**Development of Multiplatform Applications**
**2nd Course**

# Programming of Services and Processes

## UNIT 4:
## Multithreading Programming Synchronization

***Professor***: *Empar Carrasquer*

**1. OBJECTIVES 3**

**2. SYNCHRONIZATION 4**

**2.1 EXAMPLE OF RESOURCE CONFLICT SCENARIO: RACE CONDITIONS AND CRITICAL REGION 4**
**2.2 SYNCHRONIZED INSTANCE METHODS 6**
**2.3 SYNCHRONIZED STATEMENTS 8**
**2.4 THREAD COOPERATION TO AVOID CONTENTION AND STARVATION 8**

**3. HIGH LEVEL CONCURRENCY OBJECTS 10**

**3.1 EXECUTORS 10**
3.1.1 EXECUTORS INTERFACES 11
3.1.2 THREAD POOLS 11
3.1.3 CREATING AN EXECUTORSERVICE: 12
3.1.4 WRAP UP 15
**3.2 CONCURRENT COLLECTIONS 15**
**3.3 ATOMIC VARIABLES 16**
**3.4 CONCURRENT RANDOM NUMBERS 16**
**3.5 SEMAPHORES 17**
**3.6 LOCK OBJECTS 18**

# 1. OBJECTIVES

- ✓ Learn how to communicate and synchronize threads.
- ✓ Create user-defined classes with thread capability.
- ✓ Understand the concurrent issues with thread programming
- ✓ Know the high-level concurrency objects needed in massive concurrent applications that fully exploit multiprocessor and multi-core systems.
- ✓ Write multithreaded applications.

# 2. SYNCHRONIZATION

Threads communicate primarily by **sharing** access to fields and the objects fields refer to. Sharing resources is a very powerful form of communication but sometimes very hard to control as ***Thread Interference***, ***Race Conditions*** and ***Memory Consistency*** errors arise.

http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

The only way to avoid these errors is ***synchronization***. However, synchronizing of threads introduces problems of ***thread contention*** (*occurs when two or more threads try to access the same resource simultaneously in such a way that at least one of the contending threads runs more slowly than it would if the other thread(s) were not running*) like ***starvation***, ***deadlock*** or ***livelock*****.** For more information, see this link:

http://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html.

We say that an object or class is **thread-safe** when its data is protected from uncontrolled concurrent access by more than one thread and one of them might write to it. If this is not done, it could result in *data corruption* and other unpredictable consequences.

Making an object thread-safe requires using *synchronization* to *coordinate* access to its data. The primary **mechanism** for synchronization in Java is the ***synchronized*** keyword, which provides a *monitor* with ***exclusive locking***.

> A ***monitor*** is a set of atomic methods that provide a simple mechanism to assure exclusive locking to a shared resource.

The term "***synchronization***" also includes the use of ***volatile variables***, explicit ***locks***, and ***atomic variables.***

The ***synchronized*** keyword can be applied to ***methods*** or to a ***region of code.***

## 2.1 Example of resource conflict scenario: race conditions and critical region

If multiple threads run inside a program, they might call the same method in a shared object. If they call this method at the same time without exclusive access rights, ***what happens****?* Well, threads could:

