100 ) );

  t1.start();
  t2.start();
  t3.start();
  t4.start();
 }
}
```

# `Executor` Framework (I)

- Recommended that you use the `Executor` *interface* to manage the execution of `Runnable` objects
- An `Executor` object creates and manages a *thread pool* to execute `Runnable`s
- `Executor` advantages over creating threads yourself
  - Reuse existing threads to eliminate new thread overhead
  - Improve performance by optimizing the number of threads to ensure that the processor stays busy
- `Executor` method `execute` accepts a `Runnable` as an argument.
  - If `e` is an `Executor` object you can replace

```
(new Thread(r)).start();
```

with

```
e.execute(r);
```

# `Executor` Framework (II)

- Executor interface is very basal; An `ExecutorService` is more useful
  - extends `Executor`
  - declares methods for managing the life cycle of an `Executor`
  - Objects of this type are created using factory methods declared in class `Executors`.
- `Executors` method `newCachedThreadPool` obtains an `ExecutorService` that creates new threads as they are needed
- `ExecutorService` method `execute` returns immediately from each invocation
- `ExecutorService` method `shutdown` notifies the `ExecutorService` to stop accepting new tasks, but continues executing tasks that have already been submitted

# Executor demo

```
public static void main( String[] args ) {
   Counter counter = new Counter( );

   Incrementer incTask1 = new Incrementer ( counter, 60 );
   Incrementer incTask2 = new Incrementer ( counter, 60 );
   // create ExecutorService to manage threads
   ExecutorService executor = Executors.newCachedThreadPool();

   executor.execute( incTask1 ); // start task1
   executor.execute( incTask2 ); // start task2

   executor.shutdown();
```

```
   try {
      // wait 1 minute for both incrementors to finish executing
      boolean tasksEnded = executor.awaitTermination( 1, TimeUnit.MINUTES );
      if ( tasksEnded ) {
         System.out.println( "Counter: " + counter.getCounter() );
      } else {
         System.out.println( "Timed out while waiting for tasks to finish." );
      }
   } catch ( InterruptedException ex ) {
     System.out.println( "Interrupted while waiting for tasks to finish." );
   }
}
```

# Check-then-act problem

- Situation where threads check the state of a shared object (read) and, based on the result, they try to modify that state (write).
- By the time a thread performs the write, the state of the object may have been changed by some other thread.
- One way to solve this problem is to surround reading and writing by a synchronized statement
- However, this solution is not very elegant, and can be inefficient.

# Example

- Suppose we have a bank account from which a *spender* regularly wants to withdraw a certain amount of money.
- Withdrawing is only permitted if the available balance is sufficient (being in the red is not allowed)
- There is also an *earner* who now and then deposits some money into the account

# The class Account

```
public class Account {
    private int balance;


    public Account( int initial_balance ) {
        balance  = initial_balance;
    }

    public int getBalance() {
        return balance;
    }

    public void deposit( int amount ) {
        int old_balance = balance;
        balance = old_balance + amount;
    }

    public void withdraw( int amount ) {
        int old_balance = balance;
        balance = old_balance - amount;
    }

}
```

# The classes Spender and Earner

```java
public class Spender implements Runnable {
  private Account account;
  private int limit;

  public Spender( Account account, int limit ) {
    this.account = account;
    this.limit = limit;
  }

  @Override
  public void run() {
    int total_withdrawn = 0;
    while ( total_withdrawn < limit ) {
      int amount_to_withdraw =
             Util.randomInRange( 50, 150 );
      account.withdraw( amount_to_withdraw );
      total_withdrawn += amount_to_withdraw;
    }
    System.out.println( "Withdrawn in total: " +
                          total_withdrawn);
  }
}
```

```java
public class Earner implements Runnable {
  private Account account;
  private int limit;

  public Earner( Account account, int limit ) {
    this.account = account;
    this.limit = limit;
  }

  @Override
  public void run() {
    int total_deposited = 0;
    while ( total_deposited < limit ) {
      int amount_to_deposit =
             Util.randomInRange ( 100, 300 );
      total_deposited += amount_to_deposit;
      account.deposit( amount_to_deposit );
    }
    System.out.println( "Deposited in total: " +
                          total_deposited );
    }
}
```

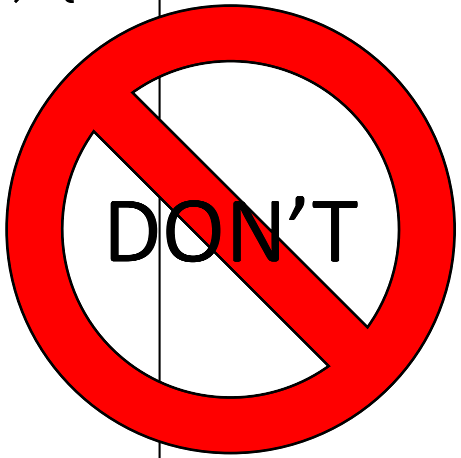
# The class AccountTest

```
public static void main( String[] args ) {
   Account account   = new Account( 1000 );

   Earner   earner   = new Earner( account, 5000 );
   Spender  spender  = new Spender( account, 5000 );

   ExecutorService executor = Executors.newCachedThreadPool();

   executor.execute( earner );
   executor.execute( spender );

   executor.shutdown();

   try {
      boolean tasksEnded = executor.awaitTermination(1, TimeUnit.MINUTES);
      if (tasksEnded) {
          System.out.println( "Final balance: " + account.getBalance() );
      }
   } catch ( InterruptedException ex ) {
   }

   System.out.println( "Final balance: " + account.getBalance() );

 }
```

```
run:
Deposited in total: 5092
Withdrawn in total: 5062
Final balance: -4062
```

# The class Account

```
public class Account {
    private int balance;


    public Account( int initial_balance ) {
        balance  = initial_balance;
    }

    public int getBalance() {
        return balance;
    }

    public void deposit( int amount ) {
        int old_balance = balance;
        balance = old_balance + amount;
    }

    public void withdraw( int amount ) {
        int old_balance = balance;
        balance = old_balance - amount;
    }

}
```



# The class Account synchronized

```
public class Account {
    private int balance;
    private Lock lock = new ReentrantLock();

    public Account( int initial_balance ) {
        this.balance = initial_balance;
    }

    public int getBalance () {
        lock.lock();
        int currentBalance = balance;
        lock.unlock();
        return currentBalance ;
    }

    public void deposit ( int amount ) {
        lock.lock();
        balance = balance + amount;
        lock.unlock();
    }

    public void withdraw ( int amount ) {
        lock.lock();
        balance = balance - amount;
        lock.unlock();
    }
}
```

# The class Account synchronized (recommended)

