Functional Programming

1. The three commitments

functions are values — you pass them, return them, and build big ones out of small ones — composition becomes the main tool instead of inheritance or mutation

data doesn't change — you make new values instead of updating old ones — once you can't mutate, most FP techniques stop being style choices and become necessities

effects are visible, pushed to the edges — I/O, throwing, the clock — the interesting parts, kept out of the middle of the program; some languages/libraries name this in the type system, plain TS does it by convention (a pure core, an effectful shell)

2. Foundations

expressions vs statements — prefer a form that hands back a value (a ternary, a lookup table) over one that requires mutating an outer variable in each branch to observe the result

pure function — the two conditions — same input always gives the same output, and the call touches nothing outside itself (no I/O, no mutation, no hidden state)

referential transparency — a call can be replaced by its result without changing the program — the reason a pure call's result can be safely cached or hoisted

hoisting a call out of a loop — safe only when the call's inputs and output never change between iterations — no dependency on the loop variable, and no hidden randomness/clock

idempotence vs purity — different axes — idempotent means repeat calls leave the resource in the same end state; purity means the call touches nothing outside itself. Neither implies the other

which HTTP verbs are idempotent by convention — GET, HEAD, PUT, DELETE, OPTIONS are idempotent by convention; POST is not, and PATCH isn't guaranteed either

structural sharing — the reason immutable updates are cheap — the new object reuses the untouched subtrees, so cost is proportional to the path changed, not the data size

recursion in place of loops; accumulators; tail calls — a loop with a mutable counter becomes a recursive function passing the running total as a parameter; a tail call is a recursive call in return position — V8 has no tail-call elimination, so this is a style idea in JS, not a performance guarantee

3. Functions as building blocks

higher-order function — a function that takes or returns a function

closures — a function remembers the variables from the scope it was created in, even after that scope has returned — the mechanism behind partial application, currying, memoization, and private state without a class

currying vs partial application — currying turns an n-ary function into n unary ones; partial application fixes some arguments and returns a function of the rest — `bind` is partial application, not currying

`compose` vs `pipe` — direction — `pipe(f,g)(x)` is `g(f(x))`, left to right; `compose(f,g)(x)` is `f(g(x))`, right to left

argument order is a design decision — data-last (`filter(pred)(list)`) makes a pipeline composable; data-first (`filter(list, pred)`) makes it readable standalone — FP libraries put the data last

point-free, and when it stops paying — `users.map(getName)` over `users.map(u => getName(u))` is a win; a long combinator chain with no named intermediate is not

the combinators worth naming — `identity`, `constant`, `tap` (run an effect, return the input untouched), `once`, `negate`

`reduce` is the general fold — `map` and `filter` are both expressible as a `reduce`; the reverse is not true — but a `reduce` that mutates its accumulator for the whole array should just be a loop instead

4. Types and data modeling

sum type vs product type — a product holds all of its fields at once (a record, a tuple); a sum holds exactly one of several alternatives (a tagged/discriminated union)

where `never` exhaustiveness belongs — a closed set of cases argues for an exhaustive union; an open set (e.g. plugin kinds arriving at runtime) argues for an open/registry shape instead

the expression problem — a union + a match makes adding an OPERATION cheap and a CASE expensive; a class hierarchy with virtual methods is the exact mirror — cheap case, expensive operation

modelling state as a union, not booleans — make illegal states unrepresentable — a tagged union of loading/ok/error rules out the boolean-flags version's impossible combinations (e.g. `loading:true` with both `data` and `error` set)

optional field vs union member — three optional fields on one type = eight possible states, most of them nonsense — a union of only the valid shapes allows just the real ones

generic functions and what a signature alone tells you — parametricity ("theorems for free") — a function typed `<T>(x: T) => T` can only be identity (or loop/throw), because it has to work for every possible `T`

how type inference works, at a high level — the compiler assigns each expression the most general type consistent with how it's used; this is far easier to do well over expression-oriented code than over code full of mutation and control-flow statements

5. Values that hold other values

`Result<T,E>` vs `throw` — expected failures are values (`Result`), bugs are throws — a return type makes failure visible in the signature and forces the caller to handle it