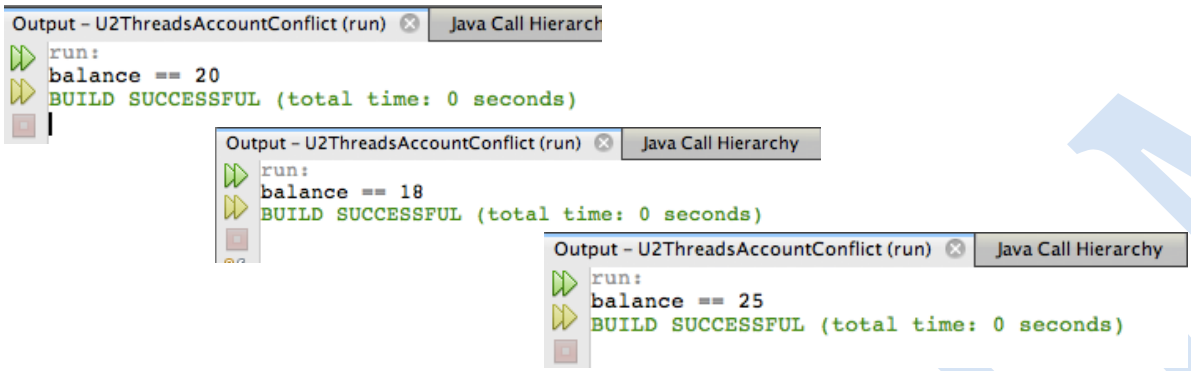
- Read different results from member fields.
- Overwrite field values with incorrect values.

To make sure that only one thread executes inside a method you must **lock** the object using the ***synchronized*** keyword.

**Example**: *Imagine an application that launches 100 threads, each of them adds a euro to a shared account. The account is initially empty. The code for the application could be like the following:*

```
public class U2ThreadsAccountConflict {
    private final int NUM_THREADS = 100;
    private Account account = new Account();     // a "not-very-safe" account
    private Thread thread[] = new Thread[NUM_THREADS];


    // nested classes
    /** Nested class for the threads: contains the run method */
    class AddAEuro implements Runnable {
        @Override
        public void run() {
            account.deposit(1);
        }
    }

    /** Nested class for the account: "the shared resource". */
    class Account {
        private int balance = 0;      // current balance, initially=0

        public int getBalance() {
            return balance;
        }

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

            try {
                Thread.sleep(1);     // sleep is deliberately added to make
                                     // the data corruption problem more evident
            }
            catch ( InterruptedException ex ) {
            }

            balance = newBalance;    // finally, update the balance

        }
    }
    // nested classes
    /** createThreads: creates and launches the threads */
    private void createThreads() {
        // Create and launch NUM_THREADS threads that deposit 1 euro.
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            thread[i] = new Thread(new AddAEuro(), "Thread-" + i);
            thread[i].start();
        }

        // main() must wait until all threads finish:
        // calls join() on all the threads to wait for them
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            try {
                thread[i].join();
            }
            catch ( InterruptedException ex ) {
                //ex.printStackTrace();
                System.out.println( ex );
            }
        }
    }

    // main: runs and prints the account balance
    // at the end, when every thread is terminated
    public static void main(String[] args) {
        U2ThreadsAccountConflict myAccount = new U2ThreadsAccountConflict();
        myAccount.createThreads();
        System.out.println( "balance == " + myAccount.account.getBalance());
    }
}
```

If we execute the program more than one time, then we get different values for the account balance:




What is happening is that all the 100 threads are modifying the account balance at the same time before one thread has finished its updating and hence, the final value is **different** for every execution of the program. This is known as a ***race condition*** and therefore, the class is not ***thread-safe.***

To avoid race conditions, the program code must assure that only one thread can be executing in a ***critical region*** i.e. the piece of code that modifies the shared resource.

In the previous code, the method ***deposit***() is the *critical region* as increase the value of the balance by amount. Therefore, only one thread should be allowed to execute in this method at a time.

## 2.2 Synchronized instance methods

Synchronizing an instance method means that a thread **must** get a ***lock*** on the instance object before beginning to execute it.

When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object get **blocked** (execution suspended) until the first thread is done with the object.