```
public class Account {
    private int balance;
    private Lock lock = new ReentrantLock();

    public Account( int initial_balance ) {
        this.balance = initial_balance;
    }

    public int getBalance() {
        lock.lock();
        try {
            return balance;
        } finally {
            lock.unlock();
        }
    }

    public void deposit( int amount ) {
        lock.lock();
        try {
            balance = balance + amount;
        } finally {
            lock.unlock();
        }
    }
```

```
  public void withdraw( int amount ) throws
                          InsufficientFundException {
    lock.lock();
    try {
      if ( balance < amount ) {
        throw new InsufficientFundException();
      }
      balance = balance - amount;
    } finally {
      lock.unlock();
    }
  }

  public static class InsufficientFundException
                                   extends Exception {
    public InsufficientFundException() {
      super("Insufficient balance");
    }
  }
}
```

ensures that the lock will always be released

# Preventing negative balance

```
public class Spender implements Runnable {
  private Account account;
  private int limit;

  public Spender( Account account, int limit ) {
    this.account = account;
    this.limit = limit;
  }

  @Override
  public void run() {
    int total_withdrawn = 0;
    while ( total_withdrawn < limit ) {
      int amount_to_withdraw = Util.randomInRange( 50, 150 );
      if ( account.getBalance() >= amount_to_withdraw ) {
        try {
          account.withdraw(amount_to_withdraw);
          total_withdrawn += amount_to_withdraw;
        } catch (Account.InsufficientFundException ex) {
          System.out.println(ex);
        }
      }
    }
    System.out.println( "Withdrawn in total: " +  total_withdrawn);
  }
}
```

what if the spender really needs the amount he tries to withdraw?

# Preventing negative balance (2)

```
public class Spender implements Runnable {
  private Account account;
  private int limit;

  public Spender( Account account, int limit ) {
    this.account = account;
    this.limit = limit;
  }

  @Override
  public void run() {
    int total_withdrawn = 0;
    while ( total_withdrawn < limit ) {
      int amount_to_withdraw = Util.randomInRange( 50, 150 );
      while ( account.getBalance() < amount_to_withdraw ) {
      }
      try {
        account.withdraw(amount_to_withdraw);
        total_withdrawn += amount_to_withdraw;
      } catch (Account.InsufficientFundException ex) {
        System.out.println(ex);
      }
    }
    System.out.println( "Withdrawn in total: " +  total_withdrawn);
  }
}
```

this called: busy waiting

- usually a bad idea

# Multiple spenders (1)

```
public static void main( String[] args ) {
  Account account    = new Account( 0 );
  Earner   earner    = new Earner( account, 5300 );
  Spender  spender1  = new Spender( account, 2500 );
  Spender  spender2  = new Spender( account, 2500 );

  ExecutorService executor = Executors.newCachedThreadPool();

  executor.execute( earner );
  executor.execute( spender1 );
  executor.execute( spender2 );

  executor.shutdown();

  try {
      boolean tasksEnded = executor.awaitTermination(1, TimeUnit.MINUTES);
      if (tasksEnded) {
          System.out.println( "Final balance: " + account.getBalance() );
      }
  } catch ( InterruptedException ex ) {
  }

}
```

# Multiple spenders (2)

- After a few runs, we get

```
run:
banking.notnegNOK.Account$InsufficientFundException: Insufficient balance
Withdrawn in total: 2574
Deposited in total: 5439
Withdrawn in total: 2616
Final balance: 382
BUILD SUCCESSFUL (total time: 0 seconds)
```

```java
  @Override
  public void run() {
    int total_withdrawn = 0;
    while ( total_withdrawn < limit ) {
      int amount_to_withdraw = Util.randomInRange( 50, 150 );
      while (account.getBalance() < amount_to_withdraw) {
      }
      try {
        account.withdraw(amount_to_withdraw);
        total_withdrawn += amount_to_withdraw;
      } catch (Account.InsufficientFundException ex) {
        System.out.println(ex);
      }
    }
    System.out.println( "Withdrawn in total: " +  total_withdrawn);
  }
}
```

check-then-act issue

between `getBalance` and `withdraw` a contextswitch may occur

# Needed Solution

- Need a way of having threads “wait” on a resource (e.g. sufficient balance)
- Also need a way to “notify” waiting threads when they can wake up (after the balance has been increased)
- Java provides a very robust Condition mechanism to fill these needs

# Conditions

- A lock has a factory method newCondition to create new condition instances.

| «interface» *java.util.concurrent.Condition* | |
|---|---|
| `+await(): void` | Causes the current thread to wait until the condition is signaled. |
| `+signal(): void` | Wakes up one waiting thread. |
| `+signalAll(): void` | Wakes up all waiting threads. |

- Condition methods can only be called if the lock from which these instances were created is held. Otherwise an Exception will be thrown.

- `await`
  - releases the lock (but not any other locks held by this thread)
  - adds this thread to waiting list for the condition
  - blocks the thread

# Conditions (2)

| «interface» *java.util.concurrent.Condition* | |
|---|---|
| +await(): void | Causes the current thread to wait until the condition is signaled. |
| +signal(): void | Wakes up one waiting thread. |
| +signalAll(): void | Wakes up all waiting threads. |

- `signalAll`
  - Releases all threads on condition's waiting list
  - Those threads must reacquire lock before continuing
    - this is done implicitly; you don't need to do it explicitly
- `signal`
  - wakes up only one waiting thread
  - Be careful: Tricky to use correctly

# The class Account (alternative solution)

```
public class Account {
  private int balance;
  private Lock lock = new ReentrantLock();
  private Condition newDeposit = lock.newCondition();
  public Account( int initial_balance ) {
     this.balance = initial_balance;
  }

  public int getBalance() {
     lock.lock();
     try {
        return balance;
     } finally {
        lock.unlock();
     }
  }

  public void deposit( int amount ) {
     lock.lock();
     try {
        balance = balance + amount;
        newDeposit.signalAll();
     } finally {
        lock.unlock();
     }
  }
```

