JavaScript and TypeScript from scratch

You write modern JavaScript and type it with TypeScript so the compiler finds your mistakes before deployment.

Mike Bild

Exercises: github.com/MikeBild/introduction-javascript

values, functions, objects, prototypes

When are two values equal in JavaScript? · What does a closure remember? · Create new values instead of changing old ones · Behind every class is a prototype

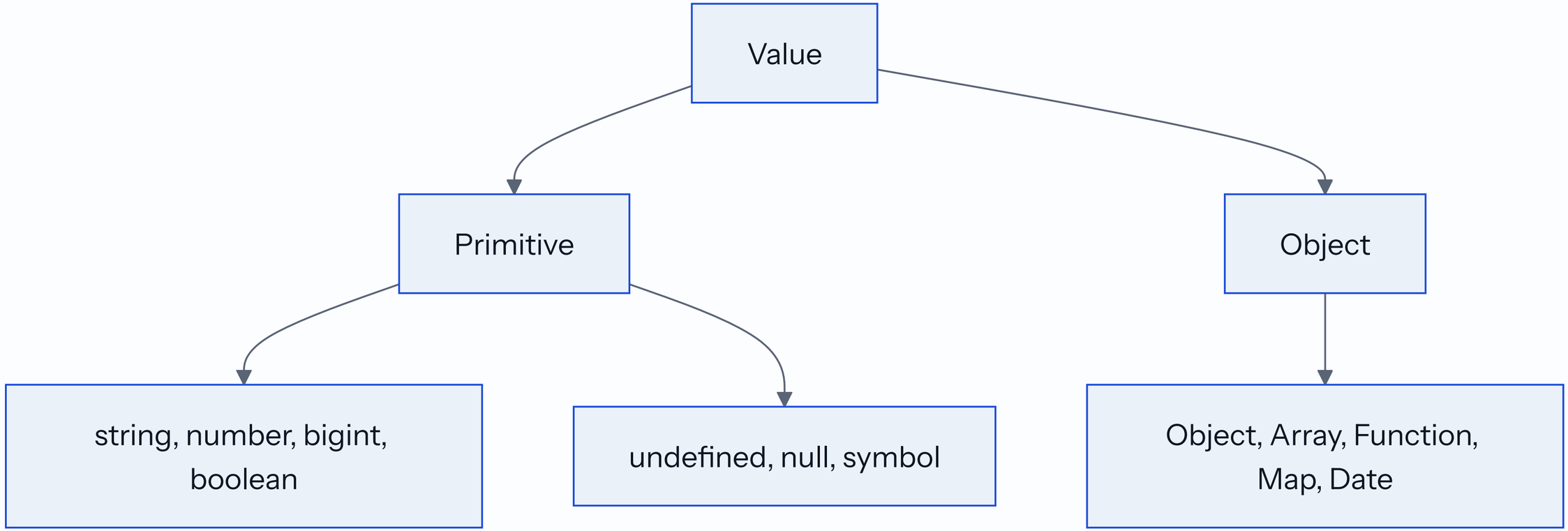
Why does JavaScript sometimes surprise you?

- Variables do not have a fixed type, values do
- Operators silently perform type coercion
- If you know the rules, you will not be surprised

```
'5' + 1       // '51'  - Plus concatenates strings
'5' - 1       // 4     - Minus calculates with numbers
true + 1      // 2
0.1 + 0.2     // 0.30000000000000004
```

Primitives by value, objects by reference

- Primitives are immutable and are compared by value
- Objects, arrays, and functions are compared by reference
- typeof helps, but it also reports object for null



Strict equality instead of loose equality

- === compares type and value, without type coercion
- == coerces first, with coercion rules that are hard to remember
- Object.is handles NaN and -0 correctly

```
[1] == 1              // true
1 === '1'             // false
1 == '1'              // true
null == undefined     // true
NaN === NaN           // false
Object.is(NaN, NaN)   // true
```

How do you handle missing values?

- undefined means never set; null means deliberately empty
- || replaces every falsy value, including 0 and empty strings
- ?? replaces only null and undefined

```
const retries = 0
retries || 3           // 3  - the 0 is lost
retries ?? 3           // 0  - the 0 remains

const user = { address: null }
user.address?.city     // undefined instead of TypeError

Number('abc')          // NaN
Number.isNaN(Number('abc')) // true
```

Your turn: A discount of zero stays zero

- Find the value that || swallows
- Replace || with ??
- Check both orders in the example

Functions are values

- Functions can be stored, passed, and returned
- Arrow functions are short and have no this of their own
- Default and rest parameters replace arguments

```
const greet = (name, greeting = 'Hallo') => `${greeting} ${name}`

const applyToAll = (names, format) => names.map(format)

applyToAll(['Ada', 'Grace'], greet)
// ['Hallo Ada', 'Hallo Grace']
```

Closures remember their scope

- The inner function sees the variables of the outer one
- The variables live on as long as the function lives
- Private state emerges without classes

```
const createCounter = () => {
  let count = 0
  return {
    increment: () => ++count,
    current: () => count,
  }
}

const counter = createCounter()
counter.increment()
counter.current() // 1
```

Higher-Order Functions build functions

- Partial Application sets parameters in advance
- Currying creates one function per parameter
- Small functions are easy to reuse

```
const multiply = (factor) => (value) => value * factor
const double = multiply(2)
const triple = multiply(3)

;[1, 2, 3].map(double) // [2, 4, 6]

const hasRole = (role) => (user) => user.roles.includes(role)
const isAdmin = hasRole('admin')
users.filter(isAdmin)
```

Where loops and this surprise you

- var has function scope, all callbacks share one variable
- let has block scope and a new binding per iteration
- function sets this on call, arrow functions take it from outside

```
const callbacks = []
for (var index = 0; index < 3; index++) {
  callbacks.push(() => index)
}
callbacks.map((callback) => callback())

const user = {
  name: 'Ada',
  greetLater() {
    setTimeout(() => console.log(this.name), 0) // 'Ada'
  },
}
```

Your turn: A closure remembers the greeting

- makeGreeting takes a greeting
- The returned function expects a name
- The greeting lives on in the closure

Shared references change quietly too

- An assignment copies only the reference
- A change takes effect everywhere
- Bugs appear far away from the trigger

```
const original = { name: 'Ada', skills: ['math'] }
const copy = original

copy.name = 'Grace'
console.log(original.name) // 'Grace'
```

Spread builds anew, Destructuring reads out

- Spread builds new objects with changed properties
- Spread builds new arrays with added elements
- Destructuring reads out properties, rest collects the rest

```
const user = { name: 'Ada', role: 'dev', team: 'core' }

const promoted = { ...user, role: 'lead' }
const { team, ...withoutTeam } = promoted

const tags = ['ts', 'js']
const moreTags = [...tags, 'node']
const [firstTag, ...otherTags] = moreTags
```

Spread copies only the top level

- Nested arrays and objects stay shared
- You deliberately copy nested parts too
- structuredClone returns a real deep copy

```
const team = { name: 'Core', members: ['Ada'] }

const shallow = { ...team }
shallow.members.push('Grace') // also changes team.members

const updated = { ...team, members: [...team.members, 'Linus'] }

const deep = structuredClone(team)
```

toSorted and with leave the original as is

- sort, reverse, and splice change the original
- toSorted, toReversed, toSpliced, and with return copies
- Object.freeze makes mutations visible

```
const scores = Object.freeze([3, 1, 2])

const sorted = scores.toSorted((a, b) => a - b) // [1, 2, 3]
const replaced = scores.with(0, 10)              // [10, 1, 2]

scores.push(4) // TypeError: the array is frozen
```

Your turn: Change an order without mutation

- Rebuild the order with Spread
- Create items anew with map
- Check that the original is unchanged afterward