To synchronize a method, we will add the ***synchronized*** keyword to the method declaration. (*Constructors are synchronized methods by default as only one thread creates the object*)

In the previous example:

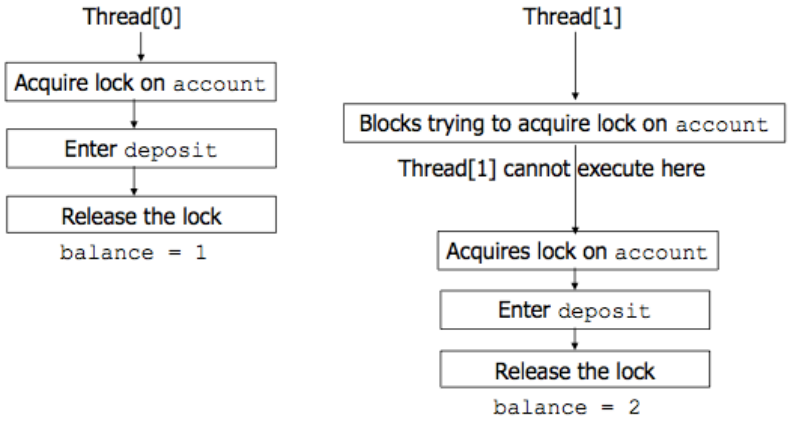
```
public synchronized void deposit(int amount) {
    int newBalance = balance + amount;

    try {
        Thread.sleep(1);     // sleep is deliberately added to make
                             // the data corruption problem more evident
    } catch (InterruptedException ex) {
    }

    balance = newBalance;    // finally, update the balance

}
```

After executing the program, the result value for the account balance is every time correct:

```
Output - U2ThreadsAccountConflict (run)   Java Call Hierarchy
run:
Balance == 100
BUILD SUCCESSFUL (total time: 2 seconds)
```

By using ***synchronized*** we can assure that meanwhile one thread executes the synchronized method, the synchronized code of this object is ***locked*** and the others must wait (*blocked*) until the first is finished.

Example: *thread[0] gets deposit the first; then thread[1] blocks; avoiding the corruption of the shared resource.*




**Exercise 7:**

Create a new Java class that launches 5 threads to count the number of vowels within a string variable. Every thread counts a different vowel and updates the shared integer variable *numberOfVowels*.

Make use of synchronized methods to avoid *race conditions* and *data corruption*.

Eventually, the main method must print the number of vowels found by the threads and must be always correct for every execution of the program for the same input string.

**Exercise 8:**

**8.1** Find out what happens when we use *synchronized* keyword in class methods (defined as *static*). Give and example in Java to demonstrate it.

http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

**8.2** Final fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed.

Program and example in Java to demonstrate that *final fields* are thread-safe and there are no race conditions.

## 2.3 Synchronized statements

Another way to create synchronized code is with ***synchronized statements***. *A synchronized block* locks only a **portion** of code. Therefore:

- The lock time might be shorter increasing concurrency.
- The critical region is smaller so performance improves.

Synchronized statements must specify the object that provides the intrinsic lock.

In our example would be:

```
// deposit using a synchronized block
public void deposit(int amount) {
    synchronized (this) {
        int newBalance = balance + amount;

        try {
            Thread.sleep(1); // sleep is deliberately added to make
                             // the data corruption problem more evident
        }
        catch (InterruptedException ex) {
        }
        balance = newBalance;    // finally, update the balance
    }
}
```

## 2.4 Thread cooperation to avoid contention and starvation

When many threads compete for CPU execution time a situation known as thread ***contention*** may occur, as a thread could not give other threads a chance to run. This might result in thread ***starvation.***

***Starvation*** describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.

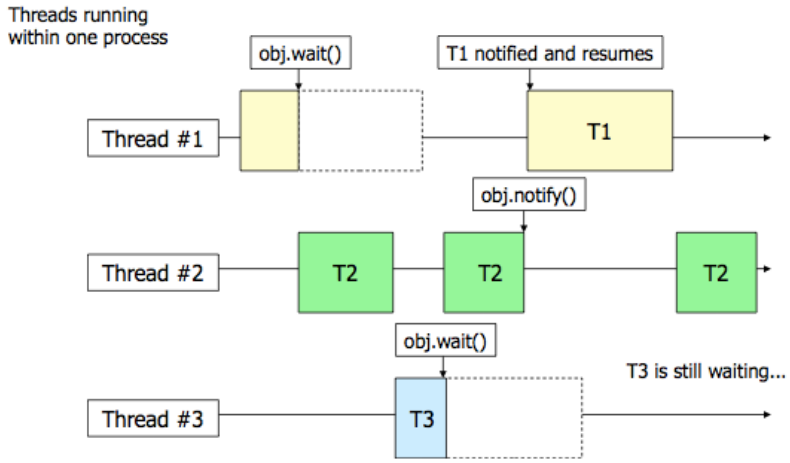
To avoid the problems above there are ways to make threads to ***coordinate*** and ***cooperate*** with each other.

