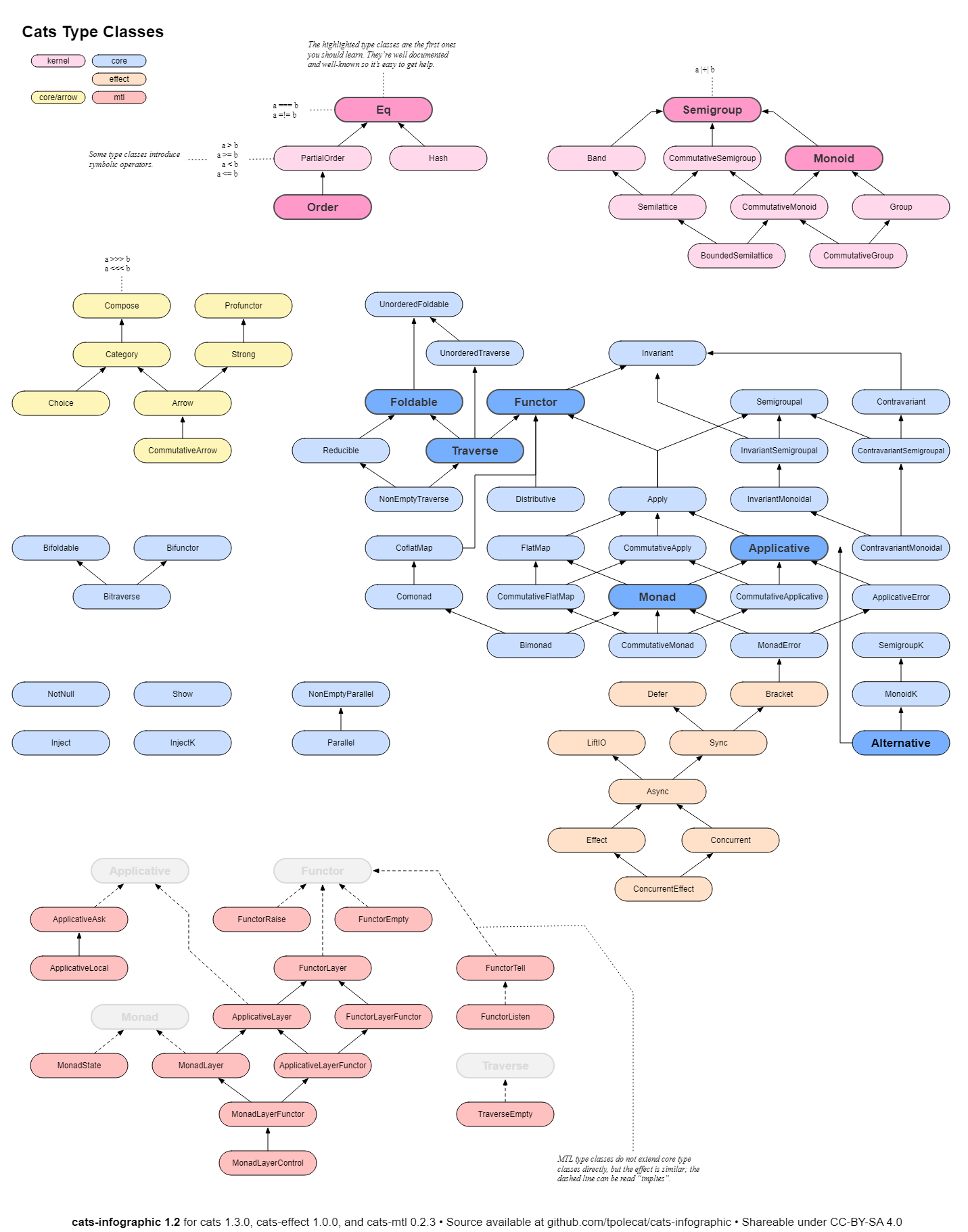
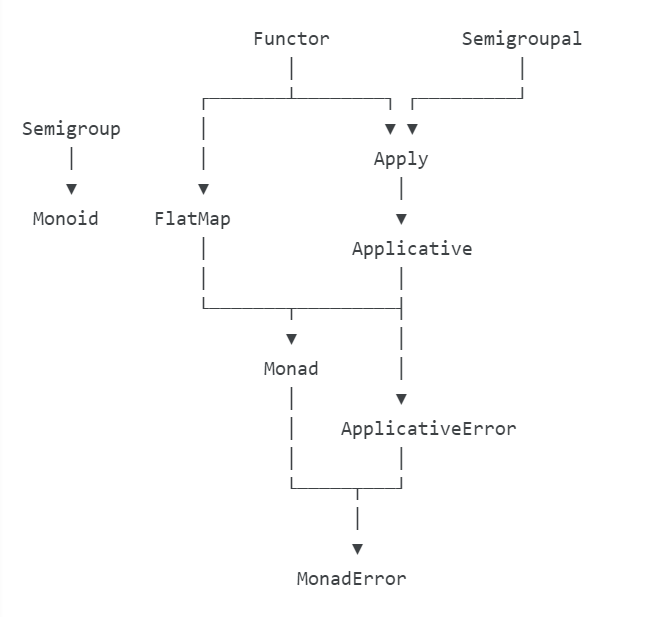
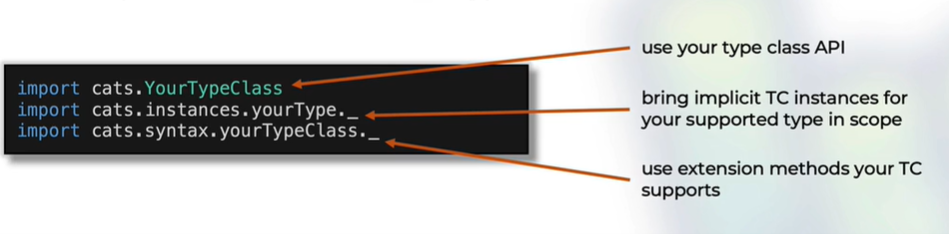
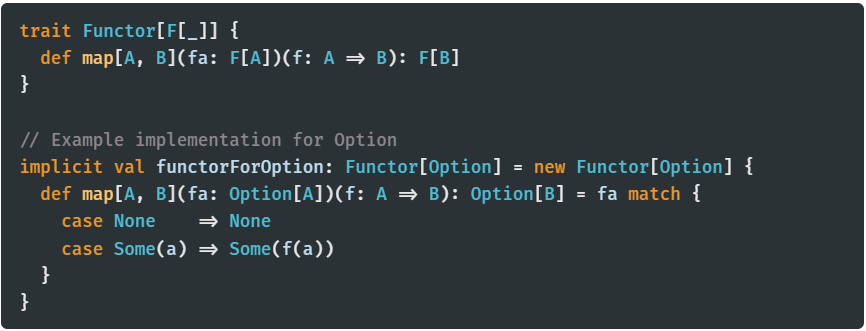
****



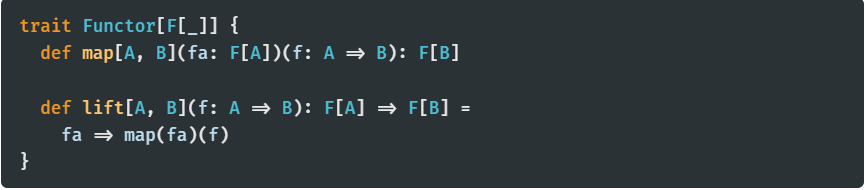
**TODO:  
Have separate sections for User-defined stuff (type-classes, implementations, extensions etc) and for Built-in Cats stuff (alias operations like |+|)**

**FUNCTORS:**

Type-class that abstracts over type constructors that can be map’ed over.  
E.g., of such type constructors are *List*, *Option*, and *Future*.