Every object points to a prototype

- Lookup follows the prototype link until null
- Object.create sets the prototype directly
- Object.hasOwn separates own from inherited properties

```
const animal = {
  describe() {
    return `${this.name} macht ${this.sound}`
  },
}

const dog = Object.create(animal)
dog.name = 'Rex'
dog.sound = 'Wuff'

dog.describe()                          // 'Rex macht Wuff'
Object.getPrototypeOf(dog) === animal // true
Object.hasOwn(dog, 'describe')          // false
Object.hasOwn(dog, 'name')              // true
```

class is syntax over prototypes

- Methods from the class body live on the prototype
- extends connects the chain, super calls the parent
- new is required, and the body runs in strict mode

```
class Animal {
  constructor(name) {
    this.name = name
  }
  speak() {
    return `${this.name} macht ein Geräusch`
  }
}

class Dog extends Animal {
  speak() {
    return `${this.name} macht Wuff`
  }
}

const rex = new Dog('Rex')
Object.getPrototypeOf(rex) === Dog.prototype              // true
Object.getPrototypeOf(Dog.prototype) === Animal.prototype // true
```

A method used as a callback loses this

- this is created at call time, not at definition time
- Without a receiver, this is undefined in strict mode
- An arrow function or bind keeps the receiver fixed

```
class Counter {
  count = 0
  add(step) {
    this.count += step
  }
}

const counter = new Counter()
;[1, 2].forEach(counter.add) // ???

;[1, 2].forEach((step) => counter.add(step))
;[1, 2].forEach(counter.add.bind(counter))
```

Composition beats mixins

- Mixins copy methods onto another prototype
- Identical method names overwrite each other silently
- Small functions each provide one behavior

```
// Mixin: changes the prototype for all instances
const Serializable = { serialize() { return JSON.stringify(this) } }
Object.assign(Dog.prototype, Serializable)

// Composition: small functions, composed
const withLogger = (log) => ({ info: (text) => log(`info: ${text}`) })
const withStore = () => {
  const items = new Map()
  return {
    save: (id, value) => items.set(id, value),
    find: (id) => items.get(id),
  }
}

const service = { ...withLogger(console.log), ...withStore() }
```

Prototype pollution affects every object

- **proto** as a key reaches Object.prototype
- A naive deep merge writes to the shared prototype
- A key filter and Map close the gap

```
const merge = (target, source) => {
  for (const [key, value] of Object.entries(source)) {
    if (value && typeof value === 'object') {
      target[key] ??= {}
      merge(target[key], value)
    } else target[key] = value
  }
  return target
}

merge({}, JSON.parse('{"__proto__": {"isAdmin": true}}'))
;({}).isAdmin // true - every object is now admin
```

Your turn: A merge without prototype pollution

- Filter dangerous keys before writing
- Read only own properties with Object.hasOwn
- Check the attack from the example again

arrays, modules, async, errors

map, filter, reduce instead of loops · ES modules instead of require and global scope · Wait with async/await without blocking · When do you throw, and when return a result?

Loops mix the what and the how

- Index, temporary array, and condition are in one block
- You have to search for the actual intent
- Every line can contain an off-by-one error

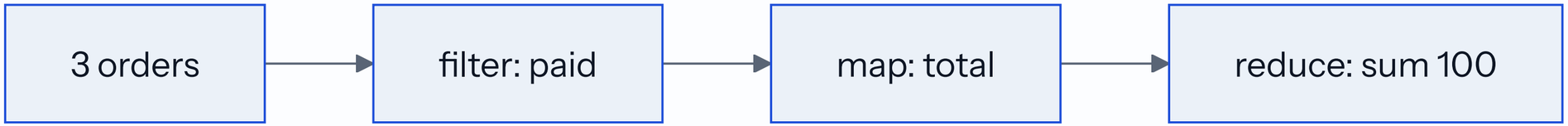
```
const paidTotals = []
for (let index = 0; index < orders.length; index++) {
  if (orders[index].paid) {
    paidTotals.push(orders[index].total)
  }
}
```

filter selects, map transforms, reduce combines

- Each step gets a small callback
- Each step returns a new array
- The original stays unchanged

```
const orders = [
  { id: 1, total: 40, paid: true },
  { id: 2, total: 15, paid: false },
  { id: 3, total: 60, paid: true },
]

const paidRevenue = orders
  .filter((order) => order.paid)
  .map((order) => order.total)
  .reduce((sum, total) => sum + total, 0) // 100
```



find, some, and groupBy replace search loops

- find and some for search and checks
- flatMap for nested lists
- Object.groupBy for grouping by key

```
const hasOpenOrder = orders.some((order) => !order.paid) // true
const firstLarge = orders.find((order) => order.total > 50) // id 3
const allPositive = orders.every((order) => order.total > 0) // true

const allTags = posts.flatMap((post) => post.tags)

const byStatus = Object.groupBy(orders, (order) =>
  order.paid ? 'paid' : 'open',
)
```

Where array methods surprise you

- reduce always needs an initial value
- sort without a compare function sorts as strings
- map passes value, index, and array

```
const empty = []
empty.reduce((sum, value) => sum + value)    // TypeError
empty.reduce((sum, value) => sum + value, 0) // 0

const numbers = [10, 9, 1]
numbers.toSorted()                 // [1, 10, 9]
numbers.toSorted((a, b) => a - b) // [1, 9, 10]

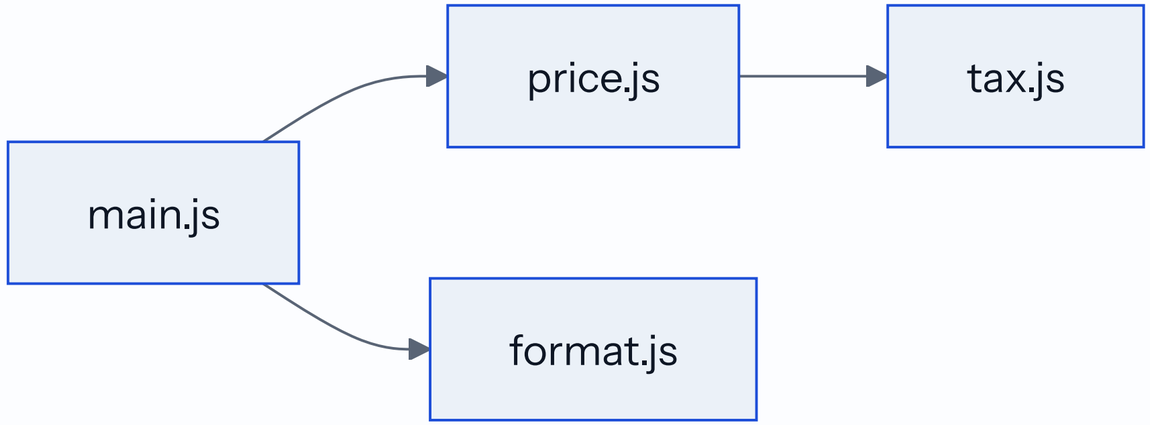
const texts = ['1', '2', '3']
texts.map(parseInt) // ?
texts.map(Number)   // [1, 2, 3]
```

Your turn: The total of open orders

- Select open orders with filter
- Add amounts with reduce and an initial value
- Give callbacks clear names

Why modules?