```
  public void withdraw( int amount ) {
    lock.lock();
    try {
      while ( balance < amount ) {
        newDeposit.await();
      }
      balance = balance - amount;
    } catch ( InterruptedException ex ) {
      ex.printStackTrace();
    } finally {
      lock.unlock();
    }
  }
}
```

wait until the balance has (sufficiently) been increased

notify waiting threads that a new deposit was added

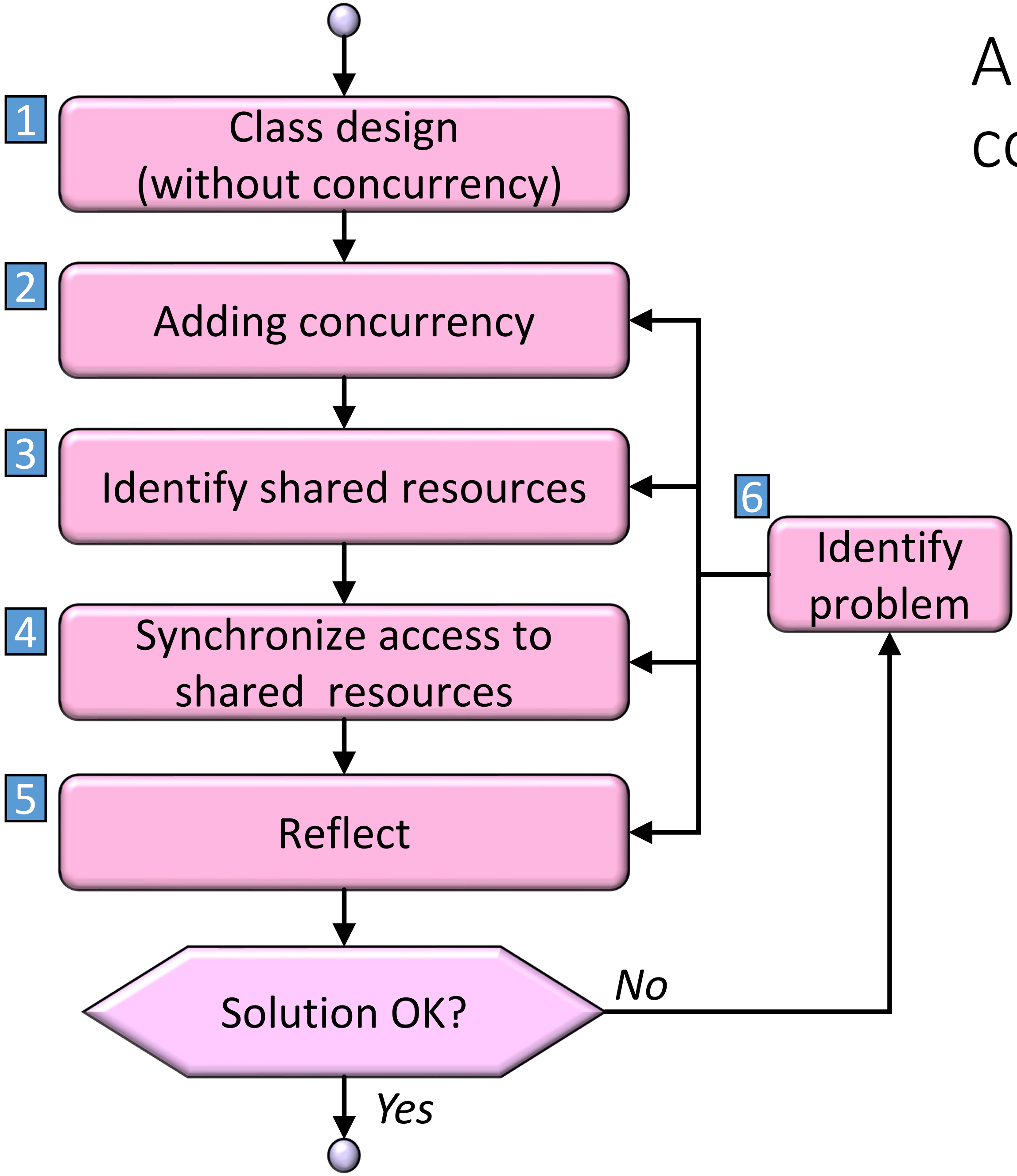
# class Spender

```
public class Spender implements Runnable {
  private Account account;
  private int limit;

  public Spender( Account account, int limit ) {
      this.account = account;
      this.limit = limit;
  }

