**WHY CATS EFFECTS:**

Cats Effects is an add-on library to Cats.

Cats Effect is a high-performance, asynchronous, composable framework for building real-world applications in a purely functional style within the Typelevel (Cats) ecosystem. It provides a concrete tool, known as "the IO monad", for capturing and controlling actions, often referred to as "effects", that your program wishes to perform within a resource-safe, typed context with seamless support for concurrency and coordination. These effects may be asynchronous (callback-driven) or synchronous (directly returning values); they may return within microseconds or run infinitely.

Even more importantly, Cats Effect defines a set of typeclasses which define what it means to be a purely functional runtime system. These abstractions power a thriving ecosystem consisting of streaming frameworks, JDBC database layers, HTTP servers and clients, asynchronous clients for systems like Redis and MongoDB, and so much more! Additionally, you can leverage these abstractions within your own application to unlock powerful capabilities with little-or-no code changes, for example solving problems such as dependency injection, multiple error channels, shared state across modules, tracing, and more.

**STRUCTURE:**

Think of it as chaining together steps/effects to form a single IO, which will contain this sequence/chain of effects and optionally a final value (e.g., an Int). Then this single IO (which hasn’t been run yet) is returned to the run function and when the run function returns this IO, then it is actually run.

**FIBERS:**

Fibers are *the* fundamental abstraction in Cats Effect.  
They are lightweight threads (or "coroutines"), which represent a sequence of actions which will ultimately be evaluated in that order by the underlying hardware.

Compared to threads, fibers have a much smaller footprint and higher level of abstraction.  
A programme can create tens of millions of fibers within the same process, and creating & starting a new fiber is extremely fast.

Cats Effect also defines first-class support for asynchronous callbacks, resource handling, and cancelation (interruption) for all fibers.  
This means that any individual "step" of a fiber may be either *synchronous* in that it runs until it produces a value or errors, or *asynchronous* in that it registers a callback which may be externally invoked at some later point.  
Therefore with fibers, there is no difference between a callback and a return.

Example:  


All 3 methods produce the same result, but use different syntax to define the sequence.  
The ‘>>’ operator puts together two effects, ignoring the result of the first (but the side-effects of the 1st effect still occur)

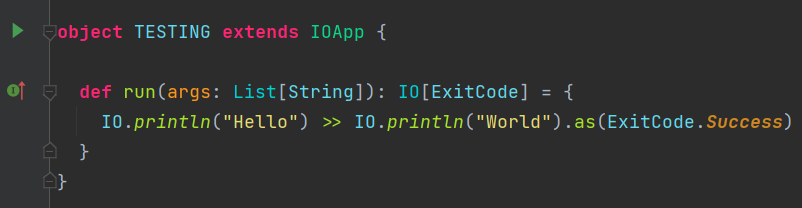
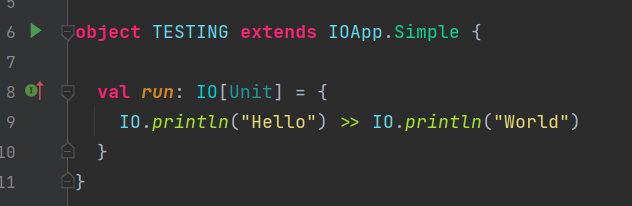
Each of these methods define the steps for a new fiber. Each step of a fiber is an effect, and composing steps together produces a larger effect, which can in turn continue to be composed.

Note: You must chain the IO fiber effects together (using 1 of the methods above) as you are just chaining processes together, as this whole chain is what needs to be returned at the end of the function. If not chained, then only the last chain of fiber effects will actually be returned and will produce a result (the rest will have no affect).  
If we directly used the code above, the effects from the #1 & #2 method won’t be chained to the #3 method, so only the #3 method part will actually be returned and ran (the rest will have no affect).

**Starting an IO Application:**

Every application has a “main” fiber that is run when the run function is called (similar to the “main thread” running when main is called in a normal Scala app).