- Each file has its own scope
- Dependencies are explicit at the top
- Bundlers detect unused code with tree shaking



Named exports make dependencies visible

- Named exports are the standard
- Imports pull names explicitly, and as renames them
- Imports are live bindings and read-only

```
// tax.js
export const vatRate = 0.19
export const addVat = (netCents) => Math.round(netCents * (1 + vatRate))

// main.js
import { addVat, vatRate as germanVat } from './tax.js'

console.log(addVat(10_000)) // 11900
console.log(germanVat)      // 0.19
```

Node needs type module and file extensions

- type module in package.json
- Relative imports always include the file extension
- Core modules use the node: prefix

```
// package.json contains: "type": "module"
import { readFile } from 'node:fs/promises'
import { join } from 'node:path'
import { addVat } from './tax.js'

const configPath = join(import.meta.dirname, 'config.json')
const config = JSON.parse(await readFile(configPath, 'utf8'))
```

What trips you up when moving to ES modules

- Your own code keeps using import instead of require
- Circular imports point to unclear boundaries
- Dynamic import loads only when needed

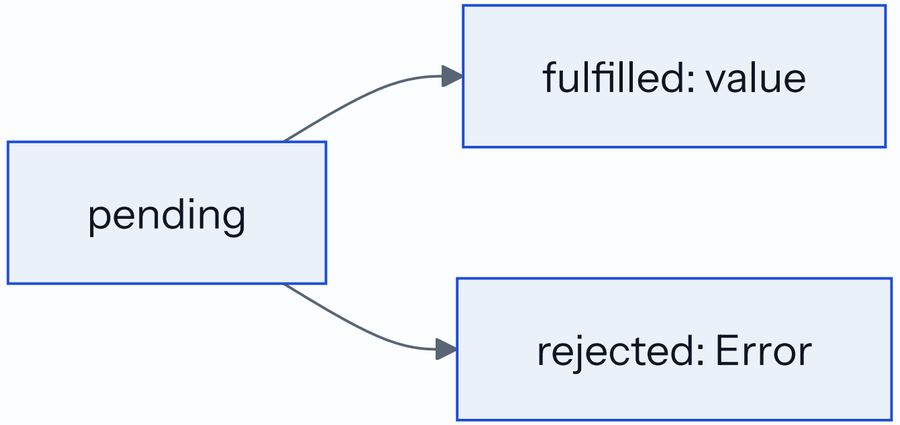
```
// Only load when it is needed
const exportReport = async (rows) => {
  const { toCsv } = await import('./csv.js')
  return toCsv(rows)
}
```

Your turn: One module, one task

- Move tax logic to tax.js
- Use named exports instead of a default export
- Import relatively with a file extension

Wait without blocking

- JavaScript runs in one thread
- I/O like network and files takes time
- A Promise represents a result that arrives later



then links steps into a chain

- then receives the value and returns a new Promise
- catch catches errors from the whole chain
- Callbacks run as microtasks after the current code

```
const wait = (milliseconds) =>
  new Promise((resolve) => setTimeout(resolve, milliseconds))

setTimeout(() => console.log('A'))
Promise.resolve().then(() => console.log('B'))
console.log('C')

wait(500)
  .then(() => fetch('https://example.org'))
  .then((response) => response.status)
  .catch((error) => console.error(error))
```

async/await reads like synchronous code

- async functions always return a Promise
- await unwraps the value, try/catch catches errors
- The thread stays free while waiting

```
import { readFile } from 'node:fs/promises'

const loadConfig = async (path) => {
  try {
    const text = await readFile(path, 'utf8')
    return JSON.parse(text)
  } catch (error) {
    throw new Error(`Konfiguration ${path} ist nicht lesbar`, { cause: error })
  }
}

const config = await loadConfig('./config.json')
```

Parallel instead of sequential

- Two awaits in a row run sequentially
- Promise.all starts everything and waits for all
- Promise.allSettled collects successes and errors

```
// sequentially: durations add up
const user = await loadUser(userId)
const orders = await loadOrders(userId)

// parallel: duration of the slowest task
const [sameUser, sameOrders] = await Promise.all([
  loadUser(userId),
  loadOrders(userId),
])

const results = await Promise.allSettled(urls.map((url) => fetch(url)))
const failed = results.filter((result) => result.status === 'rejected')
```

Where await is missing or does not wait

- A forgotten await returns a Promise instead of a value
- forEach does not wait for async callbacks
- Unhandled rejections terminate Node processes

```
// wrong: forEach ignores the Promises
files.forEach(async (file) => await upload(file))

// correct: sequentially
for (const file of files) await upload(file)

// correct: in parallel
await Promise.all(files.map((file) => upload(file)))
```

Your turn: load order and customer in parallel

- Start both calls immediately
- Wait for both with Promise.all
- Extract results with destructuring

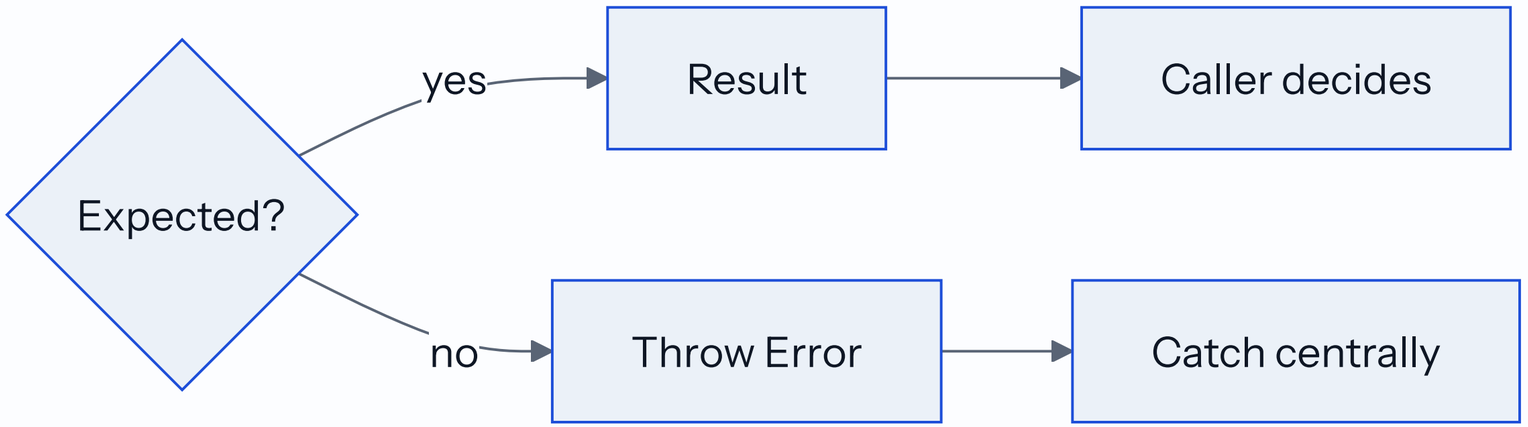
Errors are invisible

- The call does not reveal whether a function throws
- Throw allows anything in JavaScript, even strings
- The caller can easily forget to handle it

```
// Does this throw? What exactly? The call is silent.
const parsePortUnsafe = (raw) => {
  const port = Number(raw)
  if (!Number.isInteger(port)) throw 'kein Port' // bad
  return port
}
```

When result, when Exception?

- Result for expected domain errors such as validation
- Exception for the unexpected, such as a missing database
- A central place to catch, log, and respond



Throw only Error objects

- You always throw Error objects, and the cause is in cause
- In catch, you first check what arrived
- You use the Error or throw it onward

```
const readPort = (raw) => {
  const port = Number(raw)
  if (!Number.isInteger(port)) throw new Error(`Ungültiger Port: ${raw}`)
  return port
}

try {
  readPort(process.env.PORT ?? '')
} catch (error) {
  const message = Error.isError(error) ? error.message : String(error)
  console.error(message)
}
```

A Result object makes errors visible

- Success and error come back as a simple object
- ok says which case you have
- The caller checks ok before using the value

```
const parsePort = (raw) => {
  const port = Number(raw)
  if (!Number.isInteger(port)) return { ok: false, error: 'not-a-number' }
  if (port < 1 || port > 65535) return { ok: false, error: 'out-of-range' }
  return { ok: true, value: port }
}

const result = parsePort('80a')
if (result.ok) startServer(result.value)
else console.error(`Port ungültig: ${result.error}`)
```

Where do async errors get lost?

- An empty catch swallows errors
- A Promise without await and catch becomes an unhandled rejection
- fetch does not throw on HTTP errors, you check response.ok yourself

```
const loadOrders = async (customerId) => {
  const response = await fetch(
    `https://api.example.org/customers/${customerId}/orders`,
  )
  if (!response.ok) throw new Error(`HTTP ${response.status}`)
  return response.json()
}

// wrong: errors are lost
loadOrders('42').catch(() => {})
// right: handle or pass on
const orders = await loadOrders('42')
```

Your turn: Check a quantity as a Result

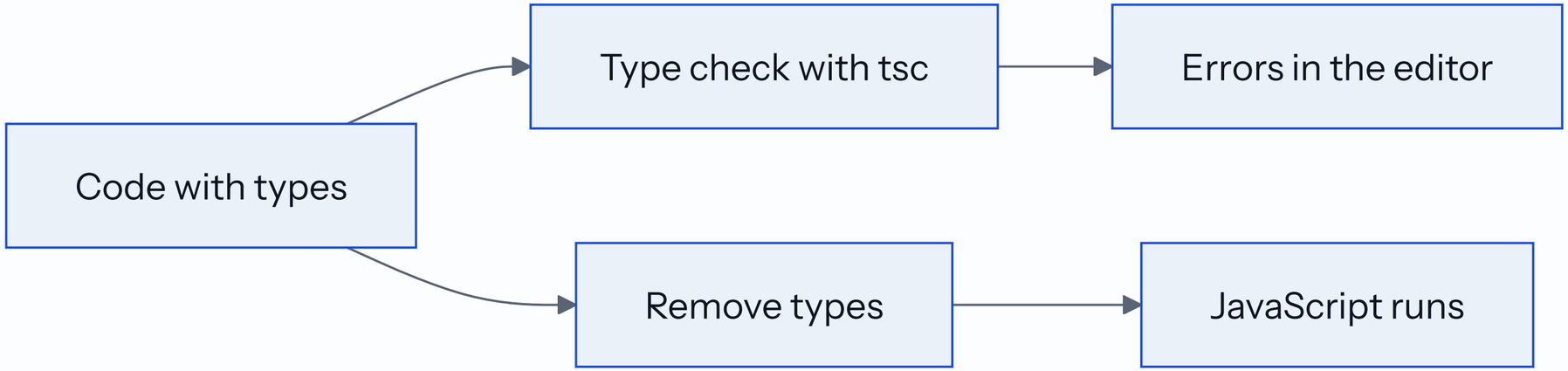
- Replace the string throw with a Result object
- Return two error codes for two error cases
- Handle the call with an if on ok

first types, setup, unions, narrowing

Annotate boundaries, infer the rest · A project setup that ends style debates · Types describe exactly the allowed values · Make invalid states impossible

Why TypeScript?

- Errors surface while writing, not for customers
- Types document intent and enable refactoring
- At runtime, it is regular JavaScript again



Annotate boundaries

- Parameters always need a type annotation
- Write down return types for public functions
- The compiler reports invalid calls immediately

```
const formatPrice = (amountInCents: number, currency: string): string =>
  `${(amountInCents / 100).toFixed(2)} ${currency}`

const label = formatPrice(1999, 'EUR') // '19.99 EUR'

// formatPrice('1999', 'EUR')
// Error: string is not assignable to number
```

Type Inference handles the rest

- Local variables usually do not need a type annotation
- const produces literal types, let the general type
- Object types and array types come from the value

```
const status = 'open'          // Type: 'open'
let mutableStatus = 'open'     // Type: string
const ids = [1, 2, 3]          // Type: number[]
const user = { name: 'Ada', age: 36 }
// Type: { name: string; age: number }

const doubled = ids.map((id) => id * 2) // number[]
```

unknown instead of any

- any turns off type checking and spreads
- unknown forces you to narrow before use
- External data is always unknown

```
const parseJson = (text: string): unknown => JSON.parse(text)

const data = parseJson('{"count": 3}')
// data.count  // Error: data is unknown

if (typeof data === 'object' && data !== null && 'count' in data) {
  console.log(data.count)
}
```

What types cannot do at runtime

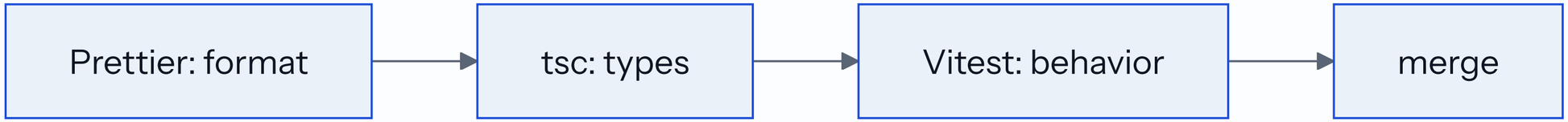
- Types disappear at runtime
- as is a type assertion, not a check
- strict stays on; since TypeScript 6 it is the default

Your turn: Type a function at its boundaries

- Add type annotations to parameters
- Write down the return type
- Leave local values to Type Inference

Three checks, one command

- Format, types, and tests
- The same commands locally and in CI
- Fast checks first



strict is only the lower bound

- strict has been the TypeScript 6 default and is only the lower bound
- Two flags harden index access and optional properties
- erasableSyntaxOnly keeps the code executable in Node

```
{
  "compilerOptions": {
    "target": "es2025",
    "module": "nodenext",
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true,
    "erasableSyntaxOnly": true,
    "verbatimModuleSyntax": true,
    "allowImportingTsExtensions": true,
    "types": ["node"],
    "noEmit": true
  },
  "include": ["src"]
}
```

One check script for local use and CI

- Each check has its own script
- check runs all of them in a fixed order
- CI runs exactly this script

```
{
  "type": "module",
  "scripts": {
    "format": "prettier --write .",
    "typecheck": "tsc --noEmit",
    "test": "vitest run",
    "check": "prettier --check . && tsc --noEmit && vitest run"
  },
  "devDependencies": {
    "prettier": "^3.9.0",
    "typescript": "^7.0.0",
    "vitest": "^5.0.0"
  }
}
```

What the compiler does not find by itself

- tsc does not find Floating Promises without await
- Turning on strict later creates many errors
- Local runs and CI use different commands

Your turn: Modernize an old tsconfig

- Set module and moduleResolution to nodenext
- Set strict and the two stricter flags
- Remove outdated options like baseUrl

Object Types describe data

- Properties have types, and a question mark makes them optional
- readonly prevents accidental assignment
- type and interface both describe object shapes

```
type User = {
  readonly id: string
  name: string
  email?: string
}

const ada: User = { id: 'u1', name: 'Ada' }

// ada.id = 'u2'
// Error: id is readonly
```

Literal Types replace magic strings

- A Union allows one value from several options
- Literal Types allow exactly certain values
- Together, they replace magic strings and enums

```
type Status = 'draft' | 'published' | 'archived'

const statusLabel = (status: Status): string =>
  status === 'draft' ? 'Entwurf'
  : status === 'published' ? 'Veröffentlicht'
  : 'Archiviert'

statusLabel('published')
// statusLabel('deleted')
// Error: 'deleted' is not a status
```

Values and types from one source

- as const freezes Literal Types
- typeof gets the type of a value
- List and type stay in sync automatically

```
const roles = ['admin', 'editor', 'viewer'] as const
type Role = (typeof roles)[number]
// 'admin' | 'editor' | 'viewer'

const isRole = (value: string): value is Role =>
  roles.some((role) => role === value)
```

The shape counts, not the name

- An object fits if the required properties are present
- Extra properties are allowed on variables
- Only fresh object literals get the excess property check

```
type Point = { x: number; y: number }
const draw = (point: Point) => point.x + point.y

const point3d = { x: 1, y: 2, z: 3 }
draw(point3d)                  // ok

// draw({ x: 1, y: 2, z: 3 })
// Error: z is unknown in Point
```

Your turn: order status from one source

- Derive the Union Type from the list
- Make the Type Guard check against the list
- Add a new status only in the list

Unions need type checks

- A union value can have multiple shapes
- TypeScript knows the exact type only after a check
- This is called Narrowing

```
const formatId = (id: string | number): string => {
  if (typeof id === 'number') {
    return id.toFixed(0)       // here: number
  }
  return id.toUpperCase()      // here: string
}

const toList = (input: string | string[]) =>
  Array.isArray(input) ? input : [input]
```

A discriminator separates the variants

- Each variant has a shared literal property
- Properties exist only where they make sense
- There is no data without success and no message without an error

```
type LoadState =
  | { status: 'loading' }
  | { status: 'success'; data: string[] }
  | { status: 'error'; message: string }

const render = (state: LoadState): string => {
  switch (state.status) {
    case 'loading': return 'Lädt'
    case 'success': return state.data.join(', ')
    case 'error': return `Fehler: ${state.message}`
  }
}
```

Force an Exhaustiveness Check

- never is the type with no values
- If anything remains in default, the compiler reports a type error
- New variants lead you to every affected location

```
const assertNever = (value: never): never => {
  throw new Error(`Unerwarteter Fall: ${JSON.stringify(value)}`)
}

const icon = (state: LoadState): string => {
  switch (state.status) {
    case 'loading': return 'spinner'
    case 'success': return 'check'
    case 'error': return 'alert'
    default: return assertNever(state)
  }
}
```



Type Guards believe you without checking

- value is T turns a check function into a Type Guard
- TypeScript does not verify your condition
- Narrowing is lost in callbacks if let is reassigned later

```
type ErrorState = Extract<LoadState, { status: 'error' }>

const isError = (state: LoadState): state is ErrorState =>
  state.status === 'error'

const messages = states.filter(isError).map((state) => state.message)
```

Your turn: A new state for LoadState

- Add the idle state to LoadState
- Follow the type errors from assertNever
- Update every switch

generics, utility types, functional

Generics preserve what any throws away · Derive types instead of copying · Pure Functions in the core, side effects at the edge

any loses information

- One function should support many types
- With any, nothing useful comes back
- Copies per type quickly get out of hand

```
const firstAny = (items: any[]) => items[0]

const name = firstAny(['Ada', 'Grace']) // any
name.toFixed(2) // compiles, crashes at runtime
```

Type parameters connect input and output

- The type is set at the call site
- TypeScript usually derives it from the arguments
- Input and output stay connected

```ts
const first = <Item>(items: readonly Item[]): Item | undefined =>
  items[0]

const name = first(['Ada', 'Grace']) // string | undefined
const count = first([1, 2, 3])       // number | undefined

const pair = <Left, Right>(left: Left, right: Right): [Left, Right] =>
  [left, right]
```

extends and keyof set limits

- extends defines what is allowed
- keyof Item returns all keys
- Item[Key] is the type of the property

```
const pluck = <Item, Key extends keyof Item>(
  items: readonly Item[],
  key: Key,
): Item[Key][] => items.map((item) => item[key])

const users = [
  { name: 'Ada', age: 36 },
  { name: 'Grace', age: 45 },
]

const names = pluck(users, 'name') // string[]
const ages = pluck(users, 'age')   // number[]
// pluck(users, 'email') // Error: not a key of users
```

Types have parameters too

- A default covers the common case, such as Error
- Result makes error cases visible in the signature
- A type parameter used only once adds nothing

```
type Result<Value, Failure = Error> =
  | { ok: true; value: Value }
  | { ok: false; error: Failure }
type PortError = 'not-a-number' | 'out-of-range'

const parsePort = (raw: string): Result<number, PortError> => {
  const port = Number(raw)
  if (!Number.isInteger(port)) return { ok: false, error: 'not-a-number' }
  if (port < 1 || port > 65535) return { ok: false, error: 'out-of-range' }
  return { ok: true, value: port }
}

// unnecessary: Value appears only once
const logGeneric = <Value>(value: Value): void => console.log(value)
```

Your turn: A generic function last

- Introduce a type parameter for the element
- Make the return type honest with undefined
- Check the call without a type argument

Copied types drift apart

- A form needs User without id
- A list needs only id and name
- Copies forget new properties

```
type User = {
  id: string
  name: string
  email: string
  role: 'admin' | 'member'
}

// Copy: outdated with the next new property
type UserFormCopy = { name: string; email: string; role: string }
```

Pick and Omit derive variants

- Partial makes all properties optional
- Pick selects, Omit leaves out
- Utility types compose like functions

```
type UserPatch = Partial<Omit<User, 'id'>>
type UserPreview = Pick<User, 'id' | 'name'>
type FrozenUser = Readonly<User>

const updateUser = (user: User, patch: UserPatch): User => ({
  ...user,
  ...patch,
})

updateUser(ada, { role: 'admin' })
```

Record enforces complete lookup tables

- Record describes lookup tables with fixed keys
- Exclude and Extract filter unions
- If a key is missing in a Record, the compiler reports it

```
type Role = User['role'] // 'admin' | 'member'

const permissions: Record<Role, readonly string[]> = {
  admin: ['read', 'write', 'delete'],
  member: ['read'],
}

type DomainEvent = 'created' | 'updated' | 'deleted'
type Change = Exclude<DomainEvent, 'created'> // 'updated' | 'deleted'
type MaybeName = string | null | undefined
type Name = NonNullable<MaybeName>             // string
```

Derive types from functions

- ReturnType gives you the return type
- Parameters returns the parameters as a tuple
- Awaited unwraps Promises

```
const loadUser = async (id: string) => ({ id, name: 'Ada', active: true })

type LoadedUser = Awaited<ReturnType<typeof loadUser>>
// { id: string; name: string; active: boolean }

type LoadUserArgs = Parameters<typeof loadUser>
// [id: string]
```

Where utility types check less than you expect

- Omit does not check keys; typos go unnoticed
- Readonly and Partial work only one level deep
- Partial allows explicit undefined; spread then overwrites

Your turn: derive a patch for orders

- Remove id and customerId with Omit
- Make the rest optional with Partial
- Add a new property only to Order

Hidden dependencies make code unpredictable

- Result depends on time, global state, or I/O
- Tests need mocks and fixed times
- Pure Functions depend only on their parameters

```
let discountRate = 0.1

// impure: global variable and current time
const priceImpure = (netCents: number): number =>
  new Date().getDay() === 0
    ? Math.round(netCents * (1 - discountRate) * 0.9)
    : Math.round(netCents * (1 - discountRate))

// pure: everything comes through parameters
const price = (netCents: number, rate: number): number =>
  Math.round(netCents * (1 - rate))
```

pipe composes small steps

- Each step has exactly one purpose
- pipe connects them from left to right
- New flows come from new combinations

```
const pipe =
  <Value>(...steps: ReadonlyArray<(value: Value) => Value>) =>
  (input: Value): Value =>
    steps.reduce((value, step) => step(value), input)

const trim = (text: string) => text.trim()
const toLower = (text: string) => text.toLowerCase()
const collapseSpaces = (text: string) => text.replace(/\s+/g, ' ')

const normalize = pipe(trim, toLower, collapseSpaces)
normalize('  Hallo   WELT ') // 'hallo welt'
```

Reuse predicates and comparators

- Functions that take or return functions
- Configuration through Partial Application
- Ready-made callbacks for filter and toSorted

```
type User = { name: string; age: number; title?: string }

const hasField = (field: string) => (item: object): boolean => field in item
const byNumber =
  <Item>(select: (item: Item) => number) =>
  (left: Item, right: Item): number =>
    select(left) - select(right)

const employedByAge = users
  .filter(hasField('title'))
  .toSorted(byNumber((user: User) => user.age))
```

Side effects belong at the edge

- Core: pure domain logic, no I/O
- Shell: reads inputs, calls the core, writes results
- The core is testable without mocks



When functional style hurts

- Excessive abstraction makes code unreadable
- Copying large structures costs time in hot loops
- Memoization is only useful for Pure Functions with a limit

```
const memoize = <Input, Output>(compute: (input: Input) => Output) => {
  const cache = new Map<Input, Output>()
  return (input: Input): Output => {
    if (cache.has(input)) return cache.get(input) as Output
    const result = compute(input)
    cache.set(input, result)
    return result
  }
}
```

Your turn: An order total without hidden dependencies

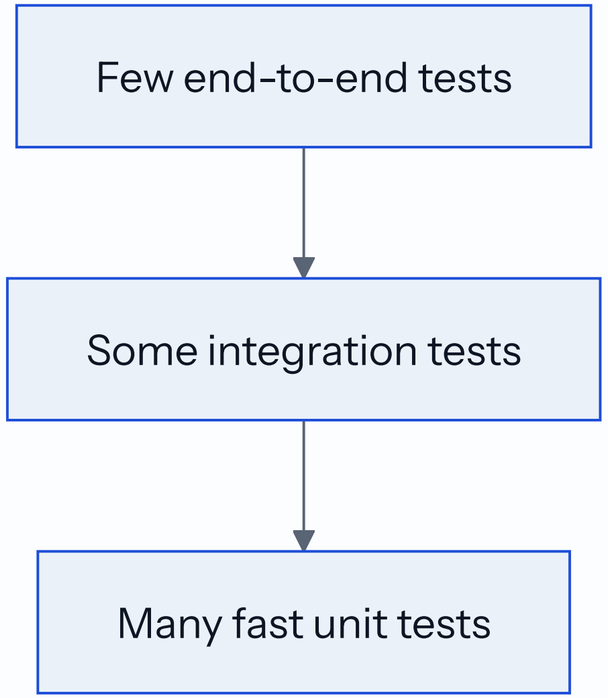
- Pass tax rate and weekday as parameters
- Read the clock only in the shell
- Test the core without a mock

tests, iterators, async for pros

Test behavior with Vitest, not details · Generators produce values only on demand · Cancel, limit, and stream async work

Why tests pay off

- Tests give fast feedback when changing code
- Pure Functions are easiest to test
- Many Unit Tests form the broad base



A test describes a behavior

- describe groups, it describes a behavior
- expect checks with matchers like toBe and toEqual
- Error cases always belong in tests

```ts
// price.test.ts
import { describe, expect, it } from 'vitest'
import { applyDiscount } from './price.ts'

describe('applyDiscount', () => {
  it('zieht den Rabatt in Cent ab', () => {
    expect(applyDiscount(10_000, 0.1)).toBe(9_000)
  })

  it('lehnt negative Rabatte ab', () => {
    expect(() => applyDiscount(10_000, -0.1)).toThrow('Rabatt')
  })
})
```

Dependencies as parameters make tests simple

- Async tests with await
- rejects and resolves for Promises
- Dependencies as parameters, with vi.fn in the test

```
const greetUser = async (
  id: string,
  fetchUser: (id: string) => Promise<{ name: string }>,
) => `Hallo ${(await fetchUser(id)).name}`

it('grüßt mit Namen', async () => {
  const fetchUser = vi.fn(async (id: string) => ({ id, name: 'Ada' }))
  expect(await greetUser('u1', fetchUser)).toBe('Hallo Ada')
  expect(fetchUser).toHaveBeenCalledWith('u1')
})

it('meldet Fehler', async () => {
  const failing = vi.fn(async () => { throw new Error('nicht gefunden') })
  await expect(greetUser('x', failing)).rejects.toThrow('nicht gefunden')
})
```

Tables and Fake Timers save tests and time

- it.each for many similar test cases
- Fake Timers instead of real waiting time
- Behavior instead of internal details

```
it.each([
  ['8080', 8080],
  ['1', 1],
])('parsePort(%s) ergibt %i', (raw, expected) => {
  expect(parsePort(raw)).toEqual({ ok: true, value: expected })
})

it('läuft nach fünf Sekunden ab', () => {
  vi.useFakeTimers()
  const onExpire = vi.fn()
  startSession(onExpire)
  vi.advanceTimersByTime(5_000)
  expect(onExpire).toHaveBeenCalledOnce()
  vi.useRealTimers()
})
```

Your turn: Test the parsePort error case

- Describe the error case as its own test
- Check the Result with toEqual
- See the test fail first

for...of queries an iterator

- for...of works with arrays, strings, Map, and Set
- Behind it is an iterator with next()
- Each call returns value and done

```
const iterator = ['a', 'b'][Symbol.iterator]()

iterator.next() // { value: 'a', done: false }
iterator.next() // { value: 'b', done: false }
iterator.next() // { value: undefined, done: true }

for (const [key, value] of new Map([['x', 1]])) {
  console.log(key, value) // x 1
}
```

Generators pause at yield

- function* creates an iterator without boilerplate
- yield returns a value and pauses
- Execution continues only at the next next()

```
function* naturals(): Generator<number> {
  let current = 1
  while (true) yield current++
}

function* take<Item>(items: Iterable<Item>, limit: number): Generator<Item> {
  if (limit <= 0) return
  let taken = 0
  for (const item of items) {
    yield item
    if (++taken >= limit) return
  }
}

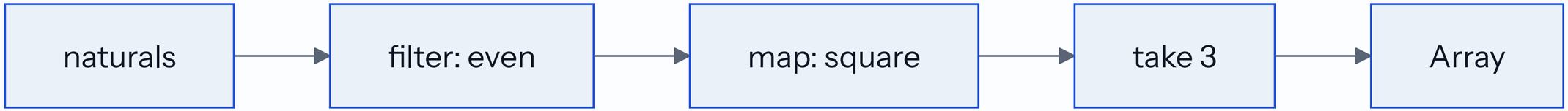
console.log([...take(naturals(), 3)]) // [1, 2, 3]
```

Iterator Helpers work lazily

- Iterator Helpers provide map, filter, and take directly on the Iterator
- Each value moves through the whole chain one by one
- No intermediate arrays are created, even for infinite sequences

```
const evenSquares = naturals()
  .filter((value) => value % 2 === 0)
  .map((value) => value * value)
  .take(3)

console.log([...evenSquares]) // [4, 16, 36]
```



for await fetches pages one by one

- async function* returns values that arrive piece by piece
- for await waits for each value
- This fits pagination and streams perfectly

```
async function* fetchPages(baseUrl: string): AsyncGenerator<unknown[]> {
  let page = 1
  while (true) {
    const response = await fetch(`${baseUrl}?page=${page}`)
    const items: unknown[] = await response.json()
    if (items.length === 0) return
    yield items
    page++
  }
}

for await (const items of fetchPages('https://api.example.org/orders')) {
  console.log(`${items.length} Bestellungen`)
}
```

What you need to watch with Generators

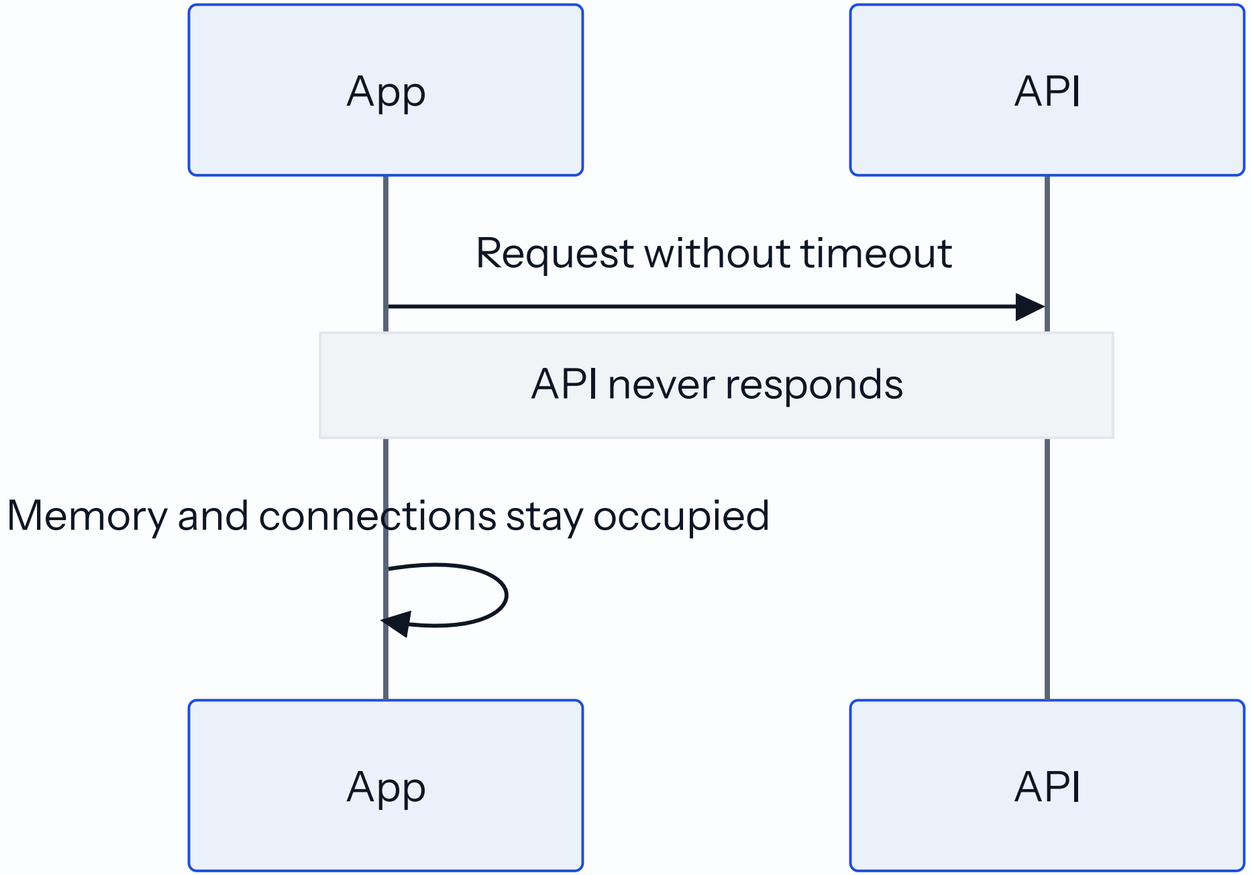
- Generators can be iterated only once
- You never collect infinite sequences completely
- break calls return; finally cleans up

Your turn: evaluate an infinite sequence lazily

- Choose even numbers with filter
- Square with map and limit with take
- Finally, sum with reduce

What goes wrong in production?

- Requests hang forever without a timeout
- A thousand parallel calls overload the target server
- Large files end up entirely in memory



Every request needs an AbortSignal

- AbortController for manual cancellation
- AbortSignal.timeout for timeouts
- AbortSignal.any for several reasons at once

```
const controller = new AbortController()
const signal = AbortSignal.any([controller.signal, AbortSignal.timeout(5_000)])

try {
  const response = await fetch('https://api.example.org/report', { signal })
  console.log(await response.text())
} catch (error: unknown) {
  if (error instanceof Error && error.name === 'TimeoutError') {
    console.error('Timeout überschritten')
  } else if (error instanceof Error && error.name === 'AbortError') {
    console.error('Abgebrochen')
  } else throw error
}
```

A pool limits concurrency

- Promise.all starts everything at once
- A fixed-size pool works through the list
- The order of the results is preserved

```
const mapWithLimit = async <Input, Output>(
  items: readonly Input[],
  limit: number,
  task: (item: Input) => Promise<Output>,
): Promise<Output[]> => {
  const results: Output[] = []
  let nextIndex = 0
  const worker = async () => {
    while (nextIndex < items.length) {
      const index = nextIndex++
      results[index] = await task(items[index]!)
    }
  }
  await Promise.all(Array.from({ length: Math.min(limit, items.length) }, worker))
  return results
}
```

Large data flows as a stream

- You read line by line instead of the whole file
- Memory usage stays constant
- pipeline connects streams with backpressure

```
import { createReadStream, createWriteStream } from 'node:fs'
import { createInterface } from 'node:readline'
import { pipeline } from 'node:stream/promises'
import { createGzip } from 'node:zlib'

const countErrors = async (path: string): Promise<number> => {
  const lines = createInterface({ input: createReadStream(path), crlfDelay: Infinity })
  let errorCount = 0
  for await (const line of lines) if (line.includes('ERROR')) errorCount++
  return errorCount
}

await pipeline(createReadStream('app.log'), createGzip(), createWriteStream('app.log.gz'))
```

Where canceling does not really cancel

- Promise.race and Promise.all do not cancel losers
- A missing signal stops cancellation halfway down
- Fire-and-forget without catch creates unhandled rejections

Your turn: Load an order with a timeout

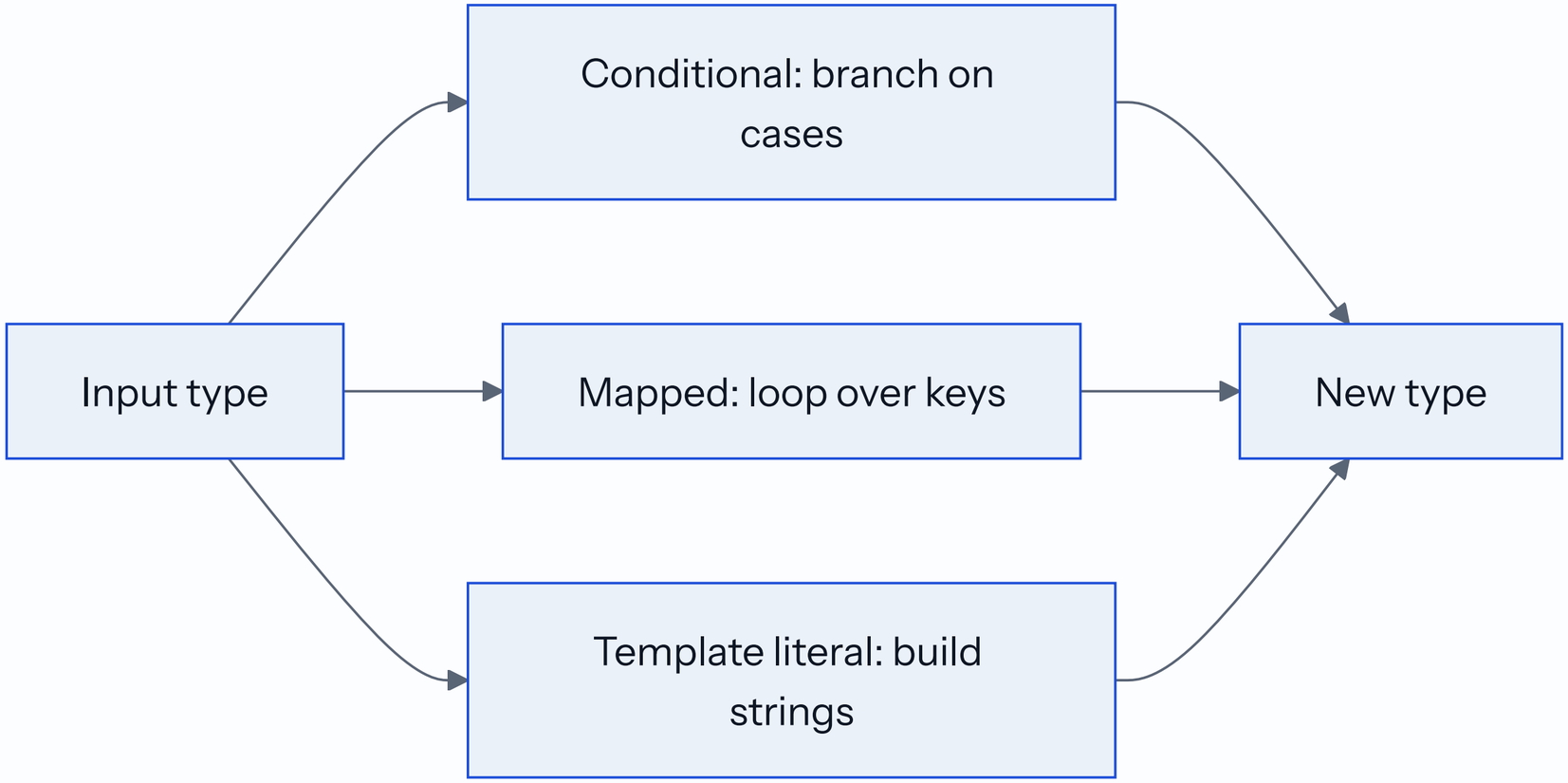
- Pass AbortSignal.timeout to fetch
- Recognize a TimeoutError by its name
- Rethrow other errors unchanged

advanced types, validation, branded types

Compute types where they prevent bugs · zod checks what types cannot see · Branded types prevent swapped IDs

