Programación de Sistemas y Concurrencia

Lesson 6: Communication and Synchronization using Shared Memory

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

Contents

- Critical Regions
- Java synchronization approaches
- Synchronized methods and Monitors
- Interruptions
- References

Conditional Critical Regions

- Disadvantages of Semaphores
  - A semaphore is a low-level primitive. Though they are very efficient from an implementation point of view, they are not suitable to resolve complex synchronization problems (readers/writers).
  - Sometimes, a semaphore is functionally overloaded (it is used to solve different problems: mutual exclusion and synchronization). This makes difficult to understand how the processes interact, and the code becomes unreadable and unmaintainable.
  - The code becomes unstable (a little missing may lead a big problem) :
    - A missing acquire in the preprotocol may violate the mutual exclusion.
    - A missing release en the postprotocol may lead to deadlocks.

Critical Regions

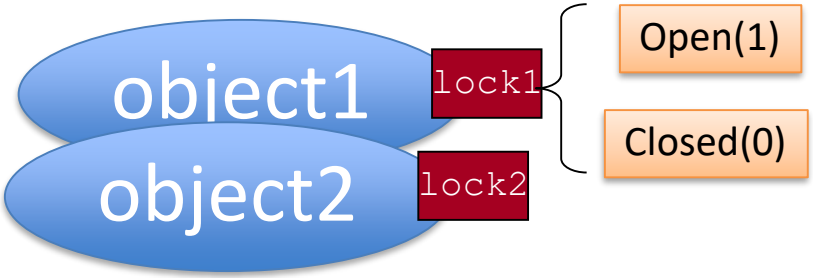
- A *critical section* is a block of sentences in a process (this block uses to access a shared resource) that *should* be executed with *mutual exclusion* regarding other critical sections of other processes.
  - **The programmer must ensure it by using mutex**
- A *critical region* is a block of sentences in a process which accesses a shared resource and, by definition, is executed with *mutual exclusion*.
  - **The language ensures it because the required mutex is included (in a transparent way) automatically**

Critical Regions

- A critical region is defined by means of a special constructor provided by the language in order to identify the block of sentences which belongs to the critical region.

```
region (Object v) {
  S;
}
```

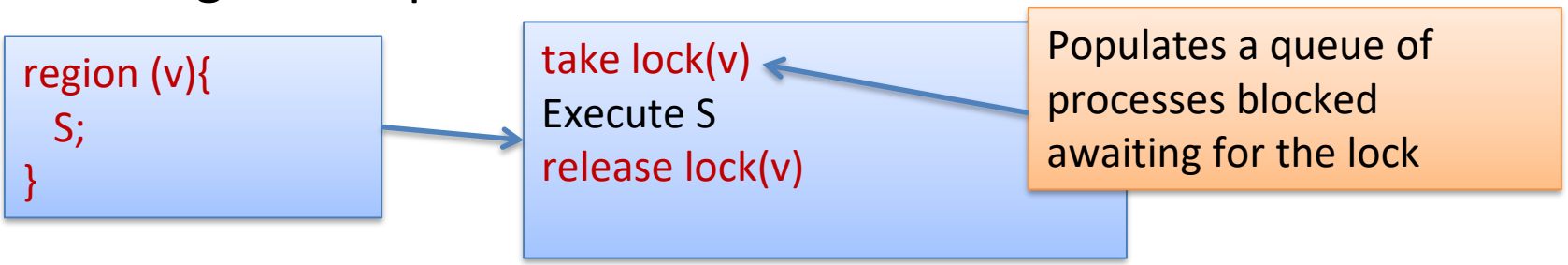
- Java has no critical regions as is; nonetheless, each Java object possesses a lock, which cannot be accessed directly.



Critical Regions

```
region (Object v) {
    S;
}
```

- In order to execute S, a process P_i must obtain previously the *lock* of the resource v.
- If P_i cannot because another P_j has such a *lock*, the P_i blocks in a queue associated to the resource v.
- When P_j ends its access to v (finishes the execution of S), it frees the *lock* and, thus, allowing to enter the first process awaiting in the queue.



Critical Regions

- Only the critical regions labelled with the same resource are executed with mutual exclusion. For example,

```
region (v1) {
   S1;
}
```

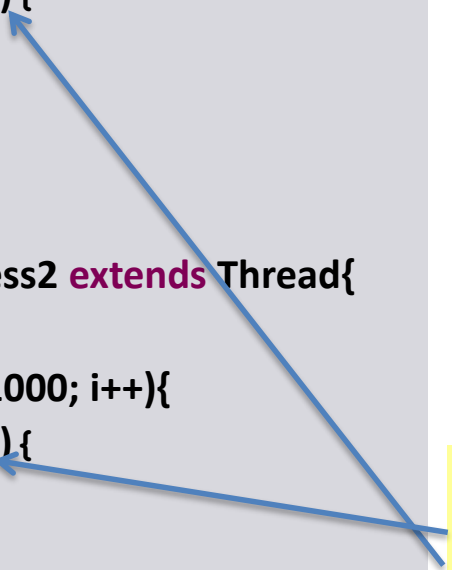
```
region (v2) {
   S2;
}
```

can be executed simultaneously, at the same time

Critical Regions: Ornamental Garden problem

```
public class MutualExclusion {
    private static int count = 0;
    public static class Process1 extends Thread{
            public void run(){
                for (int i = 0; i<1000; i++){
                  region  (count) {
                      count++;
                  }
            }
      }
      public static class Process2 extends Thread{
            public void run(){
                for (int i = 0; i<1000; i++){
                  region  (count) {
                      count++;
                  }
            }
      }
}
```

```
public static void main(String[] args){
      Process1 p1 = new Process1();
      Process2 p2 = new Process2();
      p1.start();
      p2.start();
      try{
        p1.join();
        p2.join();
      }catch (InterruptedException ie){}
      System.out.println(count);
}
}
```



Because each increment is executed with mutual exclusion, we can be sure that count = 2000 at the end of the execution

Conditional Critical Regions

- In addition to mutual exclusion, a primitive for synchronization must be able to manage synchronization conditions (as in the producer/consumer problem).
- In this sense, the critical regions must be extended with predicates which must be satisfied before a process can obtain the lock of the shared resource.

```
region (v) when (B) {
   S;
}
```

Conditional Critical Regions

```
region (v) when (B) {
   S;
}
```

- To execute a CCR, a process must:
  - Obtain the lock of the shared resource v. If not available, ten the process waits in the queue of the resource.
  - When the process obtains the lock, then evaluates the predicate B.
    - If true, then executes S.
    - If false, then frees the lock and waits again in the queue of v.
- When a process exits from the RCC, then it frees the lock of the resource

Conditional Critical Regions

```
public class ProdConsSem {
    private static final int N = 10;
    private static int[] buffer = new int[N];
    private static int items= 0;

  public static class Producer extends Thread{
      private int i = 0;
      Random r = new Random();
      public void run(){
          while (true){
              // produce aux
              region (buffer) when (items<N){
                  buffer[i] = aux;
                  i = (i + 1)% N;
                  items++;
              }
          }
  }
....
}
```

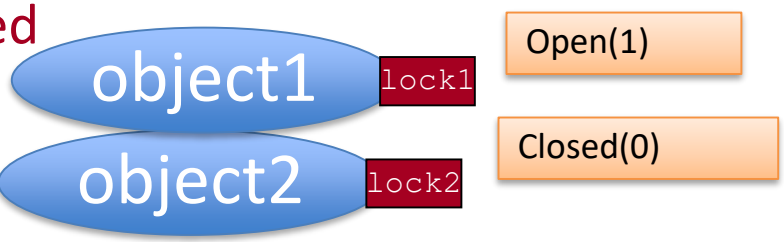
```
public static class Consumer extends Thread{
    private int i = 0;
    public void run(){
        while (true){
            region (buffer) when (items>0){
                int aux = buffer[i];
                i = (i + 1)% N;
                elem--;
            }
            // consume aux
    }
}
```

Conditional Critical Regions

- Disadvantages of CCR
  - The approach in which a process is blocked is inefficient. When a blocked process (waiting for access to the lock) is unblocked, the predicate may be false so it must be blocked again. This behaviour may seen as a kind of busy waiting.
    - A solution could be: the exiting process evaluates the predicates of the waiting processes and wakes up only one which satisfies it. Unfortunately, the predicates usually contains local variables out of the scope of the exiting process
  - The code that access the protected resource is spread over the code of the threads. This may be unsafe because the processes have access to the internal state of the resource and can lead it to an unstable state.

Java Synchronization Approaches

- Each Java object possesses a lock, which cannot be accessed directly, but it is used in
  - methods declared as synchronized
  - synchronized blocks



- Before being able of executing a **synchronized** method, must be obtained the lock of the object it belongs to.
- Therefore, synchronized methods ensures that the data of an object can be accessed with mutual exclusion, (if all its methods are synchronized)
- To execute a non synchronized method is not required to obtain the lock, so they can be executed at any moment.

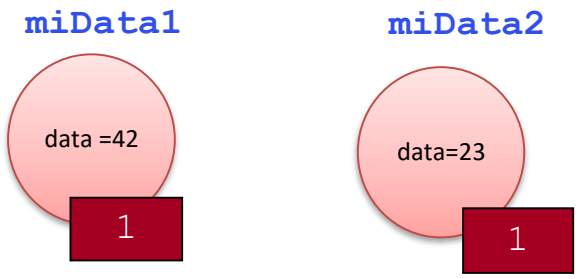
Java Synchronization Approaches

- A **synchronized** method in Java synchronizes with any other **synchronized** method of the same object.
- This way, an object can contain:
  - Resources that require access control (wrapped by **synchronized** methods)
  - Other resources that can be accessed without control, i.e. non synchronized and which can be accessed at any moment.
- From the point of view of the code distribution and encapsulation, this is a good approach because the shared resource is encapsulated inside an object. And the resource can be managed exclusively using the methods of such an object.

Java Synchronization Approaches

```
class SharedVarInt {

  private int data;
  public SharedVarInt (int initialValue) {
     data = initialValue;
  }
  public synchronized int read() {
      return data;
  }
  public synchronized void write(int aValue) {
      data = aValue;
  }
  public synchronized void inc (int increm) {
     data = data + increm;
  }
}
SharedVarInt miData1 = new SharedVarInt (42);
SharedVarInt miData2 = new SharedVarInt (23);
```

Any access to each data of an instance of the class SharedVarInt is made with mutual exclusion

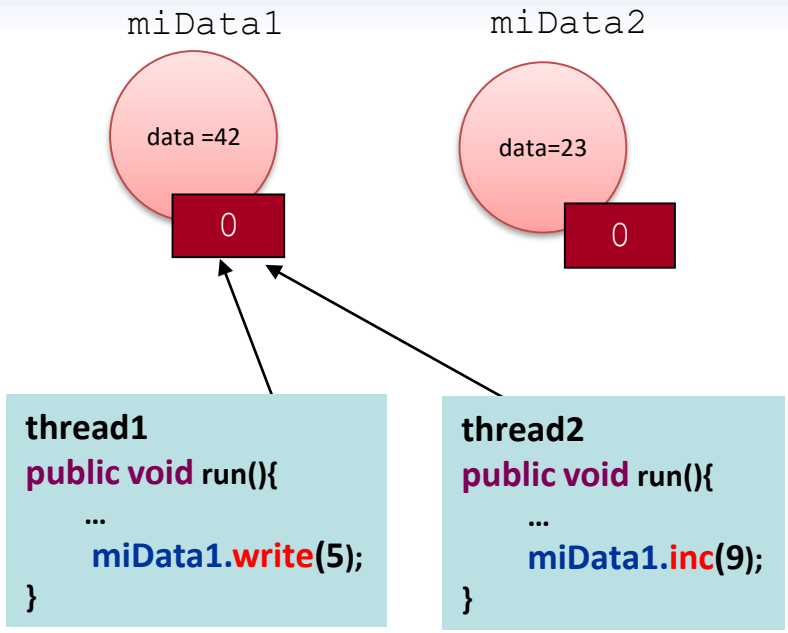


Just when an object is created, its lock is open (1)

Note: a constructor cannot be synchronized (it makes no sense)

Java Synchronization Approaches

```
class SharedVarInt {
.....

 public synchronized void write(int aValue) {
     data = aValue;
  }
  public synchronized void inc (int increm) {
     data = data + increm;
  }
.....

}

SharedVarInt myData1 = new SharedVarInt (42);
SharedVarInt myData2 = new SharedVarInt (23);
```



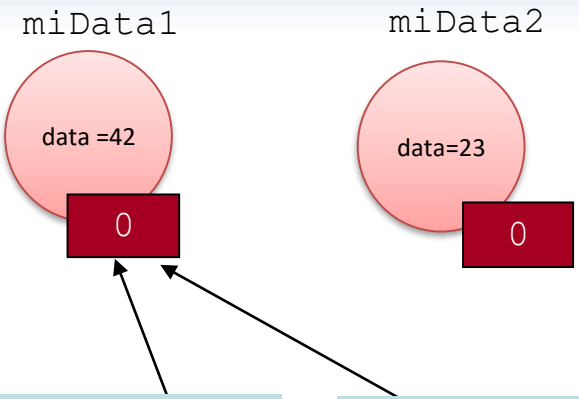

```
public synchronized void write(int aValue){
   takeLock();
   data = aValue;
   releaseLock()
 }
```

takeLock is executed atomically, so only a single thread can execute the code immediately. The other one must wait.

Java Synchronization Approaches

```
class SharedVarInt {
.....
 public synchronized void write(int aValue) {
     data = aValue;
 }
 public synchronized void inc (int increm) {
    data = data + increm;
 }
.....
}

SharedVarInt miDato1 = new SharedVarInt (42);
SharedVarInt miDato2 = new SharedVarInt (23);
```

We have two traces…




```
thread1
public void run(){
      ...
      miData1.write(5);
}
```

```
thread2
public void run(){
      ...
      miData1.inc(9);
}
```

```
public synchronized void write(int aValue){
   takeLock();
   data = aValue;
   releaseLock()
 }
```

```
thread1: takeLock (0)
thread2: takeLock (0) waits
thread1: data = 5
thread1: releaseLock (0)
thread2: data = data + 9
thread2: releaseLock (1)
```

```
thread2: takeLock (0)
thread1: takeLock (0) waits
thread2: data = data + 9 (data = 51)
thread2: releaseLock (0)
thread1: data = 5
thread1: releaseLock (1)
```

Synchronized blocks

- Any sequence of sentences (block) inside any method (synchronized or not) can be executed with mutual exclusion.
- To do so, we encompassed it with curly brackets and prefixed it with the keyword synchronized parameterized with the object whose lock we need to safely execute the block of code

```
public synchronized void write(int aValue) {
  data = aValue;
 }
```

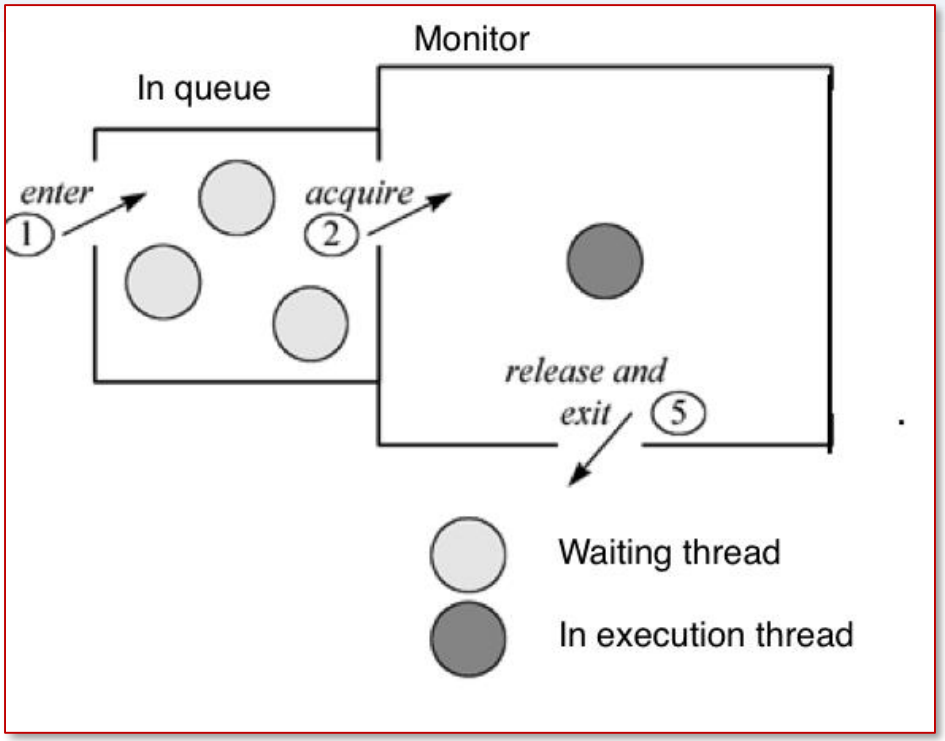
is equivalent to

```
public void write(int aValue) {
   synchronized (this){
            data = aValue;
   }
}
```

this is a reference to the current object

Java Synchronization Approaches

- This is like if each object had an incoming hall where threads wait to acquire the lock
- At each moment, any synchronized method can be executed by one thread at most
- The incoming hall can contain any number of threads
- Java specification does not force a particular implementation for this hall: FIFO, LIFO, priority queue, etc.



Synchronized methods and Monitors

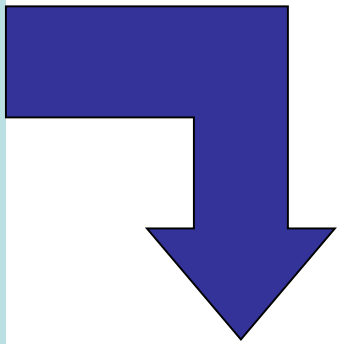
- Synchronized methods are a simplification (in earliest Java versions) of the concept of monitor defined by **Hoare** and **Hansen** in the middle 70's.
- A monitor is a structure provided by a concurrent programming language that allows to encapsulate every synchronization issue related with a resource, (resource that is usually embedded into the monitor's structure itself). By definition, every public procedure, function or method provided by the monitor is always executed with mutual exclusion. In addition, the monitor provides variables of type condition to make easier to model synchronization conditions.

- Thus, in Java we should implement all the synchronized code required by an object into the class it belongs to. This is carried out by means of synchronized methods and blocks.
- If this approach is not followed, the synchronization logic of an resource/object will be very difficult to understand because it will not be contained exclusively in its class file, i.e. other pieces of code can contain a synchronized block parameterized with such object.
- Anyway, using synchronized blocks carefully, we can model easily very complex synchronization problems.

Synchronized blocks: sample

- Let's say a class which represents two-dimensional coordinates, which can be shared by two or more threads.

```
public class Coordinates{
  private int x, y;

  public Coordinates (int initX, int initY) {
    x = initX;
    y = initY;
  }
  public synchronized void write(int newX, int newY) {
    x = newX;
    y = newY;
  }
  ...

}
```



Updating the coordinates is very easy: it is enough to assign them new values inside a synchronized method

Synchronized blocks: sample

- However, how to read the values of the coordinates (two values)?
- A Java function only can return a single value, and there is no ability to use parameters by reference
- Thus, it is impossible to write a single method which provides both **x** and **y** at the same time.
- Furthermore, if we would use two synchronized functions **readX** and **readY**, it would be possible that an uncontrolled call to **write** (made by another process) could be executed between the two calls to such methods so the returned values became inconsistent.

```
public class Coordinates{
  private int x, y;

  public Coordinates (int initX, int initY) {
    x = initX;
    y = initY;
  }
  public synchronized void write(int newX, int newY) {
    x = newX;
    y = newY;
  }
  ...

}
```

Synchronized blocks: sample

- **First solution**: Creating a method which return a clone of the actual Coordinates object (and with the same x and y values)
- The returned object Coordinate is consistent. However, the original values could be changed just after executing the method (and the returned object does not reflect the change because it is a copy)
- Once the clone has been used, it can be thrown away in order to be collected by the garbage collector
- From the point of view of the efficiency this is not a good practice because we are using objects of type "use and throw away"

```
public class Coordinates {
  private int x, y;
.....
 public synchronized Coordinates read() {
    return new Coordinates (x, y);
  }
  public int readX() { return x; }
  public int readY() { return y; }

}
```

This is a valid solution

Synchronized blocks: sample

- **Second Solution**:

The client thread may use a synchronized block to achieve atomicity

```
public class Coordinates {
  ...
  public synchronized void write(int newX, int newY) {
        x = newX; y = newY;
   }

  public int readX() { return x; } // not synchronized
  public int readY() { return y; } // not synchronized
}
Coordinates point1 = new Coordinates (0,0);


synchronized(point1) {
  Coordinatespoint2 = new Coordinates(
              point1.readX(), point1.readY());
}
```

This is a valid solution too

Static Data Synchronized

- Static variables are shared by every object which belongs to a given class
- In Java, the classes are object as well, so they have an associated synchronization lock
- This lock can be used by defining any static method as synchronized; or using the Class object itself as a parameter in a synchronized block
- This Class lock can be obtained even when an object which belongs to such a class is in use in a synchronization method (i.e., obtaining the lock of an object is independent of obtaining the lock of its class)

```
class StaticSharedVar {
  private static int sharedVar;
  ...

  public int read() {
      synchronized(StaticSharedVar.class) {
          return sharedVar;
       };
  }

  public synchronized static void write(int I) {
      sharedVar = I;
  }
}
```

Ornamental Garden problem

```
public class GardensSync {
        public static class Counter{
              private int count = 0;
              public synchronized void inc(int c){
                    count +=c;
               }
              public int count(){
                   return count ;
              }
         }
         public static class Gate extends Thread{
              private Counter count ;
              public Gate(Counter counter){
                    this.count  = counter;
               }
               public void run(){
                    for (int i=0; i<100; i++)
                         count.inc(1);
              }
         }
         ……
```

```
public static void main(String[] args){
        Counter count = new Counter ();
        Gate[] p = new Gate[25];
        for (int i = 0; i<25; i++)
             p[i] = new Gate(count);
        for (int i = 0; i<25; i++)
             p[i].start();
        try{
          for (int i = 0; i<25; i++)
               p[i].join();
         }catch (InterruptedException ie){}
        System.out.println(count.count());
}

}
```

Any increment of count is carried out with mutual exclusion

Synchronization Conditions

- To define synchronization conditions, we use the methods wait and notify defined in the class Object

```
public class Object {
  ...
  public final void notify();
  public final void notifyAll();
  public final void wait()  throws InterruptedException;
  public final void wait(long millis)throws InterruptedException;
  public final void wait(long millis, int nanos)
       throws InterruptedException;
  ...
}
```

- These methods **only** can be used once the lock of the object has been gained (inside a synchronized method or block of sentences). Otherwise is thrown the untested exception **`IllegalMonitorStateException`**

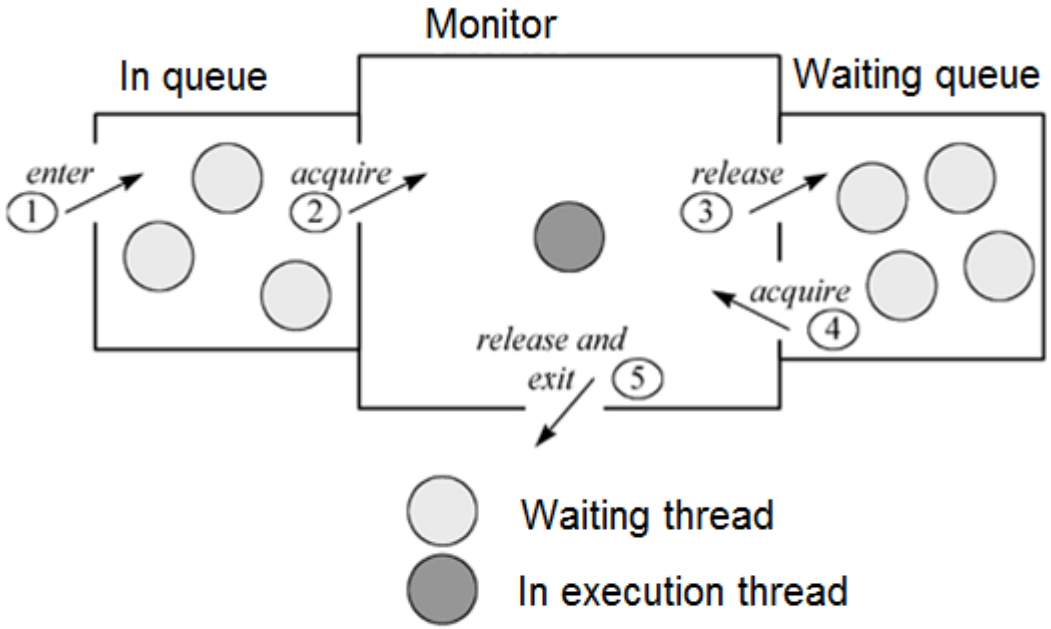
Synchronization Conditions: wait

- The method **wait()** always suspends the thread executing it
- It is used when a synchronized method is being executed but, due to some condition, the current status of the shared object is not the required one in order to continue the execution. Then, the thread is suspended waiting for a change in the object's status so the execution can be restarted
- When a thread executed **wait()**:
  - It frees the lock of the shared object
  - It is blocked into the *waiting queue*

Synchronization Conditions: wait

Associated to any shared object may be:

- Several threads waiting inside an In queue
- One thread (at the most) executing a synchronized method (or block of sentences)
- Several threads waiting inside a Waiting queue
- Several threads executing a non synchronized method



Synchronization Conditions: notify

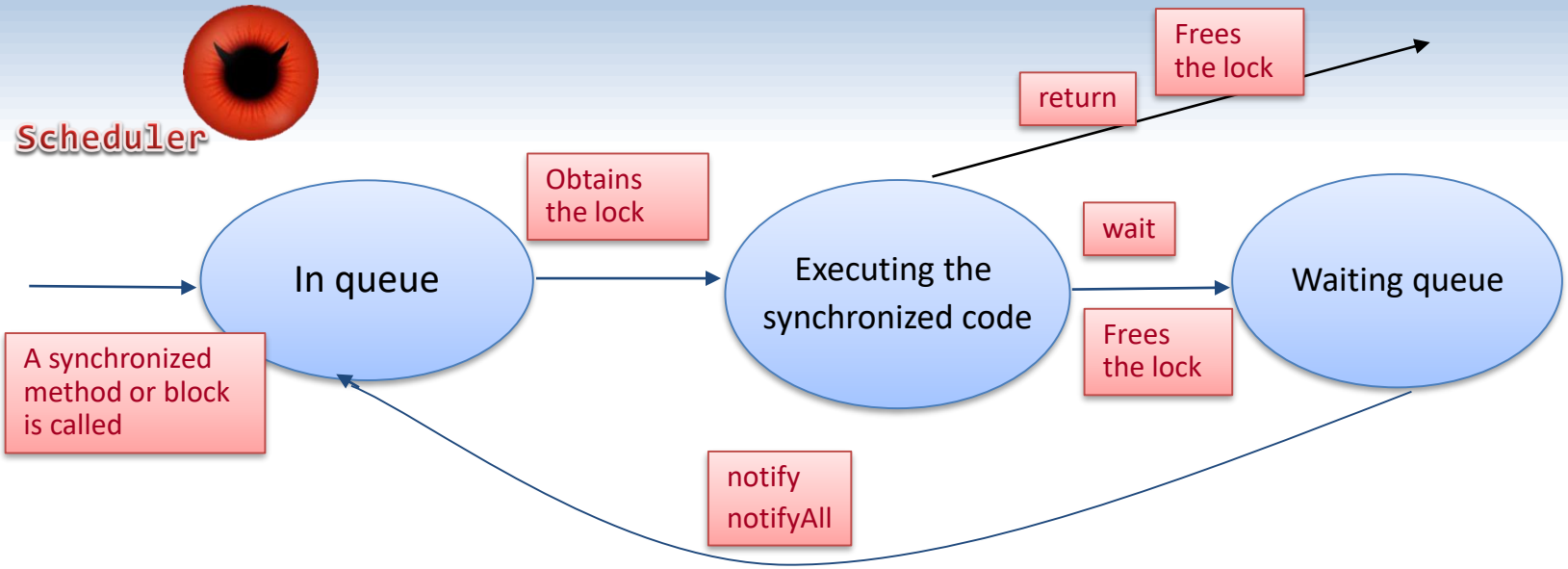
- The method **notify()** wakes up only one thread in the Waiting queue (anyone)
- The method **notifyAll()** is used too wake up all the waiting threads
- If there is not any waiting thread, these methods *has no effect*
- A waiting thread can be interrupted explicitly by any other one. In such a case, the exception **`InterruptedException`** is thrown

Synchronization Conditions: notify

- The thread which executes the **notify()** does not free the lock of the shared object, so the woken up thread must obtain it again to be able to continue its execution
- This is known as notify-and-continue discipline
- With this approach, it must be noted that a random time may elapse between the moment the thread is woken up and the moment it gains the lock. This is very important because in such a time, many other threads may have gained the lock of the shared object and, therefore, they may have changed the status of the object
- Thus, the woken up thread must re-test the status of the object in order to be sure the required condition remains being satisfied. If not, it must wait again:

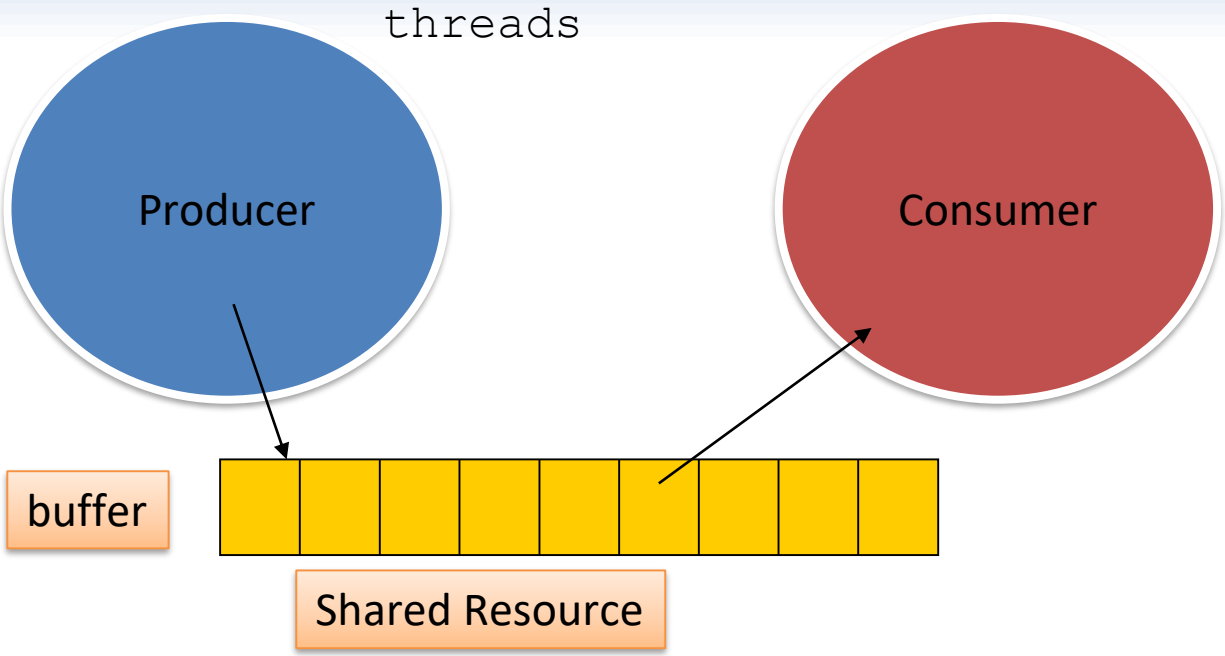
```
while (condition)
  try {
    wait();
  } catch (InterruptedException e){}
```

notify-and-continue discipline



- When a thread calls a synchronized method or block which belongs to a shared object, it waits inside the In queue if it cannot obtain the lock immediately
- When it has the lock, it executes the synchronized method or block with mutual exclusion
- Through the execution of a synchronized code, the thread may:
  - Execute wait(): the lock is freed and the thread is sent to the Waiting queue
  - Execute notify()/notifyAll(): the thread continues its execution but one/all the threads in the Waiting queue are sent to the In queue

Producer/Consumer problem with bounded buffer



Properties

- A producer can not store into the buffer if full
- A consumer can not take from the buffer if empty
- The buffer is accesses by mutual exclusion

Producer/Consumer problem with bounded buffer

```
public class Buffer <T>{

   private   T[] b;
   private int i=0,j=0;
   private int numItems = 0;
   public Buffer(int size){
       b = (T[]) new Object[size];
   }
  public synchronized T get()
              throws InterrutpedException{
       while (numItems == 0)
           wait();
         int aux = i;
         i = (i + 1) % b.length;
         numItems--;
         notifyAll();
         return b[aux];
   }
  …
```

CS2

```
public synchronized void put(T nData)
              throws InterruptedException{

    while (numItems == b.length)
        wait();

         b[j]=nData;
         j = (j + 1) % b.length;
         numItems++;
         notifyAll();

}

}
```

CS1

Synchronization conditions:

CS1: A producer can not store into the buffer if full

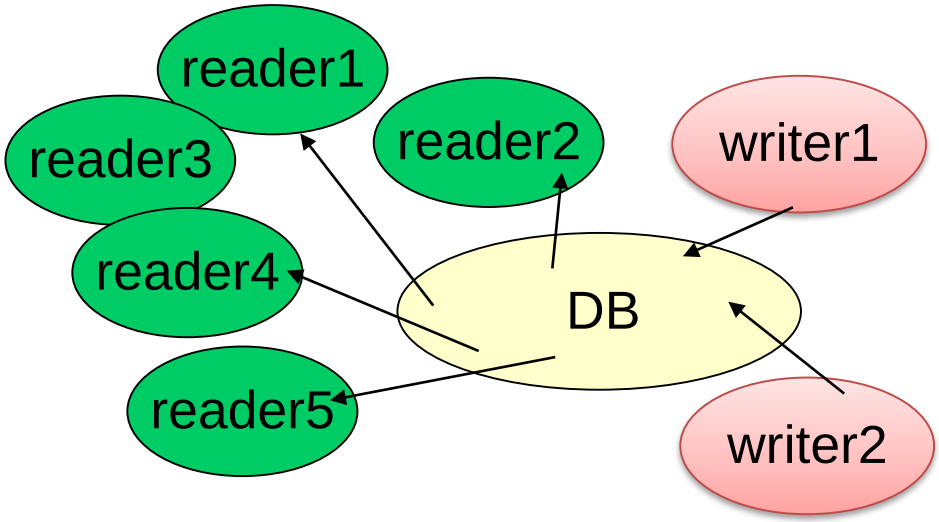
CS2: A consumer can not take from the buffer if empty

Disadvantages of a single Waiting queue

- When a thread is woken up, it cannot assume that the condition it is waiting for is satisfied. This is because notify/notifyAll wake up all the threads no matter the predicate it is waiting for
- In some algorithms this is not a problem, because the tested predicates are mutually exclusive (as in the previous case).
- In many other situations, several predicates may be true at the same time. And this may lead to problems that must be controlled.
- In any case, a time elapses between the notify and the actual taking of the lock. In this time, the condition may become false again. Because of this, checking the condition must be done inside a while loop

Readers/Writers problem

- Readers and Writers access a shared resource, usually a database (DB).
- The system must satisfy several synchronization conditions:
  - Many readers may access the DB at the same time
  - Each writer may access the DB with mutual exclusion regarding any other thread (either writer or reader)



Classes Reader and Writer

```
public class Reader extends Thread{
   private int myId; // id of the thread
   // object to control DB access
   private DBMS mng;
   // to sleep for a random time
   private static Random r = new Random();
   public Reader(int id, DBMS c){
       myId = id;
       this.mng = c;
    }

   public void run(){

       for (int i = 0; i<10;i++){
          try{
            mng.openR(myId);
           // Reader myId is inside the DB
           Thread.sleep(r.nextInt(500));
           mng.closeR(myId);
           } catch (InterruptedException ie){};
      }
}
```

```
public class Writer extends Thread{
   private int myId;
   private DBMS mng;
   private static Random r = new Random();
   public Writer(int id, DBMS c){
       myId = id;
       this.mng = c;
    }

   public void run(){

       for (int i = 0; i<10;i++){
          try{
            mng.openW(myId);
           // Writer myId is inside the DB
           Thread.sleep(r.nextInt(500));
           mng.closeW(myId);
           } catch (InterruptedException ie){};
      }
}
```

openR, closeR, openW and closeW implement the synchronization conditions

Synchronization conditions: Readers and Writers

```
public class DBMS {
    private int nReaders=0;
    private boolean writing=false;

    public synchronized void openR(int id)
                throws InterruptedException{
        while (writing)
            wait();
        nReaders++;
    }
    public synchronized void openW(int id)
            throws InterruptedException{
        while (writing || nReaders > 0)
            wait();
        writing = true;
    }
    .....
```

```
....
public synchronized void closeR(int id) {
    nReaders--;
    if (nReaders == 0) notify();
}


public synchronized void closeW(int id) {
    writing = false;
    notifyAll();
}

}
```

- If there is a writer inside the DB, any reader has to wait
- If there is a writer or any readers inside the DB, any (other) reader has to wait
- Please, notice that this class models the DB access but it does not contains it. Why?
- When a writer exits from the DB, wakes up any waiting thread:
  1) If a reader obtains the lock of the DBMS object, no writer can gain access to the DB but any other reader can
  2) If a writer obtains the lock, then it prevents the entrance of any other thread
- As a theorem, we can say that: If there is a reader inside then there is no readers in the Waiting queue
- When the last readers exists, it wakes up only a thread (if it exists then it must be a writer)

Synchronization conditions: Readers and Writers

- Problems of this approach
  - An exiting writer must launches a **notifyAll()** because if the selected thread to be woken up is a reader then any other reader must be woken up also
    - All waiting threads (readers and writers) are woken up, even if notifyAll() selects a writer to enter the critical section. In such a situation, the rest of threads must decide to wait again. This is an inefficient solution because many threads are woken up with no need
    - Even more important, the previous code is not fair for writers: readers can take over the DB all the time, avoiding any writer to gain access (so they may wait forever)

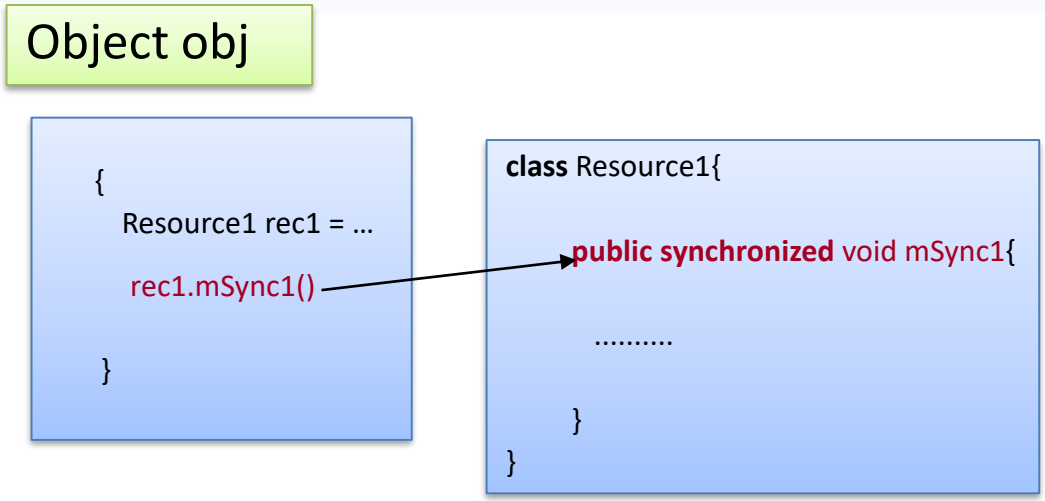
Synchronization conditions: Readers and Writers (fair version)

```
public class DBMS {
    private int nReaders=0;
    private int nWriters = 0;
    private boolean writing=false;

    public synchronized void openR(int id)
                throws InterruptedException{
        while (writing || nWriters > 0)
            wait();
        nReaders++;
    }
    public synchronized void openW(int id)
            throws InterruptedException{
        nWriters++;
        while (writing || nReaders > 0)
            wait();
        writing = true;
    }
    .....
```

```
....
public synchronized void closeR(int id) {
    nReaders--;
    if (nReaders == 0) notifyAll();
}


public synchronized void closeW(int id) {
    nWriters--;
    writing = false;
    notifyAll();
}

}
```

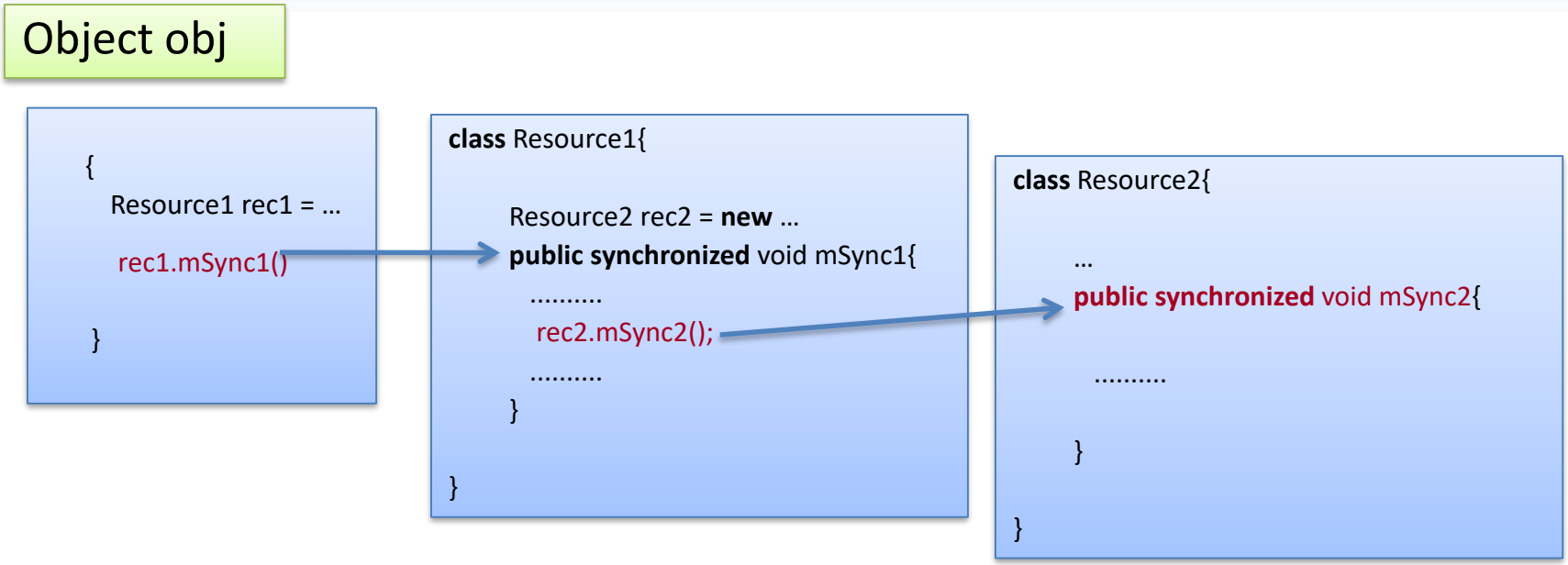
- The nWriters variable holds how many writers want to update the DB
- If there is a waiting writer, the next reader must wait even if the DB is currently locked by readers
- When a writer exits from the DB, the variable nWriters is decremented
- Although this approach is fair for writers, the problem of a single waiting queue persists: any thread waits in the same queue no matter the condition it is waiting for
- Suspending and waking up remain being inefficient

Nested calls to synchronized methods



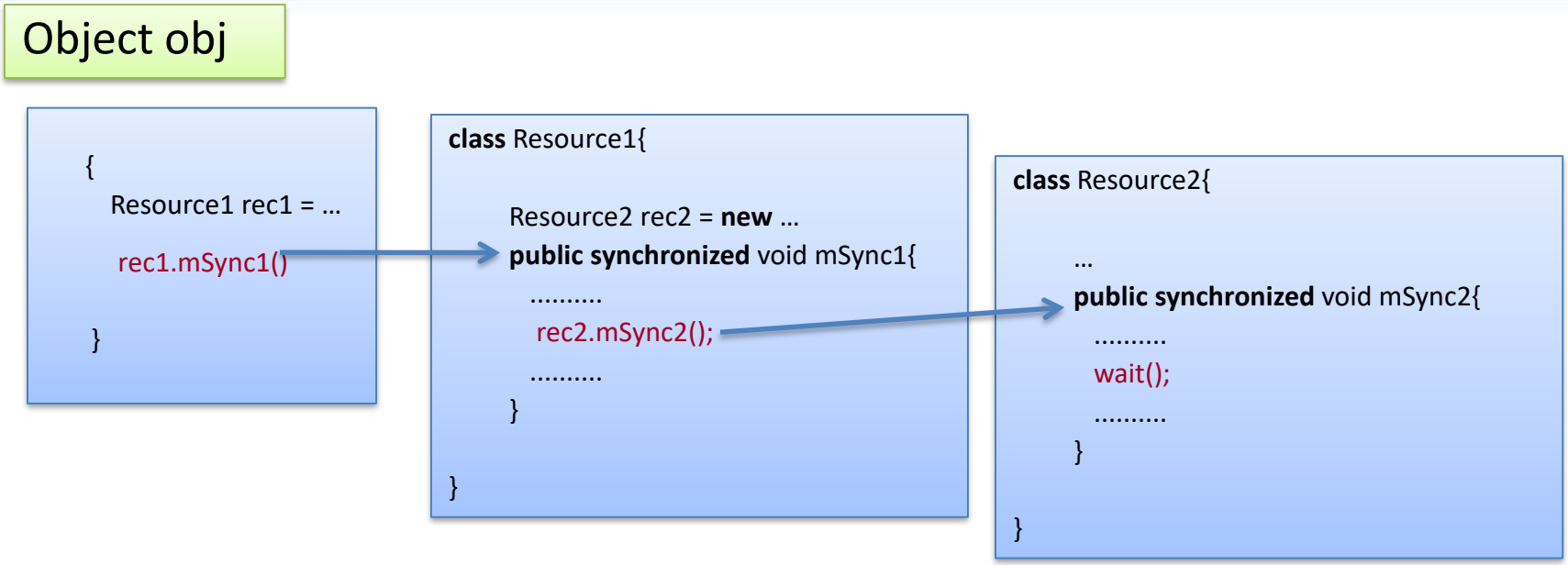
• **obj** must obtain the lock of the object **rec1** in order to be able of executing the method **mSync1**

Nested calls to synchronized methods



- **obj** must obtain the lock of the object **rec1** in order to be able of executing the method **mSync1**
- In addition, to be able of executing **mSync2**, **obj** must obtain the lock of **rec2** also. Then, **obj** has the lock of both resources **rec1** and **rec2**.

Nested calls to synchronized methods



- **obj** must obtain the lock of the object **rec1** in order to be able of executing the method **mSync1**
- In addition, to be able of executing **mSync2**, **obj** must obtain the lock of **rec2** also. Then, **obj** has the lock of both resources **rec1** and **rec2**.
- If **mSync2** invokes **wait(),** the lock of **rec2** is released but **obj** keeps holding the lock of **rec1**. This problematic situation must be controlled carefully because it can lead easily to a deadlock

Condition variables. First approach

- The next class **Condition** is an approach to model a synchronization condition to be used by threads to wait for them
- **As each object has its own lock, several variables of type Condition allow grouping the threads into several waiting queues according to the condition they need to wait for**
- As an example, okRead and okWrite could be used to block the threads Readers and Writers respectively
- The main advantage is that, when a thread is woken up, we may be completely sure of its particular type

```
public class Condition {
        public synchronized void cwait()
                throws InterruptedException{
            wait();
        }
        public synchronized void cnotify(){
            notify();
        }
        public synchronized void cnotifyAll(){
            notifyAll();
        }
}
```

```
Condition okReader = new Condition();
Condition okWriter = new Condition();
```

Condition variables. First approach

```
public class DBMSCondition {
    private int nReaders = 0;
    private int nWriters = 0;
    private boolean writing = false;
    private Condition okRead = new Condition();
    private Condition okWrite = new Condition();

    public synchronized void openR(int id)
                    throws InterruptedException{
          while (writing || nWriters > 0){
              okRead.cwait();
          }
          nReaders++;
    }
    public synchronized void openW(int id)
                      throws InterruptedException{
        nWriters++;
        while (writing || nReaders > 0){
            okWrite.cwait();
        }
        writing = true;
    }
....
}
```

```
....
public synchronized void CloseR(int id) {
    nReaders --;
    if (nReaders == 0) okWrite.cnotify();
}


public synchronized void CloseW(int id) {
    nWriters --;
    writing = false;
    if (nWriters > 0) okWrite.cnotify();
    else okRead.cnotifyAll();

}
```

Unfortunately, this solution is invalid because the system can become blocked: there are nested calls to synchronized methods

Condition variables. First approach

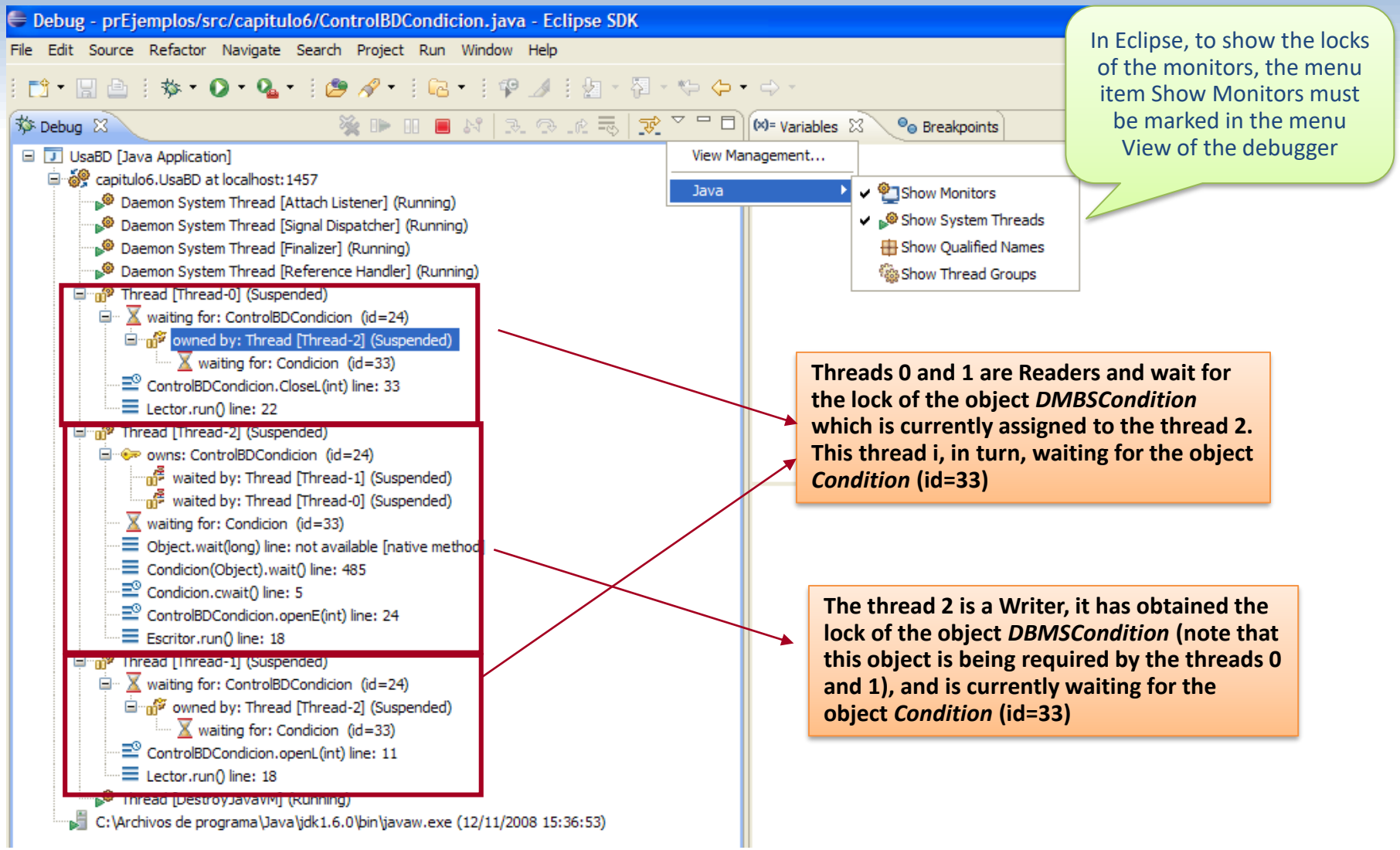
```
public class DBMSCondition {
    private int nReaders = 0;
    private int nWriters = 0;
    private boolean writing = false;
    private Condition okRead = new Condition();
    private Condition okWrite = new Condition();
    public synchronized void openR(int id)
                    throws InterruptedException{
        while (writing || nWriters > 0){
            okRead.cwait();
        }
        nReaders++;
    }
    public synchronized void openW(int id)
                    throws InterruptedException{
        nWriters++;
        while (writing || nReaders > 0){
            okWrite.cwait();
        }
        writing = true;
    }
public synchronized void closeR(int id) {
    nReaders --;
    if (nReaders == 0) okWrite.cnotify();
}
public synchronized void closeW(int id) {
    nWriters --;
    writing = false;
    if (nWriters > 0) okWrite.cnotify();
    else okRead.notifyAll();
  }
}
```

Action	nRea	nWri	Lock DBMS	lock okWrite
Initially	0	0	open	open
R1: c.openR()	1	0	open	open
R2: c.openR()	2	0	open	open
W1: c.openW()	**2**	**1**	**W1**	**W1 waits for notify**
R1: c.closeR()	2	1	W1 R1 waiting	W1 waits for notify
R2: c.closeR()	2	1	W1 **R1 waiting** **R2 waiting**	**W1 waits for notify**

```
public class Condition {
        public synchronized void cwait()
                    throws InterruptedException{
                wait();
        }
        public synchronized void cnotify(){
                notify();
        }
        public synchronized void cnotifyAll(){
                notifyAll();
        }
}
```

The three processes are blocked

Detecting deadlock debugging with Eclipse



In Eclipse, to show the locks of the monitors, the menu item Show Monitors must be marked in the menu View of the debugger

Threads 0 and 1 are Readers and wait for the lock of the object *DMBSCondition* which is currently assigned to the thread 2. This thread i, in turn, waiting for the object *Condition* (id=33)

The thread 2 is a Writer, it has obtained the lock of the object *DBMSCondition* (note that this object is being required by the threads 0 and 1), and is currently waiting for the object *Condition* (id=33)

Synchronization conditions: Java 1.5

- Java 1.5 provides new abilities for concurrent programming
- There are three main packages:
  - **java.util.concurrent**. It supplies classes to work with very usual concurrent elements like bounded buffers, sets and maps, pools of threads, etc.
  - **java.util.concurrent.atomic**. It provides safety (atomicity without explicit control of locks) in accession to primitive and usual data like atomic integers, atomic booleans etc.
  - **java.util.concurrent.locks**. It supplies several types of locks as an enhance to the basic approach of previous versions of Java: locks for read/write, condition variables, etc.

Locks

java.util.concurrent.locks supplies two interfaces:

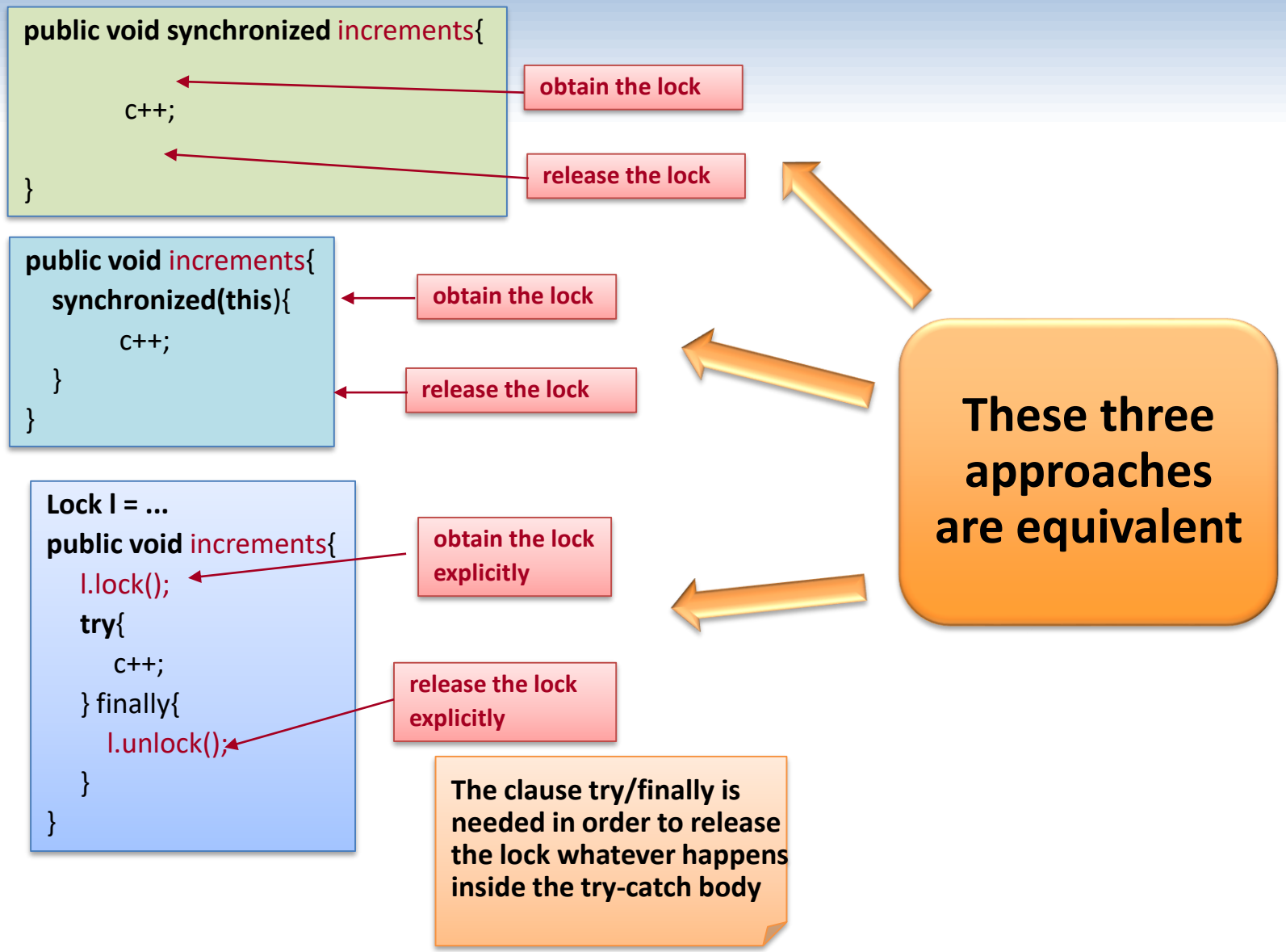
- One which allows creating new locks
- One which allows creating conditions associated to such locks

```
package java.util.concurrent.locks;
public interface Lock {

  // Waits until the lock is obtained
  public void lock();

  // Creates a new variable Condition
  // associated to the lock
  public Condition newCondition();

  // release the control of the lock
  public void unlock();
  …
}
```

```
package java.util.concurrent.locks;
public interface Condition {

  // Frees atomically the lock associated
  // to the condition and blocks the thread
  public void await() throws InterruptedException;

  // Wakes up a thread waiting for this condition
  public void signal();

  // Wakes up every thread waiting for this condition
  public void signalAll();
   …
}
```

- A Lock is like a synchronized
- A Condition is like a waiting room associated to a Lock

Locks


public void synchronized increments{
c++;
}
obtain the lock
release the lock
public void increments{
synchronized(this){
c++;
}
}
obtain the lock
release the lock
Lock l = ...
public void increments{
l.lock();
try{
c++;
} finally{
l.unlock();
}
}
obtain the lock explicitly
release the lock explicitly
The clause try/finally is needed in order to release the lock whatever happens inside the try-catch body
These three approaches are equivalent

Locks

```
public void synchronized increments{
        c++;
}
```

obtain the lock

release the lock

```
public void increments{
   synchronized(this){
        c++;
   }
}
```

obtain the lock

release the lock

```
Lock l = ...
public void increments{
   l.lock();
   try{
      c++;
   } finally{
      l.unlock();
   }
}
```

obtain the lock explicitly

release the lock explicitly

The clause try/finally is needed in order to release the lock whatever happens inside the try-catch body

- The methods/blocks synchronized allows an exclusive access to the implicit lock associated to an object, usually a resource shared by many threads
- This approach is transparent and elegant and should be used whenever possible. However, it has disadvantages as we have shown in the Readers/Writers problem
- The synchronized methods/blocks take and release the lock of the object in very specific positions of the code, and this cannot be changed. Moreover, if there are nested calls to synchronized methods/blocks, the locks of the gained objects are released in reverse order to that of taking
- Nevertheless, this approach may not be the best suitable to every situation
- The objects of type Lock resolve this problem; they make more flexible where to place a lock and unlock in a program, but is the programmer responsibility to be sure of releasing a lock (at some time) once it has been taken

Locks

- The package java.util.concurrent.lock provides several implementations of the interface Lock, with different behaviours

```
package java.util.concurrent.locks;
public class ReentrantLock implements Lock {
  public ReentrantLock();
  public ReentrantLock(boolean fair);
  ...
  public void lock();
  public Condition newCondition();
  public void unlock();
}
```

- ReentrantLock has a constructor which allows to specify that we need a fair lock
- Fairness is implemented waking up the oldest waiting thread
- This prevents starvation (waiting indefinitely to continue the execution)
- In addition, ReentrantLock solves correctly nested calls to locks

Locks: Readers/Writers problem

```
Import java.util.concurrent.locks.*;
public class DBMSLocks {
    private int nReaders=0;
    private int nWriters = 0;
    private boolean writing = false;
    private Lock l = new ReentrantLock(true);

    private Condition okRead = l.newCondition();
    private Condition okWrite = l.newCondition()

    public void openR(int id) throws InterruptedException{
        l.lock();
        try {
            while (writing || nWriters > 0){
                okRead.await();
            }
            nReaders++;
        } finally {
            l.unlock();
        }
    }
    public void closeR(int id) {
        l.lock();
        try{
            nReaders--;
            if (nReaders == 0) okWrite.signal();
        } finally {
            l.unlock();
        }
```

```
    public void openW(int id) throws InterruptedException{
        l.lock()
        try{
            nWriters++;
            while (writing || nReaders > 0){
                okWrite.await();
            }
            writing = true;
        } finally {
            l.unlock();
        }
    }

    public void closeW(int id) {
        l.lock();
        try{
            nWriters--;
            writing = false;
            if (nWriters > 0) okWriter.signal();
            else okRead.signalAll();
        } finally{
            l.unlock();
        }
    }
}
```

ReadWriteLock

- Interface ReadWriteLock and the class ReentrantReadWriteLock

```
package java.util.concurrent.locks;
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

```
package java.util.concurrent.locks;
public class ReentrantReadWriteLock {
  public ReentrantReadWriteLock();
  public ReentrantReadWriteLock(boolean fair);
  public Lock readLock();
  public Lock writeLock();
}
```

```
import java.util.concurrent.locks.*;
public class ControlDBRWLock implements Control {
    private ReadWriteLock l = new ReentrantReadWriteLock(true);
    private Lock lr = l.readLock();
    private Lock lw = l.writeLock();

    public void openR(int id) throws InterruptedException {
        lr.lock();
    }
    public void openW(int id) throws InterruptedException {
        lw.lock();
    }
    public void closeR(int id) {
        lr.unlock();
    }
    public void closeW(int id) {
        lw.unlock();
    }
}
```

Java also provides a ReadWriteLock interface to allow creating locks associated to structures following the protocol of Readers/Writers just described

Reentrant Locks

- Because locks are reentrant, if a thread requires to take a lock it already has, then the requirement is immediately satisfied
- When a thread requires a lock assigned to a different thread, then it becomes blocked but …
- Reentrancy is implemented by means of a **counter** associated to each lock and an **owner thread**
  - When the counter is zero, the lock is available
  - When a thread takes an available lock, JVM register such thread as the owner of the lock and sets the counter to 1
  - If the same thread takes the lock again, the counter is incremented
  - If the thread exits from a “locked” method, the counter is decremented
  - When the counter reaches zero, then the lock becomes available
- This behaviour is also applied to any lock belonging to the class java.util.concurrent.locks.ReentrantLock

Synchronized methods/objects vs. Locks

- Synchronized methods/objects versus Locks
  - To use locks, the programmer have to take and release them explicitly, and this may lead to errors if not enough careful:
    - Forgetting to take a lock may violate the mutual exclusion of an object
    - Forgetting to release a lock may lead to deadlocks
  - Generally speaking, if a class does not require to differentiate among several types of synchronization conditions, then the best is to use synchronized methods/objects instead of locks.
  - An important behaviour of synchronized methods/objects is that locks are released in reverse order to that of taking. And, usually, this is the expected behaviour.
  - On the other hand, if there are several synchronization conditions or the locks must be released in a different order, then it must be used some implementation of the interface Lock.

Implementing semaphores by means of monitors: two approaches

```
public static class Semaphore{
  private int value;

  public Semaphore(int v){
     value = v;
  }
  public synchronized void acquire()
              throws InterruptedException{
    while (value == 0) wait();
    value --;
  }
  public synchronized void release() {
    value ++;
    notify();
  }
}
```

With this approach, a thread just woken up may have to suspend again if a different thread overtakes it

```
public static class Semaphore{
    private int value;
    private int waiting = 0;

    public Semaphore(){
          value = 1;
    }
    public synchronized void acquire()
                    throws InterruptedException{
        if (value == 0) {
                waiting ++;
                wait();
                waiting --;
          }
          else value = 0;
      }
      public synchronized void release() {
            if (waiting > 0) notify();
            else value = 1;
      }
}
```

With this approach, the thread just woken up is the first which executes the method acquire() because it is the only knowing that the semaphore is null

Implementing semaphores by means of monitors: two approaches

Warning:

In the Java API you may find that wait() and await() can be woken up spuriously, i.e. without any notify() or notifyAll(). This is due to the internal implementation in the O.S.

Therefore, any usage of wait() or await() has to be done inside a while.

```
public static class Semaphore{
    private int value;
    private int waiting = 0, waking = 0;

    public Semaphore(){
        value = 1;
    }
    public synchronized void acquire()
                    throws InterruptedException {
        if (value == 0) {
            waiting ++;
            do {
                wait();
            } while (waking==0)
            waking --;
        }
        else value--;
    }
    public synchronized void release() {
        if (waiting > 0) { waiting--; waking++; notify(); }
        else value++;
    }
}
```

With this approach, the thread just woken up is the first which executes the method acquire() because it is the only knowing that the semaphore is null

Semaphores vs. Conditions

- Let's say a process P1 should not continue its execution until a process P2 finishes the execution of a particular block of code

Semaphore s = new Semaphore(0); {s = 0}

Action	P1	P2	S
Initially			0
P1: s.acquire()	Suspended in s		0
P2: s.release()		Wakes up to P1	0

Action	P1	P2	S
Initially			0
P2: s.release()			1
P1: s.acquire()			0

The order of execution (traces) is not relevant because the semaphore registers the release operation

Semaphores vs. Conditions

- The same synchronization task with Conditions

```
Lock l = new ReentrantLock(true);
Condition c = l.newCondition();
```

Action	P1	P2	c
Initially			
P1: c.await()	Suspended in c		P1
P2: c.signal()		Wakes up to P1	

Action	P1	P2	c
Initially			
P2: c.signal()		No effect	
P1: c.await()	Suspended in c		P1

When Conditions are used, the **execution order is very important**.

In the second trace, P1 remains suspended forever, whereas in the first one does not.

Semaphores vs. Conditions

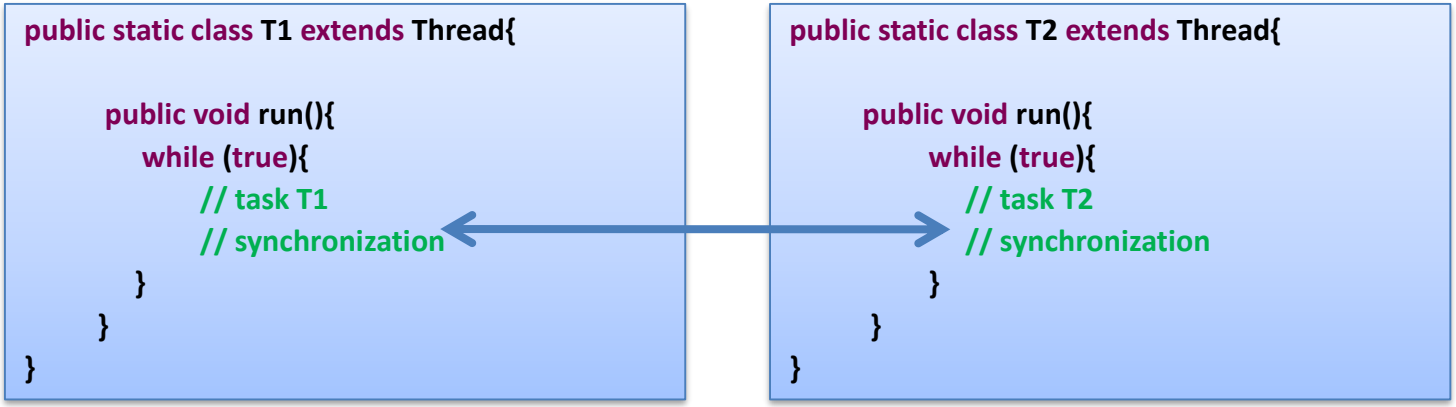
- Equivalence

boolean execP2 = false;

Action	Semaphores	Monitors
P1 suspends	s.acquire()	while (!execP2) c.await();
P1 wakes up	s.release()	execP2 = true; c.signal()
P1 wakes up	s.release()	execP2 = true; c.signal()
P1 suspends	s.acquire()	while (!execP2) c.await();

Implementing handshaking with Conditions

- Let's say two threads T1 and T2 executes indefinitely two tasks asynchronously. But, we want their executions to be synchronized (they meet) at a particular point in their respective codes



Implementing handshaking with Conditions

- The handshaking is translated into two synchronization conditions:
  - C1: T1 cannot continue until T2 finishes its task 2
  - C2: T2 cannot continue until T1 finishes its task 1

```
public static class T1 extends Thread{
        private Random r = new Random();
        private Synchro sync = new Synchro();
        public T1(Synchro sync){
           this. sync = sync;
        }
        public void run(){
             while (true){
             try {
                   Thread.sleep(r.nextInt(500));
                   System.out.println("End T1");
                   sync.arrivesT1();
                   sync.waitsForT2();
              } catch (InterruptedException e) {}
          }
      }
}
```

```
public static class T2 extends Thread{
        private Random r = new Random();
        private Synchro sync = new Synchro();
        public T2(Synchro sync){
           this.sync = sync;
        }
        public void run(){
             while (true){
             try {
                   Thread.sleep(r.nextInt(500));
                   System.out.println("End T2");
                   sync.arrivesT2();
                   sync.waitsForT1();
              } catch (InterruptedException e) {}
          }
      }
}
```

Implementing handshaking with Conditions

C1: If T1 arrives earlier then it must wait for T2 to meet with it

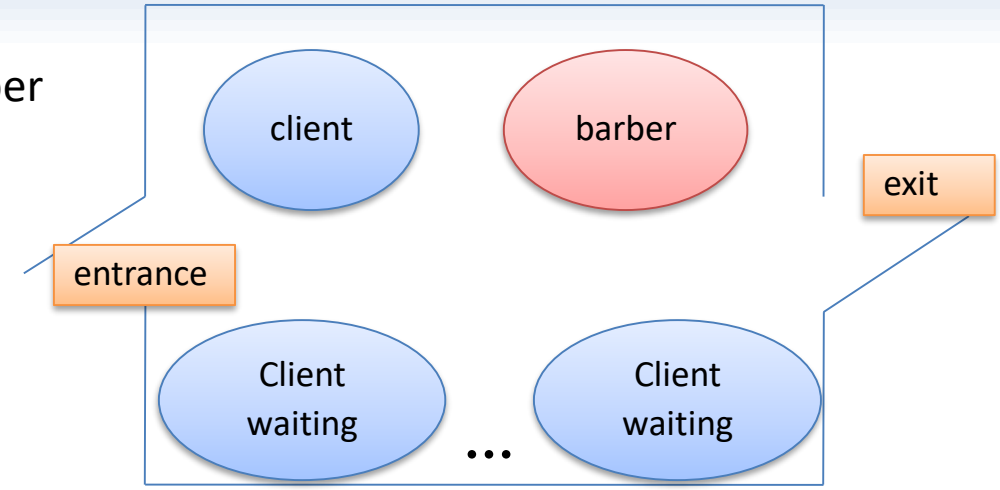
C2: If T2 arrives earlier then it must wait for T1 to meet with it

```
public static class Synchro{

    private boolean end1 = false, end2 = false;

    public synchronized void arrivesT1(){
        end1 = true;
        notify();
    }
    public synchronized void arrivesT2(){
        end2 = true;
        notify();
    }
   …
}
```

```
…
 public synchronized void waitsForT1()
                throws InterruptedException{
      while (!end1) wait();
      end1 = false;
 }
  public synchronized void waitsForT2()
                throws InterruptedException{
      while (!end2) wait();
      end2 = false;
  }
  …
```

Sleeping barber problem

- Let's say a city has a Barber's shop with two doors, a chair for the barber and many chairs for clients waiting for his services. Clients come in by one door and go out by the other.
- The barber's life is miserable: he is always shaving clients. The only incentive he has is that, when the shop is empty, he can sleep in his chair to have nice dreams (he loves drugs, sex and rock and roll).
- When a client comes in, he wakes up the barber (if sleeping), he sits on the barber's chair and receive the barber's services (barber does not take any reprisals)
- If a client comes in when the barber is occupied, then he waits sat in a client chair



- When the barber finishes to give services to a client, the barber open the exit door, the client go out and, this last, close the door
- If there is any client waiting, the barber wakes up him to work. If not, the barber returns to sleep

Sleeping barber problem

- To sum up the system:
- *barber* available → new *client* sits on a chair → *barber* shaves the client → *barber* opens the exit door → *client* go out and closes the door → *barber* available
- Thus, we have four synchronization conditions because there are two handshakes:

Sleeping barber problem

- To sum up the system:
- *barber* available → new *client* sits on a chair → *barber* shaves the client → *barber* opens the exit door → *client* go out and closes the door → *barber* available
- Thus, we have four synchronization conditions:
  - C1: A new client does not sit on the barber's chair until the barber is available (boolean **bavail**; condition **cbavail**)

Sleeping barber problem

- To sum up the system:
- *barber* available → new *client* sits on a chair → *barber* shaves the client → *barber* opens the exit door → *client* go out and closes the door → *barber* available
- Thus, we have four synchronization conditions:
  - C1: A new client does not sit on the barber's chair until the barber is available (boolean **bavail**; condition **cavail**)
  - C2: The barber does not shave the client until he is sat on the barber's chair (boolean **ctaken**; condition **cctaken**)

First handshake

Sleeping barber problem

- To sum up the system:
- *barber* available → new *client* sits on a chair → *barber* shaves the client → *barber* opens the exit door → *client* go out and closes the door → *barber* available
- Thus, we have four synchronization conditions:
  - C1: A new client does not sit on the barber's chair until the barber is available (boolean **bavail**; condition **cavail**)
  - C2: The barber does not shave the client until he is sat on the barber's chair (boolean **ctaken**; condition **cctaken**)
  - C3: The client does not go out until the barber opens him the exit door (boolean **dopen**; condition **cdopen**)

Sleeping barber problem

- To sum up the system:
- *barber* available → new *client* sits on a chair → *barber* shaves the client → *barber* opens the exit door → *client* go out and closes the door → *barber* available
- Thus, we have four synchronization conditions:
  - C1: A new client does not sit on the barber's chair until the barber is available (boolean **bavail**; condition **cavail**)
  - C2: The barber does not shave the client until he is sat on the barber's chair (boolean **ctaken**; condition **cctaken**)
  - C3: The client does not go out until the barber opens him the exit door (boolean **dopen**; condition **cdopen**)
  - C4: The barber is not available again until the client closes the exit door (condition **ccycle**)

Second handshake

Sleeping barber problem

```
public static class Barber extends Thread{
        private Random r = new Random();
        private BarberShop shop;
        public Barber(BarberShop shop){
           this.shop = shop;
        }
       public void run(){
            while (true){
               try {
                   shop.next ();
                   Thread.sleep(r.nextInt(500));
                  //barber is shaving
                   shop.endShave ();
                } catch (InterruptedException e) {}
            }
        }
}
```

```
public static class Client extends Thread{
        private Random r = new Random();
        private BarberShop shop;
        private int id;
        public Client(BarberShop shop, int id){
                this.shop = shop;
                this.id = id;
         }
        public void run(){
             try {
                shop.shave(id);
             } catch (InterruptedException e) {}
         }
}
```

Sleeping barber problem

```
public static class BarberShop{
    private Lock l = new ReentrantLock(true);
    private boolean bAvail = false, cTaken = false,
                    dOpen = false;
    private Condition cbAvail = l.newCondition(),
                      ccTaken = l.newCondition(),
                      cdOpen = l.newCondition(),
                      cCycle = l.newCondition();
public void shave(int id) throws InterruptedException{
      try{
        l.lock();
        while (!bAvail)                          C1
            cbAvail.await();
        bAvail = false;

        cTaken = true;                           C2
        ccTaken.signal();

        while (!dOpen)                           C3
            cdOpen.await();
        dOpen = false;                           C4
        cCycle.signal();
      } finally {
        l.unlock();
    }
  } ....
}
```

```
public void next() throws InterruptedException{
    try{
      l.lock();
C1    bAvail = true;
      cbAvail.signal();

C2    while (!cTaken)
          ccTaken.await();
      cTaken = false;
    }finally{
      l.unlock();
    }
}
public void endShave() throws InterruptedException{
    try{
      l.lock();
C3    dOpen = true;
      cdOpen.signal();

C4    while (dOpen)
          cCycle.await();

     }finally{
        l.unlock();
      }
}
```

Interruptions

```
public class Thread extends Object implements Runnable {
  ...
  public void interrupt();
      // Sends an interruption to a thread
      // The thread receiving the interruption contains a boolean flag
      // which is set immediately

  public boolean isInterrupted();
    // Returns true if the thread has been interrupted
    // The flag for of interruptions is not modified

  public static boolean interrupted();
    // Returns true if currentThread() has been interrupted,
    // BUT resets the flag for of interruptions
  ...
}
```

Interruptions

- Every thread has associated an internal boolean variable, called interruption flag, which can be used to stop its execution. This flag is updated when any of the methods interrupt(), isInterrupted() or interrupted() is executed

When a thread **A** interrupts another **B**:

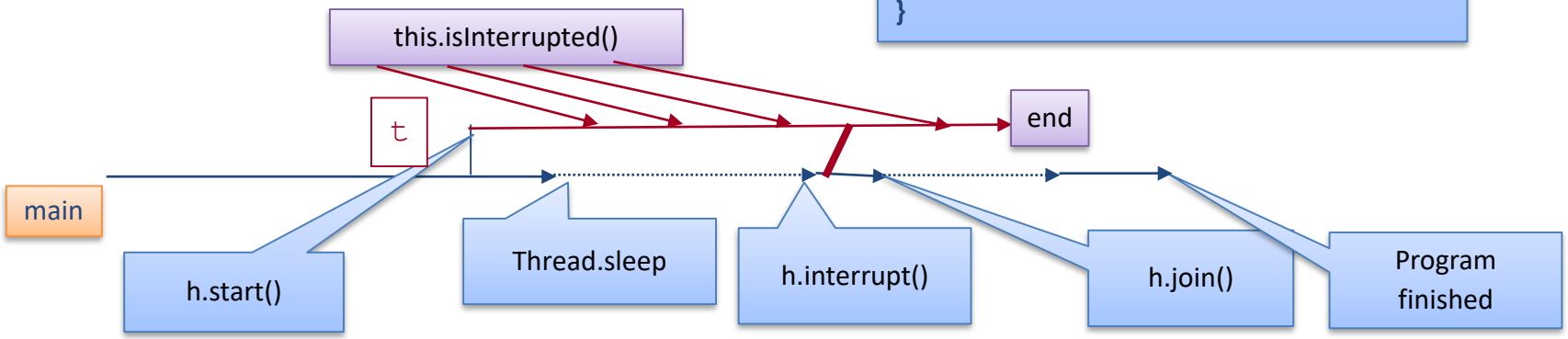
- If the thread **B** is blocked (in a wait, await, sleep, join, ...) it goes to the state "executable" and throws the controlled exception InterruptedException
- If the thread **B** is in execution,
  - The interruption flag is set to true to sign that an interruption request has been received, but the thread continues its execution (as if no interruption had been received)
  - If, afterwards, the thread try to block (in wait, await, sleep, join ,..) it goes immediately to the state "executable", throws the controlled exception InterruptedException **and** the interruption flag is reset, so it becomes false
- Hence, if the thread wants to finish its execution after being interrupted,
  - It must use the exception InterruptedException, to detect the interruption, and
  - Must check at intervals its interruption flag by means of the methods isInterrupted or interrupted

Interruptions: Example

- The class Task allows creating very simple threads: they display an infinite list of numbers. This task runs forever unless the task is interrupted
- The class Principal creates an object of type Task and starts it. After a few seconds, such Task object is interrupted. Finally, Principal waits until Tasks really finishes ad displays the message "Program finished".

```
public class Task extends Thread{
  private int i = 0;
  public void run(){
     while (!isInterrupted()){
        System.out.println(i);
         i++;
      }
   }
}
```

```
public class Driver {
   public static void main(String[] args) {
      Task t = new Task();
      t.start();
      try{
         Thread.sleep(1000);
         t.interrupt();
         t.join();
      } catch (InterruptedException ie) {…}
      System.out.println("Program finished");
   }
}
```



Interruptions: Producer/Consumer problem

```
import java.util.concurrent.*;

import java.util.*;
public class Producer extends Thread {

private BlockingQueue<Integer> buffer;

private Random r = new Random();

public Producer (BlockingQueue<Integer> buffer){
    this.buffer = buffer;
}

public void run(){
    boolean end = false;
    int count = 0;
    while (!isInterrupted() && !end)
    try{
        Thread.sleep(50);
        buffer.put(r.nextInt(25));
        count++;
    } catch (InterruptedException ie){end = true;}
    System.out.println("I have generated "+count+" items");
}
}
```

- We use the interface BlockingQueue in the package java.util.concurrent
- This class implements a bounded buffer (like the one we have developed)
- The operations to work with the buffer are put and take
- Both operations suspend the thread which executes them if the buffer is not ready. The exception InterruptedException is thrown if the thread is interrupted while blocked in put or take.
- The buffer is passed as a parameter in the constructors of Producer and Consumer

Interruptions: Producer/Consumer problem

```
import java.util.concurrent.*;

import java.util.*;
public class Producer extends Thread{

private BlockingQueue<Integer> buffer;

private Random r = new Random();

public Producer (BlockingQueue<Integer> buffer){
    this.buffer = buffer;
}


public void run(){
    boolean end = false;
    int count = 0;
    while (!isInterrupted() && !end)
    try{
        Thread.sleep(50);
        buffer.put(r.nextInt(25));
        count++;
    } catch (InterruptedException ie){end = true;}
    System.out.println("I have generated "+count+" items");
}
}
```

- The producer inserts random numbers into the buffer until another thread interrupts its execution
- If the thread is interrupted and is not blocked in the method put, then the interruption flag remains being true and, thus, the method isInterrupted returns true and the thread finishes its execution
- However, if the thread is blocked in put (or it is going to execute it) the exception InterruptedException is thrown and the interruption flag is reset to false. Thus, a call to isInterrupted returns false, and we need the boolean variable end to finishes the current iteration of the thread

Interruptions: Producer/Consumer problem

```
public class Consumer extends Thread {

private BlockingQueue<Integer> buffer;


public Consumer (BlockingQueue<Integer> buffer){
      this.buffer = buffer;
}


public void run(){

    boolean end = false;
    int count = 0;
    while (!isInterrupted() && !end) || buffer.size() > 0){
    try{
        Thread.sleep(50);
        System.out.println(isInterrupted());
        nt i = buffer.take();
        count++;
        System.out.println(i);
     } catch (InterruptedException ie){end = true;}
   System.out.println("I have consumed "+count+" items");
}
 }
```

- To end the consumer thread is slightly more complex
- It must finishes when it isinterrupted and, in addition, the buffer is empty (so there is no data to be consumed)

Interruptions: Producer/Consumer problem

```
public static void main(String[] args){
    BlockingQueue<Integer> buffer = new ArrayBlockingQueue<Integer>(5);

      Producer p = new Producer(buffer);
      Consumidor c = new Consumer(buffer);
      p.start();
      c.start();
      try{
        Thread.sleep(1000);
        p.interrupt();
        p.join();
        c.interrupt();
        c.join();
      }catch (InterruptedException ie){}

      System.out.println("Program finished");
 }
}
```

- This is an example of how to initialize the system
- ArrayBlockingQueue is the buffer. It takes as a parameter the size of the buffer (look up the documentation for further information about other parameters)
- The main program creates the threads and, after a time, interrupts them in the order: first the producer and then the consumer

References

- Concurrent Programming
  Alan Burns, Geoff Davies, Ed. Addison Wesley
- Concurrent and Real Time Programming in Java
  Andy Wellings, Ed. Willey
- Foundations of Multithreaded, Parallel and Distributed Programming
  Andrews, G. R., Addison-Wesley (2000)