  @Override
  public void run() {
    int total_withdrawn = 0;
    while ( total_withdrawn < limit ) {
      int amount_to_withdraw = Util.randomInRange( 50, 150 );
      account.withdraw( amount_to_withdraw );
      total_withdrawn += amount_to_withdraw;
    }
    System.out.println("Withdrawn in total: " + total_withdrawn);
  }
}
```

# Development guidelines (6 steps)

1. Analyse the problem without concurrency. Draw a general class diagram without detailed operations. Implement the classes of the diagram
2. Analyse which tasks must be performed concurrently. Design new active classes that use existing classes to execute a task: active class design pattern. Design method run. Choose the class that creates the threads. Modify the class diagram and implement the design
3. Draw a high-level activity diagram for the part of the program where method(s) run is/are executed. Specifically: draw shared objects with their attributes that are accessed and/or updated
4. Analyse the activity diagram and check for race conditions or check-then-act problems. Apply the appropriate solution for the problems that occur. Change the implementation
5. Ask yourself if the problem(s) has(have) been solved? What result(s) did you expect? Can you explain the observed outcome(s)?
6. If your solution is incorrect, repeat (some of) the steps above.

# Approach for designing a concurrent program

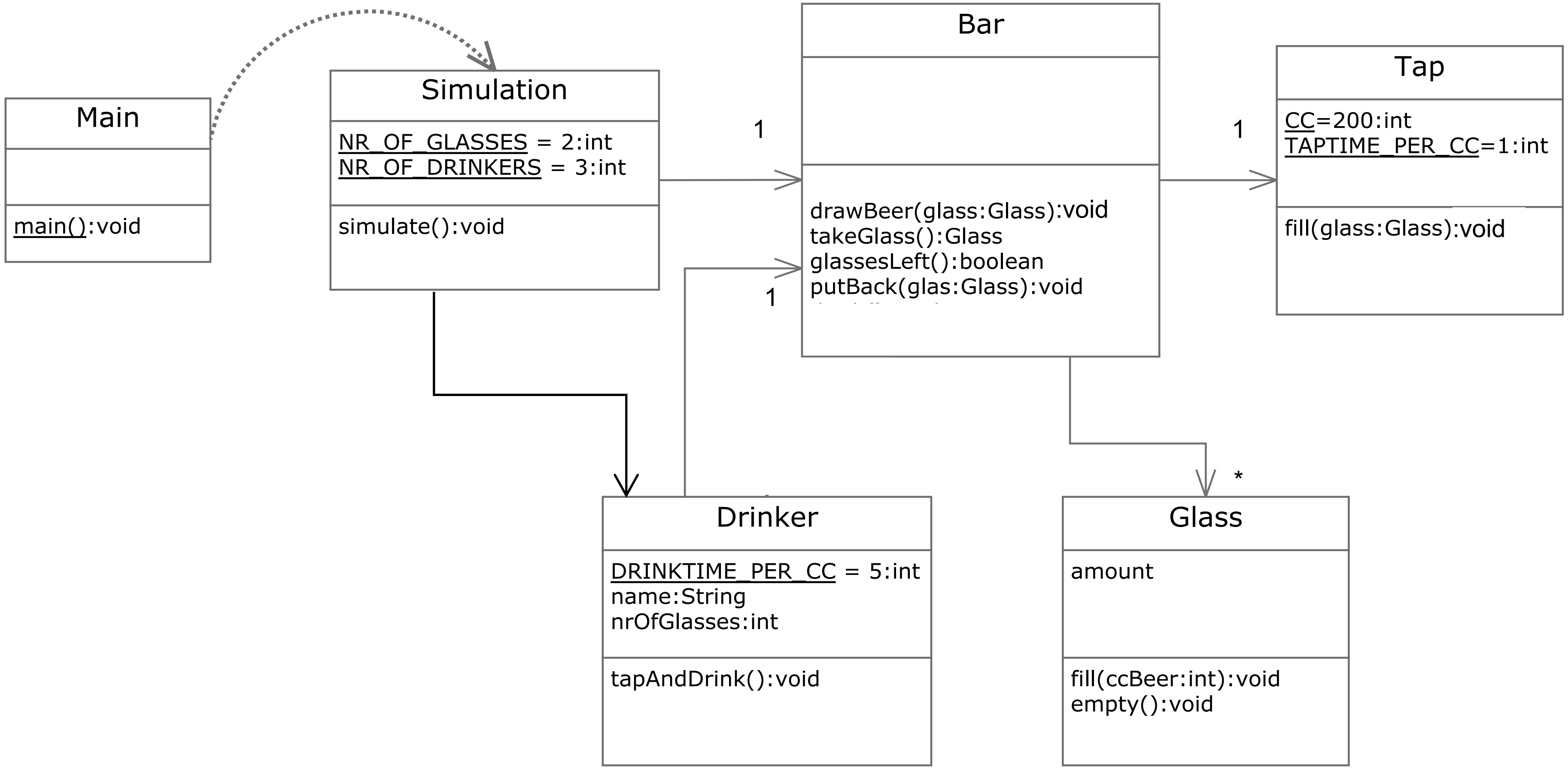


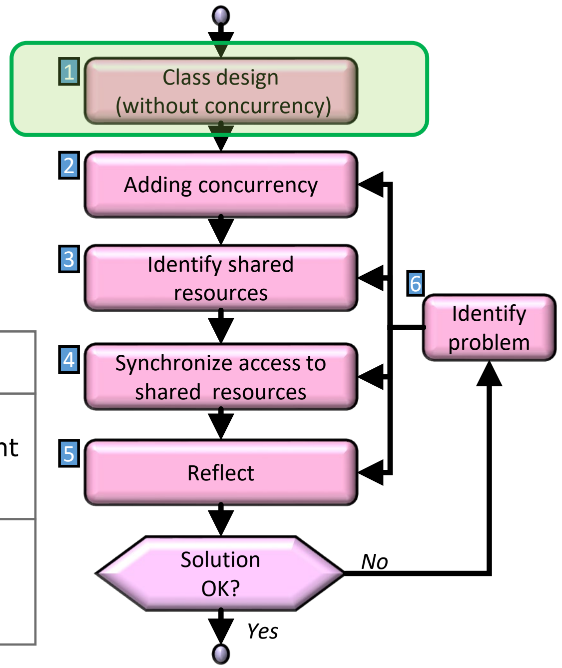

- Step1 and 2 involve creating a concurrent program;
- Step 3 and 4 are necessary to solve the possible problems that are introduced by using threads
- Step 5 is a final check

# Example II: having a drink

- Imagine a bar with only a limited number of glasses
- People are entering the bar to have a few drinks
  - The number of drinkers is larger than the number of glasses
  - Each of this drinkers takes a glass from the counter, fills the glass at the tap, empties the glass, and puts it back on the counter.
- We want to develop a concurrent simulation of bar.

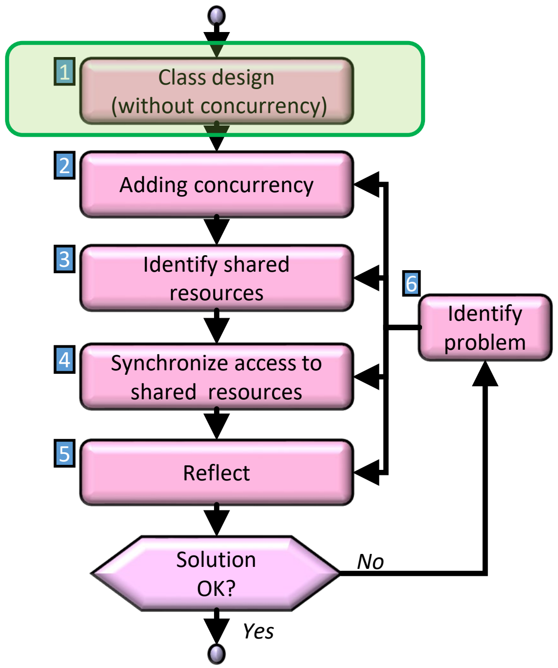
# Bar step 1a: class design



# Bar step 1b: class Drinker

```
public class Drinker {
  private int nrOfGlasses;
  private int drinkerId;
  private Bar bar;
  private static final int DRINKTIME_PER_CC = 5;

  public Drinker( int id, Bar bar ) {
     this.drinkerId = id;
     this.bar = bar;
     this.nrOfGlasses = Util.randomInRange (3, 8);
  }

  public void tapAndDrink() {
    if ( bar.glassesLeft() ) {
      Glass glass = bar.takeGlass();
      bar.drawBeer(glass);
      takeABreak( glass.getAmountBeer() * DRINKTIME_PER_CC );
      glass.empty();
      nrOfGlasses--;
      bar.putBack(glass);
    }
  }

  public boolean isSatisfied() {
     return nrOfGlasses == 0;
  }
}
```

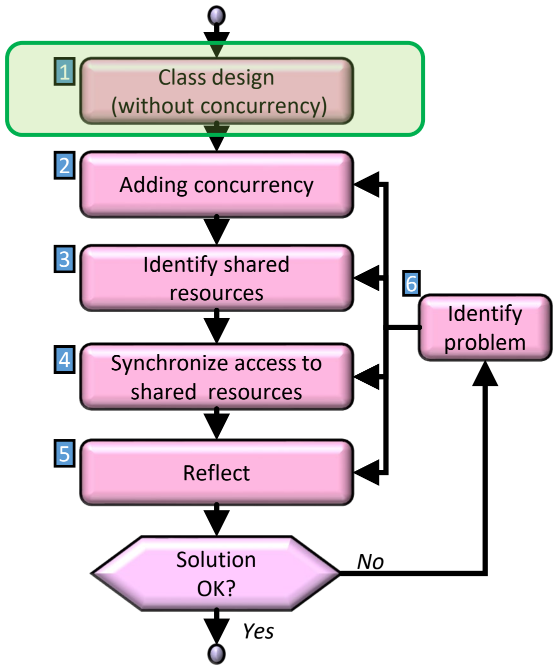





Radboud University

# Bar step 1b: class Glass

```
public class Glass {
  private int amount;

  public Glass() {
    this.amount = 0;
  }

  public void fill( int ccBeer ) {
    this.amount = ccBeer;
  }

  public void empty() {
     amount = 0;
  }

  public int getAmount() {
     return amount;
  }
}
```



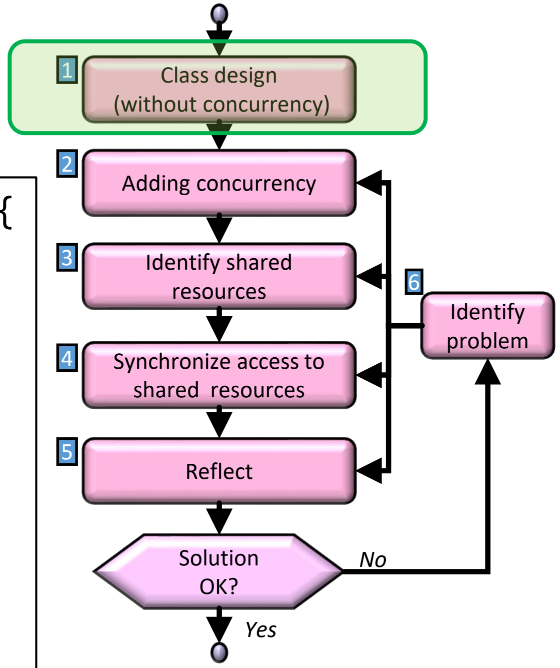



Radboud University

# Bar step 1b: class Bar

```
public class Bar {
  private Tap tap;
  private List<Glass> glasses;

  public Bar( int numberOfGlasses ) {
    tap = new Tap();
    glasses = new ArrayList<>();
    for ( int i = 0; i < numberOfGlasses; i++ ) {
      glasses.add(new Glass());
    }
  }
```

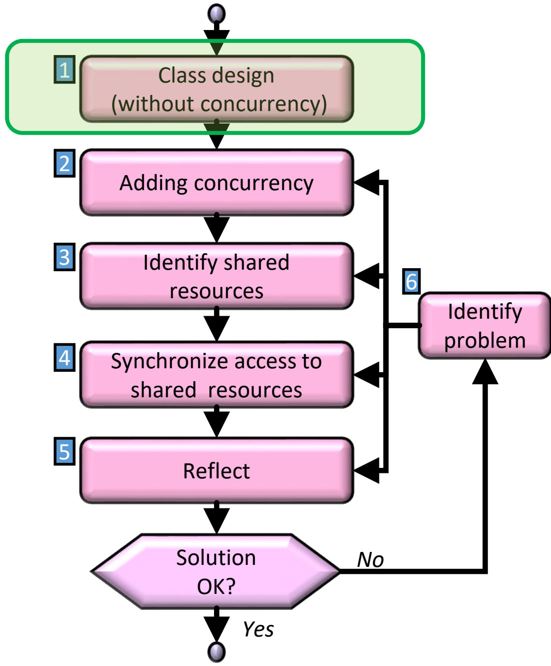
```
  public void drawBeer( Glass glas ) {
    tap.tapBeer( glas );
  }

  public boolean glassesLeft() {
    return glasses.size() > 0;
  }

  public Glass takeGlass() {
     return glasses.remove(0);
  }

  public void putBack( Glass g ) {
     glasses.add(g);
  }
}
```



# Bar step 1b: class Simulation

```java
public class Simulation {
  public static final int NR_OF_GLASSES = 2, NR_OF_DRINKERS = 3;

  public void simulate() {
    Bar bar = new Bar(NR_OF_GLASSES);
    Queue<Drinker> drinkers = new LinkedList<>();
    for (int i = 0; i < NR_OF_DRINKERS; i++) {
        drinkers.offer(new Drinker(i, bar));
    }
    while ( !drinkers.isEmpty() ) {
        Drinker drinker = drinkers.poll();
        drinker.tapAndDrink();
        if ( !drinker.isSatisfied() ) {
            drinkers.offer(drinker);
        }
    }