There are 3 *Object* methods to support coordination:

- ***wait***() → makes a thread to get blocked until *notify* or *notifyAll* is called on the object (might be blocked forever if notify never happens). The thread is told to give up the monitor so it releases the lock on the shared object and waits.
- ***notify***() → wakes up **a thread** (usually the first) waiting on the object.
- ***notifyAll***() → wakes up **all threads** waiting on the object (scheduling decides which one gets notified the first, usually FIFO).

These 3 methods **must** be always called inside a **synchronized** *method* or *block* unless *IllegalMonitorStateException* is thrown.

*Wait/notify* allow threads to safely coordinate and share data.




## Exercise 9: PRACTICE - The Producer-Consumer problem

*The producer-consumer thread problem is a classic:*

- *The producer creates new elements and deposits them in a shared buffer.*
- *The consumer takes an element from the buffer, when available, then uses it.*

Write the following Java classes to give a solution to the producer-consumer problem:

- ***MyBuffer*** class→ has methods for storing (*put*) and retrieving an integer (*get*). It can hold only one integer at a time. (Use *synchronized* methods, *wait* and *notifyAll* for thread coordination)
- ***Producer*** class → the producer generates (*every random 0-99 milliseconds)* integers between 0 and 9, one by one, and stores it in *MyBuffer.* It must print a message to inform the integer produced: *"Producer " + this.numberProducer + " put: " + i (this.numberProducer → identifier given to producer thread, i → the number produced)*
- ***Consumer*** class → consumes the number in *MyBuffer* as quickly as it is available. It must print a message to inform the integer consumed: "Consumer " + this.numberConsumer + " got: " + value *(this.numberConsumer → identifier given to consumer thread, i → the number consumed)*.
- ***ProducerConsumerTest*** class → creates an instance of MyBuffer, starts producer and consumer threads.

**A) Run** and the **test** the program:

1. Start 1 producer and 1 consumer. It should go fine.
2. Start 2 producers and 2 consumers. Are there any problems? Notice that both producers generate numbers repeated (0-9), so modify your solution to avoid:
   - Sending items that consumers miss.
   - Consumer getting the same number many times. (*TIP: producers must produce unique (not repeated) integer numbers).*

***B) Modify*** *the program to store in the buffer MAX_SIZE items. Launch the program and check out that all producer and consumer threads are properly coordinated.*

# 3. HIGH LEVEL CONCURRENCY OBJECTS

So far, we learned to use the low-level thread-related APIs that have been part of the Java platform from the very beginning.

These low-level APIs are **adequate** for very **basic tasks**, but **higher-level** building blocks are needed for more **advanced tasks** like massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

- Threads take a lot of resources
- Require of a manual management
- It is not so simple to control the # executing tasks

In Java platform version 5.0, new high-level concurrency features were introduced. Most of these features are implemented in the new **java.util.concurrent** package. There are also new concurrent data structures in the **Java Collections** Framework.

https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html

Some of them are the following:

- **Executors** → define a high-level API for launching and managing threads, especially, for *large-scale applications*.
- **Concurrent collections** → make it easier to manage large collections of data and reduce the need for synchronization.
- **Atomic variables** → minimizes need for synchronization and help to avoid memory consistency errors.
- **Concurrent Random Numbers** → **ThreadLocalRandom** (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.
- **Lock objects** → help to simplify many concurrent applications.
- **Semaphores** → useful in different scenarios where you have to limit the amount of concurrent access to certain parts of your application.