Another way of viewing a Functor[F] is that F allows the lifting of a pure function A => B into the effectful function F[A] => F[B]. This can be seen from the lift method (*which is just for demonstration*).



**FUNCTOR LAWS:**

* **Composition:**  
  Mapping with f and then again with g is the same as mapping once with the composition of f and g



* **Identity:**  
  Mapping with the identity function is a no-op (will return the initial value)



**MAP METHOD**

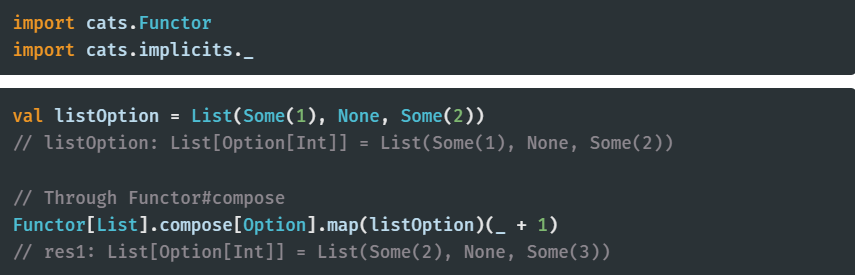
The map method in Functor has different implementations for different data-types

* **Functor[ Option[ A]]:**  
  With option, map will abstract away potentially missing values.  
  Map applies the function only if the Option has the Some case (and applies this function on the value inside the Some), but just returns None if the Option contains None.
* **Functor[ List[ A]]:**  
  With list, map will ‘traverse’ through the list’s elements and apply the function to each elem.  
  If list is empty, returns empty list

Examples…

**COMPOSE METHOD**

For nested data-types, such as *Option[List[A]]* or *List[Either[String, Future[A]]]*, use Functor’s compose method to ‘map’ over a nested list.  
if F and G have Functor instances, then so does F[G[\_]]

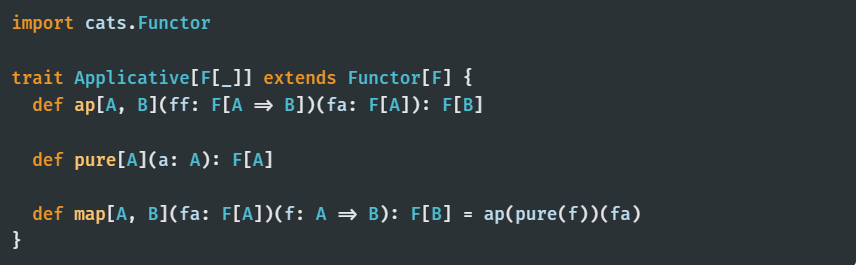


<https://typelevel.org/cats/typeclasses/functor.html>

…

**APPLICATIVE:**

Extends Functor, with an *ap* and *pure* method



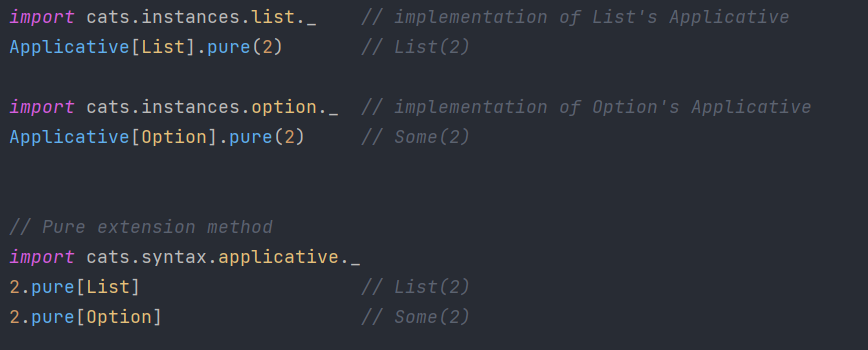
**APPLICATIVE LAWS:**

* **Associativity:**No matter the order in which these 3 values get product together, the result is isomorphic  
     
    
  With map, this can be made into an equality with  
  
* **Left Identity:**Zipping a value on the left with unit results in something isomorphic to the original value  
    
  As an equality (with map)
* **Right Identity:**Zipping a value on the right with unit results in something isomorphic to the original valueAs an equality (with map)  
  

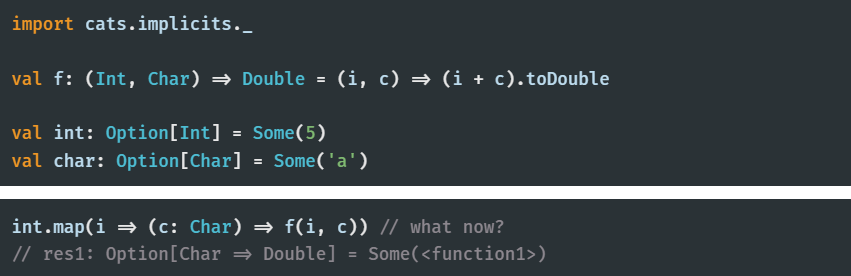
**EFFECT MANAGEMENT……**

**PURE METHOD**

Lifts a plain value to a Monadic type.  
E.g. Int -> Option[Int], String -> List[String]



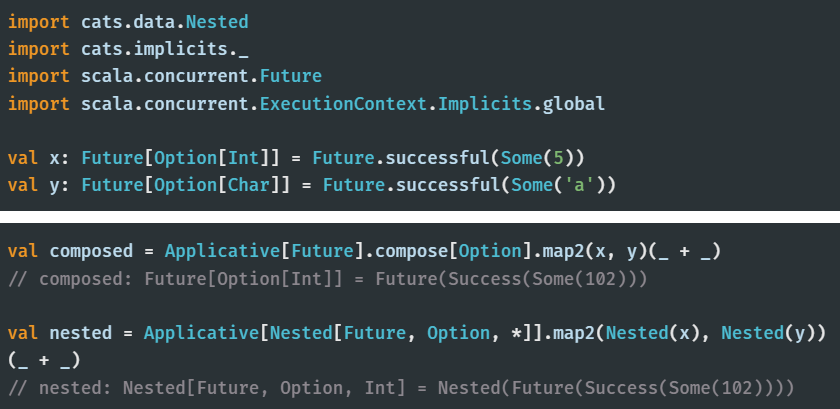
**AP METHOD**

Invokes/applies a wrapped function on a wrapped value and return a wrapped result  
E.g. ff: Option[Char => Double] and fa = Option[ Char], we want to apply ff() on fa  
  


Bfbfnfn………….

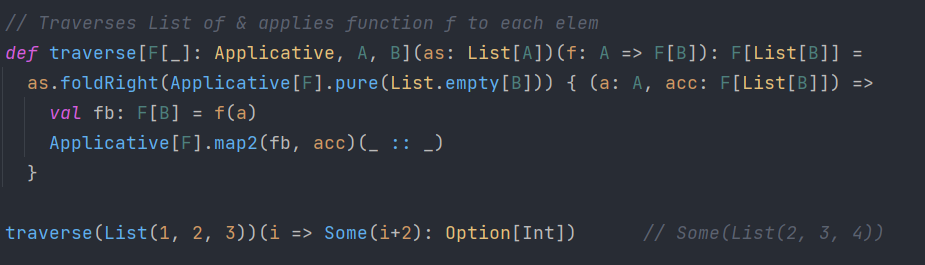
**COMPOSE METHOD**

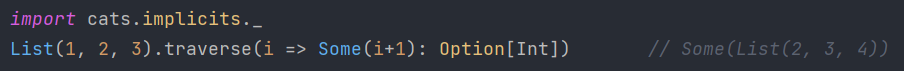
Like Functor, Applicatives compose. If F and G have Applicative instances, then so does F[G[\_]].