  }
```

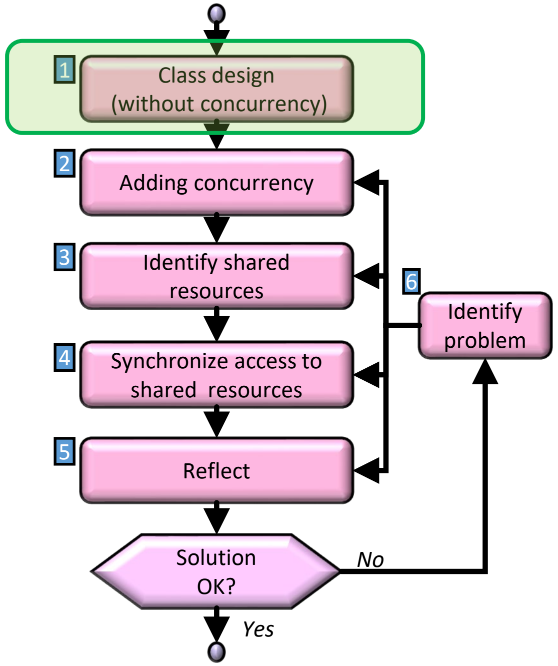
Sequential simulation!
code will change considerably



# Bar step 1b: classes Tap and Main

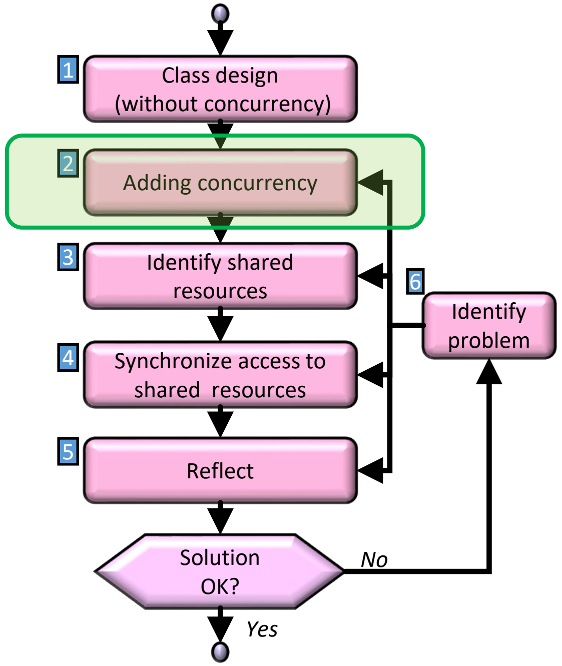
```
public class Tap {
  private static final int CC = 200;
  private static final int DRAW_TIME_PER_CC = 1;

  public void tapBeer( Glass glass ) {
    glass.fill( CC );
    takeABreak( glass.getVolume() * DRAW_TIME_PER_CC );
    return glass;
  }
}
```

```
public class Main {

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



# Active class step 2: DrinkRunner

```java
public class DrinkRunner implements Runnable {
  private Drinker drinker;

  public DrinkRunner( Drinker drinker ) {
    this.drinker = drinker ;
  }

  public void run() {
    while( ! drinker.isSatisfied() ) {
      drinker.tapAndDrink();
    }
  }
}
```