## 3.1 Executors

Working with the Thread class can be very **tedious** and **error** prone, especially when developing large-scale applications.

The Concurrency API introduces the concept of an **ExecutorService** as a higher-level replacement for working with threads directly.

**Executors** are capable of running asynchronous tasks and typically manage a **pool of threads**, so we don't have to create new threads manually.

All threads of the internal pool will be **reused** for the all application upcoming tasks.

### 3.1.1 Executors interfaces

There are three executor interfaces defined in *java.util.concurrent* package:

- **Executor**: simple interface to launch new tasks.
- **ExecutorService**, a subinterface of Executor capable to manage the lifecycle, both of the individual tasks and of the executor itself.
- **ScheduledExecutorService**, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

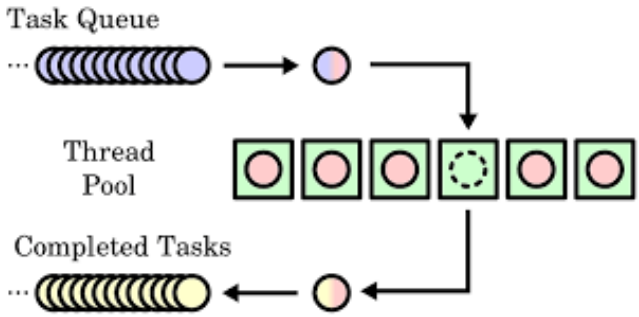
Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

### 3.1.2 Thread Pools

Most of the executor implementations use ***thread pools***, which consist of worker threads. Using worker threads minimizes the overhead due to thread creation.

In a **large-scale** application, allocating and deallocating many thread objects creates a significant memory management **overhead**.

One common type of thread pool is the **fixed thread pool**: it always keeps a fixed (specified) number of threads running. If a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread.




An internal **task queue** holds extra tasks whenever there are more active tasks than threads.

**Exercise 10:**
Consider a web server application handling HTTP requests. For every new upcoming request, the application creates a new thread.
Think about these questions:
- What happens if the application receives more requests than it can handle?
- Is there any advantage using a pool of threads instead?

A very simple way to create an instance of ExecutorService is to use the ***Executors* factory class methods** like:

- **newFixedThreadPool*(n)*** → Creates a fixed pool of threads.

```
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
```

- **newSingleThreadExecutor**() → Executes a single task at a time.

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
```

- **newScheduledThreadPool**() → Creates a thread pool that can schedule tasks to run after a given delay, or to execute periodically.

```
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

We'll find more Executors factory class methods at:
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html

Also, if needed, we can use the built-in implementations of ExecutorService:

- ***ThreadPoolExecutor***
- ***ScheduledThreadPoolExecutor***

### 3.1.3 Creating an ExecutorService:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class U2ExecutorsExample1 {

    public static void main(String[] args) {

        // create an executor service with a fixed pool of threads = availableProcessors
        ExecutorService executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        // create an anonymous Runnable object that is executed by one of the threads in the pool
        executorService.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println("This is an Asynchronous task");
            }
        });

        // terminate all threads running in the ExecutorService
        // unless, the JVM process keeps running
        executorService.shutdown();
    }