…

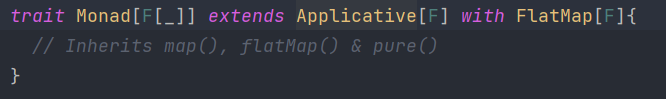
**TRAVERSE METHOD:**

User-created traverse function, for a List  


Cats has this built-in, which can be used by  


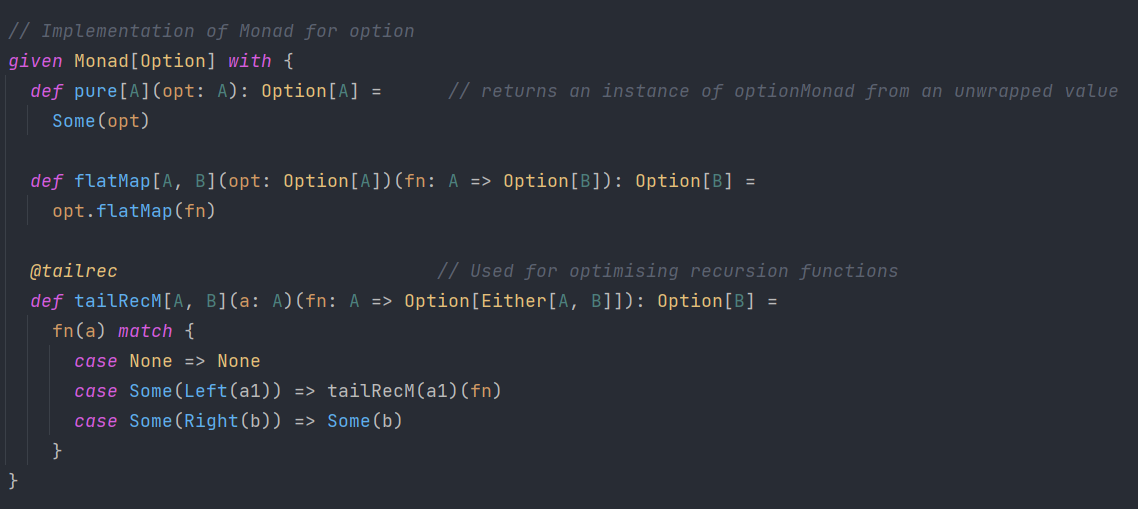
…. SHOULD BE IN OWN TYPE-CLASS

**MONAD:**



**IMPLEMENTATIONS OF MONAD FOR DIFFERENT TYPES:**

To implement an instance of Monad, need to define a pure(), flatMap() and tailRecM() method



**FLATMAP METHOD:**

The core function of Monad. Used in the implementations of map & flatten.  
FlatMap also used in for-comprehensions in Scala.

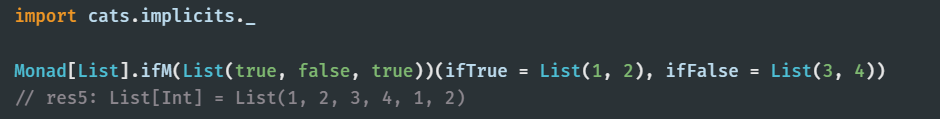
…

**TAILRECM METHOD:**

Encodes a stack-safe monadic recursion (stack-safe as it won’t lead to stack-overflow from recursion, due to optimisations in the Scala compiler)

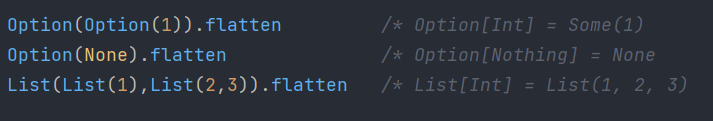
**IFM METHOD:**

To choose later operations in a sequence based on the results of earlier ones.  
ifM lifts an if statement into the monadic context.



**FLATTEN METHOD:**

Takes a value in a nested context (e.g. F[F[A]] where F is the context) and "joins" the contexts together so that we have a single context (ie. F[A]).



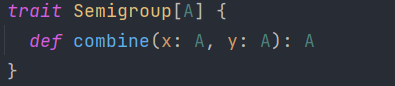
**COMPOSITION:**

Unlike Functors and Applicatives, not all Monads compose. Therefore, even if M[\_] and N[\_] are both Monads, M[N[\_]] is not guaranteed to be a Monad.

However, many common cases do. One way of expressing this is to provide instructions on how to compose any outer monad (F in the following example) with a specific inner monad (Option in the following example).

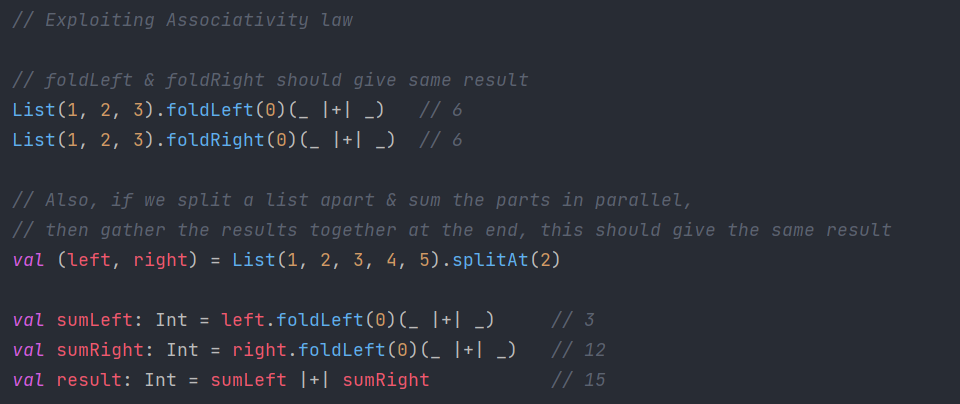
Use OptionT……….

**SEMIGROUP:**



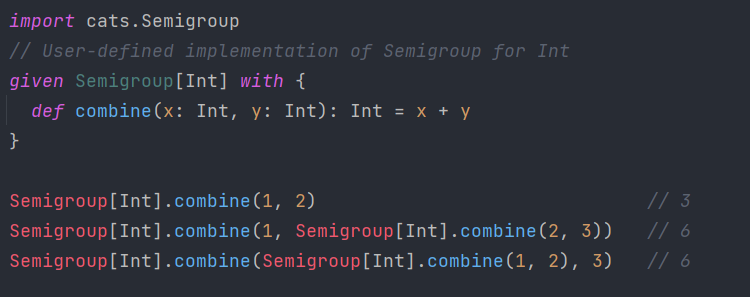
**SEMIGROUP LAWS:**

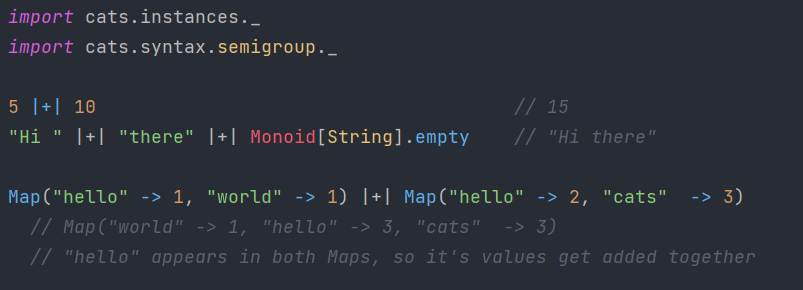
* **Associativity:**  
  The following equality must hold for any choice of x, y, and z