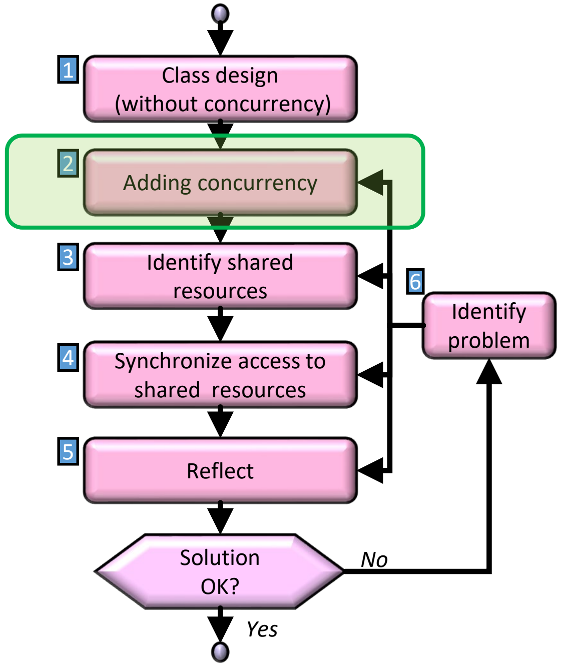




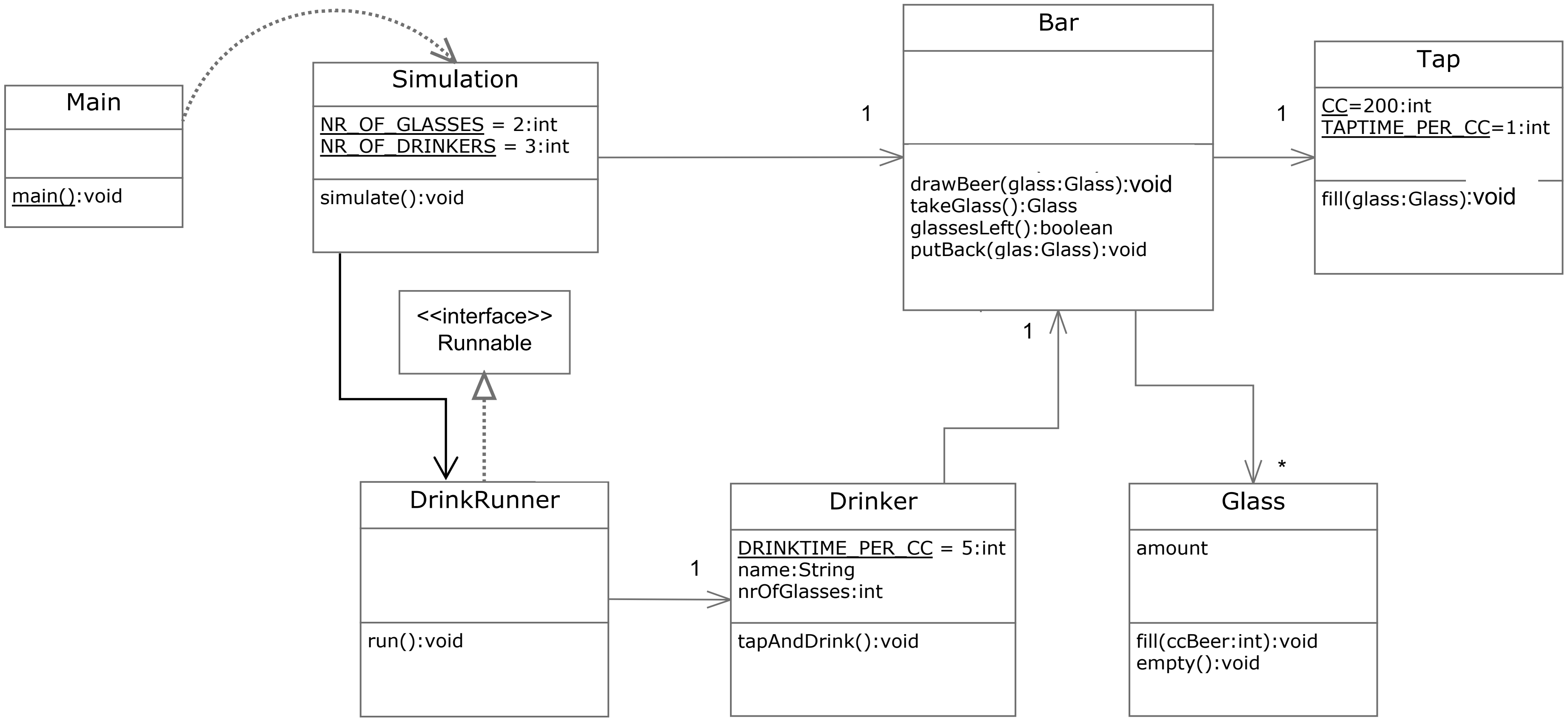

Radboud University

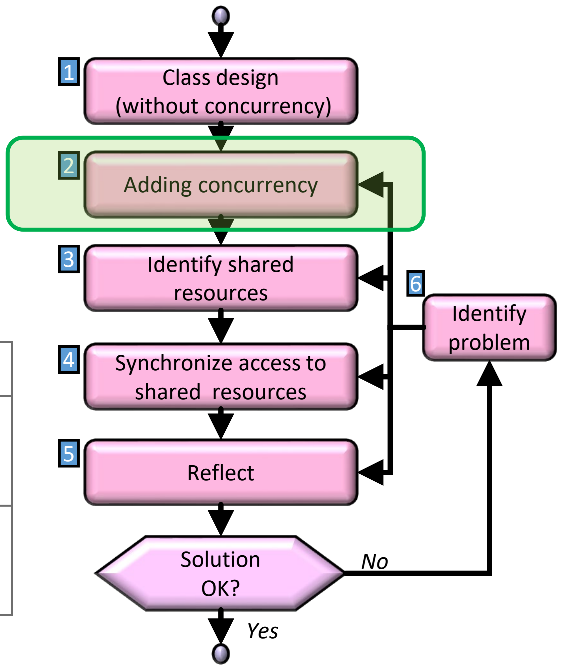
# Bar step 2b: class Simulation

```
public class Simulation {
  public static final int NR_OF_GLASSES = 2, NR_OF_DRINKERS = 3;

  public void simulate() {
    Bar bar = new Bar(NR_OF_GLASSES);
    ExecutorService executor = Executors.newCachedThreadPool();

    IntStream.rangeClosed( 1, NR_OF_DRINKERS ).
            mapToObj( id -> new DrinkerRunner( new Drinker( id, bar ) ) ).
            forEach( executor::execute );

    executor.shutdown();
  }

}
```

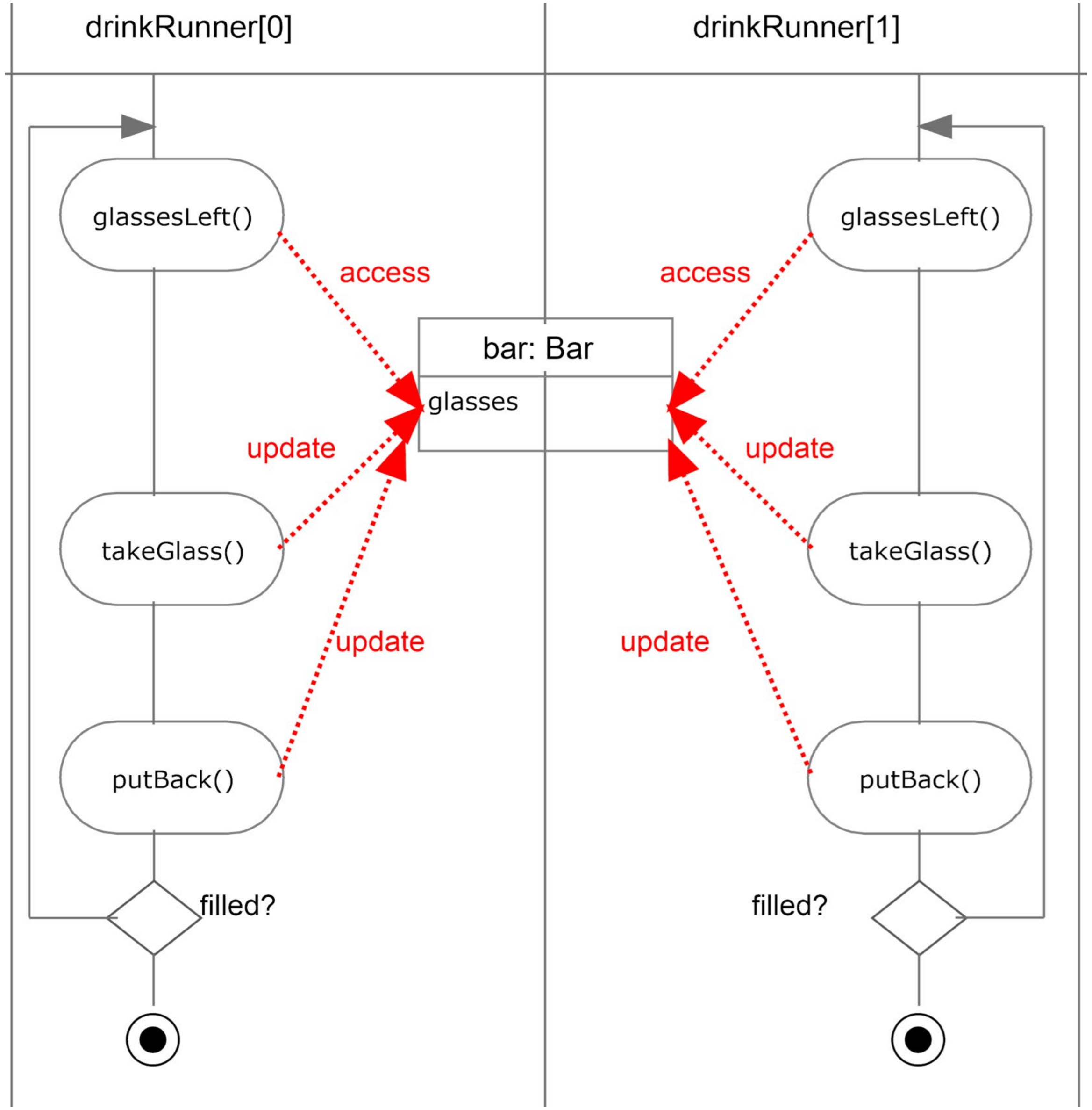


# Bar step 2b: class diagram

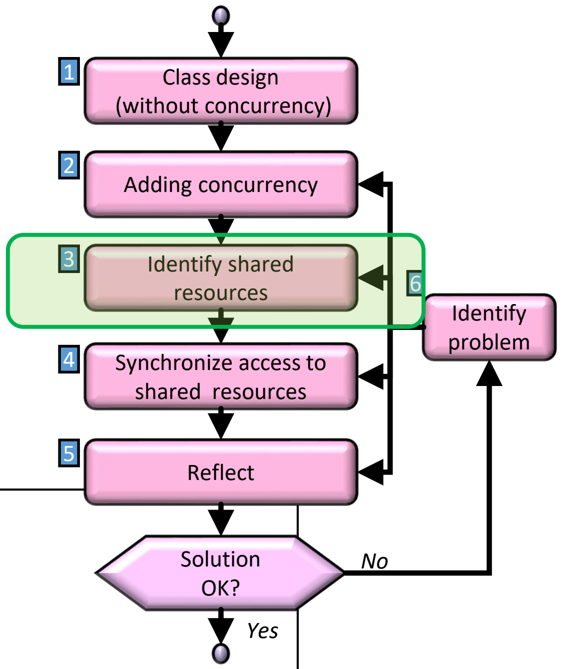


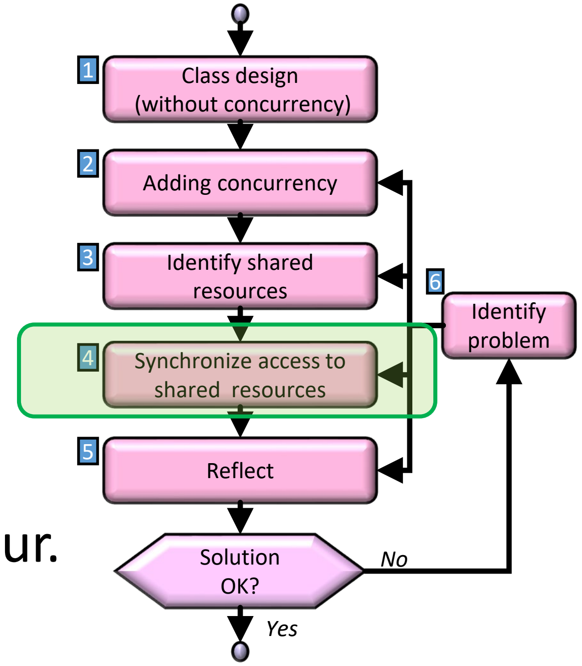
# Bar step 3: activity diagram




```
public class Drinker {
  …
  public void tapAndDrink() {
    if ( bar.glassesLeft() ) {
      Glass glass = bar.takeGlass();
      bar.drawBeer(glass);
      takeABreak( glass.getAmountBeer() *
                  DRINKTIME_PER_CC );
      glass.empty();
      nrOfGlasses--;
      bar.putBack(glass);
    }
  }
  …
}
```



# Bar step 4: analysis

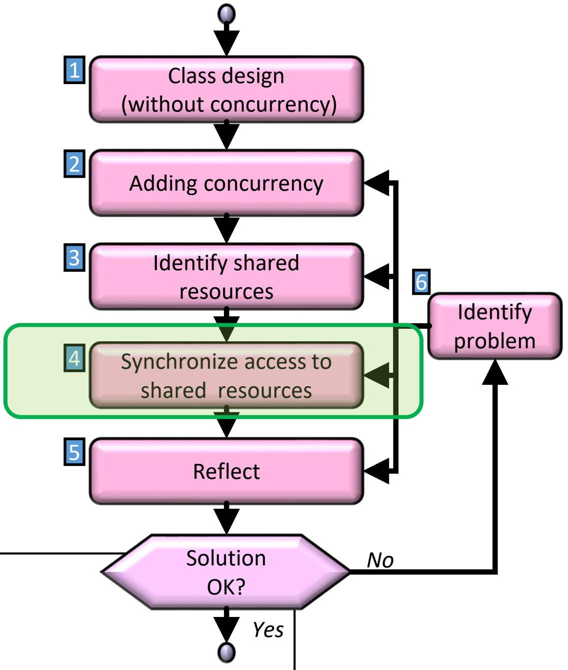



1. Both read and write access to the shared Bar object
   - synchronization necessary
2. There is another issue: the drinker continuously queries the bar to check if an empty glass is available
   - Btw: between `glassesLeft` and `takeGlass` a context switch might occur.
     - this is a check-then-act situation

1. Solving 1 is easy: make operations on the list of glasses synchronized.
2. For 2 we introduce a condition on which the drinker will wait if there is no glass until another drinker puts his glass back on de bar.
   - this will also solve the CTE-issue

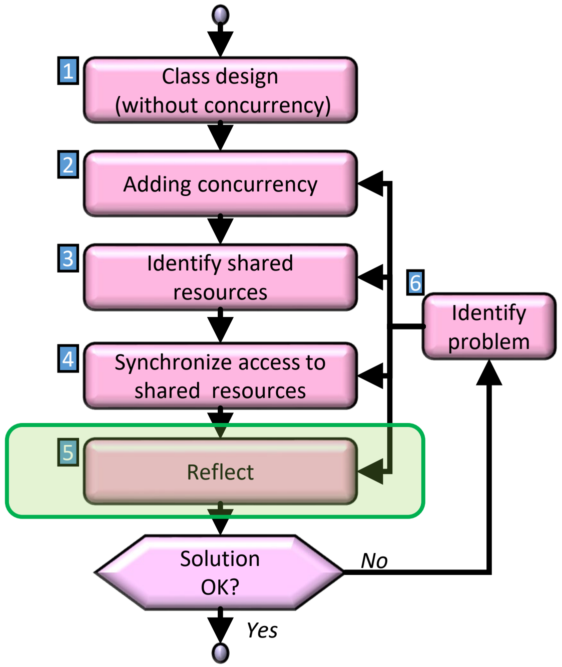
# Bar step 4: implementing adjustments

```java
public class Bar {
  ...
  private Lock myLock         = new ReentrantLock();
  private Condition newGlass = myLock.newCondition();

  public Glass getGlass() throws InterruptedException {
    myLock.lock();
    try {
      while ( glasses.isEmpty() ) {
          newGlass.await();
      }
      return glasses.remove(0);
    } finally {
        myLock.unlock();
    }
  }

  public void putBack( Glass g ) {
    myLock.lock();
    try {
      glasses.add( g );
      newGlass.signalAll();
    } finally {
      myLock.unlock();
    }
  }
  ...
}
```




```java
public class Drinker {
  ...
  public void tapAndDrink() {
    try {
      Glass glass = bar.getGlass();
      bar.drawBeer(glass);
      glass.empty();
      nrOfGlasses--;
      bar.putBack(glass);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
  }
  ...
}
```

# Bar step 5: evaluation




- Did we solve our problem?


Concurrency (III)