}
```

- **To create the fixed thread pool:**
  Call to ***Executors.newFixedThreadPool(n)*** method, where **n** is the number of worker threads.

- **To Execute tasks:**
  The ***execute(Runnable)*** method takes a *java.lang.Runnable* object, and executes it asynchronously.
- **To Stop the ExecutorService:**

**ExecutorService** keeps running if not **shutted down**. That means the JVM process will be running meanwhile there is an **active** ExecutorService as the active threads inside this ExecutorService prevents the JVM from shutting down.

To **terminate** the threads inside the ExecutorService use one of these methods:

- ***shutdown()*** method: The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished pending and current tasks, the ExecutorService shuts down.

- ***shutdownNow()*** method: **shuts down** the ExecutorService **immediately**. Stops all executing tasks right away (sends an *Interruption* and there are no guarantees if they execute until the end), and skips all submitted but non-processed tasks.

- **Testing if ExecutorService is done with all tasks:**
  After calling shut down, the ***isTerminated()*** method returns *true* if all tasks have completed.

- **Wait for ExecutorService termination:**
  After a shutdown request, the ***awaitTermination(timeout, timeUnit)*** method blocks until:
  - All tasks have completed execution, or
  - The timeout occurs, or
  - The current thread is interrupted.

---

## Exercise 11: Executing tasks in a pool of threads.

### FIRST PART

Create a new Java project and implement the following class:

```
public class PrintNumber implements Runnable {
    private String taskName;
    private int lastNumber;
    private int sleepTime;

    public PrintNumber(String taskName, int lastNumber, int sleepTime) {
        this.taskName = taskName;
        this.lastNumber = lastNumber;
        this.sleepTime = sleepTime;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName() + " " +  taskName ;
        for (int i = 1; i <= lastNumber; i++) {
            System.out.println(name + ": #"+ i);
            try {
                Thread.sleep(sleepTime);
            } catch (InterruptedException ex) {

            }
        }
        System.out.println("\t\t\t" + name + " IS TERMINATED");
    }
}
```

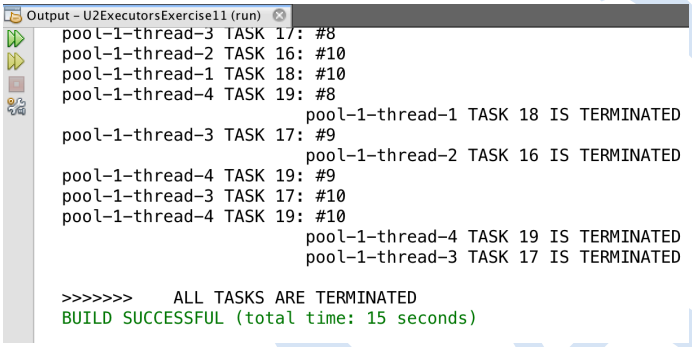
Add the following constants to the main program class:

```
private static final int NUM_THREADS = 5;
private static final int NUM_ASYNC_TASKS = 20;
private static final int LAST_NUMBER = 10;
private static final int MIN_SLEEP_TIME = 200;
private static final int MAX_SLEEP_TIME = 500;
```

Modify the main() program method to:

- Create an ExecutorService with fixed number of threads: NUM_THREADS
- Create and execute the number of required tasks NUM_ASYNC_TASKS in the pool.
    - Calculate a random sleep time for every task between MIN_SLEEP_TIME and MAX_SLEEP_TIME.
- Once all the tasks are executing in the pool, stop the ExecutorService.
- Then, wait until all tasks are done.
- Print out the following message: "ALL TASKS ARE TERMINATED". This message must be printed in the last place.

An example of output:

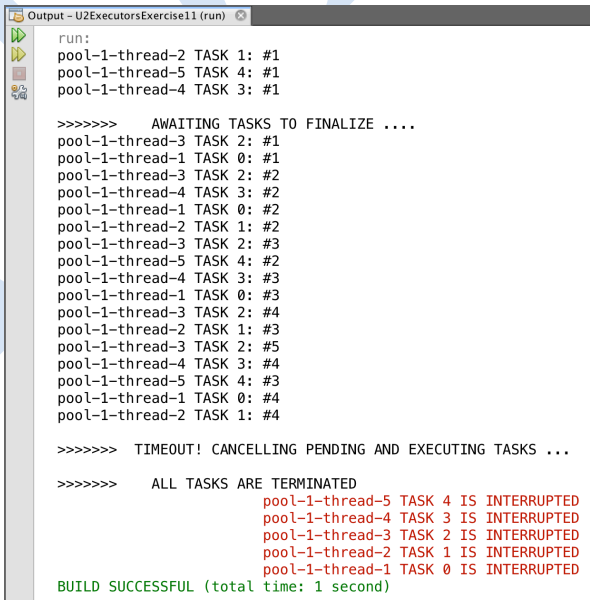



**SECOND PART**

Modify the previous program:

- After calling shutdown(), await at maximum 1 second for tasks termination (call *awaitTermination* method)
- If waiting time spires, cancel all pending and executing tasks:
    - Stop the Executor Service immediately (The current tasks executing in pool-threads must finalize immediately).
- Catch interruption while task is sleeping and print and error message: "TASK n IS INTERRUPTED" then return.

Example of output:



### 3.1.4 Wrap up

Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

The idea behind thread-pools is to have worker threads always existing, minimizing the overhead due to thread creation, especially in a large-scale application.

We might create independent groups of thread-pool based on purpose. For example, one thread-pool for background schedules, other for email, another for database updates, etc.

Thread-pools are **appropriate** when there's a stream of jobs to process, even though there could be some time when there are no jobs.

In a **client/server application**, with undetermined client requests, **creating a thread every time** there's a new request **is not the best solution** because:

- Thread creation takes time → client may notice a slight delay.
- Threads consume resources (memory,…) → Uncontrolled, the system can run out of resources.

→ It is much better to create a pool of threads when server applications starts-up so we can limit how many threads are running at any time.

And, **what should be the size of the thread-pool?** Depends on the application and other factors like:

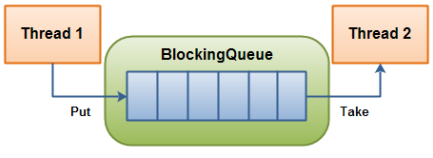
- Frequency of client requests: the more requests the more threads in the pool.
- A faster server response to the client requests means more threads in the pool.
- Of course, the system resources available will limit the number of threads in the thread-pool.

## 3.2 Concurrent collections

The *java.util.concurrent* package includes collections that help to avoid the *Memory Consistency Errors* (occur when different threads have inconsistent views of what should be the same data)

The **collection of interfaces** provided are the following:

- **BlockingQueue:** defines a **safe** FIFO data structure that *blocks* or *times out* when a adding to a full queue or retrieving from an empty queue. A BlockingQueue is typically used when threads produce objects, which another threads consume.




**Implementations**: ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue.

**Example**:

```
BlockingQueue queue = new ArrayBlockingQueue(1024);
```

- **ConcurrentMap**: It is a subinterface of *java.util.Map* that allows concurrent access what avoids the need to synchronize.

  **Implementation:** *ConcurrentHashMap*, which is a concurrent version of HashMap and implements a better concurrency than *HastTable*: It does not lock when reading and only locks the part of Map that is being written.

  **Example:**

```
ConcurrentMap concurrentMap = new ConcurrentHashMap();
concurrentMap.put("key", "value");
Object value = concurrentMap.get("key");
```

  **ConcurrentNavigableMap** is a subinterface of ConcurrentMap that supports approximate matches.

  **Implementation:** *ConcurrentSkipListMap*, which is a concurrent analog of TreeMap.

## 3.3 Atomic variables

The *java.util.concurrent.atomic* package defines classes (**AtomicInteger**, **AtomicBoolean**, **AtomicLong**, etc) to support atomic operations on single variables. All classes have get and set methods.

**Example**:

```
AtomicInteger atomicInteger = new AtomicInteger(59);//atomic integer value=59
int value = atomicInteger.get();  // value = 59
atomicInteger.set(1000);  // now the value of atomic integer is 1000
```

- Other methods: addAndGet(), getAndAdd(), getAndIncrement(), incrementAndGet(), decrementAndGet(), getAndDecrement()

---

**Exercise 12: Volatile variables**

Search the Internet for information about *volatile* keyword.

- What is *volatile* used for?
- What's the difference between *volatile* and *atomic variables*?
- When is it recommended using *volatile* variables?

---

## 3.4 Concurrent Random Numbers

In JDK 7, for concurrent access, using **ThreadLocalRandom** class instead of Math.random() results in less contention and, eventually, better performance.

**Example**:

