**Wrapped values**

A wrapped value is a value that is contained within a data structure or a type constructor

data Maybe a = Nothing | Just a

Just 5 is a wrapped value where 5 is unwrapped inside the Just constructor of the Maybe type.

[1, 2, 3] is a list where the integers 1, 2, and 3 are wrapped inside the list structure.

**Unwrapped Values**

An unwrapped value is the raw value itself, not contained within any additional structure. Using the previous examples:

* The unwrapped value of Just 5 is 5.
* The unwrapped values within the list [1, 2, 3] are 1, 2, and 3
* Just 10 is a wrapped value.
* Nothing is also a wrapped value but represents the absence of a value.
* The unwrapped value within Just 10 is 10.

**Type class**

Type class is abstract class in C++, genric interface, having different functions set of funcitons

**Instance :**

A type becomes an instance of a type class when it provides implementations for the functions defined by the type class.

Ager hm uski clas ki hi implementation krde

**Common Type Classes in Haskell**

* **Eq**: Types that can be compared for equality.
* **Ord**: Types that have an ordering.
* **Functor**: Types that can be mapped over.
* **Monad**: Types that support binding and sequencing of operations.

Type class is powerful tool for writing the generic, reuseable code work for different types.

In Haskell, you don't typically convert all values into wrapped values. Instead, you use wrapped values to handle optional or context-dependent situations while keeping the actual values in their natural form. You work with both wrapped and unwrapped values as needed, using pattern matching, functions like mapMaybe, and other techniques to manipulate and process them accordingly.

When working with wrapped values, you often need to apply functions to the unwrapped values inside them. This is where type classes like Functor, Applicative, and Monad come into play.

* Functor, Applicative, and Monad are type classes that provide abstractions for working with values in different contexts.
* They enable code reuse, compositionality, and concise expression of complex computations.
* Each type class builds on the previous one, extending the capabilities of the abstraction. For example, Monad extends Applicative, which extends Functor.

### Functor

The Functor type class abstracts over types that can be mapped over, applying a function to the values inside the context. It provides the fmap function, which allows you to transform the values while preserving the context.

When you need to operate on the actual values, you extract them from the wrappers as needed.

For example, if you have a list of Maybe values representing optional integers, you might want to filter out the Just values and sum the integers:

#### Why Functor?

* **Abstraction**: Provides a uniform way to apply a function to values in different contexts (e.g., Maybe, lists, trees).
* **Code Reusability**: Allows you to write functions that work with any type that is a Functor, promoting code reuse.
* **Composition**: Enables function composition over contexts using fmap, leading to concise and modular code.

**Functor : function + functor value (list ,maybe,either)**

* The Functor type class provides a way to apply a function to values inside a data structure while preserving its structure.
* It has a single function fmap that applies a function to each element inside the functor.
* Functors must satisfy the identity and composition laws to ensure correct behavior.
* Examples of functors include lists, Maybe, trees, and many others.

**Laws**

fmap (g . h) = fmap g . fmap h

fmap id = id

Tree

Node 1

/ \

Node 2 Node 3

/ \ / \

Empty Empty Empty Empty

Node (f 1)

/ \

Node (f 2) Node (f 3)

/ \ / \

Empty Empty Empty Empty

* **Applicative**: Extends Functor by allowing you to apply functions wrapped in a context to values wrapped in the same context. It introduces pure to lift values into the context and <\*> to apply wrapped functions to wrapped values.

By understanding and using these concepts, you can work with complex data types and abstract over different kinds of computations in a uniform way.