Haskell Intermediate/Advanced

A Complete 8-Chapter Programming Course

Topics covered:

Functors & Applicatives · Monads — the track's central chapter

IO in depth & the pure-core/IO-shell pattern · Typeclasses in depth

Monad transformers, honestly · GADTs & phantom types

Capstone: a type-safe expression interpreter

Exercises: 24 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · closing the Rust Option/Result/traits lineage thread

Course 2 of 2 · completes the Haskell track

Table of Contents

CHAPTER 1 OF 8

Functors

COURSE 2 · CH 1

Functors

"A box you can map over" — the first step of the three-chapter arc that ends in Chapter 3's own monad reveal

`haskell1-8` closed Course 1 with typeclasses. This chapter opens Course 2 with the first of three — Functor, then Applicative, then Monad — building toward the single biggest payoff in this entire track.

The Functor Typeclass

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

A real typeclass, same shape as `haskell1-8`'s own `Eq` / `Ord` / `Show` — but with a genuine new wrinkle: `f` here isn't a concrete type, it's a **type constructor** — something that itself takes a type parameter, like `Maybe` or `[]`. `Functor` is a capability for "things that hold a type parameter," not for ordinary types directly.

"A Box You Can Map Over" — The Core Intuition

```
fmap (+1) (Just 5)   -- Just 6
fmap (+1) [1,2,3]    -- [2,3,4]
fmap (+1) Nothing    -- Nothing — the function is simply never applied, safely
```

`Maybe`, `[]`, and other "container-like" types let a function be applied to whatever's *inside*, without unwrapping and rewrapping by hand. This is genuinely one step further than `java2-1`'s/`csharp2-1`'s own generics — a generic type stores a `T`; a `Functor` additionally supplies a uniform way to transform whatever it's holding, without needing to know anything about the container itself.

fmap vs. map — A Real, Honest Naming Wrinkle

`map` is Haskell's own older, list-specific function, predating the generalized `Functor` typeclass. `fmap` is the generalized version, working for *any* `Functor`, not just lists. A real, slightly awkward wart from decades of

language evolution: `map` still exists separately, mostly for historical and beginner-friendliness reasons, and newcomers often reasonably ask why there are two. Worth naming honestly rather than glossing over.

The <$> Operator — fmap's Infix Alias

```
(+1) <$> Just 5   -- identical to fmap (+1) (Just 5) — just nicer to read
```

`<$>` is `fmap`'s infix alias, seen constantly in real Haskell code from here on — worth introducing now rather than later.

Writing a Functor Instance for Your Own Type

```
-- haskell1-6's own Tree a, now made Functor-capable:
instance Functor Tree where
    fmap _ Leaf = Leaf
    fmap f (Node l v r) = Node (fmap f l) (f v) (fmap f r)
```

`Functor` isn't limited to `Maybe` and `[]` — any appropriately-shaped type can genuinely participate. This instance reuses `haskell1-6`'s own `Tree a` directly, applying `f` to every value while leaving the tree's own shape untouched.

The Functor Laws

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

A real, honest limitation: these two laws — mapping `id` changes nothing, and mapping a composed function equals composing the mapped functions — are **not enforced by the compiler at all**. They're purely a convention instance authors are expected to uphold, the same honest-acknowledgment spirit as Course 1's own record-syntax field-collision wart.

ASPECT	JAVA/C# GENERICS (JAVA2-1, CSHARP2-1)	HASKELL FUNCTOR
What it provides	a container that holds a T	a container PLUS a uniform way to transform what's inside
Requires knowing the container's internals	often, to transform contents	never — fmap works the same regardless

ASPECT	JAVA/C# GENERICS (JAVA2-1, CSHARP2-1)	HASKELL FUNCTOR
Compiler-enforced correctness	**type-safety only**	**type-safety only — the Functor laws are unenforced**

REACH FOR <$> IN REAL CODE

`<$>` is the idiomatic form seen constantly in real Haskell — `fmap` itself is more useful for explanation and teaching contexts, exactly the balance this chapter uses.

THE FUNCTOR LAWS AREN'T CHECKED — A BROKEN INSTANCE STILL COMPILES

A lawless `Functor` instance (one that doesn't actually satisfy `fmap id = id`) compiles and runs without any warning at all, and can produce genuinely surprising, wrong behavior anywhere code assumes the laws hold — including, later, inside do-notation. There is no compiler safety net here.

Coding Challenges

CHALLENGE 1

Use fmap to double every value inside a Just 21, a Nothing, and a list [1,2,3,4], printing all three results.

View solution

CHALLENGE 2

Using haskell1-6's own Tree a data type, write a Functor instance for it, build a small sample tree, apply fmap (*10) to it, and print the result.

View solution

CHALLENGE 3

Write a short comment explaining why f in "class Functor f where" must be a type constructor (like Maybe) rather than a concrete type (like Int), tying your answer to what fmap's own type signature requires.

View solution

CHAPTER 1 QUICK REFERENCE

- Functor's f is a type constructor, not a concrete type — a genuine new wrinkle beyond ordinary typeclasses
- fmap applies a function to whatever's inside a container without unwrapping/rewrapping by hand
- map is Haskell's older, list-only function; fmap is the generalized version for any Functor — a real historical wart, honestly named
- <$> is fmap's infix alias, the idiomatic form in real code
- Any appropriately-shaped type can get a Functor instance, including haskell1-6's own Tree a
- The Functor laws (fmap id = id, etc.) are pure convention — never compiler-enforced
- **Next chapter:** Applicatives — combining independent computations, the bridge to Chapter 3's own monad reveal

CHAPTER 2 OF 8

Applicatives

COURSE 2 · CH 2

Applicatives

Combining independent computations — the deliberate bridge between Functor and Chapter 3's own monad reveal

`haskell2-1`'s `fmap` handles a plain function applied to one wrapped value. This chapter handles the next real gap: what happens when the function *itself* is wrapped too.

