**Async**

When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

That being said, in the context of computers this translates into executing a process or task on another "thread." A thread is a series of commands (a block of code) that exists as a unit of work.

Now, if you introduce multiple cores/processors into the mix, then things CAN actually happen at the same time.

SYNCHRONOUS

You are in a queue to get a movie ticket. You cannot get one until everybody in front of you gets one, and the same applies to the people queued behind you.

ASYNCHRONOUS

You are in a restaurant with many other people. You order your food. Other people can also order their food, they don't have to wait for your food to be cooked and served to you before they can order. In the kitchen restaurant workers are continuously cooking, serving, and taking orders. People will get their food served as soon as it is cooked.

* Event listeners:

# **Event and Listener (Java Event Handling)**

|  |
| --- |
| Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. |

### **Java Event Handling Code**

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

## Bevezetés

A Java grafikus felületein többféle esemény, többféle módon kezelhető. Az eseménykezeléshez használhatunk interfészeket és/vagy osztályokat. Az egyes megvalósítások lehetnek névtelenül.

* callbacks:

A callback is a piece of code that you pass as an argument to some other code so that it executes it. Since Java doesn't yet support function pointers, they are implemented as Command objects.

A callback method in java is a method that gets called when an event (call it E) occurs. Usually you can implement that by passing an implementation of a certain interface to the system that is responsible for triggering the event E

CallBack Function is a function which passed into another function as an argument and is expected to execute after some kind of event. The purpose of the callback is to inform a class Sync/Async if some work in other class is done. This is very useful when working with Asynchronous tasks. Suppose we want to perform some routine tasks like perform some operation or display content after some clicking a button, or fetching data from internet. This is also used in event handling, as we get notified when a button clicked via callback function.

In Java, Callbacks can be implemented using interface.

**Synchronous Callback**

The code execution will block or wait for the event before continuing. Until your event returns a response your program will not execute any further. So Basically the callback performs all its work before returning to the call statement. The problem with synchronous callback is that they appear to lag.

**Asynchronous Callback**

An Asynchronous call do not block the program from the code execution. when the call returns from the event the call returns back to the callback function. So in the context of java we have to Create a new thread invoke the callback method inside that thread. Callback may be invoked from a thread but is not a requirement. A Callback may also start a new thread thus making themselves asynchronous.

* http
* threads:

### Background process - Thread and Runnable

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Threading can ensure that while a process is running, other processes are not locked. For example, a web download program might need long time to complete a download task. If you don't put the download task in a thread, the user interface will be locked while the download process is still running. To avoid this problem you need to put the download task in **background process** or in a separated thread.  Another style to place the background process in a separated thread is to implement the **Runnable**interface. **Runnable interface:**  |  | | --- | | The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |  |  |  | | --- | --- | | 1. **public void run():**is used to perform action for a thread. | | | String getName() | | Retrieves the name of running thread in the current context in String format | | | void start() | | This method will start a new thread of execution by calling run() method of Thread/runnable object. | | | void run() | | This method is the entry point of the thread. Execution of thread starts from this method. | | | void yield() | | By invoking this method the current thread pause its execution temporarily and allow other threads to execute. | | | void join() | | This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution | | | boolean isAlive() | | This method will check if thread is alive or dead | | |

## Life Cycle of a Thread

Following are the stages of the life cycle −

* **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
* **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
* **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
* **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
* **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

## Thread Priorities

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

* background jobs, processes:

In [multitasking](https://en.wikipedia.org/wiki/Computer_multitasking) computer [operating systems](https://en.wikipedia.org/wiki/Operating_system), a **daemon** ([/ˈdiːmən/](https://en.wikipedia.org/wiki/Help:IPA/English) or [/ˈdeɪmən/](https://en.wikipedia.org/wiki/Help:IPA/English))[[1]](https://en.wikipedia.org/wiki/Daemon_(computing)#cite_note-jargon-1) is a [computer program](https://en.wikipedia.org/wiki/Computer_program" \o "Computer program) that runs as a [background process](https://en.wikipedia.org/wiki/Background_process), rather than being under the direct control of an interactive user. Traditionally, the process names of a daemon end with the letter *d*, for clarification that the process is, in fact, a daemon, and for differentiation between a daemon and a normal computer program. For example, [syslogd](https://en.wikipedia.org/wiki/Syslogd) is the daemon that implements the system logging facility, and sshd is a daemon that serves incoming [SSH](https://en.wikipedia.org/wiki/Secure_Shell)connections.

A **background process** is a [computer process](https://en.wikipedia.org/wiki/Process_(computing)) that runs *behind the scenes* (i.e., in the background) and without user intervention.[[1]](https://en.wikipedia.org/wiki/Background_process#cite_note-tlt-1) Typical tasks for these processes include logging, system monitoring, scheduling,[[2]](https://en.wikipedia.org/wiki/Background_process#cite_note-2) and user notification

A daemon is a type of background process designed to run continually in the background, waiting for event(s) to occur or condition(s) to be met.[[9]](https://en.wikipedia.org/wiki/Background_process#cite_note-9) These processes typically use minimal system resources and perform tasks which require little to no input from the user. When launched with the *daemon* function, daemons are disassociated from their parent terminal.[[10]](https://en.wikipedia.org/wiki/Background_process#cite_note-10)

* Examples:
  + User clicks on something, what happens
  + Sending emails:

Setup an Executor bean that uses a thread pool executor in your spring context and use it to enqueue a work item that will send the email. It will then be dispatched on a thread pool thread asynchronously and therefore your request thread will not block.

* + Calling external APIs:
  + Processing background tasks

**Expected skills (2/3)**

* Able to explain main difference between sync and async
* Able to demonstrate async piece of code
* Able to explain execution order line by line