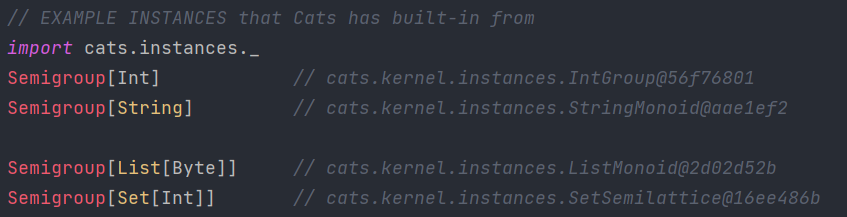
**COMBINE METHOD:**

Combines/merges 2 elements together.  
E.g. For Int, combine will add elements together.  
 For Strings, combine will concatenate the Strings together to 1 String  
 For Lists, combine will put the elements of the 2 Lists together into 1 List

  
  
Imports for easier syntax. |+| is an alias operation for combine()



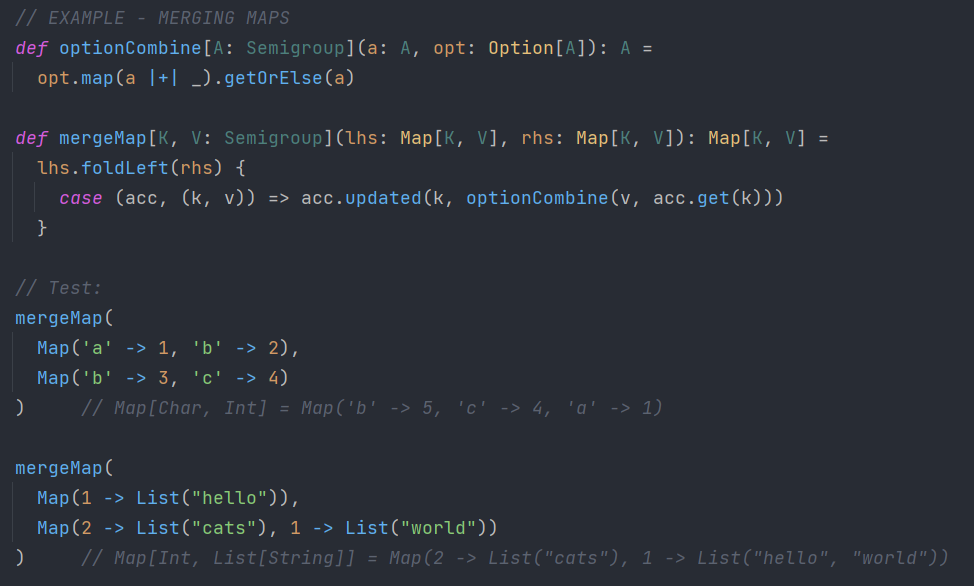
Cats provides loads of implementations of all type-classes for different data-types.  
Examples of some Cats built-in implementations for Semigroup are below.



**EXAMPLE: Merging Maps**

Function that merges two Maps that combines values if they share the same key. It is straightforward to write these for Maps with values of type say, Int or List[String], but we can write it once and for all for any type with a Semigroup instance.

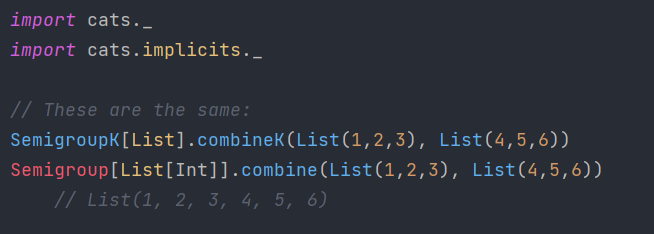
Cats has this built-in with the |+| operator, but below is our implantation of it.



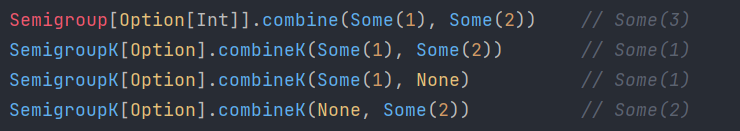
**SEMIGROUP\_K:**

SemigroupK has a very similar structure to Semigroup, the difference is that SemigroupK operates on type constructors of one argument.  
For example, Semigroup is for types which are fully specified like Int or List[Int] or Option[Int], SemigroupK is for type constructors like List and Option.  
These types are type constructors in that you can think of them as "functions" in the type space. You can think of the List type as a function which takes a concrete type, like Int, and returns a concrete type: List[Int].

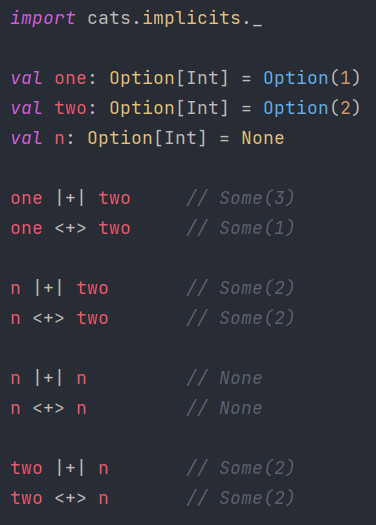
**FOR LIST:**Both Semigroup’s combine & SemigroupK’s combineK both concatenate the list (are equivalent)



**FOR OPTION:**  
comine & combineK differ.  
As SemigroupK doesn’t know the inner type of the Option (unlike Semigroup), this inner type could be anything, so combineK uses a “universal implementation” for all types inside Option.  
combineK method uses the *orElse* method of Option.  
Whereas combine knows the type inside Option, so will sum for Option[Int] or concatenate for Option[String]…

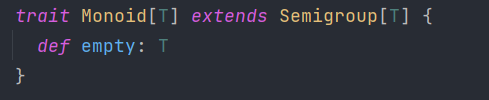


There is inline syntax available for both Semigroup & SemigroupK.  
|+| operator is from semigroup, <+> operator is from SemigroupK

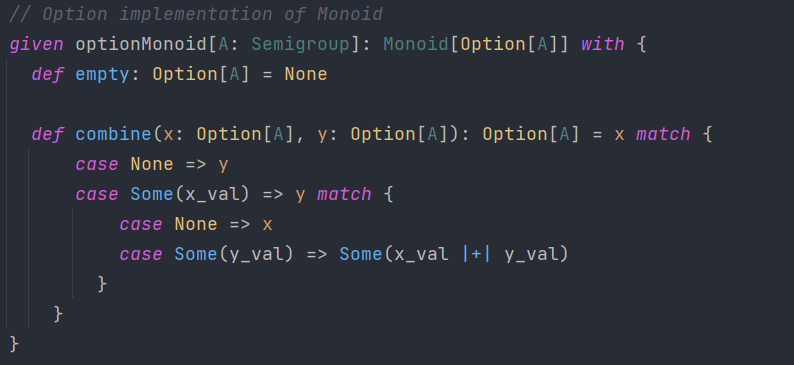
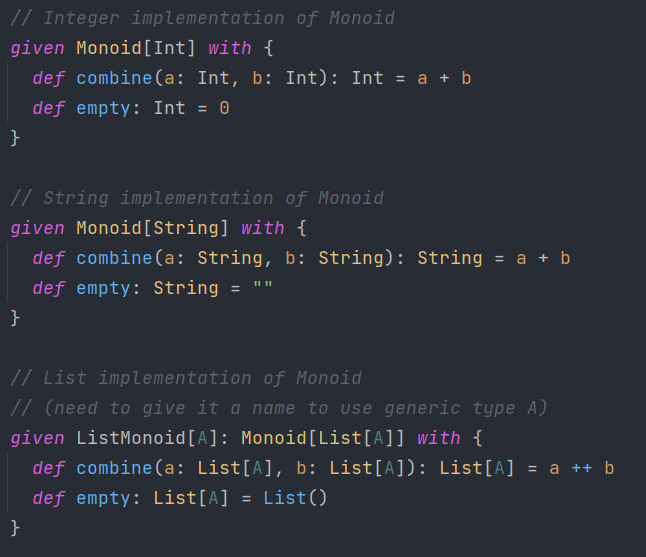


Note: We use these operators on Option[2] not Some(2) or None, as this will throw compiler error.

**MONOID:**

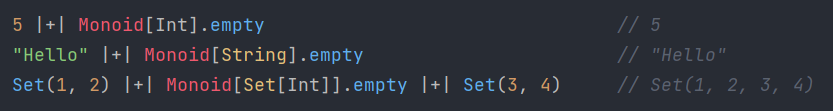


**User-defined implementations of Monoid (Cats also has these built-in)**



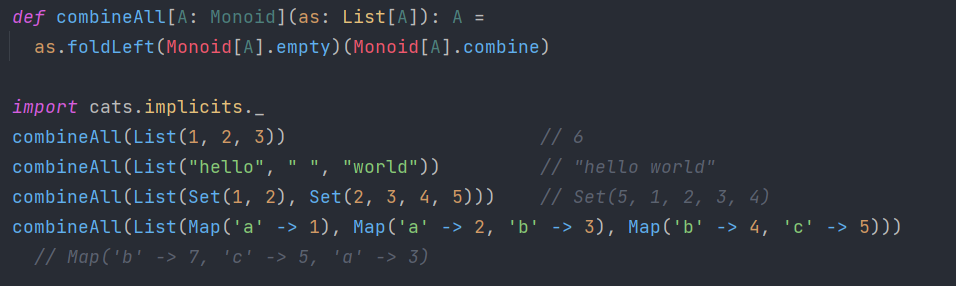
**MONOID LAWS:**

* **Identity:**  
  The empty method acts as the identity.  
  Combining a value with the empty value will return the original value

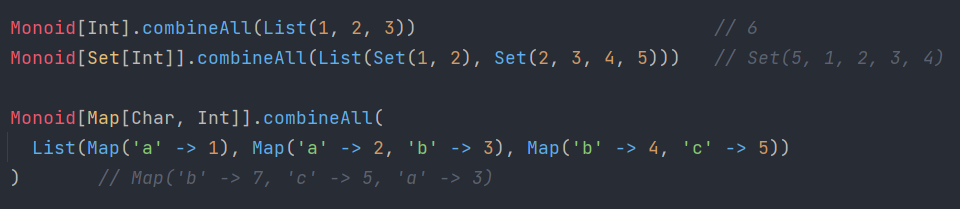


**COMBINE\_ALL METHOD:**

Given a list, will ‘combine’/fold the elements in List down to a single element  
Our user-defined implementation of combineAll method

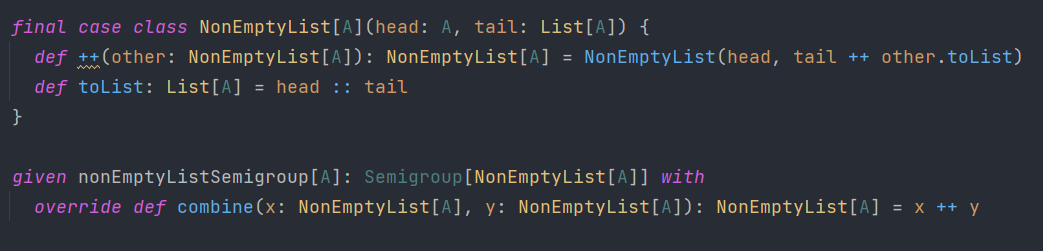


Cats also has this method built-in

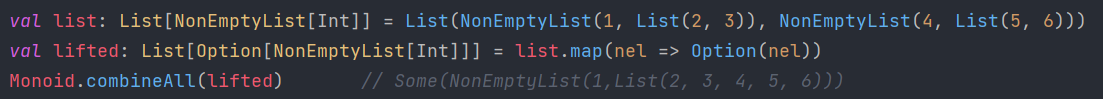


**MONOID FUNCTIONALITY FOR SEMIGROUPS – Option Monoid**

Some types can from Semigroup but not a Monoid. For example, NonEmptyList type can form a Semigroup through *++*, but there is no corresponding identity element to form a monoid.



Solution to this is to use the Option Monad as a wrapper on NonEmptyList (the type that isn’t a Monoid)



This lifting and combining of Semigroups into Option is provided by Cats as *Semigroup.combineAllOption*

**MONOID\_K:**

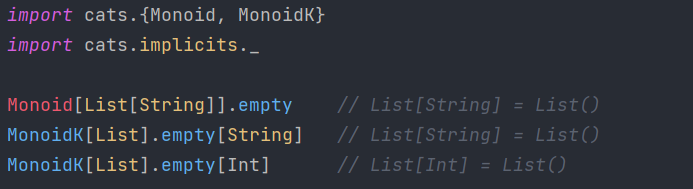
A universal monoid which operates on type constructors of one argument.

Similar to SemigroupK, this type class is useful when its type parameter F[\_] has a structure that can be combined for any particular type (e.g. List, Option), and this container type has no specified type inside.

Here's how to distinguish Monoid and MonoidK:

* Monoid[A] allows A values to be combined, and also means there is an "empty" A value that functions as an identity.
* MonoidK[F] allows two F[A] values to be combined, for any A. It also means that for any A, there is an "empty" F[A] value. The combination operation and empty value just depend on the structure of F, but not on the structure of A.

**EMPTY METHOD**  
Just like Monoid, MonoidK has an empty method

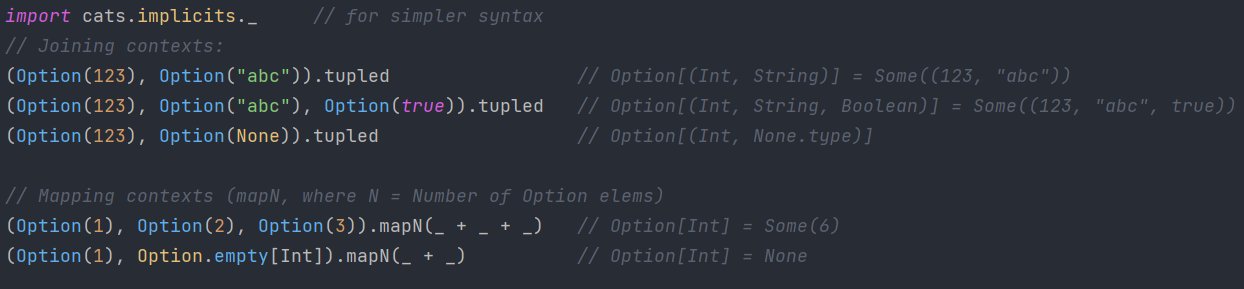


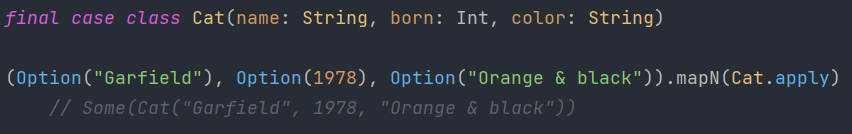
**COMBINE\_K METHOD**Works the same as Semigroup’s combineK method



**SEMIGROUPAL:**

tupled -> Converts the inputs into a single tuple  
mapN -> Maps all the inputs through a single function & returns its output  
 *(with adding Options together using mapN, if 1 element is Empty then final result is None)*



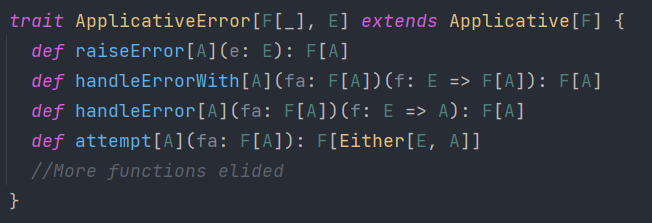


**Parallel Versions - parTupled and parMapN:**

Regular tupled/mapN evaluate their effects from left to right ("sequentially"), while parTupled/parMapN evaluate in an indeterminate order, or in parallel. The parallel variants enable you to compose tuples of tasks into a single task that will run its sub-tasks concurrently.

Used in Cats-Effects Parallelism**APPLICATIVE ERROR:**

ApplicativeError extends Applicative to provide error handling for types that represent an exception or an error (E.g. Either[E, A] or Try )



**PURE METHOD:**

Carries out the calculation / process as if it was a success (no error).  
The return type depends on type of F[\_]  
E.g. If F[\_] is an Either -> Right(), If F[\_] is a Try -> Success()

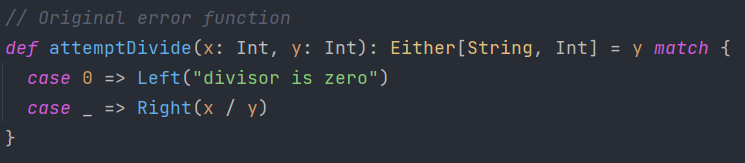
**RAISE ERROR METHOD:**

Generate the specific error type depending on what F[\_] represents.  
E.g. If F[\_] is an Either -> Left(), if F[\_] is a Validation -> Invalid.

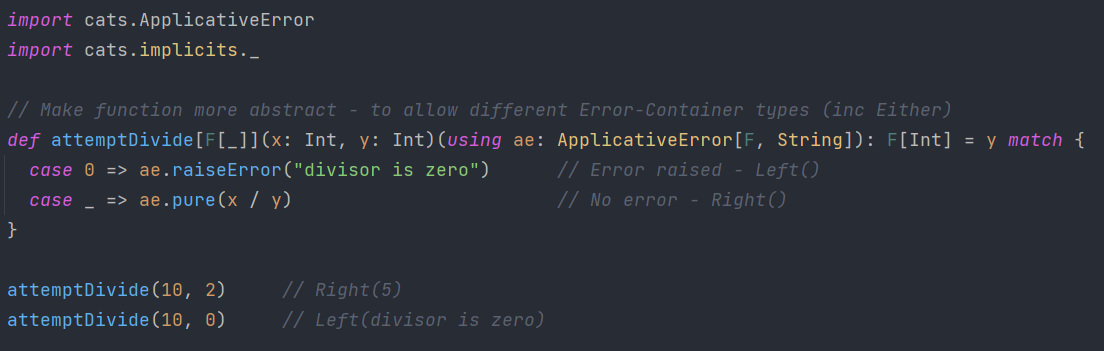
**INHERITS APPLICATIVE’S METHODS:**

Can use all methods of Applicative, e.g. tupled & mapN

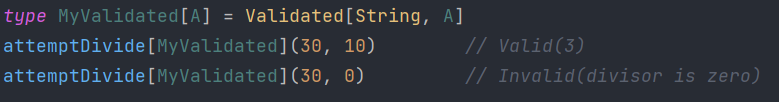
**EXAMPLE:**



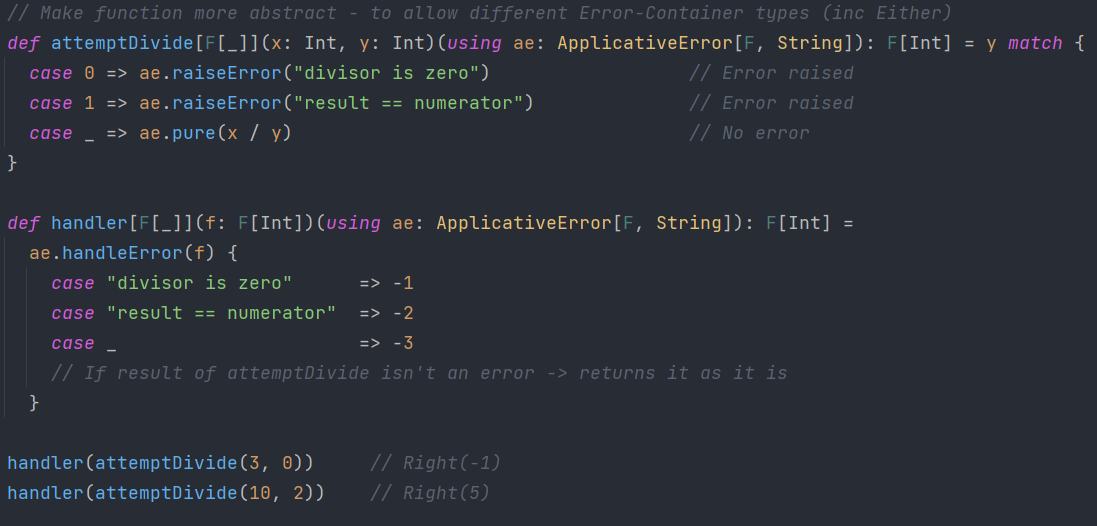
This function only works with Either. Make it more abstract using Cats ApplicativeError



By default, ApplicativeError uses type Either for the error-container type.  
Need to specify if using another error-container.

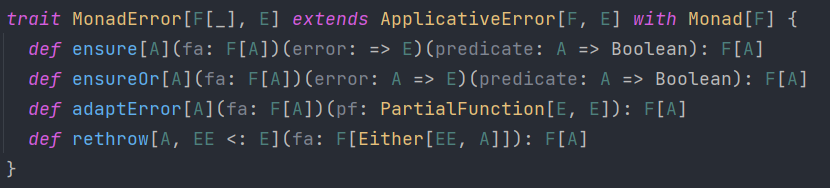


Error handling –   
Can have a wrapper/handler function that deals with any errors raised.  
handleError() only affects raiseError() values, so allows pure() values to ‘pass through’ to the output



**MONAD ERROR:**

Main use of MonadError over ApplicativeError is that MonadError has the flatMap method (inherited from Monad), so can be used in for-comprehensions.



**Example:**

*getCityClosestToCoordinate()* contains ‘throws’ an exception (inside MonadError) if the input co-ords are invalid, else will return a hard-coded success value (“Minneapolis, MN”).  
*getTemperatureByCity()* receives this output and ‘performs’ extra calculations/processes on it & returns the temp (in this case, it is hardcoded as 78). However, if the input to this func is a MonadError containing an error, then it will just return it as it is.



**FOLDABLE:**

Foldable type class instances can be defined for data structures that can be folded to a summary value. (An abstraction over things that can be folded).

For collections (E.g., List or Vector), these methods will fold together (combine) the values contained in the collection to produce a single result. Most collection types have foldLeft methods, which will usually be used by the associated *Foldable[\_]* instance.

*Foldable[F]* is implemented in terms of two basic methods:

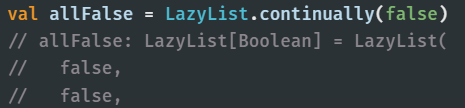
* *foldLeft(fa, b)(f)* -> eagerly performs a left-associative fold over fa.  
  
* *foldRight(fa, b)(f)* -> lazily performs a right-associative fold over fa.  
  

Therefore, *foldRight* is used in recursion & is stack-safe (unlike foldLeft)

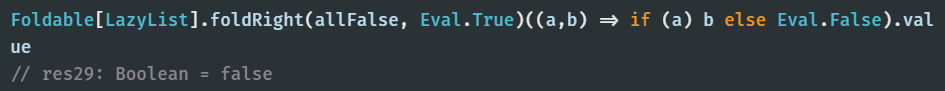
**FoldLeft Vs FoldRight:**

Given a list like List(1, 2, 3). Sum the numbers of this list using folds where 0 is the starting value (b) and integer addition (+) is the combination operation (f).  
Since *foldLeft* is left-associative, the execution of this fold would look something like ((0 + 1) + 2) + 3. The execution of a similar foldRight-based solution would look something like 0 + (1 + (2 + 3)).  
In this case, since integer addition is associative, both approaches will yield the same result. However, for non-associative operations (e.g., subtraction & division), the two methods can produce different results.

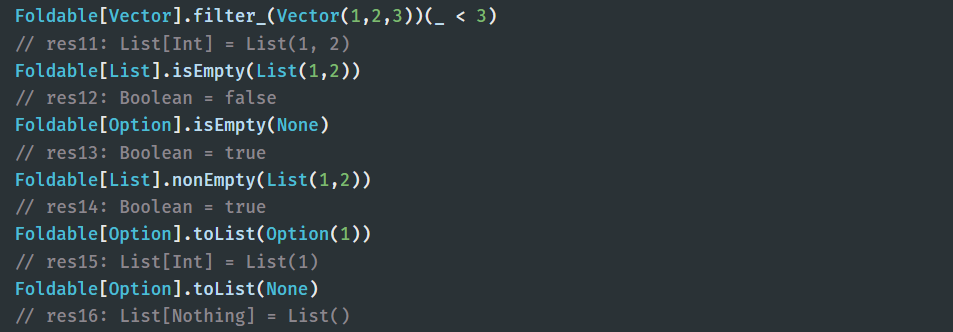
**FoldRight from Cats Vs from Standard Library:**



This ‘allFalse’ val has an infinite number of false. To fold this to a single false Boolean using ‘&&’:

* Using Standard Library foldRight -> leads to memory error as it goes through each ‘false’ item in allFalse
* Using Cat’s foldRight from Foldable -> Calculation terminates after finding the 1st ‘false’ item, as this foldRight is lazily evaluated & stack-safe

**Using Foldable:**



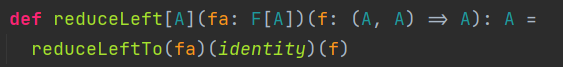
**REDUCIBLE:**

Reducible extends the Foldable type class with additional reduce methods.

The Scala Standard Library has reduceLeft &reduceOption defined in Scala's standard collections.  
Reducible offers these methods with the guarantee that the collection won't throw an exception due to a collection being empty, without having to reduce to an Option. This can be utilized effectively to derive maximum and minimum methods from Reducible instead of the maximumOption and minimumOption found on Foldable.

In essence, reduce is like a non-empty fold, requiring no initial value. This makes Reducible very useful for abstracting over non-empty collections such as NonEmptyList or NonEmptyVector.

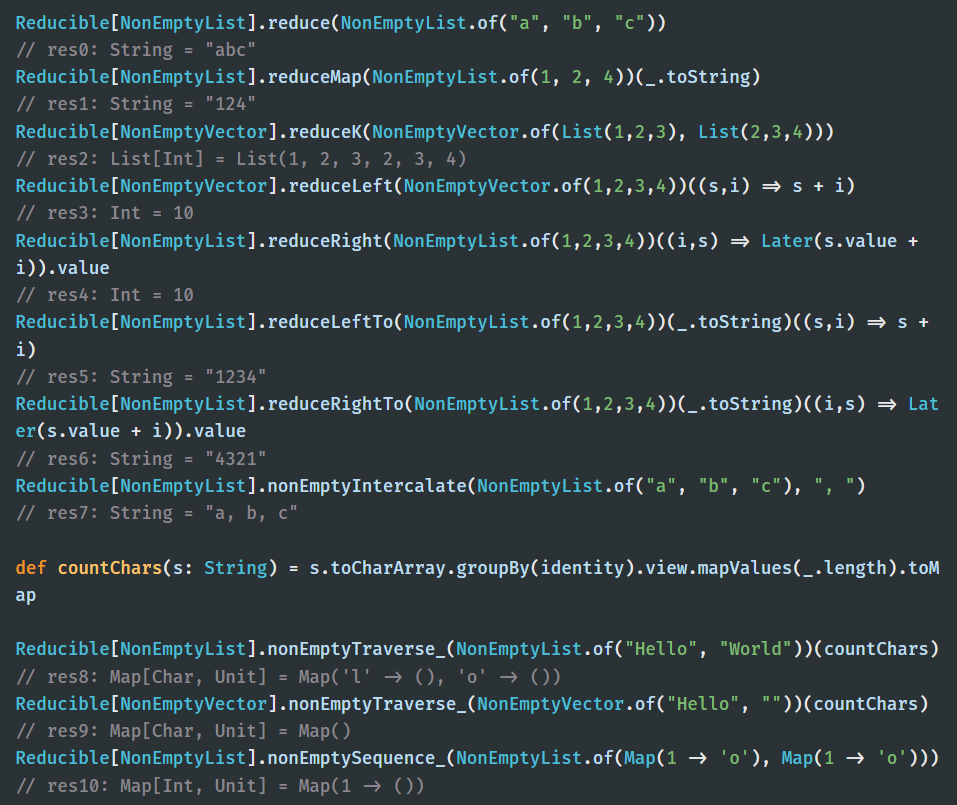
Analogous to the Foldable type class, Reducible[F] is implemented in terms of two basic methods in addition to those required by Foldable:

* *reduceLeftTo(fa)(f)(g)* -> eagerly reduces with an additional mapping function  
  
* *reduceRightTo(fa)(f)(g)* -> lazily reduces with an additional mapping function  
  

Now, because Reducible does not require an empty value, the equivalent of fold and foldMap, reduce and reduceMap, do not require an instance of Monoid, but of Semigroup.

Furthermore, just like with foldRight, reduceRight uses the Eval data type to lazily reduce the collection.

**Using Reducible:**

****

nonEmptyTraverse\_ and nonEmptySequence\_:

* Require Semigroup (instead of Monoid)
* Require Apply instead of Applicative.

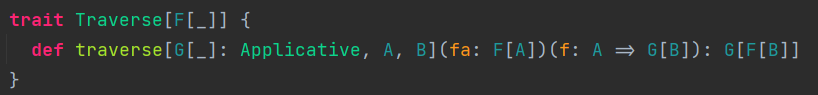
Also, Reducible offers a reduceLeftM method, that is just like foldM, but requires a FlatMap instance of a Monad instance.

**TRAVERSE:**

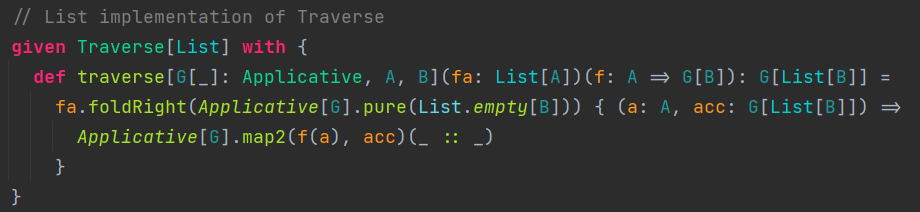
For datatypes that are Applicatives.

An abstraction over ‘things that can be traversed over’

Traverses through each element in a container (F[\_]) and applies a function ( A => G[B] ) to each element.  
The output will be a List of the new transformed elements inside a container F[\_].

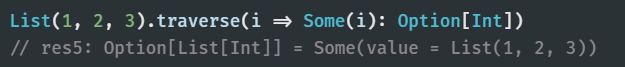


Implementations of Traverse for different Applicative objects:  
(The List & Option implementation is built-in to cats, but the custom Tree implementation isn’t)





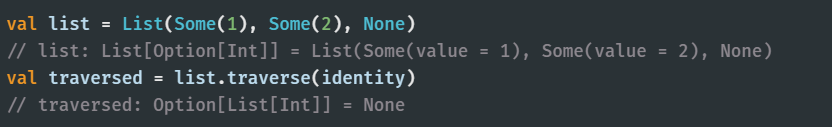
**Using traverse** (Convert List -> Option[List]):



Traversables are also Foldables.

**SEQUENCE:**

Traversing a Traversable object that has effectful values already (e.g. List[Option[A]]) with identity will turn the traversable "inside out."



This is the same as the *sequence* method



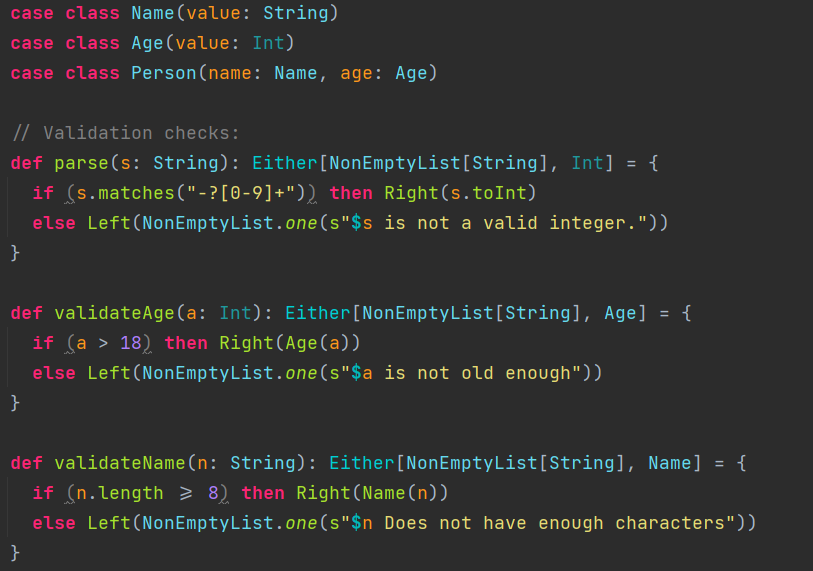
In general *t.map(f).sequence* can be replaced with *t.traverse(f)*.

**PARALLEL:**

Some data types within Cats have the same structure as a Monad, but instead are instances of Applicative (e.g., Either and Validated).

This is because defining a Monad instance for data types like Validated would be inconsistent with its error-accumulating behaviour. In short, Monads describe dependent computations and Applicatives describe independent computations.

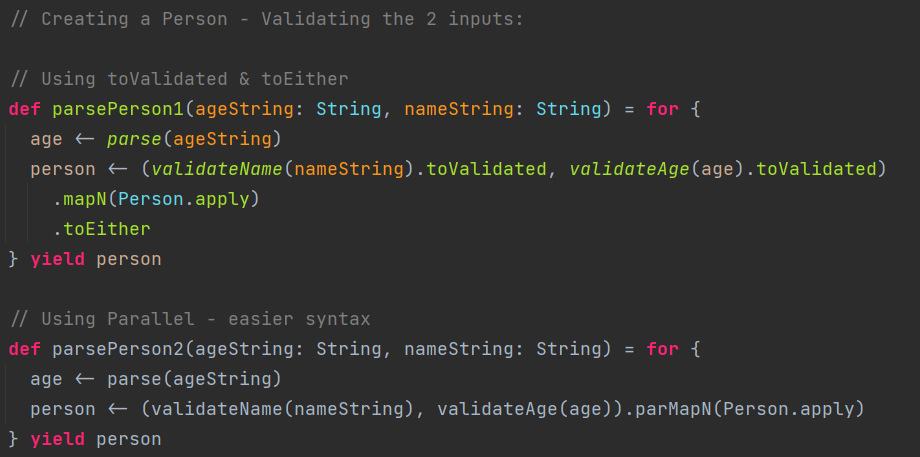
Sometimes however, we want to use both in conjunction with each other. In the example of Either and Validated it is trivial albeit cumbersome to convert between the two types.  
Below is a short example of a situation where we might run into this. For simplicity, we'll use String as our type to represent errors.



Then create a function to create a new Person, whilst validating for the different paramters Person() requires.

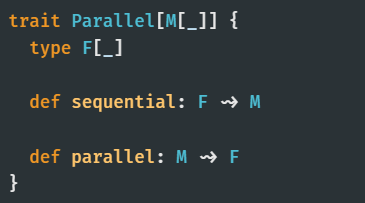
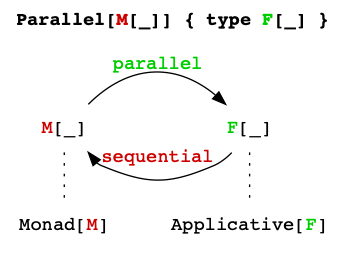
Without using parallel (in parsePerson1), converted to and from Validated manually. While this is still manageable, it gets worse the more Eithers we want to combine in parallel.

To mitigate this pain, Cats introduces a type class Parallel that abstracts over Monads which also support parallel composition. This is used in parsePerson2.

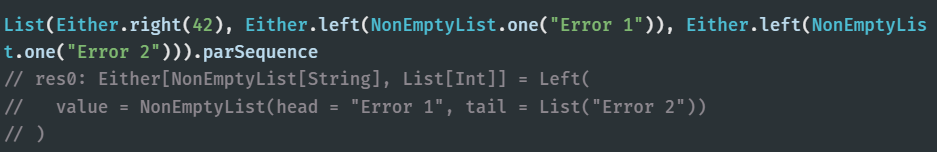


**Structure of Parallel:**

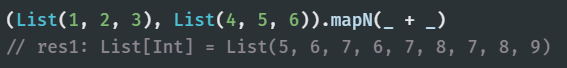
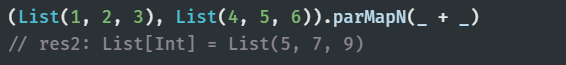
Abstracts over Monads which also support parallel composition.  
It is simply defined in terms of conversion functions between the two data types,  
where *M[\_]* is an instance of Monad, *F[\_]* an instance of Applicative and *~>* is an alias for FunctionK:

**Using Parallel:**

Can traverse over a Traverse using Parallel:  


Parallel also zips collections together.  
For the Applicative instance for List, Vector, etc:

* *mapN* results in the cartesian product of the individual collections  
  (adds the 1st index of the 1st collection with each element in the 2nd collection, then adds the 2nd index of the 1st collection with each element in the 2nd collection…)  
  
* *parMapN* traverses through the individual collections in parallel  
  (adds the 1st index of each collection, then adds then 2nd index of each collection…)  
  

Resources:

Rock the JVM…