The Problem Functor Can't Solve

```
fmap :: (a -> b) -> f a -> f b
-- but what about: Just (+3) applied to Just 5 ?
-- the function ITSELF is wrapped — fmap's signature has no place for that
```

`fmap` requires a plain, unwrapped `a -> b`. It genuinely cannot express applying `Just (+3)` to `Just 5` — the function is wrapped too, and nothing in `fmap`'s own type signature has anywhere to put a wrapped function.

The Applicative Typeclass & <*>

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

Just (+3) <*> Just 5   -- Just 8
```

`Applicative` requires `Functor` as a superclass — a real hierarchy: every `Applicative` is automatically a `Functor` too. `<*>` applies a wrapped function to a wrapped value, exactly the case `fmap` couldn't handle.

pure — Wrapping a Plain Value

```haskell
pure 5 :: Maybe Int   -- Just 5
pure 5 :: [Int]       -- [5]
```

`pure` lifts an ordinary value into the applicative context using the minimal, default wrapping — genuinely useful for starting a chain of applicative operations from a plain, unwrapped value.

Combining Independent Computations

```haskell
(+) <$> Just 3 <*> Just 4   -- Just 7
```

Here's the real, practical use case: combining *two separately-wrapped* values with an ordinary multi-argument function. `java2-1`'s/ `csharp2-1`'s own generics have no equivalent mechanism for this at all — combining two independently-wrapped values with a plain function requires manual unwrapping in both, since neither language's generics carry this kind of combining operation as part of the type itself.

A Real Practical Example — Validating Multiple Fields

```haskell
data Person = Person { name :: String, age :: Int }

validateName :: String -> Maybe String
validateAge :: Int -> Maybe Int

mkPerson :: String -> Int -> Maybe Person
mkPerson n a = Person <$> validateName n <*> validateAge a
-- Nothing if EITHER field fails validation — Just a Person only if BOTH succeed
```

A genuinely realistic, common Haskell idiom — building a record from several independently-validated fields, where the whole construction fails if any single field does.

The Bridge to Monad

Stated explicitly, not just implied: `Applicative` combines **independent** computations — neither can depend on the other's actual result, only on whether each independently succeeded or failed. Chapter 3's own `Monad` is exactly what's needed once a *later* computation needs to depend on an *earlier* one's real, concrete value. Applicative isn't a lesser version of Monad — it's a genuinely useful, distinct stepping stone for the many real cases where independence is all that's actually needed.

TYPECLASS	FUNCTION IS WRAPPED?	CAN LATER STEPS DEPEND ON EARLIER RESULTS?
Functor (haskell2-1)	`no — plain a -> b`	`n/a — only one value involved`
Applicative	`yes — f (a -> b)`	`no — combines independently`
Monad (Chapter 3)	`n/a — uses >>= instead`	`yes — the whole point`

USE F <$> A <*> B <*> C FOR COMBINING INDEPENDENT WRAPPED VALUES

This chained applicative style is genuinely idiomatic and common — reach for it whenever a plain multi-argument function needs to be applied across several separately-wrapped values.

APPLICATIVE CANNOT EXPRESS "DO THIS ONLY IF THE FIRST ONE SUCCEEDED, USING ITS VALUE"

This is a real, genuine limitation, not a bug — Applicative structurally has no way for one computation's result to influence which computation runs next. That's precisely the gap Chapter 3's own Monad exists to close.

Coding Challenges

CHALLENGE 1

Use <*> to apply Just (*2) to Just 10, and separately apply Nothing (of the correct function type) to Just 10, printing both results.

View solution

CHALLENGE 2

Write a three-argument record constructor and combine three independently-Maybe-wrapped fields into one Maybe of the record using <$> and <*> chained together, testing both a fully-successful case and a case where one field is Nothing.

View solution

CHALLENGE 3

Write a short comment giving a concrete, realistic example of a task that CANNOT be expressed using Applicative alone (where a later step genuinely needs an earlier step's actual value), explaining exactly why <*> falls short for it.

View solution

CHAPTER 2 QUICK REFERENCE

- Applicative solves what Functor can't: applying a WRAPPED function to a wrapped value, via <*>
- Applicative requires Functor as a superclass — every Applicative is automatically a Functor
- pure lifts a plain value into the applicative context with minimal wrapping
- f <$> a <*> b combines independently-wrapped values with an ordinary function — no equivalent in java2-1's/csharp2-1's own generics
- Applicative combines INDEPENDENT computations only — no later step can depend on an earlier step's actual result
- **Next chapter:** Monads — the track's central payoff, where dependent computations finally become possible

CHAPTER 3 OF 8

Monads

COURSE 2 · CH 3 — CENTRAL CHAPTER

Monads

The single biggest payoff in this entire 16-chapter track

Everything since `haskell1-1` has been building toward this. `haskell2-2` closed with an explicit promise: Applicative can't let a later computation depend on an earlier one's real value, and Monad is exactly what closes that gap. Here it is — and the payoff is bigger than just closing one gap.

The Gap Applicative Left Open

`haskell2-2`'s own Challenge 3 asked for exactly this: looking up a user, then using that user's *real* email to look up their order. `<*>` structurally cannot do this — both its arguments must already be wrapped, independently, before it ever runs.

The Monad Typeclass & >>= (bind)

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
```

`Monad` requires `Applicative` as its own superclass, extending `haskell2-2`'s hierarchy further: Functor ← Applicative ← Monad. `>>=` ("bind") takes a wrapped value *and a function that receives the real, unwrapped value*, returning a new wrapped result. This is exactly the missing mechanism — the function passed to `>>=` genuinely sees the earlier computation's actual result.