`Option`/`Maybe` vs `T | undefined` — TS's union plus strict null checks already does most of what `Option` does, without a wrapper — an `Option` class is usually not worth it in TS

the conversion boundary — pick one layer (usually the transport/handler edge) and convert once between throw-style and `Result`-style — mixing both styles everywhere is the real failure mode

functor = something with a lawful `map` — `Array`, `Promise`, `Result`, and `Option` all have one — a container you can `map` over without unwrapping it first

applicative: combining several independent values — combines several independent wrapped values with a plain multi-argument function (e.g. two validated fields into one `Result<Person,E>`, collecting every failure) — unlike monad, the values don't depend on each other

monad = `of` plus `flatMap` — `flatMap` over `map` when the callback itself returns the container, so nesting doesn't accumulate — unlike applicative, each step can depend on the previous one's result

why `Promise` is not quite a monad — `.then` auto-flattens whether or not you return a promise, so it's both `map` and `flatMap` at once — and it makes `Promise<Promise<T>>` unrepresentable

`flat`/`flatMap` as the practical half — the one place the functor/monad vocabulary pays off in ordinary code — depth argument defaults to 1

railway-oriented / short-circuit chains — a chain of steps where the first failure skips the rest — what `Promise` rejection and monadic `flatMap` already give for free

the laws, and why the laws are the point — functor: mapping identity changes nothing, two maps compose into one. Monad: left/right identity, associativity. The laws are what let you refactor a chain — split it, reorder, extract a helper — without re-testing the whole pipeline

`traverse` and `sequence` — `sequence` flips `Result<T,E>[]` into `Result<T[],E>` — success only if every item succeeded; `traverse` is `map` then `sequence` in one pass

6. Effects and the real world

what counts as a side effect — anything besides computing the return value from the inputs — I/O, the console, mutating something outside its own scope, throwing, reading the clock or a random source

functional core, imperative shell — pure decisions in the middle, I/O at the boundary — keep the testable logic separate from where it touches the outside world

describing work as a value; eager vs lazy — a plain function call runs immediately; wrapping the same work as a value (a thunk, an `IO`/`Task`) lets you build it up, pass it around, and decide when — or whether — to run it

async as just another effect; `Promise` vs `Task` — a `Promise` starts running the moment it's created and caches its one result; a `Task` (`() => Promise<T>`) is a lazy description of the same work — nothing runs until invoked, so it composes, retries, and cancels like any other value

error handling without exceptions — an effect that can fail is exactly the case `Result` is for

the Reader idea — a function shaped `(config) => result` composes with the others and the config gets supplied once, at the edge — instead of threading it through every function by hand

the State idea — each step is a pure function `(state) => [result, newState]`, and the next step receives the state the previous one produced — the state changes, nothing is mutated

7. Structure and laws

Semigroup and Monoid — a Semigroup is a type plus an associative combine operation (order of grouping doesn't matter); a Monoid is a Semigroup with an identity element too (`0` for `+`, `""` for concat, `[]` for arrays) — `reduce` needs exactly a Monoid

equality and ordering as values you pass around — instead of a type's built-in `===` or `<`, equality/ordering become ordinary values — a comparator function, an `{equals}`/`{compare}` object — passed as a parameter

type classes / traits vs interfaces — an interface is declared by the type itself upfront; a type class (Haskell) / trait (Rust) attaches behavior to a type from the outside, after the fact, without owning its source — ad-hoc polymorphism instead of inheritance

property-based testing — assert that a LAW holds for hundreds of randomly generated inputs (associativity, identity, functor/monad laws) instead of asserting specific examples — `fast-check` in TS, `QuickCheck` in Haskell

the law count for each structure — the same two ideas (identity, associativity) all the way down, monad just splits identity in two. Category: 2 (identity, associativity). Monoid: 2 (identity, associativity). Functor: 2 (identity, composition). Monad: 3 (two identity laws, one associativity)