```
int randomNumber = ThreadLocalRandom.current().nextInt(50, 90);
```

**Exercise 13**

Modify the Threads Account Conflict example using high-level concurrency mechanisms:

- Define an **AtomicInteger** in the Account class; now methods synchronization is not needed.

- In the *deposit()* method: after adding the *amount,* introduce a random sleep between 500 and 1000 milliseconds. Use **ThreadLocalRandom** class for better performance.

- After calling *deposit()*, the thread must print:

*currentThreadName + task-x + " DEPOSITED 1 EURO"*

- Create a **pool** of 25 threads, set a name to every runnable instance created: TASK-X

- Wait until the pool is done with all the 100 tasks.

- Print out the final balance account.

Execute the program,

```
.
pool-1-thread-19-TASK-93 DEPOSITED 1 EURO
pool-1-thread-13-TASK-95 DEPOSITED 1 EURO
pool-1-thread-4-TASK-100 DEPOSITED 1 EURO
pool-1-thread-10-TASK-99 DEPOSITED 1 EURO
pool-1-thread-11-TASK-98 DEPOSITED 1 EURO
Balance == 100
BUILD SUCCESSFUL (total time: 3 seconds)
```

- You must get always the expected result: 100 euros, thus no race conditions.
- Watch how the threads in the pool are reused to complete all the 100 tasks.
- Notice that random sleep has nothing to do with the concurrent account update.

## 3.5 Semaphores

The *java.util.concurrent.Semaphore* **class** is a **counting semaphore**: it is initialized with a given number of "*permits*" (it is a simple counter). The semaphore has two main operation methods:

- *acquire*(): a permit is taken by the calling thread.
- *release*(): a permit is returned to the semaphore.

At most, N threads can pass the *acquire*() method without any *release*() calls. (N is the number of permits the semaphore was initialized with)

Typical uses of semaphores:

- **Guarding critical sections**

  A thread wanting to enter the critical section must first try to acquire a permit. When it is done, release the permit.

**Example**:

```
Semaphore mySemaphore = new Semaphore(1);// only one thread can take permit

//critical section
mySemaphore.acquire();

     …  // do critical job

mySemaphore.release();
```

- **Sending signals between two threads**

  One thread calls the *acquire*() method, and the other thread call the release() method.

  → If no permits are available, the acquire() call will block until a permit is released by another thread.

  → a release() call is blocked if no more permits can be released into this semaphore.

**Fairness**

By default, acquiring a permit on a semaphore is not fair: The first thread calling to *acquire*() is not guaranteed that is the first to get it.

To create a semaphore enforcing fairness:

```
Semaphore fairSemaphore = new Semaphore(1, true);
```

However, fairness has a performance/concurrency **penalty**, only enable fainess if you really need it!

## 3.6 Lock objects

*java.util.concurrent.locks.Lock* is an interface used as a thread synchronization mechanism like synchronized blocks.

However, a **Lock** is **more flexible** and **more sophisticated** than a synchronized block:

- Threads waiting to entering a synchronized block have no guarantees about the sequence followed to grant the access.
- In a synchronized block we can't set a timeout when trying to get access.
- The synchronized block must be fully contained within a single method. A Lock can have its calls to *lock*() and *unlock*() in separate methods.

**Methods**: *lock(), lockInterruptibly(), tryLock(), tryLock(long timeout, TimeUnit timeUnit), unlock()*.

**Example**:

```
Lock lock = new ReentrantLock();   // implementation of Lock interface

lock.lock();  // blocks thread if not available

      …..  //critical section

lock.unlock();  // only the thread that has the Lock is allowed
                //  to call this method, unless, RuntimeException
```