

*These are the type classes that Cats Effects has.*

*Defer & MonadError are from Cats and the rest are from Cats Effects*

**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 type-classes 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.

**TYPE-CLASSESS OVERVIEW:**

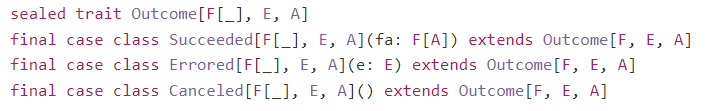
Broadly, the rules for functional effects can be broken down into the following categories:

* Resource safety and cancelation
* Parallel evaluation
* State sharing between parallel processes
* Interactions with time, including current time and sleep
* Safe capture of side-effects which return values
* Safe capture of side-effects which invoke a callback

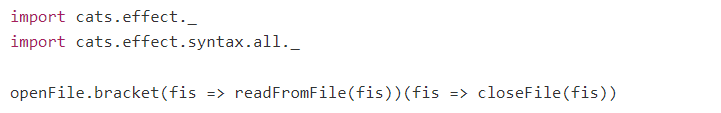
These are listed roughly in order of increasing power. For an object or type-class to have a specific effect, it must also have all the effects above this effect (as this effect’s implementation will inherit from less powerful effects).

**MONAD-CANCEL:**

A fiber can terminate in three different states, reflected by the different subtypes of Outcome:



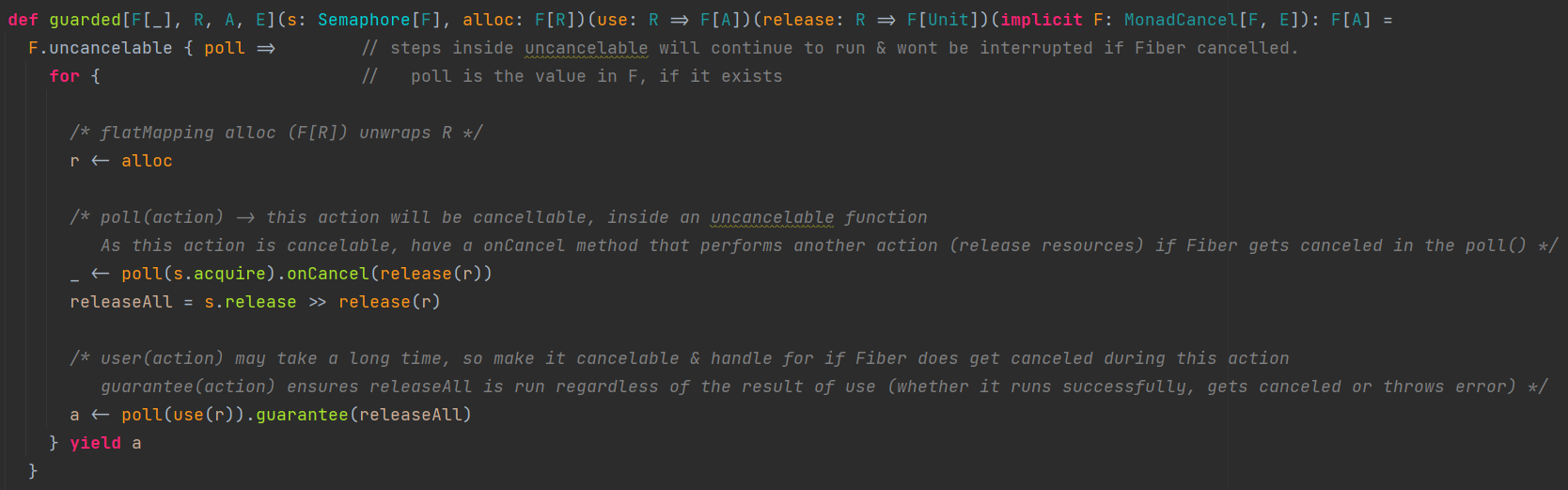
This means that when writing resource-safe code, we have to worry about cancelation as well as exceptions. The MonadCancel typeclass addresses this by extending MonadError (in Cats) to provide the capability to guarantee the running of finalizers when a fiber is canceled. Using it, we can define effects which safely acquire and release resources:



The bracket combinator works similar to try/finally:  
If openFile runs, then closeFile will be run, no matter what. This will happen even if readFromFile  
produces an error, or even if the whole process is canceled by some other fiber.