Maybe as a Monad — Chaining Optional Computations

```
findUser 5 >>= \u -> findOrderByEmail (email u)
-- if findUser 5 is Nothing, the whole chain short-circuits automatically
-- if it's Just u, the lambda genuinely receives the real u and can use it
```

Directly resolving `haskell2-2`'s own Challenge 3 example. If the lookup fails, `>>=` short-circuits to `Nothing` without ever calling the lambda; if it succeeds, the lambda receives the real, unwrapped `User`.

The Central Reveal — Maybe/Either/[]/IO Are All the Same Abstraction

This is the single most important paragraph in the entire track. `Maybe`'s `>>=` short-circuits on `Nothing`. `Either`'s `>>=` short-circuits on `Left`. `[]`'s `>>=` represents nondeterminism — every combination of possible results, chained together. `IO`'s `>>=` sequences side-effecting actions one after another. Four completely different-*feeling* use cases — optional values, error handling, multiple results, real-world side effects — are secretly implementations of the **exact same single interface**, differing only in what `>>=` actually does underneath. This is the "aha" this whole course has been building toward since `haskell1-1`'s own `main :: IO ()`.

do-Notation — Syntactic Sugar Over >>=

```
do
    u <- findUser 5
    o <- findOrderByEmail (email u)
    return o

-- desugars EXACTLY to:
findUser 5 >>= \u -> findOrderByEmail (email u) >>= \o -> return o
```

`do`-notation isn't a separate control-flow feature — it's pure syntax sugar over `>>=`, no different in kind from `haskell1-2`'s own currying being "secretly" a chain of one-argument functions. This retroactively explains every `do` block used without comment since `haskell1-1`'s own `main = do { ... }`.

Closing the Loop — Rust's Option/Result Are Monad-Shaped

Back to the thread run through `haskell1-6` and `haskell1-8`: Rust's own `.and_then()` method on `Option<T>` / `Result<T, E>`, and its `?` operator, are directly equivalent to Haskell's `>>=`. Rust genuinely never uses the word "monad" anywhere in its own documentation or community culture — but the underlying structure and behavior *is* a monad in the formal sense. Stated plainly, as this course's own closing statement on the Rust lineage thread: the idea travelled, even where the name didn't.

The Monad Laws

A real, honest brief mention, same spirit as `haskell2-1`'s own Functor laws: left identity, right identity, and associativity are expected of every lawful `Monad` instance — and, again, none of them are compiler-enforced. Pure convention, same as before.

TYPE	WHAT >>= ACTUALLY DOES	RUST EQUIVALENT
Maybe	short-circuits on Nothing	Option::and_then / ?
Either	short-circuits on Left	Result::and_then / ?
[]	explores every combination of results	no direct one-line equivalent
IO	sequences side-effecting actions	ordinary sequential statements

REACH FOR DO-NOTATION FOR READABILITY IN REAL CODE

It desugars to `>>=` automatically — no performance or capability difference, purely presentation. Nearly all real Haskell code uses `do`-notation for anything beyond the simplest one-step chain.

DO-NOTATION'S SEQUENTIAL LOOK CAN OBSCURE VERY DIFFERENT UNDERLYING BEHAVIOR

A `do` block reads the same regardless of which monad it's written in — but a `Maybe` block might silently short-circuit, an `[]` block genuinely explores every combination rather than running once, and only `IO`'s own version behaves like ordinary imperative sequencing. The sugar looks identical; the real behavior underneath genuinely isn't.

Coding Challenges

CHALLENGE 1

Write findUser :: Int -> Maybe String and findOrderByEmail :: String -> Maybe String (stub implementations are fine), then chain them with >>= to resolve haskell2-2's own Challenge 3 scenario for real, testing both a successful lookup chain and one that fails partway through.

View solution

CHALLENGE 2

Rewrite Challenge 1's >>= chain using do-notation instead, and confirm both versions produce identical results for both the success and failure cases.

View solution

CHALLENGE 3

Write a short comment explaining, using a concrete example, how Rust's ? operator on a Result is behaviorally equivalent to Haskell's >>= on Either, even though Rust code never uses the word "monad" anywhere.

View solution

CHAPTER 3 QUICK REFERENCE — THE TRACK'S CENTRAL CHAPTER

- Monad requires Applicative (which requires Functor) — the full Functor ← Applicative ← Monad hierarchy
- >>= (bind) passes the REAL unwrapped result of one computation to a function producing the next — closing Applicative's own dependency gap
- THE central reveal: Maybe/Either/[]/IO all implement the same >>= interface, differing only in what bind actually does underneath
- do-notation is pure sugar over >>= — no different in kind from currying being sugar over chained one-argument functions
- Rust's Option/Result, .and_then(), and ? are genuinely monad-shaped, even though Rust never uses the word — closing the loop from haskell1-6/haskell1-8
- The Monad laws (left/right identity, associativity) are pure convention, never compiler-enforced, same as haskell2-1's Functor laws
- **Next chapter:** IO in depth — how IO stays a real, distinct type that "infects" every calling signature, paying off Chapter 1's own throughline in full

CHAPTER 4 OF 8

IO in Depth

COURSE 2 · CH 4

IO in Depth

Chapter 1's own throughline, paid off in full

`haskell1-1` opened this entire course with a claim: `main :: IO ()` tells you, in the type itself, that side effects are possible. Sixteen chapters later, with `haskell2-3`'s own monad reveal now in hand, here's the full mechanism behind that claim — and it's stronger than it first looked.

IO as a Monad — Recap

`haskell2-3` already named `IO`'s own `>>=` as sequencing side-effecting actions one after another. This chapter goes deeper into why that specific design choice matters as much as it does.

"Infection" — Why Calling IO Code Makes You IO Too

```
impureFunction :: Int -> Int
impureFunction x = x + getLine   -- DOES NOT TYPE-CHECK, ever
```

Here's the central mechanic: `getLine` has type `IO String`, not `String` — it genuinely cannot be added to an `Int`, no matter what. The only way to actually get the `String` *out* of an `IO String` is `>>=` or `do`-notation — and both of those require the surrounding function to itself return something wrapped in `IO`. Calling IO code doesn't just fail quietly; it forces the caller's own type signature to admit it.

Purity Is Contagious the Other Way Too — main Can't Escape Its Own Type Either

Even `main` itself gets no free pass — its signature is `IO ()`, declared honestly like everything else. There is genuinely no "trusted root" that silently does IO without its type saying so. Contrast this directly against every other language covered on this site: Java's `main`, C#'s `Main`, Python's top-level code all execute freely, with zero tracking of what they actually touch.

The Real, Concrete Consequence — You Can See Purity by Reading a Signature

```
parseConfig :: String -> Config   -- GUARANTEED pure — no file read, no network call, no console output,
```

This is the payoff `haskell1-1`'s own comparison table promised, now fully justified: `parseConfig`'s signature is a compiler-checked guarantee, not a hope or a convention. If it needed to read a file or touch the network, its own type would be forced to say `IO Config` instead — there's no way to sneak IO past the signature.

unsafePerformIO — The Honest Escape Hatch That Breaks the Promise

```
unsafePerformIO :: IO a -> a   -- genuinely exists — a real way to lie to the type system
```

An honest acknowledgment, matching this course's own established pattern of naming real limitations rather than pretending everything is unbreakable: `unsafePerformIO` genuinely exists, letting a programmer forcibly extract a value from `IO` and misrepresent a function as pure when it isn't. Almost never appropriate in real code — it exists mainly for narrow, expert-level interop cases, not everyday use.

Practical IO — A Pure Core, IO Shell Shape

```
main :: IO ()
main = do
    input <- getLine
    let result = pureBusinessLogic input   -- zero IO in this function's own type
    putStrLn result
```

A genuinely practical, real-world pattern: keep as much logic as possible in ordinary pure functions, and push all actual IO to a thin outer layer that calls into that pure core. `pureBusinessLogic` is fully testable with no IO mocking at all — a real, concrete engineering benefit this whole design buys.

ASPECT	JAVA / C# / PYTHON	HASKELL
Compiler tracks side effects	**no, never**	**yes — via IO in the type**
A "trusted" entry point exempt from tracking	**yes — main runs freely**	**no — main :: IO () is honest too**
Can a signature alone prove purity	**no**	**yes — a compiler-checked guarantee**

STRUCTURE REAL PROGRAMS WITH A PURE CORE AND A THIN IO SHELL

Push actual IO to the outermost layer and keep genuine business logic pure — the pure core becomes trivially testable with ordinary function calls, no mocking framework required at all.

UNSAFEPERFORMIO IS A REAL, NOT PURELY THEORETICAL, WAY TO BREAK THE GUARANTEE

A function whose type signature promises purity but internally uses `unsafePerformIO` to sneak around it genuinely can lie to every caller relying on that signature — a real, if rare, way this chapter's own central guarantee can be violated, worth knowing exists rather than assuming the boundary is literally unbreakable.

Coding Challenges

CHALLENGE 1

Attempt to write a function that adds the result of getLine directly to an Int without any do-notation or >>=, show the resulting compile error, and explain in a comment exactly why it fails.

View solution

CHALLENGE 2

Write a small program with a pure function calculateTotal :: [Int] -> Int and a main that reads a line, parses it into a list of numbers, calls calculateTotal, and prints the result — keeping calculateTotal itself completely free of IO in its type.

View solution

CHALLENGE 3

Write a short comment contrasting Haskell's main :: IO () against Java's public static void main(String[] args), specifically addressing whether either language's own main gets a "free pass" from its own safety mechanism.

View solution

CHAPTER 4 QUICK REFERENCE

- Calling IO code forces the caller's own type to admit IO too — there's no way to quietly extract a value from IO without >>= or do-notation, both of which require IO in the surrounding signature
- Even main :: IO () is honest about itself — no trusted root gets a free pass, unlike every other language on this site
- A pure signature (String -> Config) is a real, compiler-checked guarantee of no hidden side effects — the payoff haskell1-1 promised
- unsafePerformIO genuinely exists as a real, if rarely appropriate, way to break the purity guarantee
- Pure core, IO shell: push IO to a thin outer layer, keep real logic pure and trivially testable

- **Next chapter:** typeclasses in depth — writing custom instances, compared against rust2-2's traits and Kotlin/Java's interfaces

CHAPTER 5 OF 8

Typeclasses In Depth

COURSE 2 · CH 5

Typeclasses In Depth

Beyond Eq/Ord/Show — and the real, honest differences from rust2-2's own trait system

`haskell1-8` introduced typeclasses through the standard trio. This chapter writes real, multi-method typeclasses from scratch, and gets specific about where Haskell's version genuinely diverges from `rust2-2`'s own traits — not just naming the shared lineage again, but the real differences too.

Beyond Eq/Ord/Show — A Real Custom Typeclass

```
class Shape a where
    area :: a -> Double
    perimeter :: a -> Double

instance Shape Circle where
    area (Circle r) = pi * r * r
    perimeter (Circle r) = 2 * pi * r
```

A genuinely useful multi-method typeclass — every instance must supply both `area` and `perimeter`.

Default Method Implementations

```
class Shape a where
    area :: a -> Double
    perimeter :: a -> Double
    describe :: a -> String
    describe s = "Area: " ++ show (area s)   -- a real default body
```

A typeclass method can carry a default body — the same direct genetic connection `haskell1-8` already named between typeclasses and both `rust2-2`'s trait default methods and `java1-6`'s Java 8 interface defaults. An instance can override `describe`, or simply inherit the default.

Minimal Complete Definitions

```
class MyEq a where
    {-# MINIMAL (==) | (/=) #-}   -- only ONE of these strictly needs a real implementation
    (==), (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

A genuinely nice, practical Haskell-specific idiom: the `{-# MINIMAL #-}` pragma tells instance authors exactly which subset of methods actually needs a real implementation, with the rest derivable from those. Not present in quite this form in Rust's, Java's, or Kotlin's own interface/trait systems.

Superclass Constraints — Building Real Hierarchies

```
class Shape a => Shape3D a where
    volume :: a -> Double   -- Shape3D REQUIRES Shape first — the same pattern haskell2-2's Monad requir
```

Already used implicitly since `haskell2-2`'s own `Applicative`/`Monad` hierarchy — a typeclass can require another as a prerequisite, comparable to `rust2-2`'s own supertrait bounds and interface inheritance in Java/Kotlin/C#.

Rust Traits vs. Haskell Typeclasses — The Real Differences

Not just "same idea, different name" — a few genuine, concrete differences worth naming honestly. Rust's orphan rule generally requires either the trait or the type to be defined in your own crate to write an `impl`; Haskell's typeclass system has historically been more permissive by default, though GHC extensions exist to tighten or loosen this in different scenarios. Rust also distinguishes `dyn Trait` (dynamic dispatch) from generic/`impl Trait` (static dispatch) as two separate mechanisms a programmer chooses between explicitly — `haskell1-8`'s own dictionary-passing dispatch is more uniform, working the same way underneath regardless of how static- or dynamic-feeling the usage looks.

Multi-Parameter Type Classes, Briefly

```
{-# LANGUAGE MultiParamTypeClasses #-}
class Convert a b where
    convert :: a -> b
```

A real, honest mention: a typeclass can be parameterized over more than one type at once, but this genuinely requires a GHC extension — it's not part of standard Haskell 2010. Flagged plainly as an extension, not core

language, the same honest-wart spirit as earlier chapters' own acknowledgments.

ASPECT	RUST TRAITS (RUST2-2)	JAVA/KOTLIN INTERFACES (JAVA1-6)	HASKELL TYPECLASSES
Default methods	yes	yes (Java 8+)	yes
Superclass/supertrait constraints	yes	yes (interface extends)	yes
Implementing for types outside your own code	restricted (orphan rule)	no	more permissive by default
Static vs. dynamic dispatch	explicit choice (dyn Trait vs generics)	vtable, always	dictionary passing, uniform
Multiple type parameters	supported directly	generic interfaces	needs a GHC extension

USE {-# MINIMAL #-} WHEN YOUR OWN TYPECLASS HAS DERIVABLE METHODS

Documenting the true minimal set an instance author must supply — with everything else provided as a sensible default — keeps a custom typeclass genuinely easy to implement correctly.

NOTHING STOPS AN INSTANCE FROM DISAGREEING WITH ITSELF

An instance could technically define `==` and `/=` as something other than logical negations of each other — nothing at compile time catches this. A real, genuine correctness risk resting purely on the instance author's own discipline, the same honest-limitation spirit as the unenforced Functor and Monad laws from earlier chapters.

Coding Challenges

CHALLENGE 1

Define a Shape typeclass with area and perimeter methods, plus a describe default method, and write instances for two different shapes, calling describe on both without overriding it.

View solution

CHALLENGE 2

Extend Challenge 1's setup with a Shape3D typeclass requiring Shape as a superclass constraint, adding a volume method, and write one instance for a genuine 3D shape.

View solution

CHALLENGE 3

Write a short comment explaining the real difference between Rust's dyn Trait/generic split and Haskell's own uniform dictionary-passing dispatch, referencing haskell1-8's own dispatch material directly.

View solution

CHAPTER 5 QUICK REFERENCE

- A typeclass can require multiple methods, with default bodies available for some — the same rust2-2/java1-6 lineage haskell1-8 already named
- {-# MINIMAL #-} documents exactly which subset of methods an instance must genuinely implement
- Superclass constraints (class Shape a => Shape3D a) build real hierarchies, the same pattern Applicative/Monad already used
- Real differences from Rust traits: the orphan rule vs. Haskell's more permissive defaults, and Rust's explicit dyn/generic dispatch choice vs. Haskell's uniform dictionary passing
- Multi-parameter type classes are a real GHC extension, not core Haskell 2010
- Nothing enforces logical consistency between related methods in one instance — a real, honest correctness risk
- **Next chapter:** monad transformers — stacking effects, and an honest look at where the elegance gets harder

CHAPTER 6 OF 8

Monad Transformers

COURSE 2 · CH 6

Monad Transformers

An honest chapter about where the elegance genuinely gets harder

`haskell2-3` delivered the track's biggest reveal with real, clean elegance. This chapter is deliberately different in tone: a genuine, well-documented real-world complication, named honestly rather than smoothed over.

The Real Problem — Stacking Two Monads at Once

```
loadConfig :: FilePath -> IO (Maybe Config)
-- works, but: back to manually unwrapping IO, then manually pattern-matching on Maybe
-- haskell2-3's own clean >>= chaining is lost the moment two monads nest like this
```

A genuinely common real need: a computation that might fail *and* performs IO. `IO (Maybe Config)` works, but the moment two monads are simply nested rather than unified, `haskell2-3`'s own clean chaining disappears — back to manual unwrapping at both layers.

Enter Monad Transformers

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
-- MaybeT IO a — combines Maybe's short-circuiting WITH IO's effects, as ONE unified monad
```

A **transformer** wraps another monad, combining its own effect with the wrapped one, while still being a single, unified `Monad` — restoring one clean `>>=` chain or `do`-block across both effects at once, no manual double-unwrapping required.

ExceptT — Errors + Another Effect

```
loadConfig :: FilePath -> ExceptT String IO Config
loadConfig path = do
    exists <- liftIO (doesFileExist path)   -- lift a plain IO action into the combined context
    if exists then do ...
              else throwError "file not found"
```

`ExceptT String IO a` genuinely combines `Either`-style error handling with IO in one unified `do`-block. `liftIO` is real, new ceremony this chapter isn't going to hide — a plain `IO` action must be explicitly lifted into the combined transformer context before it can be used.

StateT — Threading State Through IO

```
type App a = StateT Int IO a   -- threading an Int "state" through a sequence of IO actions
```

A real, practical use case: threading state through a sequence of IO actions, without ever actually breaking `haskell1-3`'s own immutability guarantee underneath — genuinely useful for something like a simple interpreter, previewing Chapter 8's own capstone.

The Honest Part — This Is Where It Gets Genuinely Harder

Stated plainly, matching this chapter's own framing directly: real monad transformer *stacks* — combining `StateT` + `ExceptT` + `IO` all at once — get genuinely hard to read, hard to reason about, and hard to debug. `lift` / `liftIO` calls proliferate, and the exact *order* transformers are stacked in changes real behavior in ways that aren't always obvious. This is a real, well-known, honestly-documented pain point across the Haskell community itself — not something invented for this course. Monad transformers are a genuinely useful, real tool, but they are not the effortless continuation of Functor/Applicative/Monad's own clean elegance.

mtl — A Real, Practical Mitigation

Briefly worth naming: the `mtl` library's typeclass-based approach (`MonadState`, `MonadError`, and similar) is the pragmatic answer the Haskell community actually reaches for to reduce some of the `lift`-noise — without pretending it fully dissolves the underlying complexity. Full treatment is out of scope here.

SITUATION	ERGONOMICS
A single monad (haskell2-3)	**clean, genuinely elegant**

SITUATION	ERGONOMICS
One transformer over one base monad (ExceptT e IO)	`functional, modest real ceremony (lift/liftIO)`
A deep multi-layer transformer stack	`genuinely hard to read/reason about — a real, honest limitation`

REACH FOR ONE WELL-CHOSEN TRANSFORMER, NOT A DEEP STACK, FOR THE COMMON CASE

Most real problems only genuinely need one or two combined effects (error handling plus IO, most commonly) — keep the stack as shallow as the actual problem allows rather than reaching for a deep, speculative stack up front.

STACK ORDER GENUINELY CHANGES BEHAVIOR — THIS IS WELL-DOCUMENTED, NOT A BEGINNER TRAP ALONE

`ExceptT e (StateT s IO)` and `StateT s (ExceptT e IO)` do not behave identically — which effect "wins" when both an error and a state change are in play differs based on stacking order. A real, well-known source of confusion across the Haskell community, not something unique to newcomers.

Coding Challenges

CHALLENGE 1

Write a function using the plain nested IO (Maybe Int) shape that reads a line and returns Just its parsed integer value or Nothing if parsing fails, manually unwrapping both layers.

View solution

CHALLENGE 2

Rewrite Challenge 1 using ExceptT String IO Int instead, using throwError for the parse-failure case and liftIO to perform the actual line-reading, and compare the resulting code's readability to Challenge 1's manual version.

View solution

CHALLENGE 3

Write a short comment explaining, conceptually, why ExceptT e (StateT s IO) and StateT s (ExceptT e IO) can produce different results when an error occurs partway through a sequence of state updates — specifically, what happens to state changes already made before the error in each ordering.

View solution

CHAPTER 6 QUICK REFERENCE

- Nesting monads (IO (Maybe a)) loses haskell2-3's own clean >>= chaining — back to manual double-unwrapping
- A transformer (MaybeT, ExceptT, StateT) wraps another monad, restoring one unified do-block across both effects
- liftIO/lift are real, necessary ceremony to bring a plain action into a transformer's combined context
- Deep transformer stacks are honestly, genuinely harder to read and reason about — a real, documented community pain point, not smoothed over here
- mtl's typeclass-based approach reduces some lift-noise pragmatically, without eliminating the underlying complexity
- Stack order changes real behavior — ExceptT e (StateT s IO) ≠ StateT s (ExceptT e IO)
- **Next chapter:** type-level programming — GADTs and phantom types, compared to ts4-5's own branded types

CHAPTER 7 OF 8

Type-Level Programming

COURSE 2 · CH 7

Type-Level Programming

A lighter chapter after Chapter 6's honest heaviness — and a genuine convergence with ts4-5's own branded types

A deliberately lighter touch after `haskell2-6`'s own honest complexity. This chapter previews what "type-level programming" even means, and lands on a real, satisfying convergence with a completely different type system this site already covers.

What "Type-Level Programming" Means, Briefly

Ordinary programming works with values at runtime. Type-level programming means using the type system itself to encode and enforce constraints, with the compiler doing the real work during type-checking rather than at runtime. Kept light and practical here, not a deep academic treatment.

Phantom Types

```
data Tagged tag a = Tagged a   -- tag never actually appears on the right-hand side!
```

`tag` appears in the declaration but is never actually stored or used in any of the real data — a **phantom type**. It exists purely to let the compiler distinguish otherwise-identical values, like `Tagged "Meters" Double` versus `Tagged "Feet" Double`, catching an accidental mix-up at compile time with zero runtime cost, since `tag` is never stored anywhere real.

Direct Comparison to ts4-5's Own Branded Types

`ts4-5`'s own TypeScript branded types use genuinely the same trick — a phantom property existing only at the type level to prevent two structurally-identical types from being accidentally interchanged. This is the same core idea, arrived at independently in a structural type system (TypeScript) and a nominal one (Haskell) — worth naming the convergence explicitly, since it shows the idea is useful enough to be reinvented across genuinely different type-system philosophies.

A Practical Example — Preventing Unit Mix-Ups

```
newtype Meters = Meters Double
newtype Feet = Feet Double

addMeters :: Meters -> Meters -> Meters
addMeters (Meters a) (Meters b) = Meters (a + b)

-- addMeters (Meters 5) (Feet 3)   -- REAL compile error, not a runtime mismatch
```

A real, concrete, motivating example: the compiler catches an attempt to add `Meters` and `Feet` directly, a genuine practical safety win with zero runtime overhead — `newtype` wrappers compile away completely.

GADTs — Generalized Algebraic Data Types

```
{-# LANGUAGE GADTs #-}

data Expr a where
    IntLit  :: Int -> Expr Int
    BoolLit :: Bool -> Expr Bool
    Add     :: Expr Int -> Expr Int -> Expr Int

-- Add (IntLit 5) (BoolLit True)   -- REAL compile error — Add requires two Expr Int
```

A real, honest step up in power from `haskell1-6`'s own plain `data` declarations: GADT syntax lets each constructor specify its own, more specific return type, rather than every constructor sharing the same generic one. This makes a genuinely type-safe mini expression language possible, where an ill-typed expression like adding an `Int` to a `Bool` is a real compile error — directly useful groundwork for Chapter 8's own capstone interpreter.

{-# LANGUAGE GADTs #-} — Another Honest Extension Flag

Same honest spirit as `haskell2-5`'s own `MultiParamTypeClasses` acknowledgment: GADTs require an explicit extension flag, not part of core Haskell 2010 — worth naming plainly rather than presenting as if it had always just been part of the language.

CONCEPT	HASKELL1-6'S PLAIN ADTS	GADTS	TYPESCRIPT (TS4-5)
Constructor return types	all share the same generic type	each constructor specifies its own	n/a — structural typing

CONCEPT	HASKELL1-6'S PLAIN ADTS	GADTS	TYPESCRIPT (TS4-5)
Phantom-type-style tagging	possible, but GADTs unneeded for it	n/a	branded types — same core idea
Core language or extension	core Haskell 2010	requires {-# LANGUAGE GADTs #-}	core TypeScript feature

REACH FOR A PHANTOM TYPE OR NEWTYPE WHENEVER VALUES SHARE A REPRESENTATION BUT SHOULDN'T MIX

Units, IDs belonging to different entity types, and similar cases are exactly where a lightweight phantom type or `newtype` wrapper buys real compile-time safety for zero runtime cost.

DON'T REACH FOR GADTS UNLESS PLAIN ADTS GENUINELY CAN'T EXPRESS THE CONSTRAINT

GADT syntax is real, genuine additional complexity — worth it specifically when different constructors truly need different, more specific return types (like the `Expr a` example), not as a default replacement for ordinary `data` declarations.

Coding Challenges

CHALLENGE 1

Define newtype wrappers UserId and ProductId, both wrapping an Int, and write a function that only accepts a UserId. Attempt to call it with a ProductId instead and show the resulting compile error.

View solution

CHALLENGE 2

Define the Expr a GADT from the chapter (IntLit, BoolLit, Add) plus a new constructor If :: Expr Bool -> Expr a -> Expr a -> Expr a, and write an eval :: Expr a -> a function that correctly evaluates a small expression using it.

View solution

CHALLENGE 3

Write a short comment explaining the real convergence between Haskell's phantom types and ts4-5's own branded types, addressing specifically how the same "compiler-only tag" idea gets expressed differently in a nominal type system versus a structural one.

View solution

CHAPTER 7 QUICK REFERENCE

- Type-level programming uses the type system itself to enforce constraints, resolved during compilation, not at runtime
- Phantom types carry a type parameter that's never actually stored, existing purely to prevent mix-ups at compile time, zero runtime cost
- Genuinely the same core idea as ts4-5's own branded types — independently converged upon across a nominal and a structural type system
- newtype wrappers (Meters/Feet) are a real, practical, zero-cost way to prevent unit/ID mix-ups
- GADTs let each constructor of a data type specify its own, more specific return type — a real step up from haskell1-6's plain ADTs, useful for a type-safe mini expression language
- GADTs require an explicit language extension, same honest-wart spirit as haskell2-5's own MultiParamTypeClasses
- **Next chapter:** the capstone — a real small Haskell program combining ADTs, typeclasses, monads, and IO from across both courses

CHAPTER 8 OF 8

Capstone: Building a Small Project

COURSE 2 · CH 8 — CAPSTONE

Building a Small Project

A type-safe expression interpreter combining nearly every chapter across both Haskell courses

Sixteen chapters, two courses — this capstone builds a small, real expression interpreter touching almost all of it: `haskell2-7`'s own GADT expression language, monadic error handling with `Either`, a custom typeclass for pretty-printing, and a thin IO shell around a fully pure core.

The Expression Language, Extended

```
{-# LANGUAGE GADTs #-}

data Expr a where   -- haskell2-7's own GADT, extended with Div
    IntLit  :: Int -> Expr Int
    BoolLit :: Bool -> Expr Bool
    Add     :: Expr Int -> Expr Int -> Expr Int
    Div     :: Expr Int -> Expr Int -> Expr Int   -- new — genuinely CAN fail at eval time
    If      :: Expr Bool -> Expr a -> Expr a -> Expr a
```

`Div` type-checks fine for any two `Expr Int` — but dividing by zero is a real, runtime possibility no type signature alone can rule out, which is exactly why evaluation needs to be monadic.

Monadic Evaluation — evalM Returns Either String a

```haskell
evalM :: Expr a -> Either String a   -- haskell1-6's own Either, haskell2-3's own >>= chaining
evalM (IntLit n)  = Right n
evalM (BoolLit b) = Right b
evalM (Add e1 e2) = do
    v1 <- evalM e1
    v2 <- evalM e2
    Right (v1 + v2)
evalM (Div e1 e2) = do
    v1 <- evalM e1
    v2 <- evalM e2
    if v2 == 0 then Left "division by zero" else Right (v1 `div` v2)
evalM (If cond t f) = do
    c <- evalM cond
    if c then evalM t else evalM f
```

Each sub-evaluation is chained with `>>=`-powered `do`-notation — if any sub-expression fails, `haskell2-3`'s own short-circuit-on-`Left` behavior propagates the error automatically, with no manual error-checking at every step.

A Pretty-Printing Typeclass

```haskell
class Pretty a where   -- a real custom typeclass, per haskell1-8 / haskell2-5
    pretty :: a -> String

instance Pretty (Expr a) where
    pretty (IntLit n)  = show n
    pretty (BoolLit b) = show b
    pretty (Add e1 e2) = "(" ++ pretty e1 ++ " + " ++ pretty e2 ++ ")"
    pretty (Div e1 e2) = "(" ++ pretty e1 ++ " / " ++ pretty e2 ++ ")"
    pretty (If c t f)  = "if " ++ pretty c ++ " then " ++ pretty t ++ " else " ++ pretty f
```

A Small IO Shell

```haskell
runExample :: Show a => Expr a -> IO ()   -- haskell2-4's own pure-core/IO-shell pattern
runExample expr = do
    putStrLn (pretty expr)
    case evalM expr of
        Right v  -> putStrLn ("  = " ++ show v)
        Left err -> putStrLn ("  ERROR: " ++ err)

main :: IO ()
main = do
    runExample (Add (IntLit 3) (IntLit 4))
    runExample (Div (IntLit 10) (IntLit 0))
    runExample (If (BoolLit True) (IntLit 1) (IntLit 2))
```

`evalM` and `pretty` are both completely free of `IO` in their own types — fully testable with ordinary function calls, no mocking required. Only `main` itself touches `IO`, exactly `haskell2-4`'s own recommended shape.

Chapter Attribution

CAPSTONE PIECE	CHAPTER
Expr a GADT	**haskell2-7**
Either-based error handling	**haskell1-6**
evalM's do-notation / >>= chaining	**haskell2-3**
Pattern matching on constructors	**haskell1-7**
The custom Pretty typeclass	**haskell1-8, haskell2-5**
Pure core / thin IO shell (main only touches IO)	**haskell2-4**

What's Still Out of Scope

Honestly: no real parser or lexer — expressions are built directly in Haskell code, not typed as text by a user, so this isn't a genuine interactive REPL. No monad transformers used here at all, despite `haskell2-6` covering them — `evalM`'s own error handling only needs plain `Either`, a deliberate scope decision, not an oversight. No phantom types beyond the GADT's own per-constructor typing. This capstone proves the pieces fit together, not that the result is a production interpreter.

> **A CAPSTONE'S VALUE IS IN THE SEAMS, NOT THE SIZE**

This project is deliberately small — the point was never scale, it was confirming that GADTs, monadic error handling, custom typeclasses, and a disciplined pure-core/IO-shell split genuinely compose cleanly together in real code.

THIS IS A TEACHING CAPSTONE, NOT A TEMPLATE FOR A REAL INTERPRETER

A real interpreter would need actual parsing, a proper error-reporting story (source locations, not just a message string), and tests — treat this as proof the language features fit together, not as production-ready code to copy.

Coding Challenges

CHALLENGE 1

Add a new GADT constructor Mul :: Expr Int -> Expr Int -> Expr Int alongside Add, update evalM and pretty to handle it, and test it with a small expression combining Add and Mul.

View solution

CHALLENGE 2

Write a nested expression that divides by zero INSIDE a larger Add expression (e.g. Add (IntLit 5) (Div (IntLit 10) (IntLit 0))), run it through evalM, and confirm the error propagates out of the whole expression correctly, explaining why in a comment.

View solution

CHALLENGE 3

Write a short paragraph (as a comment) explaining what would need to change in this capstone if it needed to ALSO thread a variable-count "operations performed" counter through evaluation, referencing haskell2-6's own StateT material directly.

View solution

CHAPTER 8 QUICK REFERENCE — HASKELL TRACK COMPLETE

- A GADT lets each constructor declare its own specific return type, enabling a genuinely type-safe expression language (haskell2-7)
- evalM's Either-based do-notation chains sub-evaluations, short-circuiting automatically on failure (haskell1-6, haskell2-3)
- A custom Pretty typeclass demonstrates real, practical ad-hoc polymorphism (haskell1-8, haskell2-5)

- evalM and pretty stay completely free of IO — only main touches it, haskell2-4's own recommended shape
- Monad transformers, real parsing, and phantom types are honestly named as still out of scope
- **Both Haskell courses are now complete — 16 chapters total, framed throughout as the real origin of ideas already met on this site as Rust's Option/Result/traits.**