To make a cats-effects app, the object must extend IOApp or IOApp.Simple and have a run method (similar to main function in normal Scala)



The difference is the non-simple IOApp must return an ExitCode value at the end of its run function.  
The *as()* method will convert the IO to the type in the IO argument: in this case it is ExitCode, but it could be String or Int).

Can also run a cats-effects app without extending IOApp, by calling *unsafeRunSync()* on a value / function that returns an IO (e.g *val run: IO[Unit]*, as below).  
However, this is unsafe and should use the IOApp method.  
 

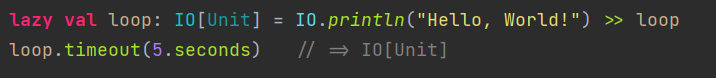
**Fibers interacting with each other:**

When one fiber starts another fiber, the 1st fiber is the "parent" of the second one.  
The parent can terminate before the child without causing any inconsistencies.  
Fibers can observe and recover from errors (using something like handleErrorWith or attempt). Fibers may also observe their own cancelation, but they cannot recover from it, meaning that they cannot continue executing after cancelation. Parent fibers may initiate cancelation in a child (via the cancel method), and can observe the final outcome of that child (which may be Canceled) and may continue executing after the child has terminated.

**Cancellation:**

By default, fibers are cancellable at any point during their execution. This means that unneeded calculations can be promptly terminated and their resources gracefully released as early as possible within an application.

In practice, fiber cancelation most often happens in response to one of two situations: timeouts and concurrent errors. An example of timeout is:



This constructs an IO which starts a fiber defined by loop. This fiber prints "Hello, World!" infinitely, and left alone, the loop fiber will run forever. However, the timeout function delays for 5 seconds, after which it calls cancel on the fiber, interrupting its execution and freeing any resources it currently holds (in this case, none).

This is a similar in concept to the Thread’s *interrupt* method in the Java standard library. Despite the similarity, there are some important differences which make cancelation considerably more robust, more reliable, and much safer:

* **It is cooperative**. When a fiber calls cancel on another fiber, it is a request to the target fiber to cancel when it is able to and the cancelling fiber asynchronously waits for cancelation to become possible. Once cancelation starts, the target fiber will run all of its finalizers (usually used to release resources such as file handles) before yielding control back to the canceler.
* **Cancelation can be suppressed within scoped regions**. If a fiber is performing a series of actions which must be executed atomically (i.e. either all actions execute, or none of them do), it can use the IO.uncancelable method to mask the cancelation signal within the scope, ensuring that cancelation is deferred until the fiber has completed its critical section. This is commonly used in conjunction with compound resource acquisition, where a scarce resource might leak if the fiber were to be cancelled "in the middle".
* **The fiber model offers more control and tighter guarantees around cancelation**, it is possible and safe to dramatically increase the granularity of cancelation within the target fiber. Every step of a fiber contains a cancelation check (checking whether the current fiber has been cancelled). This is how the loop fiber in the example above is cancelled despite the fact that the thread never blocks.

**TERMINOLOGY**

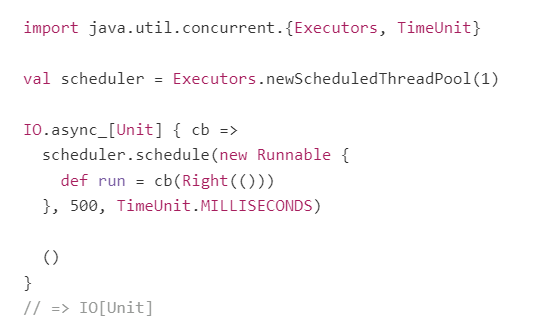
*(From* [*https://typelevel.org/cats-effect/docs/concepts*](https://typelevel.org/cats-effect/docs/concepts)*)*

**ASYNCHRONOUS VS SYNCHRONOUS:**

Asynchronous opposite of synchronous. They describe the manner in which a given effect produces a value.  
Synchronous effects are defined using *apply* (or *delay*, *blocking*, *interruptible* or *interruptibleMany*) and produce their results using return, or alternatively raise errors using *throw*.



*(NOTE: the above should really be defined using IO.interruptible since Thread.sleep blocks the underlying Thread, but we're using apply to illustrate the point)*

Asynchronous effects are defined using async (or async\_) and produce their results using a callback, where a successful result is wrapped in Right while an error is wrapped in Left:  


**Thread.sleep VS schedule effects:**  
- Have same semantics – delay for 500 milliseconds before allowing the next step in the fiber to take place.  
  
- However, Thread.sleep is synchronous while schedule is asynchronous.  
 Therefore Thread.sleep with block the Thread until its delay has expired, so that this Thread can’t  
 perform any more processes, which means that the resources for this Thread are not being utilised for  
 this duration (therefore resources are wasted).  
 Conversely, schedule returns immediately when run and simply invokes the callback in the future once  
 the given time has elapsed. This means that the underlying kernel Thread is not wasted and can be  
 repurposed to evaluate other work in the interim.

Asynchronous effects are more efficient than synchronous effects (whenever they are applicable, such as for network I/O or timers), but are harder to work with in real applications due to needing to manually manage callbacks and event listeners.  
Fibers eliminate this disadvantage due to their built-in support for asynchronous effects. In both of the above examples, the effect in question is simply a value of type IO[Unit], and from the outside, both effects behave identically. Thus, the difference between return/throw and a callback is encapsulated entirely at the definition site, while the rest of your application control flow remains entirely oblivious. This is a large part of the power of fibers.

It is critical to note that nothing in the definition of "asynchronous" implies "parallel" or "simultaneous", nor does it negate the meaning of "sequential" (remember: all fibers are sequences of effects). "Asynchronous" simply means "produces values or errors using a callback rather than return/throw". It is an implementation detail of an effect, managed by a fiber, rather than a larger fundamental pattern to be designed around.

**CONCURRENT:**

*Multiple tasks interleaved. Concurrency doesn't have to be multithreaded. We can write concurrent applications on single processor using methods such as event loops.*

*Example: Communicating with external services through HTTP.*

*Main concern: interaction with multiple, independent and external agents.*

Refers to two or more actions which are defined to be independent in their control flow. It is the opposite of "sequential". Critically, it is possible for things that are "concurrent" to evaluate sequentially if the underlying runtime decides this is optimal, whereas actions which are sequential will always be evaluated one after the other.

Concurrency is often conflated with asynchronous execution due to the fact that, in practice, the implementation of concurrency often relies upon some mechanism for asynchronous evaluation. But as noted above, asynchrony is just that: an implementation detail, and one which says nothing about concurrent vs sequential semantics.

Cats Effect has numerous mechanisms for defining concurrent effects. One of the most straightforward of these is parTupled, which evaluates a pair of independent effects and produces a tuple of their results:  


As with all concurrency support, parTupled is a way of declaring to the underlying runtime that two effects (callServiceA(params1) and callServiceB(params2)) are independent and can be evaluated in parallel. Cats Effect will never assume that two effects can be evaluated in parallel.

All concurrency in Cats Effect is implemented in terms of underlying primitives which create and manipulate fibers: start and join. These concurrency primitives are very similar to the equivalently-named operations on Thread, but as with most things in Cats Effect, they are considerably faster and safer.

**Structured Concurrency**

A form of control flow in which all concurrent operations must form a closed hierarchy.  
Conceptually, it means that any operation which forks some actions to run concurrently must forcibly ensure that those actions are completed before moving forward. Furthermore, the results of a concurrent operation must only be made available upon its completion, and only to its parent in the hierarchy. parTupled above is a simple example of this: the IO[(Response, Response)] is unavailable as a result until both service calls have completed, and those responses are only accessible within the resulting tuple.

Cats Effect has a number of structured concurrency tools: parTupled, parMapN, and parTraverse.  
It offers robust structured concurrency operators: background, Supervisor, and Dispatcher.  
Cats Effect uses structured concurrency by default, but structured concurrency can be limiting so Cats Effect does not prevent unstructured concurrency when it is needed.

In particular, fibers may be started without the caller being forced to wait for their completion. This low-level flexibility is sometimes needed, but is dangerous since it can result in fiber "leaks" (in which a fiber is started and all references to it outside of the runtime are abandoned). It is generally better to rely on structured (but very flexible) tools such as background or Supervisor.

Cats Effect provides a pair of general tools for modelling shared concurrent state: Ref and Deferred.  
These are fundamentally unstructured in nature, and can result in business logic which is difficult to follow. However, as with many powerful tools, they do have a time and place. These tools can be used to create powerful higher-level abstractions such as Queue, Semaphore, and such.

Overall, Cats Effect encourages structured concurrency and provides users with a large number of flexible tools for achieving it, but it does not prevent unstructured concurrent compositions such as start or Ref.

**PARALLEL:**

*Using multiple computational resources (like more processor cores) to perform a computation faster, usually executing at the same time.*

*Example: summing a list of Integers by dividing it in half and calculating both halves in parallel.*

*Main concern: efficiency.*

Similar to async execution, parallelism is an implementation detail of the runtime. When two things are evaluated in parallel, it means that the underlying runtime and hardware are free to schedule the associated computations simultaneously on the underlying processors. This concept is very related to that of concurrency in that concurrency is how users of Cats Effect declare to the runtime that things can be executed in parallel.

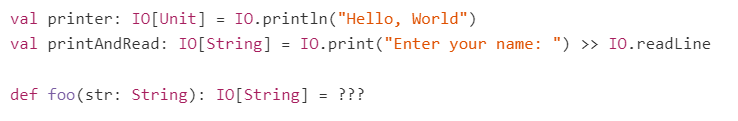
It is generally easier to understand the distinction between concurrency and parallelism by examining scenarios in which concurrent effects would not be evaluated in parallel. One obvious scenario is when the application is running on JavaScript rather than on the JVM. Since JavaScript is a single-threaded language, it is impossible for anything to evaluate in parallel, even when defined to be concurrent. Now, this doesn't mean that concurrency is useless on JavaScript, since it is still helpful for the runtime to understand that it doesn't need to wait for A to finish before it executes B, but it does mean that everything will, on the underlying hardware, evaluate sequentially.

A more complex example of non-parallel evaluation of concurrent effects can happen when the number of fibers exceeds the number of underlying Threads within the runtime. In general, Cats Effect's runtime attempts to keep the number of underlying threads matched to the number of physical threads provided by the hardware, while the number of fibers may grow into the tens of millions (or even higher on systems with a large amount of available memory). Since there are only a small number of actual carrier threads, the runtime will schedule some of the concurrent fibers on the same underlying carrier thread, meaning that those fibers will execute in series rather than in parallel.

As another implementation detail, it is worth noting that fibers are prevented from "hogging" their carrier thread, even when the underlying runtime only has a single thread of execution (such as JavaScript). Whenever a fiber sequences an asynchronous effect, it yields its thread to the next fiber in the queue. Additionally, if a fiber has had a long series of sequential effects without yielding, the runtime will detect the situation and insert an artificial yield to ensure that other pending fibers have a chance to make progress. This is one important element of fairness.

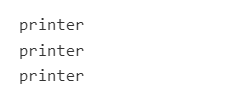
**EFFECTS:**

An effect is a description of an action (or actions) that will be taken when evaluation happens. One very common sort of effect is IO:

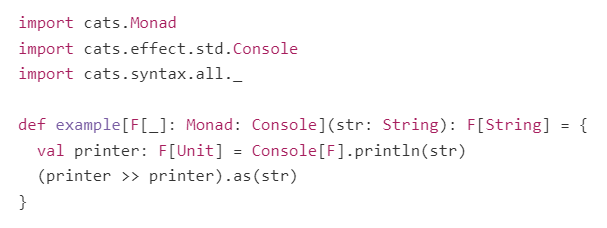


In the above snippet, printer and printAndRead are both effects: they describe an action (or in the case of printAndRead, actions plural) which will be taken when they are evaluated. foo is an example of a function which returns an effect. As a short-hand, such functions are often called "effectful": foo is an effectful function.

This is very distinct from saying that foo is a function which performs effects, in the same way that the printer effect is very distinct from actually printing (the function and values will just return / store the IO, which is the steps/effects that will be taken when the IO is ran).

This is illustrated neatly if we write something like the following:  


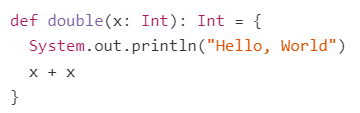
When this code is evaluated, the text "Hello, World" will be printed exactly zero times, since printer is just a descriptive value; it doesn't do anything on its own. Cats Effect is all about making it possible to express effects as values.

In advanced usage of Cats Effect, it is also common to use effect types which are not simply IO:

In the above, *example* is an effectful function, and *printer* is an effect (as is printer >> printer and (printer >> printer).as(str)).  
The effect type here is F, which is generic (so could be IO). Therefore, the caller of *example* is free to choose the effect at the call site, for example by writing something like example[IO]("Hello, World"), which in turn would return an IO[String].

**SIDE-EFFECTS**

When running a piece of code causes changes outside of just returning a value, we generally say that code "has side-effects". More intuitively, code where you care whether it runs more than once, and/or when it runs, almost always has side-effects. The classic example of this is System.out.println:

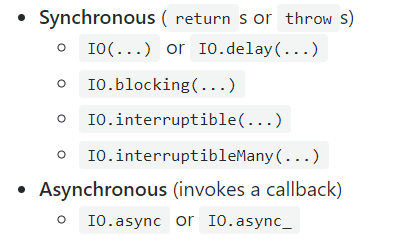


The double function takes an Int and returns that same Int added to itself and it prints "Hello, World" to standard out. This is what is meant by the "side" in "side-effect": something else is being done "on the side". The same thing could be said about logging, changing the value of a var, making a network call, etc.

Critically, a side-effect is not the same thing as an effect. An effect is a description of some action, where the action may perform side-effects when executed. The fact that effects are just descriptions of actions is what makes them much safer and more controllable. When a piece of code contains a side-effect, that action just happens. You can't make it evaluate in parallel, or on a different thread pool, or on a schedule, or make it retry if it fails. Since an effect is just a description of what actions to take, can freely change the semantics of how it eventually executes to meet the needs of your specific use-case.

Overall: Effects are the value that is contained in the F[?] or IO[?].  
Side-effects are any other effects that occur that aren’t this final value and that affect other parts of programme (aren’t purely functional).

In Cats Effect, code containing side-effects should always be wrapped in one of the "special" constructors. In particular:



When side-effecting code is wrapped in one of these constructors, the code itself still contains side-effects, but outside the lexical scope of the constructor we can reason about the whole thing (e.g. including the IO(...)) as an effect, rather than as a side-effect.

For example, we can wrap the System.out.println side-effecting code from earlier to convert it into an effect value:  


ALWAYS WRAP SIDE-EFFECT PRODUCING CODE!! – better optimisation.

**THREAD POOLS:**



* **Computation pool** – where the main / heavy processes & computations run on.
* **Event Dispatcher pool** – Where events to be run are added to, and this pool then decides when to run them and what resources to allocate to them (and then sends them to Computation pool to actually run)
* **Blocking IO pool** – Where processes are sent that are waiting for something to occur or have been delayed, so that they do not block the Computation pool.

*Cats Effect’s fibers handle this for you behind the scenes.*