Additionally, openFile itself is atomic:  
it either doesn't evaluate at all (e.g. if the current fiber is cancelled prior to any of this even happening), or it fully evaluates. This allows openFile to do complicated things in the process of acquiring the resource without fear of something external getting in the way.

In addition to bracket, MonadCancel also provides a lower-level operation, uncancellable, which makes it possible to perform extremely complex, cancelation-sensitive actions in a safe and composable manner. For example, imagine that we have a block of code which must be guarded by a Semaphore, ensuring the runtime has exclusive access when evaluating. The problem here is that the acquisition of the Semaphore, which is a resource, may also result in blocking the fiber, and thus may need to be cancelled externally.  
Put another way: resource acquisition needs to be uncancellable, but this particular resource acquisition has a very specific point at which it needs to allow cancelation, otherwise it might end up locking up the JVM.

uncancelable provides a mechanism to achieve this: 

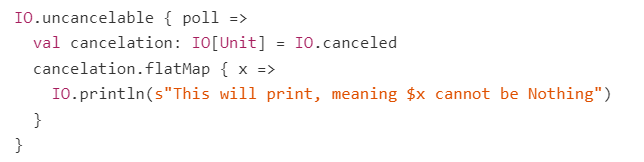
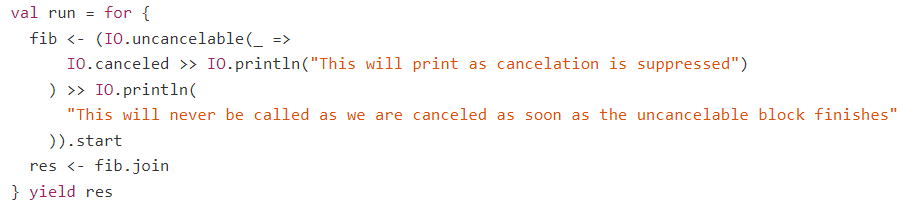
**Self-Cancelation:**

MonadCancel can self-cancel  


The above will result in a cancelled evaluation, and fa will never be run, provided that it isn't wrapped in an uncancelable block. If the above is wrapped in uncancelable, then the canceled effect will be ignored.

Self-cancelation is somewhat similar to raising an error with raiseError in that it will short-circuit evaluation and begin "popping" back up the stack until it hits a handler. Just as raiseError can be observed using the onError method, canceled can be observed using onCancel.

self-cancelation Vs raiseError:

* uncancelable suppresses canceled within its body unless polled.  
  For this reason, canceled has a return type of F[Unit] and not F[Nothing]  
    
    
  Note: cancelation will be observed as soon as uncancelable terminates, i.e. uncancelable only suppresses the cancelation until the end of its body, not indefinitely.  
  
* If you sequence an error with raiseError, it's always possible to use attempt or handleError to handle the error and resume normal execution.  
    
  No such functionality is available for cancelation, therefore cancelation can’t be undone.  
  It can be suppressed, but once it is observed, it must be respected by the canceled fiber.

Self-cancelation is intended for use-cases such as supervisor nets and other complex constructs which require the ability to manipulate their own evaluation in this fashion. It isn't something that will be found often in application code.

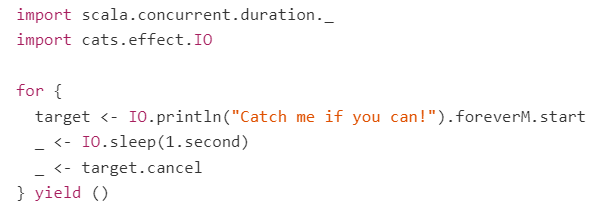
**SPAWN:**

Provides a lightweight Thread-like abstraction, Fiber, which can be used to implement parallel evaluation semantics. Fibers have extremely low overhead and optimized runtime mapping  
Much like Thread, Fiber is not often directly useful in user code, but instead is best when used as an implementation detail for higher-level functionality, such as the Parallel typeclass in Cats.

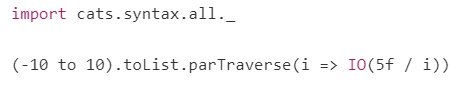
1st example in code…….

**CANCELATION:**

Unlike JVM Threads, Fibers are *cancelable*. So, can safely cancel a running fiber and it will clean up whatever resources it has allocated and bring itself to a halt in short order, ensuring that you don't have errant processes running in the background, eating up resources that could have otherwise been released.



In this example:  
- A fiber is created (target) and it runs indefinitely in a loop, printing "Catch me if you can!".  
- At the same time, the main fiber will sleep for 1 second, then after the main fiber will cancel the target fiber (so it stops printing)  
- Once the main fiber cancels target fiber, it should be practically instant until target fiber actually stops  
- Then the programme terminates.

  
  
*parTraverse* is a higher-level concurrency tool provided by Cats, backed by Spawn and Fiber behind the scenes.  
For each of the integer within the List, we create a new IO which uses that value as a divisor under the float 5f. The IO computes the result of this division, and since we're using a form of traverse, it will be evaluated and merged together into a single List inside of an outer IO. Thus, the result of this line is an IO[List[Float]].

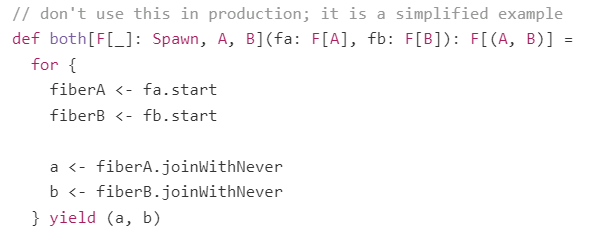
The par part of *parTraverse* means that, rather than performing each IO action in sequence (from left to right), it will spawn a new fiber for each action and run them all in parallel. This is usually a much nicer way of doing concurrency than manually fiddling around with start and cancel. It's still Fiber under the surface, but the API is much higher level and easier to work with.

One of these divisions will fail and an exception will be raised. When this happens, the result of the whole evaluation is discarded and the IO[List[Float]] will actually just produce the exception itself. Naturally, once any one of the constituent IOs has failed, there is no point in continuing to evaluate the other nineteen, and so their fibers are all immediately canceled.

A practical example of this auto-cancelation would be *parTraverseing* a long List of URLs, where each one was being fetched in parallel then failing fast and canceling all other actions on the first error.

**JOINING:**

Use *join* to when want to fork off a few fibers to perform some task, then wait for them to finish, accept their results, and move forward.



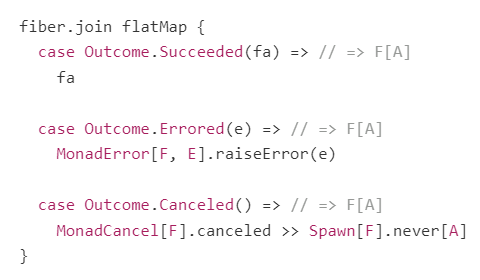
The *joinWithNever* function is a convenience method built on top of join, which is much more general. Fiber’s join method returns F[Outcome[F, E, A]] (where E is the error type for F).

Outcome has the following shape:

* Succeeded (containing a value of type F[A])
* Errored (containing a value of type E, usually Throwable)
* Canceled (which contains nothing)

These are the three possible termination states for a fiber, and by producing them within join, Cats Effect gives you the ability to react to each differently.

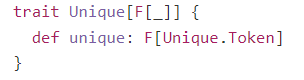
* If the child fiber completed successfully, produce its result
* If it errored, re-raise the error within the current fiber
* If it canceled, attempt to self-cancel, and if the self-cancelation fails, **deadlock**



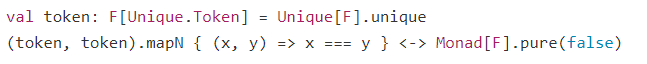
Sometimes this is an appropriate semantic, and the *joinWithNever* function implements it for you.

**UNIQUE:**

A typeclass which is a source of unique tokens



Each evaluation of unique is guaranteed to produce a value that is distinct from any other currently allocated tokens.

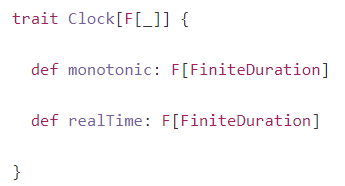


After a token becomes eligible for garbage collection, a subsequent evaluation of unique may produce a new token with the same hash. Similarly, the guarantee of uniqueness only applies within a single JVM runtime. If you need a token that is unique across all space and time, use a UUID instead.

Both Sync[F] and Spawn[F] extend Unique[F] as both typeclasses trivially have the ability to create unique values via delay(new Unique.Token()) and start respectively (fibers are always unique).

**CLOCK:**

A typeclass that provides effectful monotonic and system time analogous to *System.nanoTime()* and *System.currentTimeMillis()*



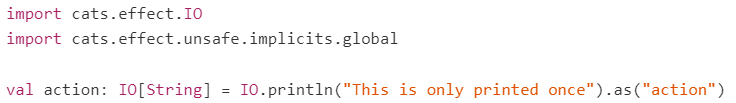
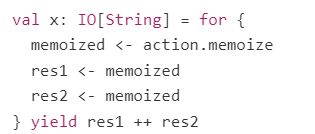
**CONCURRENT:**

This typeclass extends Spawn with the capability to allocate concurrent state in the form of Ref and Deferred and to perform various operations which require the allocation of concurrent state, including *memoize* and *parTraverseN*.  
*Ref* and *Deferred* are the concurrent primitives necessary to implement arbitrarily complicated concurrent state machines.

Concurrent brings the ability to cancel or start concurrently the side-effect in F

**MEMOIZATION:**

We can memoize an effect so that it's only run once and the result used repeatedly.  


Example:  
  
  


**Why *Ref* and *Deferred*?**

Generally, when implementing concurrent data structures, we need access to the following:

* A way of allocating and atomically modifying state -> Ref
* A means of waiting on a condition (semantic blocking) -> Deferred

Example: CountDownLatch, which is instantiated with n > 0 latches and allows fibers to semantically block until all n latches are released. We can model this situation with the following state

<https://typelevel.org/cats-effect/docs/typeclasses/concurrent>

**TEMPORAL:**

Temporal extends *Concurrent* with the ability to suspend a fiber by sleeping for a specified duration.



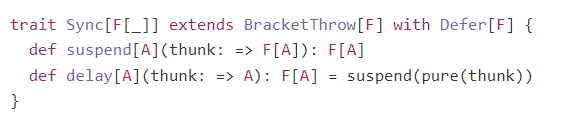
An alternative to this would be using *Sync[F].delay(Thread.sleep(duration)),* but this will block a thread from the compute pool, which is bad.  
Instead, *Temporal[F].sleep()* semantically blocks the execution of the calling fiber by de-scheduling it. Internally a scheduler is used to wait for the specified duration before rescheduling the fiber.

The ability to sleep for a specified duration enables us to define powerful time-dependent derived combinators like timeoutTo:

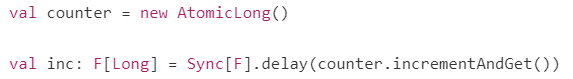


**SYNC:**

Sync is the synchronous FFI for suspending side-effectful operations. The means of suspension is dependent on whether the side effect you want to suspend is blocking or not.



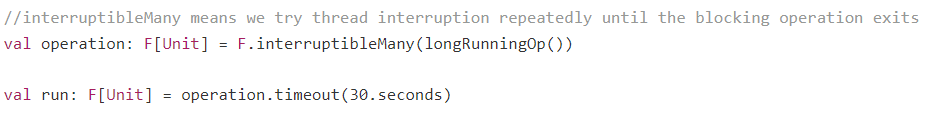
**METHODS OF SUSPENSION:**

If side effect is not thread-blocking then you can use *Sync[F].delay*  


If side effect is thread blocking then use *Sync[F].blocking*, which not only suspends the side effect but also shifts the evaluation of the effect to a separate thread-pool to avoid blocking the compute thread-pool. Execution is shifted back to the compute pool once the blocking operation completes.



A downside of thread-blocking calls is that the fiber executing them is not cancelable until the blocking call completes. If you have a very long-running blocking operation then you may want to suspend it using *Sync[F].interruptible* or *Sync[F].interruptibleMany* instead. This behaves the same as blocking but will attempt to interrupt the blocking operation via a thread interrupt in the event on cancelation.



**ASYNC:**

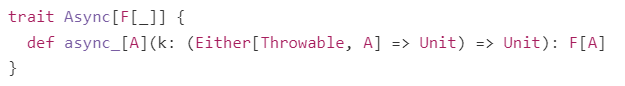
Async is the asynchronous FFI for suspending side-effectful operations that are completed elsewhere (often on another threadpool via a future-like API). This typeclass allows us to sequence asynchronous operations without stumbling into callback hell and also gives us the ability to shift execution to other execution contexts.

It is used to model datatypes that:

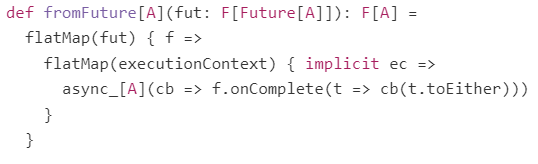
* Can start asynchronous processes.
* Can emit one result on completion.
* Can end in error

**ASYNC INTERFACE:**

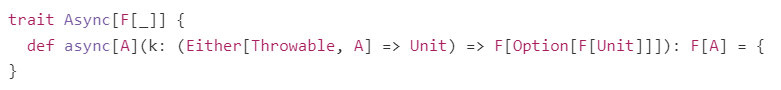
Asynchronous task: when the results of the task are computed somewhere else. We await the results of that execution by giving it a callback to be invoked with the result.  
That computation may fail hence the callback is of type *Either[Throwable, A] => ()*.  
This awaiting is semantic only - no threads are blocked, the current fiber is simply descheduled until the callback completes.

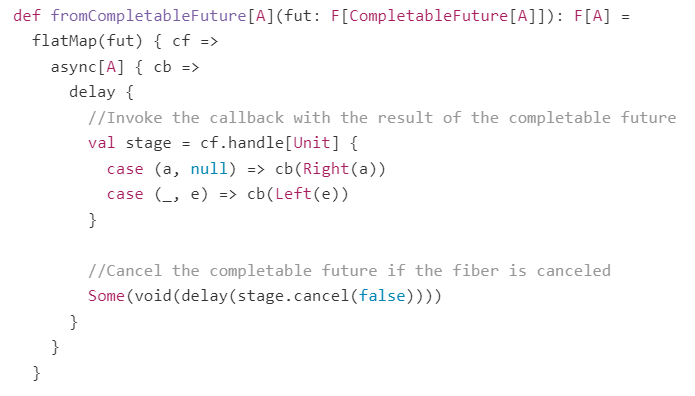
Simple asynchronous interface:  


*Async[F].fromFuture* (as shown below) is an example of this action and uses *Future.onComplete* to invoke the supplied callback

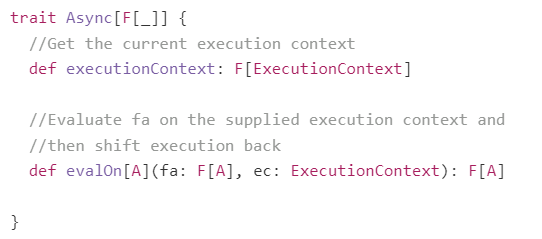


*async\_* is constrained, as can't perform any F effects in the process of registering the callback and can't register a finalizer to cancel the asynchronous task in the event that the fiber running async\_ is canceled.

Async therefore provides the more general async as well:  


It takes the same callback as before but can perform effects suspended in F. The *Option[F[Unit]]* allows us to return a finalizer to be invoked if the fiber is canceled.  
For example, here's a simplified version of Async[F].fromCompletableFuture  


**THREAD SHIFTING:**

Async has the ability to shift execution to a different thread pool.  


Works similar to Cats Reader monad:

* *executionContext* is like *ask* and gives us the current execution context.
* *evalOn* is like *local* and allows us to locally change the execution context for a given computation.
* After this computation is complete, execution will return to the context specified by *executionContext*

