# What is a higher kinded type in Scala?

Asked 13 years, 6 months ago  Modified 10 months ago  Viewed 82k times

309

You can find the following on the web:

1. **Higher kinded type == type constructor?**

   ```
   class AClass[T]{...} // For example, class List[T]
   ```

   Some say this is a higher kinded type, because it abstracts over types which would be compliant with the definition.

   > *Higher kinded types* are types which take other types and construct a new type

   These though are also known as *type constructor*. (For example, in Programming in Scala).

2. **Higher kinded type == type constructor which takes type constructor as a type parameter?**

   In the paper Generics of a Higher Kind, you can read

   > ... types that abstract over types that abstract over types ('higher-kinded types') ..."

   which suggests that

   ```
   class XClass[M[T]]{...} // or

   trait YTrait[N[_]]{...} // e.g. trait Functor[F[_]]
   ```

   is a higher kinded type.

So with this in mind, it is difficult to distinguish between *type constructor*, *higher kinded type* and *type constructor which takes type constructors as type parameter*, therefore the question above.

scala  generics  types  higher-kinded-types  type-constructor

Share  Improve this question

Follow

edited Jun 13, 2020 at 23:50

Mario Galic

48.4k ● 6 ● 62 ● 108

asked Jun 5, 2011 at 23:50

Lutz

4,725 ● 4 ● 19 ● 23

Added Landei's Functor as example. – Lutz Jun 6, 2011 at 8:43

## 5 Answers

Sorted by: Highest score (default)

315

Let me make up for starting some of this confusion by pitching in with some disambiguation. I like to use the analogy to the value level to explain this, as people tend to be more familiar with it.

> A type constructor is a type that you can apply to type arguments to "construct" a type.
>
> A value constructor is a value that you can apply to value arguments to "construct" a value.

Value constructors are usually called "functions" or "methods". These "constructors" are also said to be "polymorphic" (because they can be used to construct "stuff" of varying "shape"), or "abstractions" (since they abstract over what varies between different polymorphic instantiations).

In the context of abstraction/polymorphism, first-order refers to "single use" of abstraction: you abstract over a type once, but that type itself cannot abstract over anything. Java 5 generics are first-order.

The first-order interpretation of the above characterizations of abstractions are:

> A type constructor is a type that you can apply to proper type arguments to "construct" a proper type.
>
> A value constructor is a value that you can apply to proper value arguments to "construct" a proper value.

To emphasize there's no abstraction involved (I guess you could call this "zero-order", but I have not seen this used anywhere), such as the value `1` or the type `String`, we usually say something is a "proper" value or type.

A proper value is "immediately usable" in the sense that it is not waiting for arguments (it does not abstract over them). Think of them as values that you can easily print/inspect (serializing a function is cheating!).

A proper type is a type that classifies values (including value constructors), type constructors do not classify any values (they first need to be applied to the right type arguments to yield a proper type). To instantiate a type, it's necessary (but not sufficient) that it be a proper type. (It might be an abstract class, or a class that you don't have access to.)

"Higher-order" is simply a generic term that means repeated use of polymorphism/abstraction. It means the same thing for polymorphic types and values. Concretely, a higher-order abstraction abstracts over something that abstracts over something. For types, the term "higher-kinded" is a special-purpose version of the more general "higher-order".

Thus, the higher-order version of our characterization becomes:

> A type constructor is a type that you can apply to type arguments (proper types or type constructors) to "construct" a proper type (constructor).
>
> A value constructor is a value that you can apply to value arguments (proper values or value constructors) to "construct" a proper value (constructor).

Thus, "higher-order" simply means that when you say "abstracting over X", you really mean it! The `X` that is abstracted over does not lose its own "abstraction rights": it can abstract all it wants. (By the way, I use the verb "abstract" here to mean: to leave out something that is not essential for the definition of a value or type, so that it can be varied/provided by the user of the abstraction as an argument.)

