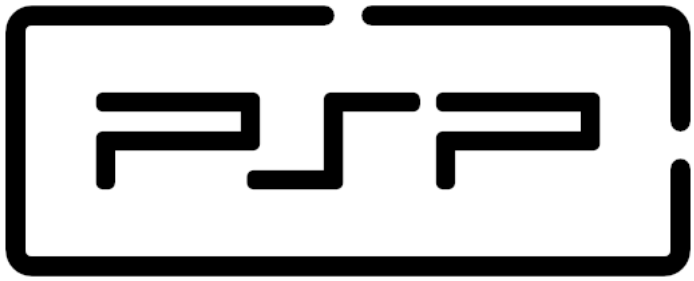


Process and Service Programming

3.2 Threads synchronization and communication

3.2 Threads synchronization and communication

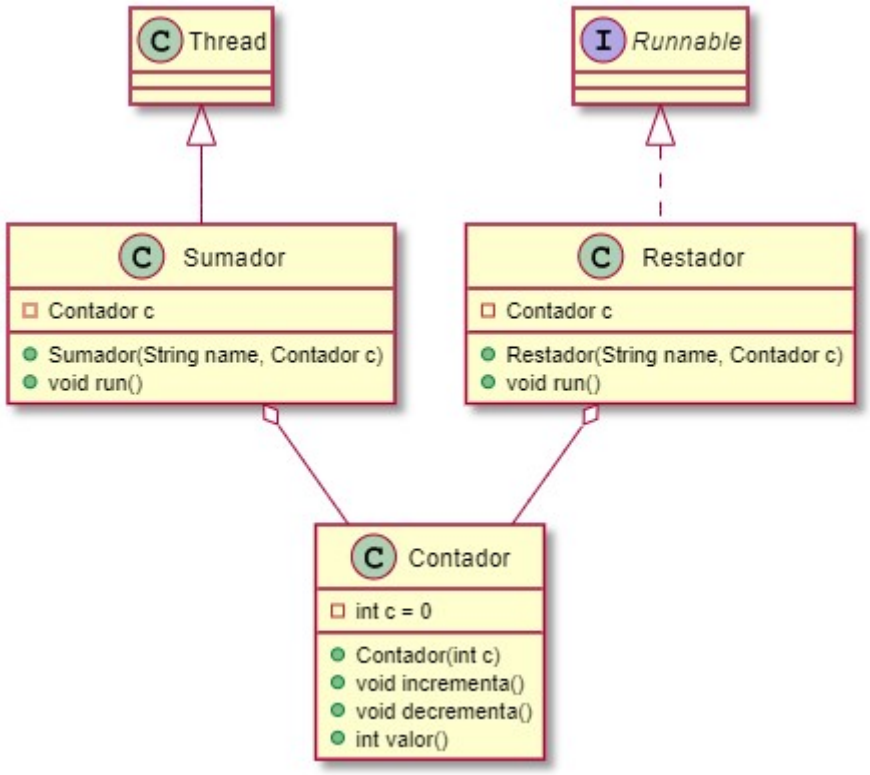
Multithread Vocabulary

- **Race condition**: A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.
- **Deadlock**: Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order.
- **Critical section**: A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section. It needs to be executed without outside interference - i.e. without another thread potentially affecting/being affected by "intermediate" states within the section.
- **Thread-safe**: A class (or chunk of code) is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and **with no additional synchronization or other coordination on the part of the calling code**.

3.2.1. Shared memory

Usually threads need to communicate with each other. The most common way of communication is sharing a common object.

Let's code an example where two threads share the same Contador instance.



To test the shared object, there must be just another class - containing the main method - to create the shared object (with init value of 100) and to launch the Sumador and Restador threads. In Sumador class we call the Contador.incrementa method in order to add 1 the Contador c property, and similarly Restador calls the decrementa method to substract 1 on Contador c property. Each thread will repeat the same action 300 times, waiting a random time between 50ms and 150ms. It is very relevant to use the same Contador object as parameter for Sumador and Restador, to make sure they are sharing the same Contador instance.

Expected behaviour

Write the four classes attending to the Class diagram. Make sure to have Sumador extending Thread and Restador implementing Runnable to test differences in how a thread is obtained from each approach.

What should happen after running the code?

Check what it really happens. Try to run the program many times.

if we check the above problem, we'll find that we are trying to run this code in parallel, from different threads, on the same object instance (shared instance):

```java
public void incrementa() {
  c++;
}
public void decrementa() {
  c--;
}
```

If we apply Bernstein conditions, we well get that none of the three conditions are met, so this code cannot be run concurrently, at least not without having concurrency problems.

So, we have to set a special configuration in our code in order to avoid this code to be run simultaneously.

3.2.2. Synchronization

As we have previously seen, threads communicate primarily by sharing access to objects and their properties. This form of communication is extremely efficient but makes two kinds of errors possible:

- thread interference
- memory consistency errors.

The tool needed to prevent these errors is synchronization.

When one thread is able to observe the effects of other threads and may be able to detect that variable accesses become visible to other threads in a different order than executed or specified in the program, we talk about reorderings, usually happening with incorrectly synchronized multithread programs. Most of the time, one thread doesn't care what the other is doing. But when it does, that's what synchronization is for.

Monitors and locks

To synchronize threads, Java uses `monitors`, which are a high-level mechanism for allowing only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of locks; **there is a lock associated with each object**.

Synchronization has several aspects. The most well-understood is `mutual exclusion` — **only one thread can hold a monitor at once**, so synchronizing on a monitor means that once one thread enters a synchronized block protected by a monitor, no other thread can enter a block protected by that monitor until the first thread exits the synchronized block.

But there is more to synchronization than mutual exclusion. Synchronization ensures that memory writes by a thread before or during a synchronized block are made visible in a predictable manner to other threads that synchronize on the same monitor.

> **Volatile-like behaviour**
>
> After we exit a synchronized block, we release the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads. Before we can enter a synchronized block, we acquire the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from the main memory.

Critical sections

Synchronized blocks in Java are marked with the `synchronized` keyword. A synchronized block in Java is synchronized with some objects. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

The synchronized keyword can be used to mark four different types of blocks:

- Instance methods
- Static methods
- Code blocks inside instance methods
- Code blocks inside static methods

Here is a synchronized instance method:

```java
public class Counter {
 private int count = 0;
 public synchronized void add(int value){
   this.count += value;
 }
}
```

Notice the use of the synchronized keyword in the add() method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance.

Only one thread per instance can execute inside a synchronized instance method. If more than one instance exists, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

This is true across all synchronized instance methods for the same object (instance). Thus, in the following example, only one thread can execute inside either of the two synchronized methods. One thread in total per instance:

```java
public class Counter {
 private int count = 0;
 public synchronized void add(int value){
   this.count += value;
 }
 public synchronized void sub(int value){
   this.count -= value;
 }
}
```

Synchronized with static methods

Synchronized static methods are synchronized on the **class object** of the class the synchronized static method belongs to. Since **only one class object exists in the Java VM per class**, only one thread can execute inside a static synchronized method in the same class.

You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods make this possible. Here is a synchronized block of Java code inside an unsynchronized Java method:

```java
public void add(int value){
  synchronized(this){
    this.count += value;
  }
 }
```

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.

Notice how the Java synchronized block construct takes an object in parentheses. In the example “this” is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor

object. The code is said to be synchronized on the monitor object. **A synchronized instance method uses the object it belongs to as a monitor object**.

Only one thread can execute inside a Java code block synchronized on the same monitor object.

The following two examples are both synchronized on the instance they are called on. **They are therefore equivalent with respect to synchronization**:

```java
public class MyClass {
  public synchronized void log1(String msg1, String msg2){
    log.writeln(msg1);
    log.writeln(msg2);
  }

  public void log2(String msg1, String msg2){
    synchronized(this){
     log.writeln(msg1);
     log.writeln(msg2);
    }
  }
}
```

Thus only a single thread can execute inside either of the two synchronized blocks in this example.

> **What Objects to Synchronize On**
>
> The synchronized block must be synchronized on some object. You can actually choose any object to synchronize on, but **it is recommended that you do not synchronize on String objects, or any primitive type wrapper objects** (Integer, Double, Boolean, ...).
>
> To be on the safe side, synchronize on this - or on a new Object() . Those are not cached or reused internally by the Java compiler, Java VM, or Java libraries.

Synchronized and Data Visibility

Without the use of the synchronized keyword (or the Java `volatile` keyword) there is no guarantee that when one thread changes the value of a variable shared with other threads (e.g. via an object all threads have access to), that the other threads can see the changed value. There are no guarantees about when a variable kept in a CPU register by one thread is "committed" to main memory, and there is no guarantee about when other threads "refresh" a variable kept in a CPU register from main memory.

The synchronized keyword changes that.

- When a thread enters a synchronized block it will refresh the values of all variables visible to the thread.
- When a thread exits a synchronized block all changes to variables visible to the thread will be committed to the main memory.

This is similar to how the volatile keyword works.

3.2.3 Inter-Thread synchronization

We can avoid several threads run the same code at the same time by using the `synchronized` keyword in order to get `mutual exclusion` in the form of `critical sections`. Sometimes it can be enough, but others we need the threads to keep certain

order in their execution, probably related to other threads previous actions or results.

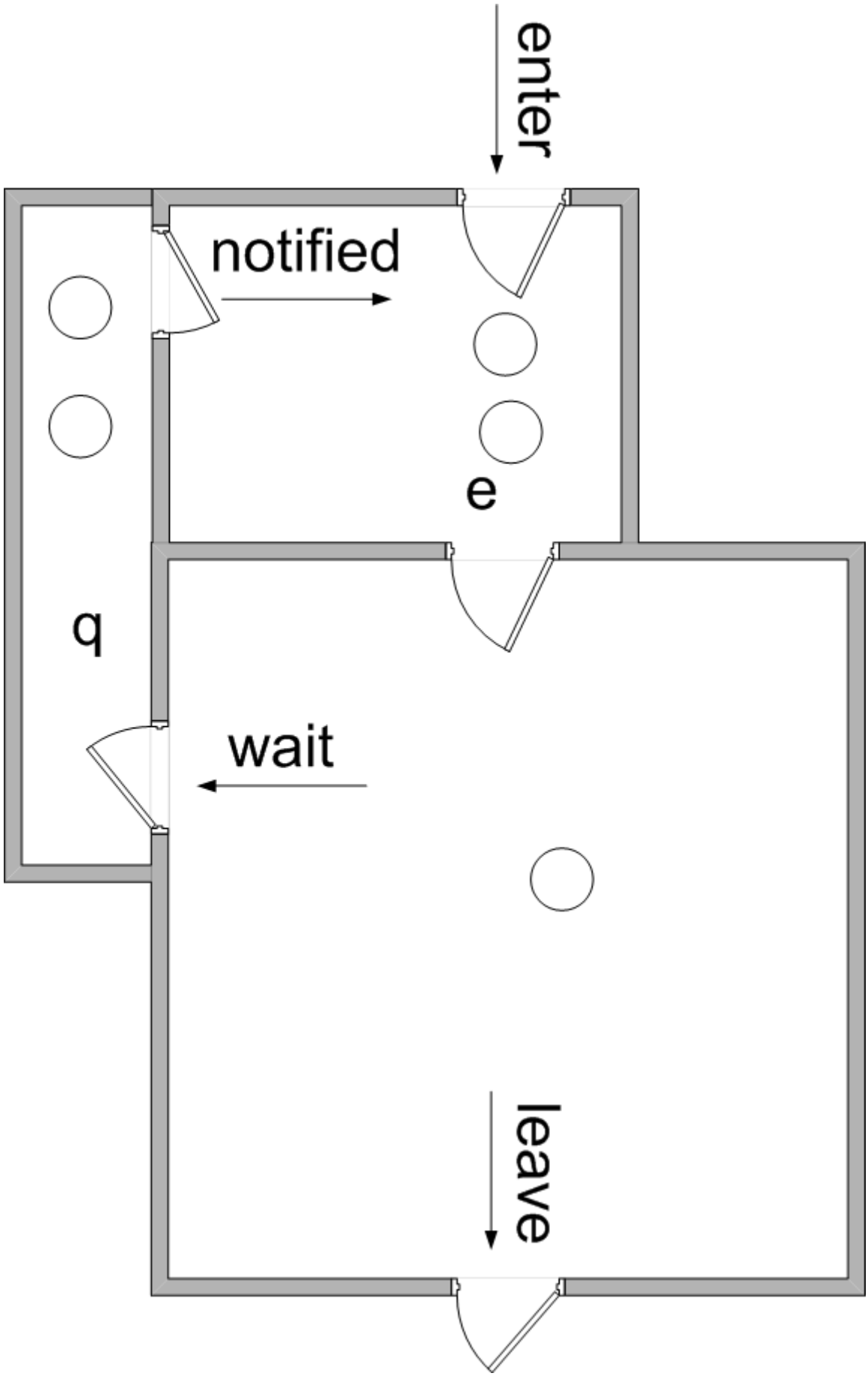
To do so, we need to use three new methods from the object class, directly related to synchronized.

- **wait()**: When you call wait method on the object then it tell threads to `give up the lock` and go to sleep state unless and until some other thread enters in same monitor and calls notify or notifyAll methods on it.
- **notify()**: When you call notify method on the object, it `wakes one of thread waiting for that object`. So if multiple threads are waiting for an object, it will wake of one of them. Now you must be wondering which one it will wake up. It actually depends on OS implementation.
- **notifyAll()**: notifyAll will `wake up all threads waiting on that object` unlike notify which wakes up only one of them. Which one will wake up first depends on thread priority and OS implementation.

wait , notify and notifyAll method are used to allow threads to communicate to each other via accessing common object. This common object can be considered a medium for `inter thread communication` via these methods. **These methods need to be called from synchronized context**,otherwise it will throw java.lang.IllegalMonitorStateException.

When **wait()** method is called, the thread is running inside the synchronized block so it will own the monitor (lock) for the object. That monitor is released by the thread and the thread is locked into another **queue (from the lock object) of threads waiting to be notified**, different than the queue of threads waiting for the object monitor (lock).

When a thread is unlocked because another thread has called **notify()/notifyAll()** on the same object, the thread goes back to the point where the wait was made, so the thread is still into a synchronized block. To keep on running, the thread goes back to the **queue of threads waiting for the object monitor (lock)** and it has to wait until it gets the lock to run the sentences after the wait().



Let's learn how these methods work by looking at the following example

```java
// It is the common java class on which thread will act and call wait and notify method.
public class Book {
```

```
 String title;
 boolean isCompleted;

 public Book(String title) {
 super();
 this.title = title;
 }

 public String getTitle() {
 return title;
 }
 public void setTitle(String title) {
 this.title = title;
 }
 public boolean isCompleted() {
 return isCompleted;
 }
 public void setCompleted(boolean isCompleted) {
 this.isCompleted = isCompleted;
 }
}
```

java

```
// It will first take a lock on book object
// Then, the thread will wait until other thread call notify method, then after it will complete its processing.
// So in this example, it will wait for BookWriter to complete the book.
public class BookReader implements Runnable{
 Book book;

 public BookReader(Book book) {
 super();
 this.book = book;
 }

 @Override
 public void run() {
 synchronized (book) {
  System.out.println(Thread.currentThread().getName()+" is waiting for the book to be completed: "+book.getTitle());
  try {
  book.wait();
  } catch (InterruptedException e) {
  e.printStackTrace();
  }
  System.out.println(Thread.currentThread().getName()+": Book has been completed now!! you can read it");
 }
 }
}
```

java

```
// This class will notify thread(in case of notify) which is waiting on book object.
// It will not give away lock as soon as notify is called, it first complete its synchronized block.
// So in this example, BookWriter will complete the book and notify it to BookReaders.
public class BookWriter implements Runnable{

 Book book;

 public BookWriter(Book book) {
 super();
 this.book = book;

```

```
  }

  @Override
  public void run() {
  synchronized (book) {
   System.out.println("Author is Starting book : " +book.getTitle() );
   try {
   Thread.sleep(1000);
   } catch (InterruptedException e) {
   e.printStackTrace();
   }
   book.setCompleted(true);
   System.out.println("Book has been completed now");

   book.notify();
   System.out.println("notify one reader");
  }
  }
 }
```

java

```
// This is our main class which will create object of above classes and run it.
public class U3S5_Books {

 public static void main(String args[])
 {
 // Book object on which wait and notify method will be called
 Book book=new Book("The Alchemist");
 BookReader johnReader=new BookReader(book);
 BookReader arpitReader=new BookReader(book);

 // BookReader threads which will wait for completion of book
 Thread johnThread=new Thread(johnReader,"John");
 Thread arpitThread=new Thread(arpitReader,"Arpit");

 arpitThread.start();
 johnThread.start();

 // To ensure both readers started waiting for the book
 try {
  Thread.sleep(3000);
 } catch (InterruptedException e) {

  e.printStackTrace();
 }

 // BookWriter thread which will notify once book get completed
 BookWriter bookWriter=new BookWriter(book);
 Thread bookWriterThread=new Thread(bookWriter);
 bookWriterThread.start();
 }
}
```

There must be a notify call for every wait to ensure we have no deadlocks in our app.

Does order matter?

If we run the previous code, we have to ask ourselves:

- a) How many wait() are done? And how many notify()?

Just by looking at the code, we can see that there are 2 wait() and 1 notify(). Something doesn't fit.

One of the readers is not notified, so a thread is waiting in a wait(). As that thread doesn't finish, the process doesn't either. We have to remember that a process doesn't finish until the last of its threads does. In Netbeans, this means the program doesn't end and we have to stop it.

Solution: In this case we have two alternatives. The first one is to use notifyAll() instead of notify(). This way, both BookReaders are activated and they wait to take the monitor lock. One will do it first and the other one later, but both will end up reading the book. The other option is, following with notify(), that each reader, when finishing reading the book, notifies other possible readers waiting so that one wakes up and reads the book.

- b) What will happen to the above code if we change the order for the Readers and Writer in the main method? That is, first we make sure the Book is finished and then we call the Readers. What if we start all threads at the same time and we don't know in which order they will run? What if we start the Readers first and then the Writer?

If the BookWriter is launched first, it finishes the book and notifies... nobody, because the BookReaders are not waiting yet. Then the BookReaders arrive and both get stuck, as no other thread will notify them.

Solution: Threads are now blocking indiscriminately, but they should only block if the book they want to read is not finished. Therefore, we have to control the blocking of the BookReader with a condition. As we have commented, the conditions must be in the shared object, in this case book, shared by BookWriter and the two BookReader. The condition that allows us to discriminate if a BookReader can continue or not is the isCompleted property that we consult through the book.isCompleted() method.

```java
try {
  if (!book.isCompleted())
    book.wait();
} catch (InterruptedException e) {
```

With these two changes, the application should work with any number of BookReaders and BookWriters, regardless of the order or the amount.

¿notify() o notifyAll()?

All will depend on the system we are programming, but as a general rule, if we want only one thread to continue after modifying the system state, we will call notify().

Otherwise, notifyAll() should be used. If everything is well programmed, the thread will check if it can continue and, if not, it will wait() again and continue waiting, so it is not a problem that more than one thread is activated.

The use of notify() poses a greater risk of indefinite blocking of threads waiting for notifications that will never arrive, this blocking being different from a deadlock. We must be very careful with the programming of synchronization mechanisms.

It should also be noted that there should be at least one notify() call for each wait() that has been made, although that does not ensure that some thread will not be blocked.

Modifify previous Sumador-Restador

> Make the necessary changes in the classes of the U3S3_SharedMemory project (save it as U3S3_SharedMemory_v2) so that:
>
> - The first thread that performs an operation on the counter is a Sumador
> - After a Sumador, a Restador is always executed and after a Restador, a Sumador is always executed, making a sequence Sumador-Restador-Sumador-Restador-...