λ

Haskell Fundamentals

A Complete 8-Chapter Programming Course

Topics covered:

GHC/GHCi & main :: IO () · Hindley-Milner inference & currying

Immutability & referential transparency · Lists & recursion-only looping

Lazy evaluation & infinite structures · Algebraic data types

Pattern matching & exhaustiveness · Typeclasses

Exercises: 24 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed as the real origin of Rust's Option/Result/traits

Course 1 of 2 · Intermediate/Advanced follows

Table of Contents

CHAPTER 1 OF 8

Getting Started

COURSE 1 · CH 1

Getting Started

main :: IO () — the type signature that carries this entire course's throughline

Every language on this site so far has been imperative or object-oriented in flavor — even the ones with functional features bolted on (Java's streams, C#'s LINQ) are still fundamentally "do this, then this" languages underneath. Haskell isn't. This course treats it as what it genuinely is: the real origin point for ideas already met elsewhere on this site under different names — Rust's `Option`/`Result`, pattern matching, even the whole idea of a trait. This first chapter states the throughline that runs through the entire track: in Haskell, a function's type doesn't just describe its inputs and outputs — it tells you whether that function can have side effects at all.

GHC and GHCi

```
$ ghci
Prelude> 2 + 2
4
Prelude> :quit
```

GHC (the Glasgow Haskell Compiler) is the standard toolchain; **GHCi** is its interactive REPL — similar in spirit to Python's or Node's own REPLs, useful for quick experiments before committing to a real file.

Your First Program

```
main :: IO ()
main = putStrLn "Hello, World!"
```

Two lines: a **type signature** (`main :: IO ()`) and the actual **definition** (`main = ...`) below it. Here's the reveal this whole course is built around: `IO ()` is not decoration or convention — it's a real, compiler-checked part of the type system, stating plainly that `main` is a computation which may perform IO and produces no meaningful value (`()`, pronounced "unit"). Every other language covered on this site writes a

`main` with a signature that says nothing at all about what it does — `void main()`, `static void Main()`, no signature whatsoever in Python. Haskell's says it, in the type, from line one.

The Type Signature Convention

`::` reads as "has type." GHC can infer nearly every type signature on its own — but by strong, near-universal community convention, every top-level function gets an explicit signature anyway, written directly above its definition. This is a real stated best practice, not busywork: it documents intent and catches a mismatch between what a function was meant to do and what it actually type-checks as, immediately, at the point of definition rather than somewhere downstream.

Laziness — Present From Line One

```
Prelude> take 3 [1..]
[1,2,3]  -- [1..] is an INFINITE list — nothing crashes or hangs
```

One small preview before Chapter 5's full treatment: `[1..]` genuinely describes an infinite list, and nothing about writing it crashes the program — Haskell only computes values as they're actually demanded, and `take 3` only ever demands three of them. This is the language's default evaluation strategy, not a special trick, and it touches nearly everything from here on.

Running Programs — Compiled or Interpreted

```
$ ghc Main.hs -o hello   # compiles to a real native binary
$ ./hello

$ runghc Main.hs          # "interpreted" — but really compiled in memory first, then run
```

`ghc` compiles straight to native machine code — no bytecode, no VM, unlike Java's or C#'s own JIT-compiled-bytecode model. `runghc` feels interpreted for quick iteration, but it's genuinely compiling in memory first; Haskell is always compiled, never truly interpreted the way Python is.

LANGUAGE	MAIN'S SIGNATURE	DOES IT REVEAL IO CAPABILITY?
Java	`public static void main(String[] args)`	no
C#	`static void Main()`	no
Python	`no signature at all`	no

LANGUAGE	MAIN'S SIGNATURE	DOES IT REVEAL IO CAPABILITY?
Haskell	`main :: IO ()`	`yes — directly, in the type`

WRITE EXPLICIT TYPE SIGNATURES, EVEN THOUGH GHC CAN INFER THEM

A top-level type signature is cheap to write and expensive to skip — it documents intent, and a mismatch between the signature and the actual definition is caught immediately as a real compile error, rather than surfacing later as a confusing inferred type somewhere downstream.

LAZINESS MEANS AN ERROR CAN HIDE IN PLAIN SIGHT

Because nothing is evaluated until it's actually demanded, an error buried in a part of an expression that's never forced may simply never surface at all — a genuinely surprising first encounter for anyone used to eager, top-to-bottom execution. Chapter 5 covers this properly.

Coding Challenges

CHALLENGE 1

Write a Main.hs with a main :: IO () that prints two separate lines of your choosing using two putStrLn calls, and run it with runghc.

View solution

CHALLENGE 2

In GHCi, evaluate take 5 [10..] and explain in a comment why this doesn't hang or crash despite [10..] describing an infinite list.

View solution

CHALLENGE 3

Write a short comment contrasting Java's public static void main(String[] args) against Haskell's main :: IO (), explaining specifically what information Haskell's version encodes that Java's does not.

View solution

CHAPTER 1 QUICK REFERENCE

- GHC compiles to native code directly; GHCi is the interactive REPL
- main :: IO () — the type signature itself states this computation may perform IO, unlike any other language's own main signature
- :: reads as "has type" — explicit top-level signatures are strong convention even though GHC can infer them
- Haskell is lazy by default — [1..] is a real infinite list, safe because nothing is computed until demanded
- ghc always compiles to native code; runghc feels interpreted but compiles in memory first
- **Next chapter:** functions and types — Hindley-Milner inference and currying as the default

CHAPTER 2 OF 8

Functions & Types

COURSE 1 · CH 2

Functions & Types

Every function secretly takes exactly one argument — the arrows in a type signature aren't decoration

`haskell1-1`'s type signatures used a single arrow. This chapter explains what happens once there's more than one — and it's a genuine, structural surprise for anyone arriving from a language where "a two-argument function" is simply a fact, not a disguise.

Function Definitions & Type Signatures

```
add :: Int -> Int -> Int
add x y = x + y
```

A type signature reads left to right, each `->` separating one more piece from the rest. This chapter is entirely about taking that arrow-chain seriously — it's the first real hint of what's coming.

Hindley-Milner Type Inference

GHC can infer almost any type without a single annotation, using an algorithm called Hindley-Milner. This is a genuinely stronger guarantee than `ts1-2`'s own TypeScript inference: Hindley-Milner is *complete* for a large, well-defined core of the language — it always finds the single most general type an expression could have, with no ambiguity and no escape hatches needed. TypeScript's inference is real and useful, but structural, local, and leans on `any`/assertions when it can't fully work something out; Haskell's core inference doesn't have that kind of fallback because it doesn't need one.

Currying — Every Function Takes Exactly One Argument

```
add :: Int -> Int -> Int
-- really means:
add :: Int -> (Int -> Int)   -- a function taking one Int, RETURNING another function

addFive = add 5       -- addFive :: Int -> Int — a genuinely new, real function
addFive 3              -- 8
```

Here's the reveal: `add :: Int -> Int -> Int` is really `Int -> (Int -> Int)` — a function that takes one `Int` and returns *another function*, which itself takes one `Int` and returns the final `Int`. Multi-argument functions don't actually exist in Haskell; every one of them is a chain of one-argument functions. `add 5` genuinely produces a real, callable function — `addFive` — not a placeholder or a partial-application trick bolted on afterward.

Partial Application in Practice

Because currying is how every function already works, partial application isn't a special feature some functions opt into — *any* function can be partially applied, always, automatically. This is a real, structural difference from `csharp2-3`'s own `Func<>` delegates, where building a specialized version of a general function means writing a new lambda wrapper by hand — Haskell just gives you the specialized function directly, for free, from the same definition.

Polymorphic Types — A First Look

```
id :: a -> a
id x = x
```

Lowercase type variables (`a`) mark a genuinely polymorphic function — comparable in spirit to `java2-1`'s own `<T>` generics, but without that chapter's own erasure-vs-reification question ever coming up the same way; Haskell's approach is closer to true parametric polymorphism. This is only a first look — full typeclass-based polymorphism arrives properly in Chapter 8.

ASPECT	TYPESCRIPT (TS1-2)	C# (CSHARP2-3)	HASKELL
Type inference	structural, local, escape hatches (any)	var, local only	Hindley-Milner — complete, most-general type
Multi-argument functions	a real, direct feature	a real, direct feature	a chain of one-argument functions (curried)
Partial application	manual wrapper needed	manual lambda wrapper needed	automatic, for every function, always

USE PARTIAL APPLICATION INSTEAD OF WRITING WRAPPER LAMBDAS

Where another language needs a hand-written lambda to specialize a general function, Haskell's currying already provides it directly — `add 5` alone is the specialized function, no wrapper required.

AN ARROW-CHAIN SIGNATURE IS NOT "N INPUTS, ONE OUTPUT" AS A SINGLE UNIT

`Int -> Int -> Int` is genuinely a chain of two one-argument functions, not shorthand for "takes two Ints, returns one." Reading it that way works for calling it normally, but breaks down the moment partial application enters the picture — worth internalizing the real structure early.

Coding Challenges

CHALLENGE 1

Write a function multiply :: Int -> Int -> Int with an explicit type signature, then use partial application to create a triple function that multiplies its argument by 3, without writing a lambda.

View solution

CHALLENGE 2

In GHCi, check the type of a partially applied function (e.g. :t (add 5)) and explain in a comment why its type is Int -> Int rather than something else.

View solution

CHALLENGE 3

Write a short comment explaining why Int -> Int -> Int is really Int -> (Int -> Int), and why this specific structure is what makes automatic partial application possible for every function.

View solution

CHAPTER 2 QUICK REFERENCE

- Hindley-Milner inference is complete for a large language core — always finds the most general type, no escape hatches needed, a stronger guarantee than ts1-2's own inference
- Every Haskell function secretly takes exactly one argument — Int -> Int -> Int really means Int -> (Int -> Int)
- Partial application is automatic for every function, always — no manual wrapper lambdas needed, unlike csharp2-3's own Func<> delegates

- Lowercase type variables (a -> a) mark genuine parametric polymorphism — a first look, full typeclass polymorphism comes in Chapter 8
- **Next chapter:** immutability and pure functions — no reassignment exists at all, not just opt-in

CHAPTER 3 OF 8

Immutability & Pure Functions

COURSE 1 · CH 3

Immutability & Pure Functions

Not opt-in like rust1-2's own mut — there is no reassignment mechanism to opt into at all

`rust1-2` made immutability the default, one keyword (`mut`) away from mutability whenever it's genuinely needed. Haskell doesn't offer that escape hatch in the first place — this chapter explains what that actually buys, precisely.

No Reassignment, Ever

```
x = 5
-- x = 6   -- not "immutable by default" — this simply doesn't mean what it looks like it means
```

`x = 5` is not an assignment — it's a **binding**, a permanent name-to-value association. There is no `mut` equivalent anywhere in the pure part of the language. Once `x` is bound, there is genuinely no way to reassign it, ever, anywhere in its scope — a stronger guarantee than `rust1-2` 's own default, which is real immutability, but only until `mut` is written.

"Variables" Aren't Variable

The word "variable" is a slight misnomer inherited from mathematics, not from imperative programming. In Haskell, `x` behaves exactly like a variable in an algebraic equation — a fixed name standing for a fixed value within its scope — not a storage *location* that can hold different things over time, the way `c1-2` 's and `java1-2` 's own variables genuinely can.

Referential Transparency

```
x = 5
result = x + x   -- can always be replaced by "10" — everywhere, safely, forever
```

An expression is **referentially transparent** if it can be replaced by its value with zero change to the program's behavior — anywhere, at any time. Since `x` is never reassigned, `x + x` can always be swapped for `10`. This is a genuine, checkable algebraic property no mutable-variable language on this site can offer: in C or Java, `x + x` might legitimately produce a different result on a second evaluation if something reassigned `x` in between.

Why This Matters — Equational Reasoning

A direct, practical consequence: Haskell code can be reasoned about the way algebra is — substitute equals for equals, refactor by simple substitution, and trust that the substitution never changes meaning. This is genuinely different from tracing through an imperative program, where "what is `x` right now" depends on execution history and ordering, not just the code as written.

"Mutation" via Shadowing and let/where

```
let x = 5 in let x = x + 1 in x  -- 6 — but this is NOT reassignment
```

This can look exactly like reassignment — it isn't. The inner `x` is a genuinely new, separate binding that happens to reuse the name `x`, **shadowing** the outer one. The original binding is untouched and still exists, just no longer reachable by that name in the inner scope. `rust1-2` also has shadowing, but Rust's shadowing coexists alongside real mutation via `mut` — two genuinely distinct mechanisms that can look similar. Haskell only has the one, since it has no true mutation to distinguish shadowing from in the first place.

ASPECT	C/JAVA	RUST (RUST1-2)	HASKELL
Default mutability	mutable	immutable	no mutation mechanism exists
Opting into mutation	always available	mut keyword	not possible for a binding
Shadowing	limited/scoped	real, coexists with mut	the only "reassignment-looking" mechanism there is
x + x always substitutable by its value	no	yes, if x is not mut	yes, always

READ = AS MATHEMATICAL EQUALITY, NOT AN IMPERATIVE STATEMENT

Treating `x = 5` as "define x to mean 5," the same way algebra uses `=`, avoids most of the early confusion that comes from expecting it to behave like an assignment statement in an imperative language.

SHADOWING CAN LOOK EXACTLY LIKE MUTATION AT A GLANCE

Repeated `let x = ...` lines inside a `do` block can easily read as reassignment on a first pass — it's genuinely a fresh binding each time, not the same `x` being updated. Worth double-checking scope carefully the first several times this comes up.

Coding Challenges

CHALLENGE 1

In GHCi, bind x = 10, then attempt to write a second line rebinding x = 20 at the top level in the same session, and report what actually happens (a genuine rebinding at the top level of GHCi vs. what would happen inside a single expression's scope).

View solution

CHALLENGE 2

Write a small expression using nested let...in bindings that shadows the same name twice, printing the final value, and explain in a comment which binding is actually in scope at each point.

View solution

CHALLENGE 3

Write a short comment explaining why x + x being always substitutable by its value is impossible to guarantee in a language like Java, tying your answer to referential transparency.

View solution

CHAPTER 3 QUICK REFERENCE

- x = 5 is a permanent binding, not an assignment — no mut equivalent exists for it anywhere
- Haskell's "variables" behave like mathematical variables — fixed names for fixed values, not storage locations
- Referential transparency: an expression can always be replaced by its value, everywhere, safely — a real, checkable property mutable-variable languages can't offer
- Equational reasoning follows directly — Haskell code can be refactored by substitution, the way algebra is
- Shadowing (let x = ... in let x = ... in ...) looks like reassignment but creates a genuinely new binding — the only such mechanism, since Rust's own shadowing (rust1-2) coexists with real mutation and Haskell's doesn't
- **Next chapter:** lists and recursion — recursion as the only way to loop, no for/while exist at all

CHAPTER 4 OF 8

Lists & Recursion

COURSE 1 · CH 4

Lists & Recursion

No for, no while — the "wait, what?" moment haskell1-1 promised

`haskell1-1` promised a genuine surprise once this chapter arrived. Here it is: Haskell has no looping construct at all. Every repeated computation in this language is written as a recursive function — and after `haskell1-3`'s own immutability chapter, the reason why is actually a direct, structural consequence, not a stylistic choice.

Haskell Lists — Syntax & Basics

```
nums = [1, 2, 3]
nums2 = 1 : 2 : 3 : []   -- identical to [1,2,3] — : is the cons operator
head nums   -- 1
tail nums   -- [2,3]
```

A list is either empty (`[]`) or an element consed onto another list (`:`). All elements share one type — `[1, 2, "three"]` doesn't type-check at all, a real difference from a dynamically-typed language's own list literals.

No for/while Loops — Recursion Is the Only Way

```
sumList :: [Int] -> Int
sumList [] = 0                    -- base case
sumList (x:xs) = x + sumList xs   -- recursive case — x is the head, xs is everything else
```

There is no `for`, no `while`, anywhere in Haskell. A "loop" is written as two pattern-matched cases: a **base case** for the smallest input, and a **recursive case** that handles one piece and delegates the rest back to the same function.

Why No Loops?

This connects directly back to `haskell1-3`: a `for` loop in an imperative language works by mutating a counter variable on each pass. Haskell has no mutation at all — the entire mechanism a `for` loop depends on simply doesn't exist to build one out of. Recursion isn't a stylistic preference here; it's the only mechanism left once mutation is off the table.

List Comprehensions

```
evenDoubles = [x * 2 | x <- [1..10], even x]  -- [4,8,12,16,20]
```

Another "this idea started here" moment: `py1-6`'s own first look at Python's list comprehensions covered syntax that was explicitly borrowed *from* Haskell's (and SETL's) own comprehension notation. `[expression | generator, condition]` reads almost identically in both languages — a real, documented lineage, not a coincidental similarity.

Standard Recursive Patterns

```
myMap :: (a -> b) -> [a] -> [b]
myMap _ [] = []
myMap f (x:xs) = f x : myMap f xs   -- map itself is just... recursion
```

`map`, `filter`, and `foldr` / `foldl` aren't special language magic — they're ordinary recursive functions living in the standard library, built exactly the same shape as `sumList` above. `myMap` here reimplements the real `map`'s own logic directly, demystifying it completely.

Tail Recursion, Briefly

A real, honest performance note: naive recursion can build up stack frames the way `sumList` above does, since each call waits on the result of the next before it can finish its own addition. This isn't free, and it interacts with Chapter 5's own laziness material in ways that are sometimes genuinely surprising — full treatment is deferred there rather than glossed over here.

ASPECT	C/JAVA/PYTHON FOR-LOOPS	PYTHON COMPREHENSIONS (PY1-6)	HASKELL
Mechanism	`mutate a counter each pass`	`a real language feature`	`no loop construct — recursion only`
Comprehension syntax	`n/a`	`[expr for x in seq if cond]`	`[expr | x <- seq, cond] — the real ancestor`

ASPECT	C/JAVA/PYTHON FOR-LOOPS	PYTHON COMPREHENSIONS (PY1-6)	HASKELL
map/filter	library functions, often loop-backed internally	library functions	library functions, themselves plain recursion

REACH FOR MAP/FILTER/FOLD BEFORE HAND-WRITING RECURSION

The same instinct that favors LINQ or Streams over a hand-written loop in other languages already covered on this site applies here — `map` / `filter` / `fold` cover the vast majority of real list-processing needs without writing a new base-case/recursive-case pair each time.

A MISSING OR UNREACHABLE BASE CASE RECURSES FOREVER

The Haskell-flavored version of the missing-`break`/infinite-loop bug class this site's own `python_lesson_infinite_loops.html` explored for Python's `while` loops: a recursive function with no base case, or one whose base case is never actually reached, recurses without end — producing a stack overflow rather than a silent hang, but the same underlying "the exit condition was never satisfied" mistake.

Coding Challenges

CHALLENGE 1

Write a recursive function myLength :: [a] -> Int that computes a list's length using only a base case and a recursive case, with no built-in length function.

View solution

CHALLENGE 2

Write a list comprehension that produces the squares of every odd number from 1 to 20, and write the equivalent using filter and map instead, showing both produce the same result.

View solution

CHALLENGE 3

Write a recursive function with a base case that can never actually be reached for certain inputs (e.g. counting down by 2 from an odd starting number toward a base case of exactly 0), explain why it never terminates for those inputs, and fix it.

View solution

CHAPTER 4 QUICK REFERENCE

- Lists are built from [] and : (cons); all elements share one type
- No for/while exist in Haskell — every loop is a recursive function with a base case and a recursive case
- No loops exist BECAUSE no mutation exists (haskell1-3) — recursion is the only mechanism left, not a stylistic choice
- List comprehensions are the real syntactic ancestor of py1-6's own Python comprehensions, not a coincidental lookalike
- map/filter/fold are themselves plain recursive functions in the standard library, not special language magic
- A missing or unreachable base case recurses forever — the same "exit condition never satisfied" mistake covered for Python's while loops
- **Next chapter:** lazy evaluation — the language's default strategy for everything, not an opt-in feature

CHAPTER 5 OF 8

Lazy Evaluation

COURSE 1 · CH 5

Lazy Evaluation

The full treatment haskell1-1 promised — and a genuine scope difference from java2-4's own opt-in Stream laziness

`haskell1-1` previewed `take 3 [1..]` and promised this chapter would explain why it works. Here's the full answer — and it goes well beyond one convenient trick.

What Laziness Actually Means

An expression isn't evaluated until its *value* is actually demanded. This is the opposite of **eager** (or "strict") evaluation, which every other language on this site uses by default — C, Java, Python, C#, Rust all evaluate a function's arguments before the function body ever runs. Haskell evaluates an argument only if, and only when, the function body actually inspects it.

Infinite Data Structures, For Real

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)   -- a list defined in terms of ITSELF

take 10 fibs   -- [0,1,1,2,3,5,8,13,21,34]
```

`fibs` is genuinely, structurally infinite and self-referential — it's defined using itself. This is only possible because laziness means the list is never "fully built" anywhere — only as much of it as actually gets demanded is ever computed. `take 10 fibs` works fine; printing `fibs` directly would run forever.

undefined and Bottom (⊥) — Errors That Never Surface

```
let x = undefined in 5   -- evaluates to 5, no error, ever — x is never demanded
```

`haskell1-1`'s own warn-box previewed this — here's the full explanation. **Bottom** (⊥) is the formal name for a computation that errors, or loops forever, or is otherwise "unfinished." Because Haskell only forces what's

demanded, a bottom value can sit embedded inside a data structure and never actually detonate, as long as nothing ever forces it. This is a real, structural consequence of laziness, not an edge case.

WHNF — Weak Head Normal Form, Briefly

Laziness doesn't mean "left fully unevaluated until printed" — it means evaluated only as far as strictly necessary to make a decision, such as pattern-matching on a constructor. That partial-evaluation point has a real name, **weak head normal form** (WHNF) — worth knowing the term exists, without needing the full academic definition to use laziness correctly day to day.

Contrasted with java2-4's Stream Laziness

`java2-4`'s own Java Streams are genuinely lazy — intermediate operations don't run until a terminal operation triggers them. But that's an **opt-in feature of one specific API**; ordinary Java expressions, method calls, and arithmetic are all strictly eager by default, Streams or not. Haskell's laziness applies to *every* expression in the entire language, all the time, with no separate "lazy API" to reach for — a genuine difference in scope, not just a similarity in spirit.

seq and Forcing Strictness When Needed

```
x `seq` y   -- forces x to WHNF before returning y — a real, honest escape hatch
```

Laziness isn't free — `haskell1-4`'s own tail-recursion aside foreshadowed this. A long chain of unevaluated computations (called **thunks**) can build up and eventually need forcing all at once, which can blow the stack. `seq` forces evaluation to WHNF immediately, on purpose, when a programmer decides laziness is working against them in a specific spot — a real, practical tool, not an admission that laziness was a mistake.

ASPECT	C/JAVA/PYTHON/C#/RUST	JAVA STREAMS (JAVA2-4)	HASKELL
Default evaluation	`eager, everywhere`	`eager, everywhere except Streams`	`lazy, everywhere`
Laziness scope	`none`	`one specific API, opt-in`	`the entire language, always on`
Genuinely infinite structures	`not directly possible`	`Stream.iterate() can approximate it`	`a routine, everyday pattern`

USE TAKE/HEAD TO SAFELY PEEK AT EXPENSIVE OR INFINITE STRUCTURES

Rather than trying to force an entire structure to check it, ask for only what's actually needed — `take 10 fibs` is always safe; forcing all of `fibs` at once never is.

LAZINESS CAN CAUSE SPACE LEAKS — DEFERRED COMPUTATION CAN COST MORE MEMORY, NOT LESS

A long, unevaluated chain of thunks can consume genuinely more memory than the equivalent eager computation would have, since each deferred step has to be remembered until something finally forces it. This is a real, well-documented Haskell-specific gotcha — a leak from deferring computation too long, not from failing to free memory the way `c2-2`'s own manual-memory leaks work.

Coding Challenges

CHALLENGE 1

Define an infinite list of all even numbers using a self-referential or generator-based definition, and use take to print the first 8.

View solution

CHALLENGE 2

Write an expression using let ... in that binds a name to undefined but never actually uses that binding in the final result, and show the expression still evaluates successfully.

View solution

CHALLENGE 3

Write a short comment explaining the real difference in SCOPE between Haskell's laziness and java2-4's own Stream laziness, using a concrete example of ordinary (non-Stream) Java code that would NOT be lazy even though Streams are.

View solution

CHAPTER 5 QUICK REFERENCE

- Values are computed only when their value is actually demanded — the opposite of every other language on this site's own eager default
- Self-referential infinite lists (fibs = 0 : 1 : zipWith (+) fibs (tail fibs)) work because nothing forces the whole structure at once

- undefined embedded but never forced never errors — the formal concept is called bottom (⊥)
- WHNF is evaluation only as far as necessary to make a decision, not necessarily all the way down
- Haskell's laziness is the language's own default for everything — a real scope difference from java2-4's opt-in, Streams-only laziness
- seq forces strict evaluation deliberately — a real escape hatch, not a contradiction of laziness's value
- Space leaks (too many deferred thunks) are a genuine, distinct Haskell performance gotcha — different in kind from other languages' own memory leaks
- **Next chapter:** algebraic data types — the real ancestor of rust1-6's own enums and Option/Result

CHAPTER 6 OF 8

Algebraic Data Types

COURSE 1 · CH 6

Algebraic Data Types

Maybe and Either aren't magic — they're ordinary data declarations, and the real ancestor of rust1-6's Option and Result

This is one of the two chapters (alongside Chapter 8) where this course's own "origin point" framing pays off most directly and concretely. What `rust1-6` introduced as `Option<T>` and `Result<T, E>` — genuinely modeled on exactly what's covered here.

data Declarations — Product Types

```
data Point = Point Int Int   -- a "product" type — bundles two values together

origin = Point 0 0
```

`Point` bundles two `Int`s together — a **product type**, named because the number of distinct `Point` values is the product of each field's own possible values. Structurally comparable to `csharp2-5`'s own positional records, decades earlier.

Sum Types — Multiple Shapes, One Type

```
data Shape = Circle Double | Rectangle Double Double   -- a "sum" type — EITHER shape, never both
```

This is the real deal: a value of type `Shape` is *either* a `Circle` *or* a `Rectangle`, never a hybrid of both — a **sum type**, named because the total possibilities are the sum of each variant's own. This is the direct genetic ancestor of `rust1-6`'s own data-carrying enums.

The Real Lineage — Maybe and Either ARE Just ADTs

```haskell
-- from the standard library — not special, not magic, just ordinary data:
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Here's the central reveal: `Maybe` and `Either` aren't built-in language magic — they're defined using the exact same `data` syntax available to any programmer, sitting in the standard library like any other type. `rust1-6`'s own `Option<T>` and `Result<T, E>` were explicitly modeled on these two — `None` / `Some` maps directly onto `Nothing` / `Just`, and `Err` / `Ok` maps directly onto `Left` / `Right`. Same shape, same idea, arriving in Haskell decades earlier.

Recursive Data Types

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)   -- a real binary tree, in one line
```

A `data` type can reference itself — a genuine binary tree needs no more than this. Recursive data alongside `haskell1-4`'s own recursive functions, and the two ideas reinforce each other constantly: processing a `Tree` almost always means writing a recursive function over it.

Record Syntax for Product Types

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

alice = Person { name = "Alice", age = 30 }
name alice   -- "Alice" — a real accessor function, generated automatically
```

Named-field syntax, comparable again to `csharp2-5`'s own record syntax — Haskell's own version predates C#'s by decades too, though honestly, it carries a real historical wart C#'s doesn't share, covered next.

CONCEPT	RUST (RUST1-6)	C# (CSHARP2-5)	HASKELL
Product type	`struct`	`record (positional)`	`data with one constructor`
Sum type	`enum with data`	`no direct built-in equivalent`	`data with | alternatives`
"Optional value" type	`Option<T> — modeled on Maybe`	`nullable reference types (csharp2-6)`	`Maybe a — the real origin`
"Success or failure" type	`Result<T, E> — modeled on Either`	`exceptions (csharp1-7)`	`Either a b — the real origin`

REACH FOR A SUM TYPE WHENEVER A VALUE IS GENUINELY ONE OF SEVERAL DISTINCT SHAPES

The same instinct `rust1-6`'s own enum design encourages applies directly here — model "this OR that, never both" as a real sum type rather than a product type with unused/optional fields simulating the same idea.

CLASSIC RECORD SYNTAX HAS A REAL FIELD-NAMING LIMITATION

Two different `data` types in the same module genuinely cannot both have a field called `name` without a real conflict — unlike `csharp2-5`'s own records or Java's classes, where every field is automatically scoped to its own type with no possibility of collision. This is a well-documented, honest wart in Haskell's classic record syntax, not a made-up gotcha.

Coding Challenges

CHALLENGE 1

Define a sum type TrafficLight with three constructors (Red, Yellow, Green, no arguments needed), and a function next :: TrafficLight -> TrafficLight that cycles Red -> Green -> Yellow -> Red.

View solution

CHALLENGE 2

Write your own version of Maybe called MyMaybe (data MyMaybe a = MyNothing | MyJust a) and a function safeDivide :: Int -> Int -> MyMaybe Int that returns MyNothing on division by zero and MyJust result otherwise.

View solution

CHALLENGE 3

Define a recursive Tree a data type and write a function treeSum :: Tree Int -> Int that recursively sums every value in the tree, testing it on a small tree with at least 4 nodes.

View solution

CHAPTER 6 QUICK REFERENCE

- Product types (data Point = Point Int Int) bundle multiple values — comparable to csharp2-5's positional records
- Sum types (data Shape = Circle Double | Rectangle Double Double) are the real ancestor of rust1-6's own data-carrying enums

- Maybe and Either are ordinary standard-library data types, not language magic — Nothing/Just and Left/Right are the real origin of Rust's None/Some and Err/Ok
- A data type can reference itself — real recursive data (Tree a), reinforcing haskell1-4's own recursive functions
- Classic record syntax has a genuine field-naming collision limitation across types in the same module — a real wart C#'s own records don't share
- **Next chapter:** pattern matching — the real origin of the style rust1-6's match and csharp2-5's own pattern matching both descend from

CHAPTER 7 OF 8

Pattern Matching

COURSE 1 · CH 7

Pattern Matching

The real origin of the style rust1-6's match and csharp2-5's own pattern matching both descend from

Every function written since `haskell1-4` has already used pattern matching, informally — `sumList`'s `[]` / `(x:xs)` equations, `haskell1-6`'s own `next`. This chapter formalizes it, and traces the real lineage to Rust's `match` and C#'s own switch patterns.

Matching on Constructors

```
area :: Shape -> Double
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h
```

Multiple function equations *are* pattern matching — each one matches a specific constructor shape, tried top to bottom until one succeeds. This has been in use since Chapter 4 without being named.

case Expressions

```
describeShape :: Shape -> String
describeShape s = case s of
    Circle r      -> "a circle"
    Rectangle w h -> "a rectangle"
```

`case` is the explicit, single-expression form of what multiple top-level equations do implicitly — useful when pattern matching is only *part* of a function body, not the whole thing.

Guards

```
classify :: Int -> String
classify n
  | n < 0    = "negative"
  | n == 0   = "zero"
  | otherwise = "positive"
```

A **guard** (`| condition`) adds a boolean condition on top of a successful pattern match. This is the direct conceptual ancestor of C#'s own `when` clause in `csharp2-5`'s switch patterns, and Rust's own match guards — Haskell had this decades before either.

As-Patterns

```
firstTwo :: [a] -> ([a], a)
firstTwo all@(x:_) = (all, x)   -- 'all' binds the WHOLE list, x binds just the head
```

`all@(x:_)` binds the entire matched value to `all` *while also* binding its parts (`x`) separately — using both the pieces and the whole without re-matching or reconstructing anything. A genuinely nice convenience with no exact equivalent named in either Rust's or C#'s own pattern syntax on this site.

Exhaustiveness Checking

```
area :: Shape -> Double
area (Circle r) = pi * r * r
-- missing the Rectangle case — GHC WARNS about this by default
```

GHC can warn — and, with a flag, hard-error — when a set of patterns doesn't cover every constructor of a type. Name this plainly: Haskell's pattern matching was exhaustiveness-checked long before `rust1-6`'s own `match` or `csharp2-7`'s own exhaustive switch over sealed types existed. Both of those are, in a real sense, downstream of this exact idea.

Nested Patterns

```
describe :: Maybe (Tree Int) -> String
describe Nothing            = "no tree"
describe (Just Leaf)        = "empty tree"
describe (Just (Node _ v _)) = "tree rooted at " ++ show v
```

Patterns nest arbitrarily deep — matching straight into a `Maybe (Tree a)` in one shot, with no chain of manual unwrapping required.

FEATURE	RUST MATCH	C# SWITCH (CSHARP2-5)	HASKELL — THE ORIGIN
Exhaustiveness enforcement	yes, compile error	yes, over sealed types	warning by default, error with a flag — decades earlier
Guards	match guards	when clause	| guard — the real ancestor of both
Bind whole + parts at once	@ bindings, similar spirit	not directly available	as-patterns (x@pattern)

TURN ON -WINCOMPLETE-PATTERNS (OR MAKE IT A HARD ERROR)

`-Wincomplete-patterns` (and `-Werror=incomplete-patterns` to make it fail the build) catches a missing case at compile time — the same safety Rust's compiler-enforced exhaustiveness gives by default, opt-in but genuinely worth enabling on every project.

A REACHED INCOMPLETE PATTERN THROWS A REAL, UNRECOVERABLE EXCEPTION

Without the flag above, an incomplete pattern match that's actually hit at runtime throws `"Non-exhaustive patterns"` — a genuine runtime crash, not something GHC's own compile-time warning catches automatically unless it's explicitly promoted to a hard error.

Coding Challenges

CHALLENGE 1

Write a function grade :: Int -> String using guards that returns "A", "B", "C", or "F" based on a numeric score, with otherwise as the final fallback.

View solution

CHALLENGE 2

Write a function summarizeList :: [Int] -> String using an as-pattern that returns a string mentioning both the full list and its first element, for a non-empty list, and a distinct message for the empty list.

View solution

CHALLENGE 3

Write a function over the Shape type from haskell1-6 that deliberately omits one constructor's pattern, compile it with -Wincomplete-patterns, and report the resulting warning text. Then fix it.

View solution

CHAPTER 7 QUICK REFERENCE

- Multiple function equations ARE pattern matching, already in use since Chapter 4; case expresses the same idea in one spot
- Guards (| condition) add a boolean check on top of a pattern — the real ancestor of Rust's match guards and C#'s own when clause
- As-patterns (x@pattern) bind the whole value and its parts simultaneously — no direct equivalent named for Rust or C# on this site
- Exhaustiveness checking predates rust1-6's match and csharp2-7's sealed-type switch by decades — both are downstream of this idea
- Patterns nest arbitrarily deep, avoiding manual unwrapping chains
- -Wincomplete-patterns (or -Werror=incomplete-patterns) catches a missing case at compile time; without it, a reached incomplete match crashes at runtime
- **Next chapter:** typeclasses, a first look — the real ancestor of rust2-2's own traits, arriving decades earlier

CHAPTER 8 OF 8

Typeclasses, A First Look

COURSE 1 · CH 8

Typeclasses, A First Look

1988 — the real historical ancestor of rust2-2's own trait system, closing the loop this course opened

Fundamentals closes with the chapter this whole course has been building toward: the mechanism behind every `==`, every `show`, every `print` call used since Chapter 1 — and the real, documented origin of an idea already met on this site under a different name.

The Problem Typeclasses Solve — Ad-Hoc Polymorphism

`==` needs to behave differently for `Int`, for `String`, for a custom `Shape` — but should still be called the exact same way regardless. This is **ad-hoc polymorphism**: one function name, genuinely different implementations per type, dispatched based on which type is actually involved. A real, important contrast with `haskell1-2`'s own **parametric** polymorphism (`id :: a -> a`), which works identically for every type with zero per-type customization at all.

class and instance — Defining a Typeclass

```
class MyEq a where
    myEq :: a -> a -> Bool

instance MyEq Bool where
    myEq True True = True
    myEq False False = True
    myEq _ _ = False
```

A `class` declares a capability; an `instance` provides the actual per-type implementation. Any type can gain `MyEq` by writing its own `instance` — the capability isn't limited to types the language ships with.

Eq, Ord, and Show — The Standard Trio

```
data Shape = Circle Double | Rectangle Double Double
  deriving (Eq, Show)   -- automatic instances for the straightforward case
```

`Eq` gives `==` / `/=`; `Ord` gives `<` / `>` / `compare`; `Show` gives the conversion to `String` that has been powering every `print` / `show` call used throughout this entire course, now properly explained. `deriving`, already used without comment in `haskell1-6` and `haskell1-7`, is GHC automatically writing the obvious structural instance for straightforward cases.

The Real Ancestor of rust2-2's Traits

Here's the closing reveal: Haskell's typeclasses, introduced in 1988, are the direct historical ancestor of `rust2-2`'s own trait system. Genuinely the same core idea — a capability defined separately from any specific type, implemented per-type, dispatched by which type is actually involved. Rust's own designers have been explicit about this lineage. Stated plainly, closing the loop this entire course opened back in `haskell1-1`: this course exists to show where these ideas really came from, and this is the clearest case of all.

How Dispatch Actually Works — Dictionary Passing, Briefly

A real, honest technical note, kept light: GHC implements typeclass polymorphism by secretly passing a "dictionary" of the relevant functions alongside any call using a typeclass-constrained function. This is genuinely different from `java1-5`'s/`csharp1-5`'s own vtable-based dynamic dispatch, and different again from `cpp2-1`'s/`rust2-3`'s monomorphization — a real, distinct third mechanism, worth naming even without going deep here.

ASPECT	JAVA INTERFACES (JAVA1-6)	RUST TRAITS (RUST2-2)	HASKELL TYPECLASSES
Arrival	Java 1.0 (1996)	Rust 1.0 (2015)	1988 — the real origin
Dispatch mechanism	vtable	monomorphization (usually)	dictionary passing
Can implement for types you don't own	no	yes, with restrictions	yes

USE DERIVING WHENEVER THE DEFAULT STRUCTURAL BEHAVIOR IS GOOD ENOUGH

`deriving (Eq, Ord, Show)` covers the overwhelming majority of real cases — reach for a hand-written `instance` only when a type genuinely needs custom comparison or formatting logic beyond straightforward structural equality.

A TYPECLASS CONSTRAINT IN A SIGNATURE IS REQUIRED, NOT OPTIONAL

`Eq a => a -> a -> Bool` — the `Eq a =>` part is a real, load-bearing part of the type signature. Omitting it while the function body still uses `==` produces a genuine compile error the moment GHC checks the body against the (now too-weak) declared signature — a good, early checkpoint that catches a real mistake immediately.

Coding Challenges

CHALLENGE 1

Define a data type Color = Red | Green | Blue with deriving (Eq, Show), and write an expression demonstrating both == comparing two Colors and show converting one to a String.

View solution

CHALLENGE 2

Define a typeclass Describable with a method describe :: a -> String, write instance declarations for two different types (e.g. Int and Bool), and call describe on a value of each.

View solution

CHALLENGE 3

Write a function that uses == inside its body but omit the Eq a => constraint from its type signature. Show the resulting compile error and explain why adding the constraint fixes it.

View solution

CHAPTER 8 QUICK REFERENCE — COURSE 1 COMPLETE

- Ad-hoc polymorphism (one name, per-type implementations) is genuinely different from haskell1-2's own parametric polymorphism (identical for every type)
- class declares a capability; instance provides the per-type implementation, for any type, including ones you don't own
- Eq/Ord/Show are the standard trio — Show is what's been powering every print/show call since Chapter 1; deriving automates the straightforward case
- Typeclasses (1988) are the real, documented historical ancestor of rust2-2's own trait system, arriving decades earlier
- Dictionary passing is Haskell's own dispatch mechanism — a genuine third alternative to Java/C#'s vtables and C++/Rust's monomorphization
- A typeclass constraint (Eq a =>) is a required, load-bearing part of a signature — omitting it while using == is a real compile error

- **Haskell Fundamentals is now complete.** Course 2 (Intermediate/Advanced) begins with Functors — the first step toward the big monad reveal.