Here are some examples (inspired by Lutz's questions by email) of proper, first-order and higher-order values and types:

```
                    proper    first-order           higher-order

values              10        (x: Int) => x         (f: (Int => Int)) => f(10)
types (classes)     String    List                  Functor
types               String    ({type λ[x] = x})#λ   ({type λ[F[x]] =
F[String]})#λ
```

Where the used classes were defined as:

```
class String
class List[T]
class Functor[F[_]]
```

To avoid the indirection through defining classes, you need to somehow express anonymous type functions, which are not expressible directly in Scala, but you can use structural types without too much syntactic overhead (the `#λ`-style is due to https://stackoverflow.com/users/160378/retronym afaik):

In some hypothetical future version of Scala that supports anonymous type functions, you could shorten that last line from the examples to:

```
types (informally) String    [x] => x              [F[x]] => F[String]) // I
repeat, this is not valid Scala, and might never be
```

(On a personal note, I regret ever having talked about "higher-kinded types", they're just types after all! When you absolutely need to disambiguate, I suggest saying things like "type

constructor parameter", "type constructor member", or "type constructor alias", to emphasize that you're not talking about just proper types.)

ps: To complicate matters further, "polymorphic" is ambiguous in a different way, since a polymorphic type sometimes means a universally quantified type, such as `Forall T, T => T`, which is a proper type, since it classifies polymorphic values (in Scala, this value can be written as the structural type `{def apply[T](x: T): T = x}`)

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Jun 21, 2011 at 14:49

Adriaan Moors

4,296 ● 1 ● 19 ● 10

1 see also: adriaanm.github.com/research/2010/10/06/… – Adriaan Moors Jun 21, 2011 at 15:03

6 Adriaan's "Type Constructor Polymorphism" article now at adriaanm.github.com/research/2010/10/06/… – Steven Shaw Jun 6, 2012 at 7:39

1 I keep reading it as higher kindred and imagining a kindred spirit – Janac Meena Mar 14, 2019 at 23:54

119

(This answer is an attempt to decorate Adriaan Moors answer by some graphical and historical information.)

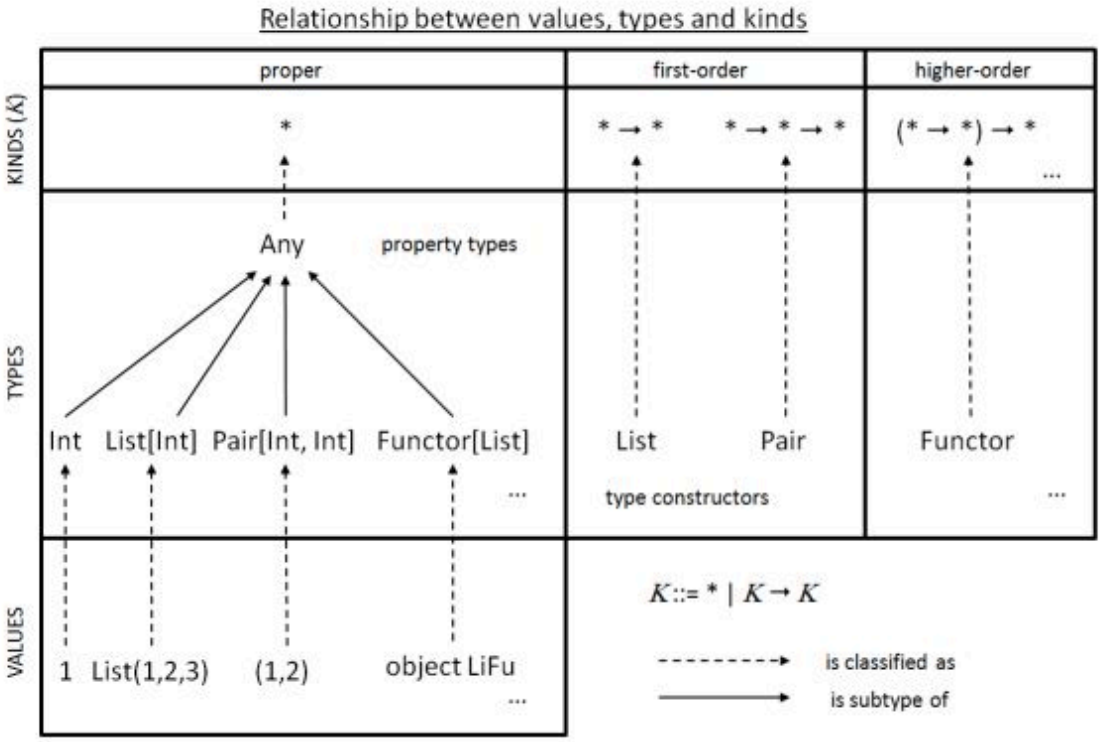
Higher kinded types are part of Scala since 2.5.

- Before that Scala, as Java till now, did not allow to use type constructor ("generics" in Java) to be used as type parameter to a type constructor. e.g.

  ```
  trait Monad [M[_]]
  ```

  was not possible.

  > In Scala 2.5 the type system had been extended by the ability to classify types on a higher level (known as *type constructor polymorphism*). These classifications are known as kinds.




http://adriaanm.github.com/files/higher.pdf + Functor and „first-order" and „higher-order".

(Picture derived from Generics of a Higher Kind)

The consequence is, that type constructor (e.g. `List`) could be used just as other types in type parameter position of type constructors and so they became first class types since Scala 2.5. (Similar to functions which are first class values in Scala).

In the context of a type system supporting higher kinds, we can distinguish **proper types**, types like `Int` or `List[Int]` from first-order types like `List` and **types of a higher kind** like `Functor` or `Monad` (types which abstract over types which abstract over types).

The type system of Java on the other side does not support kinds and therefore has no types of a "higher kind".

*So this has to be seen against the background of the supporting type system.*

- In the case of Scala you often see examples of a type constructor like

```
trait Iterable[A, Container[_]]
```

with the headline "Higher kinded types", e.g. in Scala for generic programmers, section 4.3

This is sometimes missleading, because many refer to `Container` as *higher kinded type* and not `Iterable`, but what is more precise is,

> the use of `Container` as type constructor parameter of a higher kinded (higher-order) type here `Iterable`.

Share Improve this answer Follow

edited May 12, 2015 at 8:57

answered Jun 20, 2011 at 20:50

Lutz

92

The kind of ordinary types like `Int` and `Char`, whose instances are values, is `*`. The kind of unary type constructors like `Maybe` is `* -> *`; binary type constructors like `Either` have (curried) kind `* -> * -> *`, and so on. You can view types like `Maybe` and `Either` as type-level functions: they take one or more types, and return a type.

A function is *higher-order* if it has an *order* greater than 1, where the order is (informally) the nesting depth, to the left, of function arrows:

- Order 0: `1 :: Int`
- Order 1: `chr :: Int -> Char`
- Order 2: `fix :: (a -> a) -> a`, `map :: (a -> b) -> [a] -> [b]`
- Order 3: `((A -> B) -> C) -> D`
- Order 4: `(((A -> B) -> C) -> D) -> E`

So, long story short, **a *higher-kinded* type is just a type-level higher-order function** which *abstracts over* type constructors:

- Order 0: `Int :: *`
- Order 1: `Maybe :: * -> *`
- Order 2: `Functor :: (* -> *) -> Constraint`—higher-kinded: converts unary type constructors to typeclass constraints

Share Improve this answer

Follow

edited Jul 13, 2020 at 18:49

answered Jun 6, 2011 at 0:18

Jon Purdy
54.9k ● 9 ● 98 ● 169

Ok I got it, what then would be an examples in Scala for ( * ⇒ * ) ⇒ * and ( * ⇒ * ) ⇒ ( * ⇒ * )? Would the Functor of Landei fit into the first category or rather in the second? – Lutz Jun 9, 2011 at 9:12

1 @lutz: It would be in the first category: `Functor` produces a proper type (well, trait, but same idea) `Functor[F[_]]` from a type constructor `F`. – Jon Purdy Jun 9, 2011 at 22:21

1 @Jon: A very insightful post, thank you. Can the type converter `(* => *) => (* => *)` be expressed in Scala? If not, in any other language? – Eugen Labun Jun 13, 2012 at 10:58

@JonPurdy The comparison between `* ⇒ * ⇒ *` with currying is very helpful. Thanks! – Lifu Huang Sep 18, 2016 at 10:00

`(* ⇒ *) ⇒ (* ⇒ *)` can also be spelled `(* ⇒ *) ⇒ * ⇒ *`. It can be expressed in Scala like `Foo[F[_], T]`. This the kind of a type like (in Haskell) `newtype Twice f a = Twice (f (f a))` (e.g., `Twice Maybe Int` ≅ `Maybe (Maybe Int)`, `Twice [] Char` ≅ `[[Char]]`) or something more interesting like the free monad `data Free f a = Pure a | Free (f (Free f a))`. – Jon Purdy Mar 15, 2019 at 0:13

I would say: A higher kinded type *abstracts over* a type constructor. E.g. consider

42

```
trait Functor [F[_]] {
   def map[A,B] (fn: A=>B)(fa: F[A]): F[B]
}
```

Here `Functor` is a "higher kinded type" (as used in the "Generics of a Higher Kind" paper). It is not a concrete ("first-order") type constructor like `List` (which abstracts over proper types only). It abstracts over all unary ("first-order") type constructors (as denoted with `F[_]`).

Or to put it in another way: In Java, we have clearly type constructors (e.g. `List<T>`), but we have no "higher kinded types", because we can't abstract over them (e.g. we can't write the `Functor` interface defined above - at least not directly).

The term "higher order (type constructor) polymorphism" is used to describe systems that support "higher kinded types".

Share Improve this answer

Follow

edited Jan 29 at 20:04

Jacob van Lingen

**9,502** ● 7 ● 52 ● 83

answered Jun 6, 2011 at 7:06

Landei

**54.5k** ● 13 ● 102 ● 195

This is what I thought, but it seems to contradict Jon's answer, where a "concrete type constructor" is already a "higher kinded type". – Lutz Jun 6, 2011 at 8:03

3 Yes. According to Jon's answer (as I understand it) `List<T>` in Java would be an unary type constructor as it has clearly the kind `* -> *`. What's missing in Jon's answer is that you must be able to abstract over the "whole thing" (and not just over the second `*` as in Java) in order to call it a higher kinded type. – Landei Jun 6, 2011 at 11:03

@Landai: The paper Scala for generic programmers in section 4.3 suggests the trait Iterable[A, Container[_]] to be a higher-kinded type (though it's not clear if Iterator or Container is meant) where on the other side vero690 in section 2.3.1 uses the term **higher-kinded type constructor** for someting like ( * -> * ) -> * (type operator parameterized with higher-order type constructor) which looks similar to the Iterator or your Functor trait. – Lutz Jun 7, 2011 at 8:29

1 This is probably right, but I think we're starting to split hairs here. The important point regarding higher kinded types is that not only type constructors are involved (order one type constructor polymorphism), but that we are able to abstract over the concrete type of that type constructors (higher order type constructor polymorphism). We have the ability to abstract over whatever we want without restrictions (concerning types and type constructors), which makes it less interesting to name all the possible versions of that feature. And it hurts my brain. – Landei Jun 7, 2011 at 8:44

2 In general it's important to distinguish definitions and references here. The definition `def succ(x: Int) = x+1` introduces the "value constructor" (see my other answer for what I mean by this) `succ` (no-one would refer to this value as succ(x: Int)). By analogy, `Functor` is the (indeed higher-kinded) type defined in your answer. Again, you shouldn't refer to it as `Functor[F[_]]` (what is `F`? what is `_`? they're not in scope! unfortunately, the syntactic sugar for existentials muddies the waters here by making `F[_]` short for `F[T forSome {type T}]`) – Adriaan Moors Jun 21, 2011 at 14:52

8

Scala REPL provides `:kind` command which

```
scala> :help kind
```

```
:kind [-v] <type>
Displays the kind of a given type.
```

For example,

```
scala> trait Foo[A]
trait Foo

scala> trait Bar[F[_]]
trait Bar

scala> :kind -v Foo
Foo's kind is F[A]
* -> *
This is a type constructor: a 1st-order-kinded type.

scala> :kind -v Foo[Int]
Foo[Int]'s kind is A
*
This is a proper type.

scala> :kind -v Bar
Bar's kind is X[F[A]]
(* -> *) -> *
This is a type constructor that takes type constructor(s): a higher-kinded
type.

scala> :kind -v Bar[Foo]
Bar[Foo]'s kind is A
*
This is a proper type.
```

---

The `:help` provides clear definitions so I think it is worth posting it here in its entirety (Scala 2.13.2)

```
scala> :help kind

:kind [-v] <type>
Displays the kind of a given type.

    -v      Displays verbose info.

"Kind" is a word used to classify types and type constructors
according to their level of abstractness.

Concrete, fully specified types such as `Int` and `Option[Int]`
are called "proper types" and denoted as `A` using Scala
notation, or with the `*` symbol.

    scala> :kind Option[Int]
    Option[Int]'s kind is A

In the above, `Option` is an example of a first-order type
constructor, which is denoted as `F[A]` using Scala notation, or
* -> * using the star notation. `:kind` also includes variance
information in its output, so if we ask for the kind of `Option`,
we actually see `F[+A]`:

    scala> :k -v Option
    Option's kind is F[+A]
    * -(+)-> *
```

```
    This is a type constructor: a 1st-order-kinded type.

When you have more complicated types, `:kind` can be used to find
out what you need to pass in.

    scala> trait ~>[-F1[_], +F2[_]] {}
    scala> :kind ~>
    ~>'s kind is X[-F1[A1],+F2[A2]]

This shows that `~>` accepts something of `F[A]` kind, such as
`List` or `Vector`. It's an example of a type constructor that
abstracts over type constructors, also known as a higher-order
type constructor or a higher-kinded type.
```

Share Improve this answer

Follow

edited Jun 13, 2020 at 23:45

answered Jun 13, 2020 at 23:38

Mario Galic
48.4k ● 6 ● 62 ● 108