8. In the large

data-first design and module boundaries — model the data first, write functions over it second — a module's boundary should export a data shape plus plain functions over it, not a class hiding mutable state

eager chain vs lazy pipeline — `arr.filter(...).map(...).slice(0,3)` builds two full intermediate arrays to return three items; a generator pipeline (or iterator helpers) computes three

generator as a lazy sequence — an infinite sequence is only expressible lazily

a thunk — `() => expensive()` — a value wrapped in a function so it isn't computed until needed

short-circuit as laziness — `&&`, `||`, `??` don't evaluate the right side unless they must

state over time as a fold over events — current state = `events.reduce(reducer, initialState)` — a pure function over the full history; exactly what a Redux-style reducer is, and the same idea as event sourcing

why immutability makes concurrency easier — a value that can never change can be read by multiple threads/workers at once with no lock and no race condition — shared MUTABLE state is what makes concurrency hard

the real costs — every immutable update allocates (structural sharing keeps this cheap, not zero); deep recursion without a trampoline risks a stack overflow; unfamiliar style slows a team down before it speeds them up

living alongside imperative code — most real codebases are mixed — a pure functional core surrounded by imperative frameworks; the skill is knowing where to draw the boundary, not rewriting everything functional

10. Optional deep work

lenses / optics — composable immutable get/set at depth

effect systems — the "effects as values" idea taken all the way — a type like `Effect<R,E,A>` tracks dependencies, possible errors, and result all in the type; Task + Reader + Result fused into one

free monads — separate DESCRIBING a sequence of effectful steps (build a data structure representing the program) from INTERPRETING them (run one or more interpreters over it later)

dependent types — a type that depends on a VALUE, not just another type — e.g. a vector type parameterized by its own length; TS doesn't have these

category theory — the branch of math functor/monad/applicative borrow their vocabulary and laws from — objects and structure-preserving maps (morphisms), composed associatively with an identity

transducers — composing `map`/`filter` steps into one pass with no intermediate arrays

trampolining, and why — V8 has no tail-call elimination, so deep recursion throws `RangeError` — a trampoline (or a loop) is the fix

immutable-data libraries — Immer's draft-mutate-produce shape, and where a persistent-collection library earns its weight

11. Vocabulary

total function vs partial function — a total function is defined for every input in its domain; a partial one isn't — array `head` on an empty array, integer division by zero

referential opacity — the opposite of referential transparency — a call that can't be replaced by its result without changing the program (reading the clock, a counter)

arity — the number of arguments a function takes — unary, binary, n-ary; a variadic function takes any number

bifunctor — a functor with two independent type parameters you can map over separately — e.g. `Result<T,E>` having both `map` (over `T`) and `mapErr` (over `E`)

desugaring — rewriting convenient syntax (a `for` loop, `async`/`await`) into the more primitive form it's shorthand for (a fold, a chain of `.then`s)

composition (the general word) — combining two things into one bigger thing of the same kind, without either needing to know about the other — `compose`/`pipe` is the function-level case, Semigroup/Monoid is the value-level case

record of functions — bundles related pure functions into a plain record instead of an OOP object holding hidden state — the standard FP shape for dependency injection, decoupling logic from the infrastructure it calls

11. Vocab: Intro

functional programming — programming built around pure functions, immutable data, and composition, instead of step-by-step commands that change state

imperative programming — programming as a sequence of commands that change state one step at a time

object-oriented programming — organizing code around objects that bundle state and behavior together

mutable state — data that can be changed in place after it's created, rather than replaced with a new value

assignment-based programming — programming where progress is tracked by reassigning variables as you go — the imperative default FP moves away from

side effects — anything a function does besides computing its return value from its inputs — I/O, mutating something outside itself, throwing, reading the clock or randomness

DRY — avoid duplicating the same logic in more than one place

YAGNI — don't build for a requirement you don't actually have yet

loose coupling, high cohesion — modules should depend on each other as little as possible, while what's inside one module should belong together

principle of least surprise — a design should behave the way someone would reasonably expect it to, given its name and shape

single responsibility — a unit of code should have one reason to change

associativity — grouping doesn't matter: `(a+b)+c` equals `a+(b+c)`

commutativity — order doesn't matter: `a+b` equals `b+a`

identity (property) — a value that combines with anything and leaves it unchanged — `0` for `+`, `1` for `×`

distributivity — one operation spreads over another: `a×(b+c)` equals `a×b + a×c`

composable parts — small pieces built so they combine into bigger pieces without needing to know about each other's internals

set theory — the math of collections and membership; underlies "a type is the set of values that satisfy it"

reasoning about code — predicting what code does by reading it alone, without running it or tracking hidden state — the actual payoff FP is chasing

11. Vocab: First Class Functions

first class functions — functions can be stored in variables, passed as arguments, and returned from other functions, just like any other value

callable — anything invokable with `()` — a plain function, a bound function, an object with a call signature

callback — a function passed into another function to be invoked later, usually when some event or step completes

delayed evaluation — wrapping work in a function so it doesn't run until that function is actually called

indirection — adding a layer between a caller and the real work so the caller doesn't need to know the concrete implementation

wrapper function — a function that calls another function, adding behavior around it (logging, timing, validation) without changing the original

leaky abstraction — an abstraction meant to hide details that forces the caller to know about them anyway

this binding — what `this` refers to depends on how a function is called, not where it's defined — except arrow functions, which capture it lexically

generic vs. specific code — generic code works across many types/shapes at the cost of doing less per-case; specific code does more but only for one shape

reusability — writing something once so it applies in more than one place without being copied

naming / misnomers — a name that no longer matches what the thing actually does, misleading anyone reading it

micro-optimization — tuning a small, rarely-hot piece of code for performance, usually not worth the readability cost

11. Vocab: Pure Functions

pure function — same output for the same input, and no observable effect outside itself

impure function — depends on or affects something outside its own inputs/outputs — state, I/O, randomness, the clock

mutation — changing a value in place rather than creating a new one

immutability (Object.freeze) — preventing a value from being changed after creation; `Object.freeze` is JS's shallow, runtime version of it

system state — the sum of all mutable data a program currently holds, spread across variables, objects, and external stores

cognitive load — how much a reader has to hold in their head to understand code; pure functions lower it because nothing hidden can affect the result

system complexity — how hard a system is to reason about as a whole, driven up by hidden state and interactions between parts

mathematical function — a pure function models exactly this: one input always maps to the same one output, nothing else involved

caching — storing a computed result so a repeated call can reuse it instead of recomputing

memoization — caching keyed by a function's arguments; only safe when the function is pure

portability — pure code doesn't depend on its ambient environment, so it can be moved or reused elsewhere unchanged

self-documenting code — code whose behavior is clear from its own names and shape, without needing external comments

explicit dependencies — everything a function needs arrives through its parameters, not through hidden globals or ambient state

dependency injection — supplying a function/object what it needs from outside, rather than it reaching out to construct or fetch it itself

parameterization — turning something that was hardcoded or reached-for into a parameter instead

testability — how easy code is to test in isolation; pure functions are trivially testable since there's no hidden state to set up or tear down

property-based testing — testing that a general law holds for many randomly generated inputs, rather than checking specific examples — `fast-check` in TS, `jqwik` in Java

referential transparency — a call can be replaced by its result without changing the program's behavior

equational reasoning — treating code like algebra, substituting equals for equals — safe only because referential transparency guarantees it

inlining / substitution — replacing a call site with the function's body — only valid to do freely under referential transparency

parallel code — code split to run at the same time across cores, threads, or workers

shared memory — memory more than one thread/worker can read and write — the source of most concurrency bugs

race condition — a bug where the outcome depends on timing between concurrent operations touching shared mutable state

11. Vocab: Currying

currying — turning an n-ary function into a chain of n unary functions, each taking one argument and returning the next

closure — a function that remembers the variables from the scope it was created in, even after that scope has returned

partial application — fixing some of a function's arguments now and getting back a function of the rest

higher order function — a function that takes another function as an argument, returns one, or both

argument ordering (data last) — putting the data being operated on as the last parameter, so earlier arguments can be fixed first to build a reusable step

unary function — a function that takes exactly one argument

11. Vocab: Coding by Composing

function composition (compose) — combining two functions into one, feeding one's output into the other's input; `compose(f,g)(x)` is `f(g(x))`

variadic function — a function that accepts any number of arguments

right-to-left data flow — `compose`'s reading order: the rightmost function runs first, matching mathematical `f(g(x))` order

associativity of composition — grouping doesn't matter when composing more than two functions: `compose(f, compose(g,h))` equals `compose(compose(f,g), h)`

extract function (refactoring) — pulling a piece of logic out into its own named function, usually to make it reusable or composable

pointfree style — defining a function without naming its arguments, built purely by composing other functions

identity function (id) — `x => x` — the do-nothing function that acts as composition's identity element

trace / debugging composition — inserting a `tap`-like logging step into a composed pipeline to see an intermediate value without changing the result

types as sets — treating a type as the set of all values that satisfy it, so type relationships mirror set relationships (subset, union, etc.)

11. Vocab: Declarative Coding

declarative coding — describing WHAT the result should be, leaving HOW to compute it to the underlying implementation

imperative coding — describing HOW to get the result, step by step, in the order things should happen

expression vs. step-by-step instruction — an expression evaluates to a value; a statement just performs an action and moves on

specification (what, not how) — declarative code as a spec of the desired outcome, decoupled from the mechanism that produces it

order of evaluation — the sequence in which a language actually evaluates subexpressions; imperative code depends on this being predictable

parallel / concurrent computing — parallel runs work at the same literal time across cores; concurrent structures independent tasks that may or may not run simultaneously

JIT optimization — a just-in-time compiler watches a program as it runs and compiles hot paths to faster machine code on the fly

isolating impure actions (namespacing) — keeping impure code (I/O, mutation) grouped and named separately from pure logic, so the boundary is visible

universal getter (prop) — a generic `prop(key)(obj)` function that reads any field off any object, instead of writing a getter per field

map's composition law — mapping `f` then `g` gives the same result as mapping the single composed function `g.f` — justifies fusing multiple `.map()`s into one

principled refactor — a refactor justified by an algebraic law holding, not just "it looks the same"

11. Vocab: Type Signatures

type signature — a function's declared shape: the types of its parameters and its return type

type inference — the compiler working out a value's type from how it's used, without an explicit annotation

type annotation — writing a type explicitly instead of relying on inference

compile-time type checking — catching type errors before the program runs, by checking code against its declared/inferred types

dynamic language — a language that checks types at runtime instead of compile time — JS without TS

type variable — a placeholder standing in for "any type," used in generics — the `T` in `<T>(x: T) => T`

polymorphic type — a type that works over more than one concrete type via a type variable

parametricity — a function typed generically enough can't inspect or special-case its type parameter — "theorems for free"

free theorem — a property you get for free purely from a generic signature, without looking at the implementation

rewrite rule — a compiler optimization that replaces one expression with an equivalent one it knows is safe, e.g. fusing two `map`s into one

type constraint — restricting a type variable to only types that satisfy some capability, e.g. `<T extends Comparable>`

domain restriction — narrowing which inputs a function accepts, often to make it total instead of partial

11. Vocab: Tupperware

container — a value that wraps another value, giving you a context to operate in — `Array`, `Maybe`, `Result`, `Promise`

functor — a container with a lawful `map`

map — apply a function to the value(s) inside a container without unwrapping it first

of (constructor / lifting) — putting a plain value into a container, minimally wrapped — `Array.of`, `Promise.resolve`

Maybe — a container representing a value that might be absent, without using `null`/`undefined` directly

Just / Nothing — Haskell's two cases of `Maybe`: `Just x` holds a value, `Nothing` holds none

Some / None — Rust/Scala's names for the same two cases as Just/Nothing

Option / Optional — Scala/Java's name for the Maybe type — Java's `java.util.Optional`

null check / type safety — TS's union `T | undefined` plus strict-null-checks does most of what Maybe/Option gives, checked at compile time

short-circuiting — a chain of operations on Nothing/None skips the rest and returns Nothing/None immediately

Either — a container with two possible cases, conventionally used for success-or-failure

Left / Right — Either's two cases; by convention Left holds an error/failure, Right holds a success value

pure error handling — using a container type like Either/Result instead of throwing, so failure is visible in the return type

throw / catch — the exception mechanism Either/Result is offered as an alternative to

lifting — taking a plain value or function and wrapping it to work inside a container

sum type — a type holding exactly one of several alternatives

deferred effect — an effect represented as a value/description, not run until something explicitly executes it

command pattern — OOP's version of the same idea: wrapping a request/action as an object so it can be queued, logged, or undone

queue — a sequence of pending items processed in order, often used to hold deferred effects/commands until they're run

Task / Future — a lazy description of an asynchronous computation (fp-ts's `Task`); Java's `Future`/`CompletableFuture` is the eager cousin

fork — starting a `Task`/`Future`'s execution — the point where the lazy description actually begins running

asynchronous actions — work that completes at some later time rather than immediately

promises — JS's built-in eager, cached async value

callbacks — a function passed in to be called when async work completes — the mechanism `Promise`/`Task` replace

functor laws: identity, composition — mapping `identity` changes nothing; mapping `f` then `g` equals mapping the composed function `g.f`

nested / stacked functors — one functor inside another, e.g. `Promise<Result<T,E>>` — handling requires mapping "through" both layers

event streams — a sequence of values arriving over time, instead of all at once — the async analogue of an array

observables — a library type (RxJS) representing an event stream you can `map`/`filter`/combine like a lazy, push-based collection

11. Vocab: Monadic Onions

monad — a pointed functor with `chain`/`flatMap` — `of` plus flattening

join (flatten) — collapsing one level of nesting: `M<M<T>>` to `M<T>`

chain / bind / flatMap — map then join in one step: run a function that itself returns a wrapped value, without ending up double-wrapped

nested monads — the `M<M<T>>` shape that `join`/`chain` exists to collapse

sequencing of effects — running one effectful step after another, where each may depend on the previous one's result

algebraic data types — types built by combining sums and products — TS discriminated unions are exactly this

monad transformers — a way to stack two monads together (e.g. handle both "might fail" and "is async" at once) without hand-rolling the combined type each time

callback hell / pyramid of doom — deeply nested callbacks from sequencing async steps without composition — the practical problem monadic chaining and `async`/`await` solve

11. Vocab: Applicative Functors

applicative functor — a functor that also has `ap`, letting you combine several wrapped values with a plain multi-argument function

ap — apply a wrapped function to a wrapped value: `Container<a→b>` applied to `Container<a>` gives `Container<b>`

concurrent vs. sequential evaluation — applicative combines independent values so order doesn't matter and they could run concurrently; monad's `chain` is inherently sequential

principle of least power — use the weakest abstraction that gets the job done — functor if you only need `map`, applicative if values are independent, monad only if a step depends on a prior result

11. Vocab: Natural Transformations

principled type conversion — converting between container types (e.g. `Maybe` to `Either`) in a lawful, uniform way rather than ad hoc

isomorphism — a pair of conversions between two types that are exact inverses of each other — converting there and back always returns the original value

fusion / optimization by law — merging multiple operations (e.g. two `map`s) into one pass, justified by a law guaranteeing the result is identical

11. Vocab: Traversing the Stone

sequence — flip a list of wrapped values into one wrapped list — e.g. a list of `Result`s to a `Result` of a list

traverse — `map` then `sequence` in one pass: run an effectful function over every element and combine the results into one wrapped structure

type constructor — a type that takes a type parameter to produce a concrete type — `Array` isn't a type by itself, `Array<T>` is

effect ordering — whether traversing runs effects one at a time (sequential) or all at once (concurrent) depends on the effect type

predicate — a function returning `boolean`, used to test each element — the shape `filter`/`find` take

partitioning vs. validating — splitting a list into successes and failures keeps both; validating fails the whole batch on the first failure (or collects all)

reduce / accumulator / seed value — `reduce`'s three parts: the combining function, the running total, and its starting value

laws as code guarantees — laws are what let you refactor a pipeline without re-testing it by hand, because the guarantee is structural, not empirical

encapsulation — hiding a value's internal representation behind an interface, so callers interact with it only through defined operations

11. Vocab: Monoids

monoid — a type plus an associative combine operation plus an identity element

semigroup — a type plus an associative combine operation, no identity required

concat (associative binary operation) — the conventional method name for a semigroup/monoid's combine operation

empty (identity element) — the conventional method name for a monoid's identity value — combines with anything and changes nothing

binary operation — an operation taking exactly two inputs of the same type and producing one output of that type

closed under an operation — combining two values of a type always produces another value of the same type, never escaping it

domain / codomain — a function's domain is the set of valid inputs; its codomain is the set the outputs are drawn from

program to an interface, not an implementation — depend on what an operation can do (e.g. "this has a `concat`"), not on a specific concrete type

Sum, Product, Min, Max — common numeric monoids: combine is `+`/`×`/`Math.min`/`Math.max`, identity is `0`/`1`/`+Infinity`/`-Infinity`

Any, All — boolean monoids: combine is `||`/`&&`, identity is `false`/`true`

First — a monoid that keeps the first non-empty value it sees and discards the rest

fold — collapsing a structure down to a single value using a combining operation — the general case `reduce` implements for arrays

reduce with initial value — supplying the seed explicitly instead of assuming the first element is it — required whenever the array might be empty