Compute types from types

- Until now, types describe values
- Now types create new types
- This pays off at boundaries like routers, events, and configuration



Conditional types distinguish cases

- extends is a condition at the type level
- infer pulls parts out of a type
- Unions are processed member by member

```
type ElementOf<Value> = Value extends readonly (infer Item)[] ? Item : never

type A = ElementOf<string[]>  // string
type B = ElementOf<number>    // never

type Unwrap<Value> =
  Value extends Promise<infer Inner> ? Unwrap<Inner> : Value

type C = Unwrap<Promise<Promise<number>>> // number
```

Mapped types loop over keys

- Loop over all keys of a type
- Key Remapping with as: rename or filter
- Add or remove readonly and optional modifiers

```
type User = { name: string; age: number }

type Getters<Shape> = {
  [Key in keyof Shape & string as `get${Capitalize<Key>}`]: () => Shape[Key]
}
type UserGetters = Getters<User>
// { getName: () => string; getAge: () => number }

type Mutable<Shape> = { -readonly [Key in keyof Shape]: Shape[Key] }
```

Template Literal Types build strings

- Combine strings at the type level
- Create all combinations from unions
- Split strings with infer, such as route params

```
type HttpMethod = 'GET' | 'POST'
type Resource = '/users' | '/orders'
type Endpoint = `${HttpMethod} ${Resource}`
// 'GET /users' | 'GET /orders' | 'POST /users' | 'POST /orders'

type EventName<Entity extends string> = `${Entity}:${'created' | 'deleted'}`
type UserEvent = EventName<'user'> // 'user:created' | 'user:deleted'

const on = (event: UserEvent, handler: () => void) => { /* … */ }
// on('user:updated', ...) // Fehler
```

When are computed types not worth it?

- Distribution over unions can surprise you; square brackets prevent it
- Deep recursion slows down the editor and compiler
- Unreadable error messages cost the team time

```
type IsString<Value> = Value extends string ? true : false
type D = IsString<string | number> // boolean, also true | false

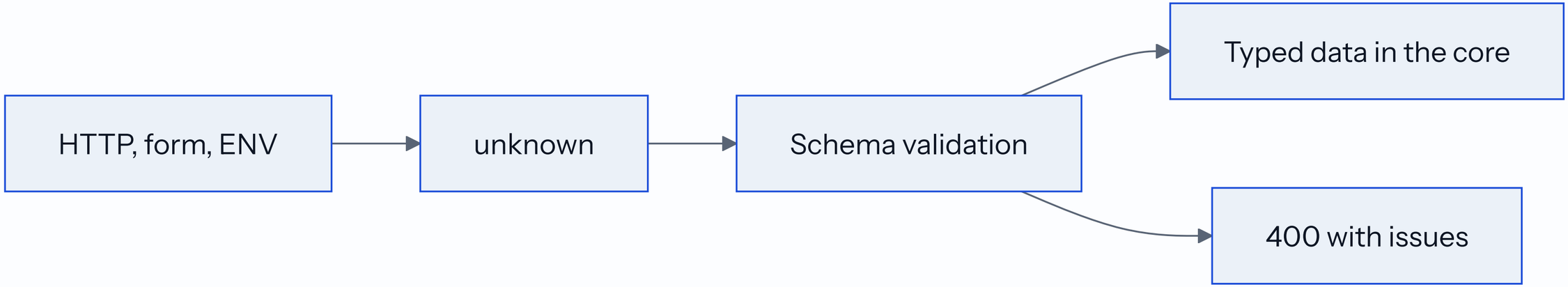
type IsStringStrict<Value> = [Value] extends [string] ? true : false
type E = IsStringStrict<string | number> // false
```

Your turn: compute event names from unions

- Build EventName from Entity and Change
- See the wrong call as a Type Error
- Check whether the type stays readable

Types disappear at runtime

- JSON, forms, and environment are unknown
- An as cast checks nothing
- Validation happens at the boundary, then you trust the data



Schema and type from one source

- The schema describes the data at runtime
- z.infer derives the TypeScript type
- Nothing is maintained twice

```ts
import { z } from 'zod'

const OrderItemInput = z.strictObject({
  productId: z.string().min(1),
  quantity: z.number().int().min(1),
})

export const PlaceOrderInput = z.strictObject({
  customerId: z.string().min(1),
  items: z.array(OrderItemInput).min(1),
})

export type PlaceOrderInput = z.infer<typeof PlaceOrderInput>
// { customerId: string; items: { productId: string; quantity: number }[] }
```

parse throws, safeParse returns a result

- parse throws for invalid data
- safeParse returns a result object
- issues describe the path and problem

```
import express from 'express'

const app = express()
app.use(express.json())

app.post('/orders', (request, response) => {
  const result = PlaceOrderInput.safeParse(request.body)
  if (!result.success) {
    response.status(400).json({ issues: result.error.issues })
    return
  }
  response.status(201).json(placeOrder(result.data))
})
```

Configuration fails early at startup

- z.coerce converts strings to Numbers or Dates
- default sets missing values
- The startup check fails early and loudly

```
const EnvSchema = z.object({
  PORT: z.coerce.number().int().positive().default(3000),
  NODE_ENV: z.enum(['development', 'production', 'test']),
  DATABASE_URL: z.string().min(1),
})

export const env = EnvSchema.parse(process.env)
// env.PORT is number, env.NODE_ENV a union of literal types
```

Where zod behaves differently than expected

- Unknown keys are removed by default
- Input type and output type differ with transforms
- Inside, validation is not repeated

Your turn: a schema for your domain

- Describe properties with suitable checks
- Derive the type with z.infer
- Reject unknown keys with z.strictObject

Everything is a string

- Customer ID and order ID have the same type
- Swapped arguments compile without problems
- The error appears only in the data

```
const cancelOrder = (customerId: string, orderId: string) => {
  /* … */
}

const customerId = 'c-42'
const orderId = 'o-7'

cancelOrder(orderId, customerId) // compiles, but is wrong
```

A Brand makes types distinguishable

- A Brand adds an invisible marker to the base type
- The marker exists only at the type level
- Swapping becomes a type error

```
declare const brand: unique symbol
type Brand<Base, Name extends string> = Base & { readonly [brand]: Name }

type CustomerId = Brand<string, 'CustomerId'>
type OrderId = Brand<string, 'OrderId'>

const cancelOrder = (customerId: CustomerId, orderId: OrderId) => {
  /* … */
}

// cancelOrder(orderId, customerId)
// Error: OrderId is not assignable to CustomerId
```

Create them only with validation

- Smart Constructor validates and brands
- The cast is in exactly one place
- You brand outside data at the boundary

```
const toOrderId = (raw: string): OrderId => {
  if (!/^o-\d+$/.test(raw)) throw new Error(`Ungültige OrderId: ${raw}`)
  return raw as OrderId
}

const orderId = toOrderId('o-7')   // OrderId
// toOrderId('c-42')               // throws: Invalid OrderId
```

More than IDs

- Units like cents, euros, and milliseconds
- Validated values like email or a non-empty string
- Explicit conversions between units

```
type Cents = Brand<number, 'Cents'>
type Euro = Brand<number, 'Euro'>

const toCents = (euro: Euro): Cents => Math.round(euro * 100) as Cents

const addVat = (net: Cents, rate: number): Cents =>
  Math.round(net * (1 + rate)) as Cents

// addVat(19.99 as Euro, 0.19) // Error: Euro is not Cents
```

Where Brands lose their protection

- Casts outside Smart Constructors bypass everything
- JSON and the database return raw values again
- Brands are worth it only where mix-ups are realistic

Your turn: cent amounts only with validation

- Allow whole, non-negative numbers
- Reject everything else with an Error
- Write the cast only in the Smart Constructor

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen