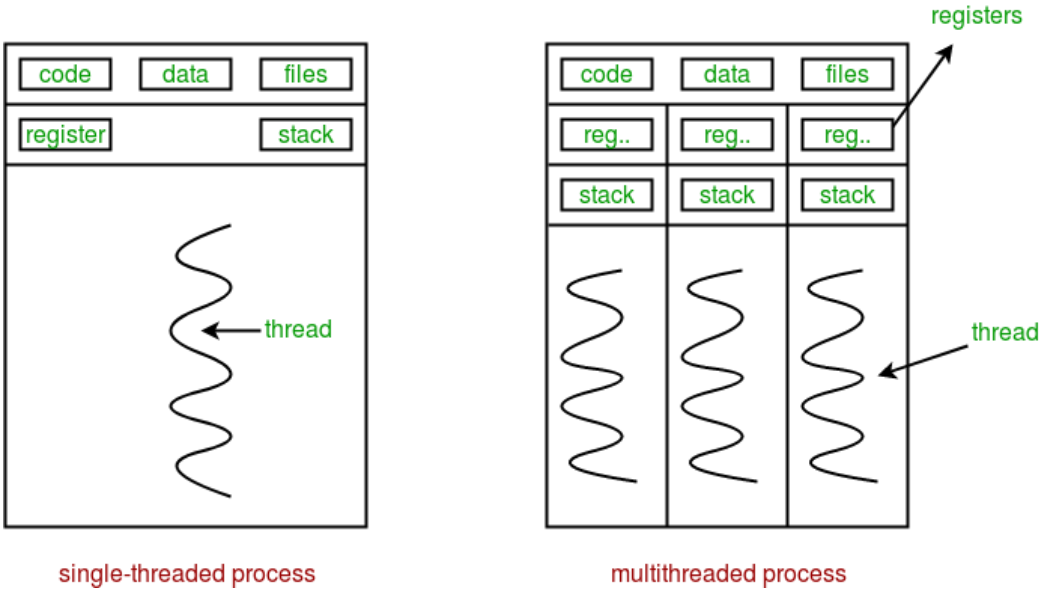
UD02: Multithread coding



1. Basics of multithreaded programming

On many occasions, the statements that make up a program must be executed sequentially, since, together, they form an ordered list of steps to follow to solve a problem. Other times, these steps do not have to be sequential, being able to perform several at the same time.

Other times, having simultaneity in execution is not optional, but necessary, as in web applications. If the requests received by a web server were not served simultaneously, each user would have to wait until all the users who arrived before him were served to obtain the requested pages or resources.

In both cases, the solution lies in multithreading, a technique used to achieve simultaneous processing.

Although full of possibilities, this technique is not exempt from conditions and restrictions. In this unit, the multithreaded processing tools available in the Java language are presented, together with the theoretical aspects related to the analysis of the processes to determine if it is possible to execute them with multiple threads, as well as to detect and prevent the conflicts that arise as a consequence of concurrent programming.

From the point of view of the operating system, a running computer program is a process that competes with other processes for access to system resources. Seen from the inside, a program is basically a sequence of statements that are executed one after the other.

The operating system is in charge of making the different processes coexist and of distributing the resources between them, so when programming it is not necessary to take into account aspects related to how the processes are going to be managed or how they are going to have access to the resources. means. Other than optimizing memory usage and making algorithms efficient, programmers don't have to worry about the rest: the operating system handles concurrency at the process level.

It is within processes that programming has something to say about concurrency, through multithreading.

> A thread is a small unit of computation that runs within the context of a process. All programs use threads.

In the case of an absolutely sequential program, the execution thread is unique, which means that each statement has to wait for the immediately preceding statement to complete its execution before starting its execution. This does not mean in any case that there are no control structures, such as `if` or `while`, but rather that the statements are executed one after the other and not simultaneously.

On the other hand, in a multithreaded program, some of the statements are executed simultaneously, since the threads created and active at a given moment access the processing resources without needing to wait for other parts of the program to finish.

Here are some of the characteristics that threads have:

- **Process dependency**. They cannot be run independently. They always have to be executed within the context of a process.
- **Lightness**. When executed within the context of a process, it does not require the generation of new processes, so they are optimal from the point of view of resource use. Large numbers of threads can be spawned without causing memory leaks.

- **Resource sharing**. Within the same process, threads share memory space. This implies that they can suffer collisions when accessing the variables, causing concurrency errors.
- **Parallelism**. They take advantage of the processor cores generating a real parallelism, always within the capabilities of the processor.
- **Attendance**. They allow to concurrently attend multiple requests. This is especially important on web and database servers, for example.

To more accurately illustrate how a single-threaded program behaves versus multi-threaded programs, the following analogy can be made.

The waiter of a cafeteria receives a client who asks for a coffee, a toast and a French omelette. If the waiter works as a single-thread process, he will put the coffee maker to prepare the coffee, he will wait until it is finished to put the bread to toast and he will not order the tortilla from the kitchen until the bread is toasted. Probably, when everything is ready to serve, the coffee and the toast will be cold and the client will be bored of waiting.

The waiters (the vast majority of them) usually work as multi-threaded processes: they put the coffee machine to prepare the coffee, the bread to toast and ask the kitchen for the dishes without waiting for each of the other tasks to be finished. Since each of them consumes different resources, it is not necessary to do so. After the time it takes for the slowest task to be ready, the customer will be served.

Continuing with the analogy, as happens in computers, the resources of a cafeteria are limited, so there is a physical restriction that prevents more tasks from being done simultaneously than the resources allow. If there is only one toaster with capacity for two slices of bread, it will not be possible to toast more than that number at the same instant of time.

The programming that allows this type of system to be carried out is called multithreading, concurrent or asynchronous.

1.1. Sequential or single-thread programming

The following example programmed in Java illustrates how a program that runs in a single thread behaves, as well as the consequences that this implies. The program is composed of a single class (represents a mouse) composed of two attributes: the name and the time in seconds it takes to eat. In the main method several objects (mice) are instantiated and the eat method of each of them is called. This method displays some text on the screen when it starts, pauses for the duration in seconds (with the sleep method of the Thread class) indicated by the FeedTime parameter, and finally displays another text on the screen when it ends.

See Example01

1.2. Concurrent or multithreaded programming

The above example can be easily scheduled concurrently, since there is no shared resource. To do this, simply convert each Mouse class object into a thread and program what you want to happen concurrently within the run method. Once the instances are created, the start method of each one is called, which causes the contents of the run method to be executed in a separate thread.

Check the Example02

In Java there are two ways to create threads:

- Implementing the `java.lang.Runnable` interface.
- Inheriting from the `java.lang.Thread` class.

The implementation of the `Runnable` interface forces you to program the method with no `run` arguments.

```
1  public class ThreadViaInterface implements Runnable {
2      @override
3      public void run() {
4      }
5  }
```

Inheriting from the Thread class this is not required because that class is already an implementation of the Runnable interface.

```
1  public class ThreadViaInheritance extends Thread {
2      @override //not mandatory but "required"
3      public void run() {
4      }
5  }
```

However, creating a thread by inheriting from `Thread` "requires" the method implementation with no `run` arguments, otherwise the class will not be multithreaded.

Check the Example3

> It is common for programmers to attempt to execute the `run` method on first encounters with threads in Java instead of executing the `start` method. From a compilation point of view there will be no problem: the program will compile without errors.
>
> **From the perspective of concurrent programming, the result will not be adequate, since the threads will be executed one after the other, sequentially, not obtaining any improvement over conventional sequential programming.**

The creation of threads through the implementation of the `Runnable` interface has a clear advantage over the inheritance of the `Thread` class: since Java is a language that does not support multiple inheritance, inheriting from said class prevents other types of inheritance, limiting the software design capabilities.

On the other hand, by implementing the `Runnable` interface many threads can be launched on a single object, as opposed to `Thread` inheritance which will create an object for each thread.

The following Example implements a thread using the `Runnable` interface to create multiple threads from a single object. On the thread, an instance attribute called `foodConsumed` is incremented by 1 during the execution of the `eat` method, called in the `run` method. You can see in the `main` method how a single instance of the `MouseSimple` class is created and four threads are created to execute it.

Check theExample04 and Example04bis

> Due to the nature of multithreaded programming, the outputs of the Example program executions may not match those presented in the notes. In fact, in different executions on the same machine the results may vary.

1.3. Thread states

During the life cycle of threads, they go through various states. In Java, they are contained within the `State` enumeration contained within the `java.lang.Thread` class.

The state of a thread is obtained using the `Thread` class's `getState()` method, which will return some of the possible values listed in the enumeration above.

The states of a thread are shown in the following table:

State	value in `Thread.State`	Description
New	`NEW`	The thread is created, but has not been started yet.
Executed	`RUNNABLE`	The thread is started and could be running or pending execution.
Blocked	`BLOCKED`	Blocked by a monitor.
Waiting	`WAITING`	The thread is waiting for another thread to perform a certain action.
Waiting for a while	`TIME_WAITING`	The thread is waiting for another thread to perform a certain action in a certain period of time.
Finalized	`TERMINATED`	The thread has finished its execution.

In the following Example code, the states passed through by a thread containing a call to the `sleep` method are stored in an `ArrayList`. The `MouseSimple` class is used which implements `Runnable` from the previous Examples.

Check the Example05

> All threads go through **NEW**, **RUNNABLE** and **TERMINATED** states. The rest of the states are conditioned by the circumstances of the execution.

1.4. Introduction to concurrency problems

In concurrent programming, difficulties arise when different threads access a shared and limited resource. If in the Example of the mice (the physical one, not the programmatic one), they ate through a device that they could only access one at a time, concurrency would be impossible. In the programmatic Example the problem is the same, but the physical restriction does not exist, so nothing prevents the threads from accessing the shared resource and that is when the problems will appear.

Access to limited shares must therefore be managed properly. Luckily, there are techniques, classes and libraries that implement solutions, so you just have to know them and know how to use them at the right time.

One important thing about concurrency problems is that they don't always cause a runtime error. This means that in successive executions of the multithreaded program the error will occur in some of them and not in others. Contrary to the determinism of sequential programs (the same input data always generate the same outputs), in the multithreaded environment the factors that determine the execution conditions can originate from elements that the programmer has no ability to influence, such as The operating system.

The difficulty in this type of programming lies in the fact that the programmer has to know in advance that the error can occur in a statement or block of statements, because of how it is executed and what resources it uses. Simply performing conventional tests to determine that the algorithm is well programmed is not enough. In addition, you have to "know" that it is well programmed.

As stated above, there are solutions to all problems, but there is a half-truth in this statement. Concurrent programming problems are solved, in certain cases, by making certain parts of the code sequential. If it is not programmed correctly, there is a risk of converting the entire program to sequential, which would effectively avoid concurrency problems since it would have ceased to exist.

2. Multithreaded programming: classes and libraries

2.1. `java.lang` package

Java has a large number of classes for multithreaded programming. These classes have multiple applications, from the very construction of threads to supporting consistent data structures when accessed by multiple threads.

These classes are grouped into two main packages: the `java.lang` package and the `java.util.concurrent` package.

The `java.lang` package is a default imported package in Java (it contains the basic classes, including `Object` as the root class of the language's class hierarchy) so you don't have to import any libraries to use its classes.

Within this package are the `Runnable` interface and the `Thread` class, as fundamental elements for the construction of threads. There are also classes `Timer` and `TimerTask`

Name	Type	Description
`Runnable`	Interface	This interface must be implemented by those classes that want to run as a `thread`. Defines a method with no arguments `run`.
`Thread`	Class	This class implements the `Runnable` interface, being a thread itself. A class that inherits from `Thread` and overrides the `run` method is executed concurrently if the `start` method is called. You can wrap an object that implements `Runnable` with the `Thread` class. causing it to run in a separate thread.
`Timer`	Class	It allows the programming of the execution of tasks in a deferred and repetitive way through the `schedule` method. This method expects schedulable tasks, represented by objects of the `TimerTask` class, as well as information about the start time of the task and the time that must elapse between each of its executions.
`TimerTask`	Abstract class	By inheriting from this class and overriding the `run` method, you can create a schedulable task with the `Timer` class.

Check the Example06

2.2. `java.util.concurrent` package

An interesting set of classes and tools related to concurrent programming are included in this package. Some of its most relevant elements are shown below.

2.2.1. Executor

It is an interface for defining multithreaded systems. Allows execution of `Runnable` type tasks. Some of its derived interfaces, as well as directly related classes, are shown in the following table.

Name	Type	Description
`ExecutorService`	Interface	Executor subinterface, allows managing asynchronous tasks
`SchedulledExecutorService`	Interface	Allows scheduling of asynchronous task execution.
`Executors`	Class	`Executor`, `ExecutorServices`, `ThreadFactory` and `Callable` object factory.
`TimeUnit`	Enumeration	Provides representations of time units with different granularity, from days to nanoseconds.

Check the Example07

2.3. Queues

Java provides components that address the full spectrum of queuing needs to create secure execution environments for messaging solutions, work queue management, and producer-consumer based systems in concurrent environments.

The most relevant classes of this group are collected in the following table.

Name	Type	Description
`ConcurrentLinkedQueue`	Class	A thread-resistant bonded tail.
`ConcurrentLinkedDeque`	Class	A thread-resistant double-bonded tail.
`BlockingQueue`	Interface	A queue that includes wait locks for space management. Some of its implementations are `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue` and `DelayQueue`.
`TransferQueue`	Interface	A type of `BlockingQueue` specially designed for messaging.
`BlockingDeque`	Interface	A doubly bound queue that includes wait locks for space management. It has an implementation in the `LinkedBlockingDeque` class.

Check theExample08 and Example08bis

2.4. Synchronizers

The `java.util.concurrent` package provides five specific classes to make thread concurrency work properly. These classes are listed in the following table.

Name	Type	Description
`Semaphore`	Class	It provides the classic mechanism for regulating user access to limited-use resources.
`CountDownLatch`	Class	Provides a synchronization aid when threads must wait for a set of operations to be performed.
`CyclingBarrier`	Class	Provides a synchronization aid when threads must wait until other threads reach a certain execution point.
`Phaser`	Class	Provides similar functionality to `CountDownLatch` and `CyclingBarrier` through more flexible usage.
`Exchanger`	Class	Allows to exchange elements between two threads.

Check the Example09

2.5. Concurrent data structures

Java has interfaces and classes to store information with almost any type of data structure. Interfaces inherited from the `Collection` interface, such as `List`, `Map`, `Set`, `Qeue` or `Deque`, are the basis of a series of classes that, by implementing these interfaces, provide the necessary support for everything type of data structures.

From a concurrency point of view, these implementations are not designed to support multiple threads simultaneously reading and writing to them, which can cause errors.

In Example10, multiple threads read from and write to a shared `ArrayList` object.

Execution of this code causes multiple concurrency errors.

Java's `Collections` class provides static methods for making data structures thread-safe, such as `synchronizedList`, `synchronizedMap` or `synchronizedSet` among others, but they are not the most efficient alternatives at all. the cases.

As a more efficient alternative if most operations are read, the `java.util.concurrent` package provides implementations specifically designed for multithreaded execution environments. Some of those classes are shown in the following table.

Name	Type	Description
`ConcurrentHashMap`	Class	Equivalent to a synchronized `HashMap`.
`ConcurrentSkipListMap`	Class	Equivalent to a synchronized `TreeMap`.
`CopyOnWriteArrayList`	Class	Equivalent to a synchronized `ArrayList`.
`CopyOnWriteArraySet`	Class	Equivalent to a synchronized `Set`.

The above example, referring to readers and writers working together on an `ArrayList`, is made thead-safe by using the `CopyOnWriteArraySet` class instead of `ArrayList`, as seen in Example11.

2.6. The interfaces `ExecutorService`, `Callable` and `Future`

Used together, these three interfaces provide mechanisms to execute code asynchronously.

`ExecutorService` provides the asynchronous code execution framework contained in objects of the `Callable` and `Runnable` interfaces. Its main methods are shown in the following table.

Method	Description
`awaitTermination`	Blocks the service when a shutdown request is received until all the tasks assigned to it have finished or the waiting time limit E (timeout) has been reached or there has been an interruption.
`invokeAll`	It allows launching a collection of tasks and fetching a list of `Future` objects.
`invokeAny`	It allows launching a collection of tasks and collecting the result of the one that completes successfully (if any of them do).
`shutdown()`	Causes the running service to stop and terminate. accept new tasks, wait for them to finish
`shutdownNow()`	Terminates the service without waiting for the tasks it has running to do so.
`submit()`	Send a `Runnable` or `Callable` task to execution.

Instance building of `ExecutorService` is done through a series of static methods of the Executors class. These methods allow you to indicate the threading strategy you want the ExecutorService to follow. Some of the static methods of Executors to create objects of the ExecutorService interface are:

- `Executors.newCachedThreadPool()`: Creates an `ExecutorService` with a pool of threads with all the necessary ones, reusing those that are free.
- `Executors.newFixedThreadPool(int nThreads)`: Creates an `ExecutorService` with a pool with a certain number of threads.
- `Executors.newSingleThreadExecutor()`: Create an `ExecutorService` with a single thread.

For its part, the `Callable` interface is an interface that works similar to `Runnable`, but with the difference that it can return a return and throw an exception. It is a functional interface as it only has the call method, so it can be used in lambda expressions. The asynchronous code of a `Callable` object is executed through the submit method of the `ExecutorService`.

The `Future` interface represents a future result generated by an asynchronous process. In a way, it is a system that allows you to suspend obtaining a result until it is available. The result of calling the `submit` method of an `ExecutorService` is a `Future` object.

The methods of the `Future` interface are listed in the following table.

Method	Description
`cancel`	Attempts to cancel the execution of the task.
`get`	Wait for the task to finish and get the result. In one of its forms, it supports a timeout to limit the waiting time.
`isCancelled()`	Indicates if the task was canceled before finishing.
`isDone()`	Indicates if the task has finished.

In summary, the relationship between these three interfaces is as follows: `ExecutorService` executes a `Callable` (or `Runnable`) object with a given multithreading strategy and deposits the future return of the task in a `Future` object.

See Example12

> In programming, one alternative is almost never always better than another. The `Runnable` and `Callable` interfaces are absolutely valid, and depending on the situation, one or the other will be the best alternative.

3. Asynchronous programming

Conceptually, asynchronous programming is independent of the programming language in which it is done, as long as it is supported by it. Instead, on a practical level, each programming language provides different tools to allow the creation and management of threads.

Java is a language with an extensive library of classes oriented to asynchronous or multithreaded programming. Thanks to many of these classes, extremely complex problems can be solved without excessive difficulty, since they implement solutions that, if they did not exist, would have to be developed.

This section shows the basic mechanisms for thread creation, execution, and synchronization. Although some of these mechanisms have already been introduced previously in this unit, they are reintroduced briefly in order to provide an overview of the basic elements of asynchronous programming in Java.

3.1. The `Runnable` interface

A class that implements the `Runnable` interface is intended to be executed within a thread. The `run` method is its only method and it has no arguments and no return. The content of the `run` method is executed asynchronously, so the main thread of the application does not stop, even if the code block contained in that method does.

When an object of a class that implements `Runnable` is instantiated, it has to be called from the `start` method of the `Thread` instance constructed from the `Runnable` object.

Example13 shows the creation and execution of 10 threads using objects from the `Runnable` interface.

Being a functional interface, `Runnable` can be used in a lambda expression, available in Java since version 8.

See Example14 which shows a modified version of Example13 using the lambda expression.

3.2. The `Thread` class

The `Thread` class represents a thread of execution. When a class inherits from `Thread` it can implement the `run` method and execute asynchronously.

To launch a `Thread` object asynchronously, just call its `start` method. Example15 shows the creation and execution of a basic thread using the `Thread` class.

In Java, normal threads are called user threads, with another type known as daemon threads. These threads are created from a conventional thread using the `setDaemon` method. A daemon thread constitutes a thread with a lower priority than a user thread, executing afterwards. The other difference is that these types of threads are useful when there are user threads, so when the latter end, the daemon threads end.

The main methods of the `Thread` class are shown in the following table.

Method	Description
`start`	Start method of the asynchronous block of the class.
`run`	Method in which the asynchronous block is programmed. It is executed when the start method is called.
`join`	Blocks the thread until the referenced thread ends.
`sleep`	Static method that temporarily stops the execution of the thread.
`getId`	Returns the identifier of the thread. It is a positive long generated when the thread is created.
`getName`	Returns the name of the thread, assigned in some of the constructor forms.
`getState`	Returns the state of the thread as a value from the `Thread.State` enumeration.
`interrupt`	Interrupt thread execution.
`interrupted`	Static method that checks if the current thread has been interrupted.
`isInterrupted`	Check if the thread you are in has been interrupted.
`isAlive`	Check if the thread is alive.
`setPriority`	Change the priority of the thread. The value must be between the `MIN_PRIORITY` and `MAX_PRIORITY` constants of the `Thread` class itself. The use made of the established priorities is determined by the operating system.
`setDaemon`	Allows you to mark the thread as a daemon thread.
`isDaemon`	Determines if a thread is a daemon thread.
`yield`	Static method that indicates to the scheduler that you are willing to give up your current processor usage. The scheduler decides whether or not to heed this suggestion.

3.3. Suspend execution: the `sleep` method

The `sleep` static method of the `Thread` class allows you to suspend the execution of the thread from which it is invoked. It accepts as a parameter the amount of time in milliseconds that this suspension is desired, the precision of which is subject to the precision of the system timers and the scheduler.

3.4. Interrupts

In computing, an interrupt is a temporary suspension of the execution of a process or a thread of execution. Interrupts do not usually belong to programs, but to the operating system, being generated by requests made by peripheral devices.

In Java, an interrupt is an indication to a thread that it should stop executing to do something else. It is the responsibility of the programmer to decide what he wants to do in the event of an interrupt, the most common being to stop the execution of the thread.

Interrupts are caught by the InterruptedException exception, which is caused by some operations such as calling the sleep method. In case the exception does not have to be handled, the static method interrupted should be called for Paraninto to handle the interruption.

In Example16 the management of an exception is carried out that is thrown in a forced way before a certain condition. On catching the exception, the `return` statement is called to end the execution of the thread.

3.5. Information Sharing

Multiple threads can be created as instances of the Thread class. The attributes of these threads, if they are not static, will be specific to each one of them, so they cannot be used to share information.

If you want several threads to share information, there are several alternatives:

- **Use static attributes**. Static attributes are common to all instances, so regardless of how the threads are built, the information is shared.
- **Using common object references** accessible from all threads. Example17 shows an Example of using an instance of an object common to several threads that are modified in each one of them.
- **Using non-static attributes** of the instance of a class that implements `RunnableSharing` and building threads from that instance. The Example18 returns an Example.

There are other ways of sharing information, either through files, databases or network or internet services. In all cases, it must be taken into account that the information shared by several threads for reading and writing is a potential source of concurrency errors.

4. Problems and solutions of concurrent programming. Synchronization

Intrinsically, concurrent programming has two characteristics that can be sources of errors: shared resources and execution order.

Regarding the sharing of resources, the problem is diverse, since it can affect how the value of a variable is modified, the data of a data structure is accessed, a block of code is executed or a resource is accessed. limited.

To illustrate this type of problem, Example19 can be used.

It is representative of the problem of concurrent programming that such important errors can arise in such an apparently simple operation.

> Concurrency problems do not always occur in the same way, since the execution is not deterministic. The algorithm that generates an error when 1000 threads increment a shared variable by 1000 units can work correctly in most cases with 10 threads and an increment of 10 units per thread. However, the risk that an algorithm might fail, even if the chance is low, is normally unacceptable in computing.

The order of execution, on the other hand, has to do with the dependencies that can occur between blocks of code depending on the order of execution. If, for Example, a block of code needs as input data generated as output by another block, this will cause a dependency, since in a multithreaded system there is no control over the order of executions unless synchronization mechanisms are established.

This section presents the most important terms and concepts related to concurrency problems, as well as these problems and their possible solutions.

4.1. Concepts

In order to understand both the problems and the solutions related to concurrent programming, it is necessary to know the meaning of some concepts. In this section the most important ones are presented.

4.1.1. Shared resources

A shared resource is, as its name suggests, a system element that is used by several execution threads simultaneously. It can be a static attribute of a class or a non-static attribute of an object shared by all threads, a static method of a class or a non-static method of an object shared by all threads, a connection to a database, a socket or anything with a limited number of instances.

4.1.2. Dependencies

As mentioned above, not all tasks can be executed in a multithreaded environment. To be able to do this, you have to be certain that the code segments to be executed in parallel are independent and the order in which they are executed is irrelevant.

A task will not be able to run in a multithreaded environment if it has dependencies. There are three main types of dependencies:

- Data dependencies. Multiple code segments use the same data.
- Flow dependencies. Since the order of execution of the program cannot be determined due to the existence of flow control instructions, there are potential dependencies that cannot be determined in a static analysis of the code.
- Resource dependencies. Multiple code segments access simultaneously to processor resources.

The existence of dependencies of any kind prevents concurrent programming unless synchronization mechanisms can be established.

4.1.3. Bernstein conditions

Formally, the fulfillment of the **Bernstein** conditions determines whether two segments of code can be executed in parallel. Given two code segments S_1, and S_2 it is determined that they are independent and can be executed in parallel if:

- The inputs of S_2 are different from the outputs of S_1. Otherwise, what is known as **flow dependency** occurs.
- The inputs of S_1 are different from the outputs of S_2. Otherwise, what is known as **anti-dependence** occurs.
- The outputs of S_1 are different from the outputs of S_2. Otherwise, what is known as **exit dependency** occurs.

If two segments of code meet the **Bernstein** conditions, their execution can be parallelized.

4.1.4. Atomic action and access

An atomic action is one that is executed without interruption, in one go. Any effect of the action is only visible at the end of the action.

In Java, some simple actions are atomic:

- Read and write variables of primitive types, except the types `long` and `double`.
- Read and write all variables declared `volatile`, including types `long` and `double`.

> Very simple actions, such as incrementing the value of a variable of type `int` by 1, are not atomic, so you have to evaluate whether it is necessary to establish a synchronization mechanism.

4.1.5. critical section

The critical section of a multithreaded program is the block of code that accesses shared resources, so it should only be accessed by a single thread of execution. Correctly determining the critical section allows the program to be correctly synchronized to avoid concurrency errors, as well as making it efficient in order to take maximum advantage of parallelism. Ensuring that only one thread accesses the critical section is known as **mutual exclusion**.

4.1.6. Mutual exclusion

This is the name given to the programming technique consisting of making, in a concurrent environment, one process exclude all the others from using a shared resource (a critical section) to guarantee the integrity of the system.

4.1.7. Thread safety

These terms refer to the property that a piece of software (a class or a data structure, for Example) has to be executed in a multi-threaded environment safely.

> In Java, the documentation for data structures often specifies whether they are `Thread safety` or are instead unsynchronized (not thread-safe). It's a good idea to review the documentation for classes you use for the first time in general, paying special attention to references to threads in multithreaded environments.

4.2. Concurrent programming problems

This section presents some of the problems that arise as a consequence of multithreading, as well as the solutions that can be adopted, both generic and specific to the Java language.

4.2.1. Interbloqueo o deadlock

Occurs when two or more threads are blocked from each other. For example, if thread A is waiting for thread B to finish to continue its process and thread B, in turn, is waiting for thread A to finish to continue its process, a deadlock occurs.

4.2.2. Death by starvation

This issue occurs when a priority policy is set that causes
that some threads never have access to the CPU.

4.2.3. Race Conditions

Occurs when two blocks of concurrent code have dependencies between input and output data and are not executed in the correct order (flow dependency or anti-dependency).

4.2.4. memory inconsistency

Occurs when two or more threads simultaneously have different values for the same variable.

4.2.5. Slip conditions

Occurs when a condition in a process is evaluated to determine if a section of code needs to be executed, and after evaluation and before execution, the condition changes value. A block of code whose condition is not being met would be executed.

4.3. Basic synchronization: `volatile` variables

In a multi-core computing environment, processors have optimization techniques and some of them are based on the use of cache memory. These techniques are usually advantageous, but in concurrent programming they can be a source of errors.

When multiple threads share the same variable, if it is stored in core caches, threads may see different copies of the same variable, which can cause memory inconsistency.

To prevent a variable from being stored in the processor cache and all threads accessing the same copy in Java, the volatile keyword is used. By declaring a variable volatile, only one copy will exist on the processor.

The following Example code shows the declaration of a volatile variable.

```
1 private volatile static long contador
```

This solution does not by itself solve all memory inconsistency problems. If several threads concurrently modify the same variable, even if it is declared volatile, it could continue to occur (synchronization mechanisms should be included).

`volatile` variables would be appropriate for systems where a single thread modifies the value of the variable and the rest just query it.

4.4. Basic synchronization: `wait`, `notify` and `notifyAll`

The `wait`, `notity` and `notifyAll` methods are specific to the `Object` class, so all Java classes have them.

All of these methods must be called from code segments in a thread that has a monitor, such as a synchronized block or segment, for example, and force an exception of type `InterruptedException` to be caught.

The `wait` method stops the execution of the thread, and the `notify` and `notifiyAll` methods cause the stopped threads to be reactivated. The `notify` method continues a single segment at random from those paused with `wait`, while `notifyAll` continues all segments paused with `wait`.

In Example20 an example of the use of these synchronization methods is shown. In it, two threads created from the same object execute two different methods. The first of the threads, when performing half of the task, enters a wait until the second of the threads finishes and notifies that it must resume execution.

4.5. Basic synchronization: the `join` method

The `join` method allows a thread to be instructed to suspend execution until another referenced thread terminates. This method must be executed within the async block of the code, otherwise it will have no effect.

The Example21 allows to illustrate the operation of this method. In it, two threads execute a loop with 3 iterations that in the absence of synchronization mechanisms would generate an output similar to this:

```
Thread 1. Interaction 0
Thread 2. Interaction 0
Thread 1. Interaction 1
Thread 2. Interaction 1
Thread 2. Interaction 2
Thread 1. Interaction 2
```

Since each thread has the same priority and the same code, they will simultaneously write the scheduled output.

Instead, in Example22, after starting both threads, thread `thread2` is told to stay suspended until `thread1` finishes executing. This requires `thread2` to have a reference to `thread1` (`referenceThread` in the Example) to call the `join` method.

The result obtained in this case is the following:

```
Thread 1. Interaction 0
Thread 1. Interaction 1
Thread 1. Interaction 2
Thread 2. Interaction 0
Thread 2. Interaction 1
Thread 2. Interaction 2
```

4.6. Basic synchronization: thread resistant data structures

The conventional data structures provided by Java satisfy any need related to storing data in memory. From the `concurrent programming point of view, it must be taken into account that the ArrayList, Vector`, `HashMap` or `HashSet` classes, all of them from the `java.util` package, are not synchronized, which means that they are not programmed to be consistent against access from multiple threads.

To be able to use these structures you have to use the synchronization techniques discussed in this unit, convert them to synchronized structures with the static methods provided by the `java.util.Collections` class (for Example `synchronizedList` for objects that implement the `List interface`) or use the data structures provided by the `java.util.concurrent` package and presented in a previous section.

4.7. Advanced synchronization: mutual exclusion, `synchronized` and monitors

One of the mechanisms provided by Java to synchronize code segments is to use the `synchronized` keyword. By using synchronized you can limit access to a segment of code to a single thread concurrently, thus achieving mutual exclusion or mutex. It allows to synchronize both methods and code segments (synchronized declarations), allowing this last alternative to delimit the critical section with more precision.

At the method level, usage is as simple as including the word in the method declaration:

```
public synchronized void calculate ()
```

The effect of this declaration is that, for an object instance, only one thread can be executing the synchronized method at any given time.

For its part, at the code segment level, synchronization is performed by delimiting a block of statements by making a synchronized declaration. The following code shows a synchronized statement.

```
public void calculate(){
    //Sentences not synchronized
    synchronized (objetoBloqueo) {
        //Block of unsyncrhonized sentences
    }
    //Sentences not synchronized
}
```

This system uses a concept known as intrinsic locking, `monitor` locking, or simply `monitor`. The `monitor` is an element associated with an instance that acts as a lock. When a synchronized method or block is executed using a given `monitor`, it is locked and cannot be used until it is released, preventing code that uses the same lock from being executed.

When using synchronized methods, the object they are in is used as the `monitor`. This assumes that when a synchronized method of an object is executing no other synchronized methods of that object can be executed. The exclusion is therefore very generic and may be inefficient in many cases.

See the Example23

As indicated above, the behavior of the synchronized methods of Example is due to the fact that they are executed on the same instance of the object that implements `Runnable`, so they use said object as a monitor, causing the block.

If they used different objects instead, the result would be the same as not marking the methods as `synchronized`, since they use different locks.

See the Example24

On its part, synchronization at the segment level also needs a `monitor`, but since it does not depend on the object in which it is being executed, it is more flexible. When using synchronized blocks it is not necessary to block all the segments of an object as is the case with object methods, but they can be grouped into different monitors.

In Example25 synchronization is performed at the block level, using two different locks in each of them. In such a way that the methods are not exclusive of each other. The synchronization is done at the method level (each of the methods can only be executed by one object at a time, but both methods can be executed by two different objects).

On the other hand, if the methods use the same object as a lock, when an object is executing one of the methods, no object can execute either method.

4.8. Advanced synchronization: semaphores

Using the `synchronized` keyword allows you to set critical sections that only one thread has access to at a given time, but there are other scenarios where limited resources allow access by more than one thread.

When a critical section can be executed by more than one thread, but the number of threads is limited, a concurrent programming element known as a semaphore is used and implemented in Java by the `Semaphore` class.

Semaphores are typically used when a resource has limited capacity and you want to control the number of consumers of that resource. It is, therefore, an element of synchronization.

When constructing the semaphore, it is provided through the constructor with a capability that refers to the number of threads it can be running concurrently.

This capacity is converted into the number of access permissions that are granted to the code blocks that want to make use of the limited resource. If all permissions are granted, threads wait for permission owners to release them.

The main methods of the semaphore class are listed below in the following table.

Method	Description
`acquire`	Acquire one or more semaphore permissions if any are available. Otherwise the thread is waiting..
`release`	Releases one or more previously granted permissions.
`tryAcquire`	Try to get one or more permissions, and may be on hold for a limited time.

In addition to these methods, the `Semaphore` class provides methods for managing and querying the state of the semaphore.

In the Example26 Example, it is built with a semaphore capable of three permissions. In turn, five threads built from the same `Runnable` instance access the critical section, request permission, perform the steps of the synchronized activity, and release the lock.

> Semaphore can not only be used when resources are limited. Sometimes it is convenient to limit the number of threads that perform certain tasks so as not to saturate the system or any of its components. For example, limiting the number of threads accessing web services or databases may be a suitable strategy in many cases.

5. Examples

5.1. Example01

Example with a single thread of execution:

```
public class Mouse {

    private String name;
    private int feedingTime;

    public Mouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            System.out.printf("The mouse %s has stopped to feed%n", name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Mouse fievel = new Mouse("Fievel", 4);
        Mouse jerry = new Mouse("Jerry", 5);
        Mouse pinky = new Mouse("Pinky", 3);
        Mouse mickey = new Mouse("Mickey", 6);
        fievel.eat();
        jerry.eat();
        pinky.eat();
        mickey.eat();
    }
}
```

The output produced by the execution is the following:

```
The mouse Fievel has started to feed
The mouse Fievel has stopped to feed
The mouse Jerry has started to feed
The mouse Jerry has stopped to feed
The mouse Pinky has started to feed
The mouse Pinky has stopped to feed
The mouse Mickey has started to feed
The mouse Mickey has stopped to feed
```

5.2. Example02

Multithreaded example:

```
public class Mouse extends Thread {

    private String name;
    private int feedingTime;

    public Mouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            System.out.printf("The mouse %s has stopped to feed%n", name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        this.eat();
    }

    public static void main(String[] args) {
        Mouse fievel = new Mouse("Fievel", 4);
        Mouse jerry = new Mouse("Jerry", 5);
        Mouse pinky = new Mouse("Pinky", 3);
        Mouse mickey = new Mouse("Mickey", 6);
        fievel.start();
        jerry.start();
        pinky.start();
        mickey.start();
    }
}
```

Example output of its execution:

```
The mouse Fievel has started to feed
The mouse Jerry has started to feed
The mouse Pinky has started to feed
The mouse Mickey has started to feed
The mouse Pinky has stopped to feed
The mouse Fievel has stopped to feed
The mouse Jerry has stopped to feed
The mouse Mickey has stopped to feed
```

All the mice have started feeding immediately, without waiting for any of the others to finish. The total process time will be approximately the time of the slowest process (in this case, 6 seconds). The reduction of the total execution time is evident.

5.3. Example03

Returning to the statement of the Example of the objects of the Mouse class, the solution by implementing the Runnable interface would have the code shown below, review the parts of the code that have been modified with respect to the previous solution in which it was inherited from Thread.

```
public class Mouse implements Runnable {

    private String name;
    private int feedingTime;

    public Mouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            System.out.printf("The mouse %s has stopped to feed%n", name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        this.eat();
    }

    public static void main(String[] args) {
        Mouse fievel = new Mouse("Fievel", 4);
        Mouse jerry = new Mouse("Jerry", 5);
```

```
        Mouse pinky = new Mouse("Pinky", 3);
        Mouse mickey = new Mouse("Mickey", 6);
        new Thread(fievel).start();
        new Thread(jerry).start();
        new Thread(pinky).start();
        new Thread(mickey).start();
    }
}
```

The execution result:

```
The mouse Fievel has started to feed
The mouse Pinky has started to feed
The mouse Jerry has started to feed
The mouse Mickey has started to feed
The mouse Pinky has stopped to feed
The mouse Fievel has stopped to feed
The mouse Jerry has stopped to feed
The mouse Mickey has stopped to feed
```

You can see that the code is very similar, except for the class declaration (interface implementation vs. inheritance) and thread creation and startup: `Runnable` objects must be "wrapped" in `Thread objects.` in order to be booted. It is important to note that in both cases the execution is done by invoking the `start` method.

5.4. Example04

Example multithreaded from a single.

```
public class SimpleMouse implements Runnable {

    private String name;
    private int feedingTime;
    private int consumedFood;

    public SimpleMouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            consumedFood++;
            System.out.printf("The mouse %s has stopped to feed%n", name);
            System.out.printf("Consumed food: %d%n", consumedFood);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }

    @Override
    public void run() {
        this.eat();
    }

    public static void main(String[] args) {
        SimpleMouse fievel = new SimpleMouse("Fievel", 4);
        new Thread(fievel).start();
        new Thread(fievel).start();
        new Thread(fievel).start();
        new Thread(fievel).start();
    }
}
```

The result of the execution is the following:

```
The mouse Fievel has started to feed
The mouse Fievel has started to feed
The mouse Fievel has started to feed
The mouse Fievel has started to feed
The mouse Fievel has stopped to feed
The mouse Fievel has stopped to feed
Consumed food: 3
The mouse Fievel has stopped to feed
Consumed food: 2
Consumed food: 4
The mouse Fievel has stopped to feed
Consumed food: 4
```

Each thread has executed the `run` method on the data of the same object. That is, a block of code from a single object has been executed simultaneously four times, sharing its attributes. In this way, in the output you can see that the value of the `consumedFood` attribute has been increased by 1 for each thread. This can be seen because the value of `consumedFood` has been incremented several times during execution. The fact that some intermediate values do not appear in the following capture of the execution output has to do with the `asynchrony` and the high execution cost of the statements that are written on the screen.

Although it will be dealt with later in this same unit, it is important to insist that the different threads of the previous Example are working on a single copy of the object in memory, so the variables (the attributes) are shared, and can suffer concurrency errors.

5.5. Example04bis

For Example, by replacing the main method of the previous Example with the following code, multiple threads can be created and executed via a for loop from a single instance of a class that implements the `Runnable` interface. The result, with a low number of iterations (for Example, 4) will usually be correct (the `consumedFood` attribute will reach the same value as the number of iterations). With a high number (for

example, 1000 iterations) the result will usually be wrong (the `consumedFood` attribute will reach a value below the number of iterations). This is due to the fact that when all the threads share the same attributes, they produce concurrency errors that must be avoided by means of specific techniques that we will see later.

```
public static void main(String[] args) {
    SimpleMouse fievel = new SimpleMouse("Fievel", 4);
    for (int i = 0; i < 1000; i++) {
        new Thread(fievel).start();
    }
}
```

In an execution of the above code, the result is 998, when it would have been 1000 if there had been no concurrency issues.

```
Consumed food: 998
```

5.6. Example05

List of states through which a thread passes:

```
import java.util.ArrayList;

public class SimpleMouse implements Runnable {

    private String name;
    private int feedingTime;
    private int consumedFood;

    public SimpleMouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            consumedFood++;
            System.out.printf("The mouse %s has stopped to feed%n", name);
            System.out.printf("Consumed food: %d%n", consumedFood);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
```

```
        this.eat();
    }

    public static void main(String[] args) {
        SimpleMouse mickey = new SimpleMouse("Mickey", 6);
        ArrayList<Thread.State> threadState = new ArrayList();
        Thread h = new Thread(mickey);
        threadState.add(h.getState());
        h.start();

        while (h.getState() != Thread.State.TERMINATED) {
            if (!threadState.contains(h.getState())) {
                threadState.add(h.getState());
            }
        }
        if (!threadState.contains(h.getState())) {
            threadState.add(h.getState());

        }
        for (Thread.State estado : threadState) {
            System.out.println(estado);
        }
    }
}
```

At the end of the execution, the states collected are shown:

```
The mouse Mickey has started to feed
The mouse Mickey has stopped to feed
Consumed food: 1
NEW
RUNNABLE
TIMED_WAITING
TERMINATED
```

5.7. Example06

The following example shows a joint use of the `Timer` and `TimerTask` classes. The program simulates controlling an automatic irrigation system. This system irrigates for the first time after `1000` milliseconds from the start of the execution and repeats the irrigation every `2000` milliseconds.

```
import java.util.Timer;
import java.util.TimerTask;

public class IrrigationSystem extends TimerTask {

    @Override
    public void run() {
        System.out.println("Watering...");
    }
```

```
10
11      public static void main(String[] args) {
12          Timer timer = new Timer();
13          timer.schedule(new IrrigationSystem(),1000,2000);
14      }
15  }
```

The output generated after a few seconds is shown below. Between the writing of each line 2000 milliseconds elapse.

```
1  Watering...
2  Watering...
3  Watering...
4  ...
```

5.8. Example07

The following example illustrates the use of `ScheduledExecutorService` as a tool for scheduling task executions. As can be seen, through the `Executors` class an instance of `ScheduledExecutorService` is obtained, which is the interface that allows recurring tasks to be scheduled in independent threads. Once the task scheduler is obtained, it is told which task to execute (`sr` object), how many time units to wait until the first task starts (1), how many time units to wait between each repetition of the task (2) and in which unit the units of time are represented(`TimeUnit.SECONDS`)

```
1  import java.util.concurrent.Executors;
2  import java.util.concurrent.ScheduledExecutorService;
3  import java.util.concurrent.TimeUnit;
4
5  public class IrrigationSystem implements Runnable {
6
7      @Override
8      public void run() {
9          System.out.println("Watering...");
10     }
11
12     public static void main(String[] args) {
13         IrrigationSystem ir = new IrrigationSystem();
14         ScheduledExecutorService ses = Executors.newSingleThreadScheduledExecutor();
15         ses.scheduleAtFixedRate(ir, 1, 2, TimeUnit.SECONDS);
16         System.out.println("Configured irrigation system");
17     }
18 }
```

The output of the execution after a few seconds is shown below. Two seconds elapse between each writing of the text “Watering”.

```
Configured irrigation system
Watering...
Watering...
```

5.9. Example08

To better illustrate the behavior of this type of solution, the following example is presented. uses A queue receives writes and reads from a set of threads. The first solution uses a `LinkedList` data structure as a support. As this structure is not safe against multiple threads, the execution produces an error.

```java
import java.util.LinkedList;
import java.util.Queue;

public class nonConcurrentQueue implements Runnable {

    private static Queue<Integer> queue = new LinkedList<Integer>();

    @Override
    public void run() {
        queue.add(10);
        for (Integer i : queue) {
            System.out.print(1 + ":");
        }
        System.out.println("Queue size:" + queue.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(new nonConcurrentQueue()).start();
        }
    }
}
```

The output will be similar to this:

```
Exception in thread "Thread-1" Exception in thread "Thread-0"
java.util.ConcurrentModificationException
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Queue size:6
Queue size:4
Queue size:5
Queue size:3
1:1:1:1:1:1:1:Queue size:7
1:1:1:1:1:1:1:1:Queue size:8
1:1:1:1:1:1:1:1:1:Queue size:9
1:1:1:1:1:1:1:1:1:1:Queue size:10
    at java.base/java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:970)
[...]
```

5.10. Example08bis

The second solution uses the same code by changing the data structure to `ConcurrentLinkedDeque` and getting an error-free execution.

```java
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedDeque;

public class ConcurrentQueue implements Runnable {

    private static Queue<Integer> queue = new ConcurrentLinkedDeque<Integer>();

    @Override
    public void run() {
        queue.add(10);
        for (Integer i : queue) {
            System.out.print(1 + ":");
        }
        System.out.println("Queue size:" + queue.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(new ConcurrentQueue()).start();
        }
    }
}
```

Output obtained:

```
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Queue size:2
Queue size:4
Queue size:2
Queue size:7
Queue size:7
Queue size:7
Queue size:3
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Queue size:9
1:Queue size:9
1:1:1:1:1:1:1:1:1:1:Queue size:10
```

Although this last execution is not ordered due to the concurrence of the threads, it can be seen that the execution has generated a queue with 10 elements corresponding to the 10 threads that were accessing to read and write in the data structure.

5.11. Example09

The following example shows the use of `Exchanger`. Two classes are created that implement `Runnable`, `TaskA` and `TaskB`. Both classes receive an instance of `Exchanger` in the constructor and use it to exchange information with each other. Calling the `exchange` method by one of the two tasks will result in a wait lock until the other task does the same, exchanging information between the two threads. For its part, the `Commuter` class builds both the `Exchanger` object and the two scheduled tasks in `TaskA` and `TaskB`. It is important to pay attention to the fact that the tasks do not have references to each other, but instead have access to the object that acts as an "exchanger" of information.

`TaskA`:

```
import java.util.concurrent.Exchanger;

public class TaskA implements Runnable {

    private Exchanger<String> exchanger;

    public TaskA(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            String receivedMessage = exchanger.exchange("Message sent by TaskA");
            System.out.println("Message received in TaskA: " + receivedMessage);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

`TaskB`:

```
import java.util.concurrent.Exchanger;

public class TaskB implements Runnable {

    private Exchanger<String> exchanger;

    public TaskB(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            String receivedMessage = exchanger.exchange("Message sent by TaskB");
            System.out.println("Message received in TaskB: " + receivedMessage);
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

`Commuter`:

```
import java.util.concurrent.Exchanger;

public class Commuter {

    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<String>();
        new Thread(new TaskA(exchanger)).start();
        new Thread(new TaskB(exchanger)).start();
    }
}
```

The output obtained will be:

```
Message received in TaskA: Message sent by TaskB
Message received in TaskB: Message sent by TaskA
```

5.12. Example10

```
import java.util.ArrayList;
import java.util.List;

public class nonSafeReaderWriter extends Thread {

    private static List<String> words = new ArrayList<String>();

    @Override
    public void run() {
        words.add("New word");
        for (String word : words) {
            words.size();
        }
        System.out.println(words.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new nonSafeReaderWriter().start();
        }
    }
}
```

References to exceptions indicating that concurrency errors have occurred in accessing the list, such as `java.util.ConcurrentModificationException`, appear in the output:

```
[...]
Exception in thread "Thread-21" java.util.ConcurrentModificationException
[...]
```

5.13. Example11

```
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class safeReaderWriter extends Thread {

    private static List<String> words = new CopyOnWriteArrayList<String>();

    @Override
    public void run() {
        words.add("New word");
        for (String word : words) {
            words.size();
        }
        System.out.println(words.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new safeReaderWriter().start();
        }
    }
}

```

That this time it runs without problems or errors.

5.14. Example12

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

public class Reader implements Callable<String> {

    @Override
    public String call() throws Exception {
        String readedText = "I like action movies";
        Thread.sleep(1000);
        return readedText;
```

```
    }

    public static void main(String[] args) {
        try {
            Reader reader = new Reader();
            ExecutorService executionService = Executors.newSingleThreadExecutor();
            Future<String> result = executionService.submit(reader);
            String text = result.get();
            if (result.isDone()) {
                System.out.println(text);
                System.out.println("Process finished");
            } else if (result.isCancelled()) {
                System.out.println("Process cancelled");
            }
            executionService.shutdown();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The output produced is the following:

```
I like action movies
Process finished
```

As stated above, the main difference between implementing `Callable` and `Runnable` is that the first option can provide a return. However, with `Runnable` there are techniques for passing values through the use of object references.

5.15. Example13

```
public class BasicRunnable implements Runnable {

    private int id;

    public BasicRunnable(int id) {
        super();
        this.id = id;
    }

    @Override
    public void run() {
        try {
            System.out.println("Processing thread " + id);
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
19
20      public static void main(String[] args) {
21          for (int i = 0; i < 10; i++) {
22              BasicRunnable br = new BasicRunnable(i);
23              new Thread(br).start();
24          }
25      }
26  }
```

The output generated for any execution will be similar to this:

```
1   Processing thread 9
2   Processing thread 1
3   Processing thread 5
4   Processing thread 6
5   Processing thread 3
6   Processing thread 7
7   Processing thread 8
8   Processing thread 4
9   Processing thread 2
10  Processing thread 0
```

As can be seen, the order of the writes does not coincide with the order of the executions of the threads.

5.16. Example14

```
1   public class BasicRunnableLambda {
2
3       private int id;
4
5       public BasicRunnableLambda(int id) {
6           super();
7           this.id = id;
8       }
9
10      public static void main(String[] args) {
11          for (int i = 0; i < 10; i++) {
12              BasicRunnableLambda br = new BasicRunnableLambda(i);
13              new Thread(() -> {
14                  try {
15                      System.out.println("Processing thread " + br.id);
16                      Thread.sleep(10000);
17                  } catch (InterruptedException e) {
18                      e.printStackTrace();
19                  }
20              }).start();
21          }
22      }
23  }
```

The output does not differ from the previous example.

5.17. Example15

```
public class BasicThread extends Thread{
        private int id;

    public BasicThread(int id) {
        super();
        this.id = id;
    }

    @Override
    public void run() {
        try {
            System.out.println("Processing thread " + id);
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            BasicThread bt = new BasicThread(i);
            new Thread(bt).start();
        }
    }

}
```

The output does not differ from the class we implemented earlier with `Runnable`.

5.18. Example16

```
public class BasicInterruption extends Thread {

    @Override
    public void run() {
        int counter = 0;
        while (true) {
            counter++;
            try {
                System.out.println(counter);
                if (counter == 3) {
                    System.out.print("Interruption");
                    this.interrupt();
                }
                Thread.sleep(1000);
            } catch (InterruptedException e) {
```

```
                return;
            }
        }

    }

    public static void main(String[] args) {
        new BasicInterruption().start();
    }
}
```

The output when executing the code should be:

```
1
2
3
Interruption
```

5.19. Example17

`SharedObject` class:

```
public class SharedObject {
    public int sharedVariable;
}
```

`UniqueInstanceSharing` class:

```
public class UniqueInstanceSharing extends Thread {

    private SharedObject so;

    public UniqueInstanceSharing(SharedObject so) {
        this.so = so;
    }

    @Override
    public void run() {
        this.so.sharedVariable++;
        System.out.println("Shared Variable: " + this.so.sharedVariable);
    }

    public static void main(String[] args) throws InterruptedException {
        SharedObject so = new SharedObject();
        UniqueInstanceSharing uis1 = new UniqueInstanceSharing(so);
        UniqueInstanceSharing uis2 = new UniqueInstanceSharing(so);
        uis1.start();
        Thread.sleep(1000);
        uis2.start();
```

```
    }
}
```

The output will be similar to this:

```
Shared Variable: 1
Shared Variable: 2
```

5.20. Example18

```
public class RunnableSharing extends Thread {

    private int counter;

    @Override
    public void run() {
        counter++;
        System.out.println("Counter: " + counter);
    }

    public static void main(String[] args) throws InterruptedException {
        RunnableSharing rs = new RunnableSharing();
        for (int i = 0; i < 1000; i++) {
            new Thread(rs).start();
        }
    }
}
```

You should get an output similar to this:

```
[...]
Counter: 998
Counter: 999
Counter: 1000
```

In this case, even if we call more than 1000 times, there are no problems accessing the `counter` variable.

5.21. Example19

In this Example 1000 threads increment a common static variable by 1000 units. The variable should return 1000000.

```
public class SharedVariable extends Thread {

    private static int counter;

    @Override
```

```
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter++;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            new SharedVariable().start();
        }
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("Counter value: " + counter);
    }
}
```

We will get an output similar to this:

```
Counter value: 993203
```

That is quite far from the expected value. This is because the increment operation with the unary operator ++ does not perform an atomic operation (which is executed without interruption), but consists of several steps and the execution can be interrupted at any of them, causing an inconsistency problem. memory

Assuming that the process of incrementing a variable by 1 is made up of the following steps:

- Reading the value of the variable.
- Increment by 1 of the value of the variable.
- Writing of the value of the variable.

If a thread reads the value of the variable being this 2, calculates the new value and before writing the resulting 3 another thread takes its place in the processor, it will read 2 again, calculate the new value that will be 3 and write it, when the first thread returns to the processor it will continue executing the third step and writing the value 3. Two threads have increased the value of the same variable by 1, but at the end only one of the increments has been reflected. This explains the increment losses shown in the example.

5.22. Example20

```
public class SimpleWaitNotify implements Runnable {

    private volatile boolean runningMethod1 = false;

    public synchronized void method1() {
        for (int i = 0; i < 10; i++) {
            System.out.printf("Method1: Running %d\n", i);
            if (i == 5) {
```

```
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public synchronized void method2() {
        for (int i = 10; i < 20; i++) {
            System.out.printf("Method2: Running %d\n", i);
        }
        this.notifyAll();
    }

    @Override
    public void run() {
        if (!runningMethod1) {
            runningMethod1 = true;
            method1();
        } else {
            method2();
        }
    }

    public static void main(String[] args) {
        SimpleWaitNotify swn = new SimpleWaitNotify();
        new Thread(swn).start();
        new Thread(swn).start();
    }
}
```

The output should be:

```
Running 0
Running 1
Running 2
Running 3
Running 4
Running 5
Running 10
Running 11
Running 12
Running 13
Running 14
Running 15
Running 16
Running 17
Running 18
Running 19
```

```
Running 6
Running 7
Running 8
Running 9
```

5.23. Example21

```java
public class Basic extends Thread {

    private int id;

    public Basic(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 3; i++) {
                System.out.printf("Thread %d. Interaction %d\n", id, i);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Basic thread1 = new Basic(1);
        Basic thread2 = new Basic(2);
        thread1.start();
        thread2.start();
    }
}
```

5.24. Example22

```java
public class BasicJoin extends Thread {

    private int id;
    private boolean suspend = false;
    private Thread referenceThread;

    public BasicJoin(int id) {
        this.id = id;
    }

    public void threadSuspend(Thread referenceThread) {
        this.suspend = true;
```

```
        this.referenceThread = referenceThread;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 3; i++) {
                if (suspend) {
                    referenceThread.join();
                }
                System.out.printf("Thread %d. Interaction %d\n", id, i);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        BasicJoin thread1 = new BasicJoin(1);
        BasicJoin thread2 = new BasicJoin(2);
        thread1.start();
        thread2.start();
        thread2.threadSuspend(thread1);
    }
}
```

5.25. Example23

```
public class MethodSyncronization implements Runnable {

    public synchronized void method1() {
        System.out.println("Method 1 start");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 1 ends");
    }

    public synchronized void method2() {
        System.out.println("Method 2 start");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 2 ends");
    }
```

```
22
23      @Override
24      public void run() {
25          method1();
26          method2();
27      }
28
29      public static void main(String[] args) {
30          MethodSyncronization ms = new MethodSyncronization();
31          new Thread(ms).start();
32          new Thread(ms).start();
33      }
34  }
```

The methods will be executed one at a time, even though there are two different threads available. The generated output is the following:

```
1  Method 1 start
2  Method 1 ends
3  Method 2 start
4  Method 2 ends
5  Method 1 start
6  Method 1 ends
7  Method 2 start
8  Method 2 ends
```

Remember that the order can vary, the important thing is that the methods are all executed one after the other.

If the methods were not synchronized (`MethodSyncronizationBis`) the two threads would simultaneously execute the first method, after its completion, the second. The output would have been the following:

```
1  Method 1 start
2  Method 1 start
3  Method 1 ends
4  Method 1 ends
5  Method 2 start
6  Method 2 start
7  Method 2 ends
8  Method 2 ends
```

5.26. Example24

```
1  public class WrongMethodSyncronization extends Thread {
2
3      public synchronized void method1() {
4          System.out.println("Method 1 start");
5          try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 1 ends");
    }

    public synchronized void method2() {
        System.out.println("Method 2 start");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 2 ends");
    }

    @Override
    public void run() {
        method1();
        method2();
    }

    public static void main(String[] args) {
        new WrongMethodSyncronization().start();
        new WrongMethodSyncronization().start();
    }
}
```

The output in this case will be:

```
Method 1 start
Method 1 start
Method 1 ends
Method 2 start
Method 1 ends
Method 2 start
Method 2 ends
Method 2 ends
```

5.27. Example25

```
public class SegmentSyncronization extends Thread {

    int id;
    static Object block1 = new Object();
    static Object block2 = new Object();

    public SegmentSyncronization(int id) {
        this.id = id;
```

```
    }

    public void method1() {
        synchronized (block1) {
            System.out.printf("Method 1 from thread %d start\n", id);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                return;
            }
            System.out.printf("Method 1 from thread %d ends\n", id);
        }
    }

    public void method2() {
        synchronized (block2) {
            System.out.printf("Method 2 from thread %d start\n", id);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                return;
            }
            System.out.printf("Method 2 from thread %d ends\n", id);
        }
    }

    @Override
    public void run() {
        if (id == 1) {
            method1();
            method2();
        } else {
            method2();
            method1();
        }
    }

    public static void main(String[] args) {
        new SegmentSyncronization(1).start();
        new SegmentSyncronization(2).start();
    }
}
```

The output will be similar to this:

```
Method 1 from thread 1 start
Method 2 from thread 2 start
Method 1 from thread 1 ends
Method 2 from thread 2 ends
Method 2 from thread 1 start
Method 1 from thread 2 start
Method 2 from thread 1 ends
Method 1 from thread 2 ends
```

5.28. Example26

```java
package UD02.Example26;

import java.util.concurrent.Semaphore;

public class BasicSemaphore implements Runnable {

    Semaphore semaphore = new Semaphore(3);

    @Override
    public void run() {
        try {
            semaphore.acquire();
            System.out.println("Step 1");
            Thread.sleep(1000);
            System.out.println("Step 2");
            Thread.sleep(1000);
            System.out.println("Step 3");
            Thread.sleep(1000);
            semaphore.release();
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }

    public static void main(String[] args) {
        BasicSemaphore sb = new BasicSemaphore();
        for (int i = 0; i < 5; i++) {
            new Thread(sb).start();
        }
    }
}

```

Output:

```
Step 1
Step 1
Step 1
```

```
Step 2
Step 2
Step 2
Step 3
Step 3
Step 3
Step 1
Step 1
Step 2
Step 2
Step 3
Step 3
```

The output shows that the first three threads that have tried to access the critical section via the semaphore lock have been executed concurrently, and the rest of the threads have been executed when the first ones have finished and released the locks.

6. Information sources

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos