```
// Understand the model. Predict the behavior.
type Result<T, E> =
  | { ok: true;  value: T }
  | { ok: false; error: E };

function assertNever(x: never): never {
  throw new Error(`Unreachable: ${x}`);
}
```

PERSIAN ENGINEERING REFERENCE

```
const parsed = UserSchema.safeParse(await res.json());
if (!parsed.success) return { ok: false, error:
type Keys<T> = { [K in keyof T]-?: K }[keyof T
type Infer<S> = S extends Schema<infer T> ? T
```

TS

مرجع جامع

TypeScript

از مدل ذهنی تا مهندسی Production

درک مدل، پیش‌بینی رفتار کامپایلر، طراحی نوع و راستی‌آزمایی پیاده‌سازی — راهنمایی برای توسعه‌دهندگانی که می‌خواهند TypeScript را به‌عنوان ابزار مهندسی به‌کار بگیرند، نه فقط حاشیه‌نویسی روی JavaScript.

۴۰ فصل در ۴ بخش

۸ پروژه عملی کامل

۱۳۰+ پرسش مصاحبه

مبتنی بر TypeScript 7



محمدرضا رضائیان

گردآوری و تدوین حرفه‌ای · برای جامعه توسعه‌دهندگان فارسی‌زبان

TypeScript 7 · React 19 · Next.js 16 · Node.js 24 · Zod 4

نسخه ۱٫۰٫۰ — ۲۰۲۶

شناسنامه کتاب

عنوان	مرجع جامع TypeScript — از مدل ذهنی تا مهندسی Production
پدیدآورنده	محمدرضا رضائیان
نسخه	۱٫۰٫۰ — پاییز ۱۴۰۵ (اکتبر ۲۰۲۶)
قطع	A4 — ۲۱۰ × ۲۹۷ میلی‌متر، PDF با متن قابل جست‌وجو و انتخاب
خط پایه فنی	TypeScript 7.0.2 و 6.0.3 · Zod 4 · Node.js 24 LTS · Next.js 16 · React 19
حروف‌چینی	وزیرمتن (Vazirmatn) برای متن فارسی، JetBrains Mono برای کد — هر دو با پروانه آزاد SIL OFL 1.1
تولید	خط لوله بازتولیدپذیر HTML/CSS → WeasyPrint، با فونت‌های نشانده‌شده (embedded)

درباره راستی‌آزمایی مثال‌ها

مثال‌های اصلی این کتاب با کامپایلر رسمی TypeScript (نسخه‌های 7.0.2 و 6.0.3) بررسی شده‌اند. قطعه‌هایی که **عمداً** خطا دارند با برچسب «خطای عمدی» مشخص‌اند و پیام خطای آن‌ها از خروجی واقعی کامپایلر برداشته شده است. هر ادعای وابسته به نسخه، به منبع رسمی آن در پیوست «منابع» ارجاع دارد.

پروانه استفاده

این اثر برای مطالعه و استفاده شخصی و آموزشی آزاد است. بازنشر تجاری بدون اجازه پدیدآورنده مجاز نیست. کدهای نمونه کتاب را می‌توانید بدون محدودیت در پروژه‌های خود — تجاری یا غیرتجاری — به‌کار ببرید.

با احترام به جامعه توسعه‌دهندگان فارسی‌زبان — پاییز ۱۴۰۵

راهنمای خواندن این کتاب

این کتاب برای سه نوع خواندن طراحی شده است. لازم نیست از صفحه اول شروع کنید؛ اما لازم است بدانید کجا ایستاده‌اید.

سه مسیر پیشنهادی

شما که هستید	مسیر پیشنهادی	فصل‌های کلیدی
توسعه‌دهنده JavaScript که تازه به TS می‌آید	بخش یکم را کامل و به‌ترتیب بخوانید؛ سپس فصل‌های ۱۳، ۱۴، ۲۲ و بخش چهارم	۱ تا ۱۲، ۲۲، ۳۴
توسعه‌دهنده React / Next.js	فصل‌های ۴ تا ۱۰ برای مدل ذهنی، سپس مستقیم به فصل‌های ۲۱، ۲۲، ۳۴ تا ۳۹	۱۰، ۲۲، ۳۷، ۳۸
مهندس باتجربه که دنبال عمق است	فصل‌های ۶، ۱۰، و تمام بخش دوم و سوم؛ پیوست C و F به‌عنوان مرجع روزمره	۱۷، ۲۴، ۲۶، ۳۲، ۳۳

اگر با یک **مشکل مشخص** آمده‌اید — یک خطای عجیب، یک جنریک که استنتاج نمی‌شود، یک پیکربندی خراب — از پیوست D (عیب‌یابی) شروع کنید؛ هر ردیف آن به فصل مربوط ارجاع می‌دهد.

قراردادهای کتاب

- **هر فصل با دو پرسش مصاحبه** و پاسخ تشریحی بسته می‌شود؛ پیوست A نیز ۵۰ پرسش عرضی سطح‌بندی‌شده دارد.
- بخش **«زیر کاپوت — چرا این‌طور است؟»** در پایان هر فصل، سازوکار درونی کامپایلر و «چرایی» تصمیم‌های طراحی زبان را باز می‌کند — اگر عجله دارید می‌توانید در خوانش اول از آن بگذرید، اما فهم عمیق همان‌جاست.
- **پنل‌های کد تیره** کد واقعی و کامل‌اند؛ متن آن‌ها قابل انتخاب و کپی است. برچسب ✓ type-checks یعنی آن قطعه با کامپایلر رسمی بررسی شده است.
- قطعه‌های دارای برچسب **«خطای عمدی ✗»** آگاهانه کامپایل نمی‌شوند؛ پیام خطای واقعی کامپایلر بلافاصله بعد از آن‌ها در کادر قرمز آمده است.
- کادرهای رنگی چهار معنا دارند: **مدل ذهنی** (طلایی — قلب هر فصل)، **نکته** (آبی)، **هشدار** (قرمز — دام‌های واقعی)، و **در عمل** (سرمه‌ای — تجربه production).
- اصطلاح‌های فنی به فارسی توضیح داده می‌شوند اما خود اصطلاح به انگلیسی حفظ می‌شود — چون در مستندات، پیام‌های خطا و مصاحبه‌ها با همان شکل انگلیسی روبه‌رو می‌شوید. واژه‌نامه کامل در پیوست B است.

وعده آموزشی

هر فصل باید چهار توانایی را در شما بسازد، به همین ترتیب:

۱. **مدل را بفهم** — سازوکار واقعی، نه قاعده حفظی. ۲. **رفتار را پیش‌بینی کن** — قبل از اجرا بگو کامپایلر چه می‌گوید.

۲. **نوع را طراحی کن** — از ابزار برای مدل‌سازی مسئله واقعی استفاده کن. ۴. **پیاده‌سازی را راستی‌آزمایی کن** — با کامپایلر، تست نوع و اعتبارسنجی runtime.

مدل ذهنی کل کتاب — در یک خط

TypeScript یعنی: **اثبات‌پذیری ایستا روی ساختار** × **پاک‌شدن کامل در runtime** × **مرزهایی که باید خودت محافظت کنی**.
هر فصل یکی از این سه ضلع را عمیق می‌کند.

فهرست مطالب

بخش یکم · بنیادها و مدل ذهنی سیستم نوع (فصل‌های ۱ تا ۱۲)

۱ TypeScript چه چیزی به JavaScript اضافه می‌کند؟ ۸

۲ راه‌اندازی پروژه و زنجیره ابزار ۱۲

۳ tsconfig.json — طراحی آگاهانه پیکربندی ۱۶

۴ استنتاج نوع و Contextual Typing ۲۱

۵ نوع‌های اولیه، Literal Types و Widening ۲۵

۶ نوع‌های شیئی و Structural Typing ۲۹

۷ null و undefined — پایان «خطای میلیارد دلاری» ۳۳

۸ void و any، unknown، never ۳۷

۹ Union و Intersection ۴۲

۱۰ Narrowing و تحلیل جریان کنترل ۴۶

۱۱ توابع — قرارداد ورودی و خروجی ۵۱

۱۲ آرایه، Tuple، readonly — و هنر خواندن خطای کامپایلر ۵۶

بخش دوم · طراحی نوع پیشرفته (فصل‌های ۱۳ تا ۲۶)

۱۳ interface و type — یک انتخاب مهندسی ۶۱

۱۴ جنریک‌ها — پارامتر نوع، Constraint و Default ۶۵

۱۵ جنریک در عمل — API و کامپوننت قابل‌استفاده مجدد ۷۰

۱۶ keyof، typeof و Indexed Access ۷۵

۱۷ Conditional Types و infer ۸۰

۱۸ Mapped Types و Key Remapping ۸۵

۱۹ Template Literal Types ۸۹

۲۰ Utility Types — استاندارد و سفارشی ۹۴

۲۱ as const، satisfies و Const Type Parameters ۹۸

۲۲ Discriminated Union و Exhaustiveness ۱۰۳

۲۳ Type Predicate، Assertion Function و Branded Types ۱۰۸

۲۴ Variance و سازگاری توابع ۱۱۳

۲۵ Declaration Merging، Overload و Module Augmentation ۱۱۸

۲۶ برنامه‌نویسی Type-Level — قدرت، هزینه و مرز عقل ۱۲۲

بخش سوم · کامپایلر، ابزار و معماری (فصل‌های ۲۷ تا ۳۳)

۲۷ خانواده strict — طراحی سطح سخت‌گیری ۱۲۷

۲۸ ماژول‌ها — CommonJS، ESM و Module Resolution ۱۳۱

۲۹ فایل‌های اعلان و @types ۱۳۵
۳۰ مهاجرت از JavaScript — تدریجی و قابل برگشت ۱۴۰
۳۱ Project References، Monorepo و CI ۱۴۴
۳۲ تست نوع، دیباگ خطاهای پیچیده و کارایی کامپایلر ۱۴۸
۳۳ گذار TypeScript 6 به 7 — کامپایلر Native ۱۵۳

بخش چهارم · TypeScript در سیستم‌های Production (فصل‌های ۳۴ تا ۴۰)

۳۴ مرز داده بیرونی — اعتبارسنجی Runtime با Zod ۱۵۸
۳۵ کلاینت API تایپ‌شده با مدل خطای صریح ۱۶۳
۳۶ مدل‌سازی دامنه و ماشین حالت ۱۶۸
۳۷ React با TypeScript ۱۷۳
۳۸ Next.js — مرز سرور و کلاینت ۱۷۹
۳۹ فرم‌ها، State ناهمگام و Event Emitter تایپ‌شده ۱۸۴
۴۰ انتشار پکیج — اعلان عمومی و تست نوع ۱۹۰

پیوست‌ها (مرجع سریع)

A ۵۰ پرسش مصاحبه با پاسخ تشریحی ۱۹۵
B واژه‌نامه فارسی–انگلیسی ۲۰۲
C مرجع تصمیم گزینه‌های کامپایلر ۲۰۵
D راهنمای عیب‌یابی سیستم نوع ۲۰۸
E چک‌لیست مهاجرت از JavaScript ۲۱۰
F الگوهای طراحی نوع قابل‌استفاده مجدد ۲۱۲
G نقشه راه نهایی ۲۱۶
H منابع و مآخذ ۲۱۸

PART 01

۱

بنیادها و مدل ذهنی سیستم نوع

پیش از آنکه بتوانی نوع‌های پیچیده طراحی کنی، باید بتوانی رفتار کامپایلر را پیش‌بینی کنی. این بخش مدل ذهنی TypeScript را می‌سازد: تحلیل ایستا در برابر رفتار زمان اجرا، استنتاج نوع، سازگاری ساختاری، و تحلیل جریان کنترل. هر فصل با مثال‌های کامپایل‌شده و خطاهای واقعی کامپایلر همراه است.

---

01 TypeScript چه چیزی به JavaScript اضافه می‌کند؟

02 راه‌اندازی پروژه و زنجیره ابزار

03 tsconfig.json — طراحی آگاهانه پیکربندی

04 استنتاج نوع و Contextual Typing

05 نوع‌های اولیه، Literal Types و Widening

06 نوع‌های شیئی و Structural Typing

07 null و undefined — پایان «خطای میلیارد دلاری»

08 any، unknown، never و void

09 Union و Intersection

10 Narrowing و تحلیل جریان کنترل

11 توابع — قرارداد ورودی و خروجی

12 آرایه، Tuple، readonly — و هنر خواندن خطای کامپایلر

TypeScript چه چیزی به JavaScript اضافه می‌کند؟

تحلیل ایستا، Type Erasure و مرز دقیق بین Compile-Time و Runtime

اهداف یادگیری این فصل

- تفاوت تحلیل ایستا با رفتار زمان اجرا را با مثال دقیق توضیح دهی
- Type Erasure را بفهمی و بدانی چه چیزی به خروجی JS نمی‌رسد
- بدانی TypeScript کدام دسته از باگ‌ها را می‌گیرد و کدام را نمی‌گیرد

اگر فقط یک چیز از این فصل بماند، همین است: TypeScript در زمان اجرا **وجود ندارد**. هر چیزی که با نوع‌ها می‌سازید، قبل از اجرا پاک می‌شود. تمام قدرت و تمام محدودیت‌های TypeScript از همین واقعیت ساده نتیجه می‌شود — و بیشترِ سوءتفاهم‌های رایج هم.

تحلیل ایستا یعنی چه؟

JavaScript خطاها را در **زمان اجرا** کشف می‌کند: برنامه اجرا می‌شود، به خط مشکل‌دار می‌رسد و می‌شکند. TypeScript لایه‌ای از **تحلیل ایستا** اضافه می‌کند: برنامه را بدون اجرا کردن می‌خواند، درباره نوع هر عبارت استدلال می‌کند و ناسازگاری‌ها را قبل از اجرا گزارش می‌دهد.

static-analysis.ts — × خطای عمدی

```ts
function fullName(user: { first: string; last: string }) {
  return user.first + " " + user.last;
}

fullName({ first: "Sara" }); //  × قبل از هر اجرایی گزارش می‌شود
```

پیام کامپایلر

```
static-analysis.ts:5:10 - error TS2345: Argument of type
'{ first: string; }' is not assignable to parameter of type
'{ first: string; last: string; }'.
  Property 'last' is missing in type '{ first: string; }'.
```

در JavaScript خالص، این برنامه اجرا می‌شد و `"Sara undefined"` برمی‌گرداند — باگی بی‌صدا که شاید هفته‌ها بعد در UI دیده می‌شد. TypeScript آن را به یک خطای فوری، در محل علت (نه محل علامت) تبدیل کرد.

Type Erasure — چه چیزی به خروجی نمی‌رسد

کامپایل TypeScript دو کار مستقل انجام می‌دهد: **بررسی نوع** و **تولید JavaScript**. در مرحله دوم، همه annotationها، interfaceها، typeها و جنریک‌ها حذف می‌شوند:

erasure-input.ts — type-checks ✓

```ts
interface User { id: number; name: string }

function greet(user: User): string {
  return `Hello, ${user.name}`;
}
```

erasure-output.js — type-checks ✓

```js
// خروجی کامپایل — هیچ اثری از نوع‌ها نیست:
function greet(user) {
  return `Hello, ${user.name}`;
}
```

مدل ذهنی — دو دنیای جدا با یک مرز پاک‌شونده

TypeScript دو فضای نام جداگانه دارد: **فضای نوع‌ها** (type، interface، جنریک‌ها) که فقط در زمان کامپایل زنده است، و **فضای مقدارها** (متغیرها، توابع، کلاس‌ها) که به JavaScript می‌رسد. کلاس و enum در هر دو فضا حضور دارند — برای همین کلاس را می‌توان هم `new` کرد و هم به‌عنوان نوع نوشت.

پیامدهای عملی erasure، که در فصل‌های بعد بارها به آن‌ها برمی‌گردیم:

- نمی‌توانید در runtime بپرسید «آیا این مقدار از نوع `User` است؟» — `User` دیگر وجود ندارد. (راه‌حل: type guard در فصل ۲۳، اعتبارسنجی runtime در فصل ۳۴)
- نوع‌نویسی **هیچ سربار اجرایی** ندارد؛ کد تولیدی همان JavaScript است.
- نوشتن `as User` روی داده شبکه، هیچ بررسی‌ای انجام نمی‌دهد — فقط به کامپایلر می‌گوید «ساکت باش».

چه باگ‌هایی گرفته می‌شوند — و چه باگ‌هایی نه

TypeScript می‌گیرد	TypeScript نمی‌گیرد
غلط املایی در نام property و متد	منطق اشتباه (`≥` به‌جای `>`)
آرگومان کم/زیاد یا ناسازگار	داده بیرونی که با نوع اعلام‌شده نمی‌خواند
دسترسی به مقدار احتمالاً `null` (با strict)	خطاهای شبکه، I/O و race conditionها
شاخه فراموش‌شده در switch (با الگوی فصل ۲۲)	سرریز عدد، دقت اعشار، مشکلات کارایی

TypeScript می‌گیرد	TypeScript نمی‌گیرد
استفاده از API حذف‌شده بعد از refactor	هر چیزی که پشت `any` پنهان شده باشد

هشدار — نوع ایستا، تضمین امنیتی نیست

نوع `string` روی ورودی کاربر، نه از SQL Injection جلوگیری می‌کند نه از XSS. نوع‌ها قرارداد **داخل** برنامه‌اند؛ هر داده‌ای که از بیرون می‌آید (شبکه، فایل، query string، localStorage) باید در مرز ورود اعتبارسنجی شود. این موضوع آن‌قدر مهم است که فصل ۳۴ کاملاً به آن اختصاص دارد.

پس ارزش واقعی کجاست؟

تجربه تیم‌های بزرگ — و منطق طراحی خود زبان — سه ارزش اصلی را نشان می‌دهد:

۱. **Refactoring بی‌ترس:** وقتی امضای تابعی را عوض می‌کنید، کامپایلر تک‌تک نقاط استفاده را قرمز می‌کند. در codebase صدهزار خطی، این یعنی تفاوت بین «یک بعدازظهر» و «یک اسپرینت با رگرسیون».

۲. **مستنداتی که دروغ نمی‌گویند:** نوع‌ها برخلاف کامنت‌ها، با کد همگام می‌مانند — چون ناهمگامی‌شان خطای کامپایل است.

۳. **ابزار هوشمند:** تکمیل خودکار، rename امن و go-to-definition همگی از همین تحلیل ایستا تغذیه می‌کنند — حتی برای کاربران JavaScript خالص در VS Code.

زیر کاپوت — چرا این‌طور است؟

TypeScript سه تصمیم بنیادین دارد که همه رفتارهایش از آن‌ها مشتق می‌شود — بدون فهم این سه، بقیه کتاب «مجموعه قواعد حفظی» می‌شود؛ با فهمشان، «نتیجه منطقی».

۱) پاک‌شونده (erasable) — چرا؟ هدف TypeScript از روز اول این بود که «JavaScript با نوع» باشد، نه زبان جدید. اگر نوع‌ها اثر runtime داشتند، هر کتابخانه TS برای مصرف‌کننده JS یک runtime اضافه تحمیل می‌کرد، خروجی کامپایل قابل پیش‌بینی نبود و پذیرش تدریجی — rename یک فایل بدون تغییر رفتار — ناممکن می‌شد. بهای این تصمیم را هم صادقانه بدانید: هیچ reflection نوعی در runtime وجود ندارد (برخلاف Java/C#)، پس «بررسی نوع در زمان اجرا» باید جداگانه ساخته شود — کل فصل ۳۴ فرزند همین تصمیم است.

۲) ساختاری — چرا؟ چون JavaScript ساختاری «زندگی می‌کند»: اشیاء بدون کلاس ساخته می‌شوند، از مرز JSON عبور می‌کنند و با شکل‌شان مصرف می‌شوند، نه با شناسنامه‌شان. سیستم نوع اسمی روی چنین زبانی یعنی جنگ دائمی با الگوهای بومی آن. پیامدها را فصل ۶ باز می‌کند.

۳) عمداً ناسالم (unsound by design) — چرا؟ تیم TypeScript آگاهانه «سلامت کامل ریاضی» را قربانی «مفید بودن» کرده است: `any`، assertion ها، covariance آرایه و bivariance متدها (فصل ۲۴) همگی حفره‌های شناخته‌شده و «انتخاب‌شده»اند. منطق: سیستمِ ۱۰۰٪ سالم روی JS واقعی یا باید نیمی از کدهای موجود دنیا را رد کند یا آن‌قدر annotation بخواهد که کسی استفاده‌اش نکند. نتیجه عملی: سبز بودن کامپایلر «اثبات» نیست، «شواهد قوی» است — و نقاط ناسالم فهرست مشخصی دارند که این کتاب یک‌به‌یک نشانتان می‌دهد.

این مثلث را به‌خاطر بسپارید: هر وقت رفتاری عجیب دیدید، بپرسید «کدام ضلع دارد خودش را نشان می‌دهد؟» — تقریباً همیشه پاسخ همین‌جاست.

تمرین

مرز compile-time و runtime را پیدا کن

برای هر گزاره مشخص کنید درست است یا غلط، و چرا:

۱. «اگر پروژه بدون خطای نوع کامپایل شود، در runtime خطای `undefined is not a function` نمی‌گیریم.»

۲. «`interface` در bundle نهایی حجم اشغال می‌کند.»

۳. «`const data = await res.json() as User` تضمین می‌کند `data.id` عدد است.»

پاسخ: ۱. غلط — کافی است جایی `any` یا assertion دروغ وجود داشته باشد، یا داده بیرونی با نوع اعلام‌شده نخواند. ۲. غلط — interface کاملاً پاک می‌شود. ۳. غلط — `as` فقط ادعای بدون بررسی است؛ `json()` نوع `any` برمی‌گرداند و هر چیزی ممکن است باشد (فصل ۳۴).

پرسش‌های مصاحبه این فصل

۱. آیا TypeScript برنامه را در «زمان اجرا» امن‌تر می‌کند؟

نه مستقیماً — نوع‌ها در کامپایل پاک می‌شوند و در runtime هیچ نگهبانی نیست. ایمنی از این می‌آید که کدِ ناسازگار اصلاً به production نمی‌رسد. هر جا داده از بیرون می‌آید (env، fetch، فرم) ایمنی runtime کار schema است نه نوع (فصل ۳۴). پاسخ کامل این تمایز، معیار ردی/قبولی همین پرسش در مصاحبه است.

۲. کامپایل موفق دقیقاً چه چیزی را تضمین می‌کند و چه چیزی را نه؟

تضمین می‌کند: همه استفاده‌ها با قراردادهای اعلام‌شده سازگارند. تضمین نمی‌کند: درستی منطق، صحت داده بیرونی، نبود exception، یا درست بودن خود قراردادها — اگر annotation دروغ بگوید (مثلاً با `as`)، کامپایل سبز و باگ سر جایش است.

راه‌اندازی پروژه و زنجیره ابزار

tsc، ادیتور، اجرای مستقیم TS و نقش هر ابزار در Toolchain

اهداف یادگیری این فصل

- نقش جداگانه bundler، language service، compiler و runtime را تفکیک کنی
- یک پروژه TypeScript را از صفر و بدون قالب آماده راه‌اندازی کنی
- بدانی چرا ادیتور و tsc ممکن است نتیجه متفاوت نشان دهند

خیلی‌ها TypeScript را با قالب‌های آماده شروع می‌کنند و هرگز نمی‌فهمند زیر کاپوت چه خبر است — تا روزی که ادیتور یک چیز می‌گوید و CI چیز دیگری. این فصل زنجیره ابزار را شفاف می‌کند تا هر خطای محیطی را بتوانید به مقصر درستش برسانید.

نقش‌ها را جدا کنید

پنج نقش مستقل در کار است؛ ابزارهای مختلف ممکن است چند نقش را هم‌زمان بازی کنند، اما نقش‌ها جدا هستند:

نقش	وظیفه	نمونه ابزار
Type Checker	بررسی درستی نوع‌ها، بدون اجرا	`tsc`، کامپایلر native نسخه ۷
Language Service	هوش ادیتور: تکمیل، rename، hover	tsserver داخل VS Code
Transpiler	حذف نوع‌ها و تبدیل syntax	tsc، esbuild، SWC
Bundler	چسباندن ماژول‌ها برای مرورگر	Vite، esbuild، webpack
Runtime	اجرای JavaScript نهایی	Node.js، Bun، مرورگر

نکته کلیدی: ابزارهای سریع مثل esbuild و SWC **فقط transpile می‌کنند و اصلاً نوع‌ها را نمی‌فهمند** — نوع‌ها را حذف می‌کنند بدون بررسی. برای همین در پروژه‌های Vite، بررسی نوع یک فرمان جداگانه است (`tsc --noEmit`) که باید در CI هم اجرا شود. اگر فقط به dev server تکیه کنید، پروژه‌تان ممکن است ماه‌ها با خطای نوع «سبز» بماند.

راه‌اندازی از صفر — بدون قالب

setup

```
mkdir ts-lab && cd ts-lab
npm init -y
npm install -D typescript
npx tsc --init          # tsconfig.json پایه می‌سازد
npx tsc --version       # Version 7.0.2
```

یک فایل بسازید و دو مسیر اصلی را امتحان کنید:

src/hello.ts — type-checks ✓

```
const now: Date = new Date();
console.log(`ساعت اجرا: ${now.toISOString()}`);
```

run

```
npx tsc                 # بررسی نوع + تولید hello.js
node src/hello.js       # اجرای خروجی

# اجرای مستقیم فایل ts: یا در Node مدرن (v24)
node src/hello.ts       # type stripping — بدون بررسی نوع!
```

نکته — اجرای مستقیم TS در Node 24

Node.js جدید فایل‌های `.ts` را با «حذف نوع» (type stripping) مستقیم اجرا می‌کند؛ اما این فقط transpile است، نه type-check، و syntaxهایی که معادل مستقیم JS ندارند (مثل `enum` و `namespace` و parameter properties) را به‌صورت پیش‌فرض نمی‌پذیرد. برای توسعه راحت، ابزارهایی مثل `tsx` هم رایج‌اند — اما در همه این مسیرها، بررسی نوع همچنان وظیفه جداگانه `tsc --noEmit` است.

ادیتور از کجا می‌فهمد؟

وقتی در VS Code روی متغیری hover می‌کنید، پاسخ از **tsserver** می‌آید — همان کامپایلر، در حالت سرویس. دو پیامد عملی:

- **نسخه ادیتور و نسخه پروژه ممکن است متفاوت باشد.** VS Code به‌صورت پیش‌فرض نسخه داخلی خودش را به‌کار می‌برد. با فرمان «TypeScript: Select TypeScript Version» نسخه workspace (همان `node_modules/typescript`) را انتخاب کنید تا ادیتور و CI یک چیز بگویند.
- **ادیتور خطا نشان نمی‌دهد ولی build می‌شکند؟** تقریباً همیشه یعنی دو پیکربندی متفاوت دیده می‌شود: tsconfig دیگری در مسیر، فایلی خارج از `include`، یا نسخه متفاوت کامپایلر.

مدل ذهنی — یک مغز، چند لباس

tsc در ترمینال، tsserver در ادیتور و `vue-tsc` /پلاگین‌ها همه از **یک موتور تحلیل** استفاده می‌کنند. اگر دو جا نتیجه متفاوت دیدید، دنبال «اختلاف نظر دو موتور» نگردید — دنبال **اختلاف ورودی** بگردید: نسخه، tsconfig، یا مجموعه فایل‌ها.

ساختار پیشنهادی پروژه کوچک

layout

```
ts-lab/
├── src/            ← فقط کد TS
│   └── index.ts
├── dist/           ← خروجی کامپایل (در .gitignore)
├── package.json
└── tsconfig.json   ← فصل بعد: خط‌به‌خط
```

دو اسکریپت استاندارد که در هر پروژه‌ای — مستقل از فریمورک — معنا دارند:

package.json

```
{
  "scripts": {
    "typecheck": "tsc --noEmit",
    "build": "tsc"
  }
}
```

`typecheck` باید در CI، کنار تست‌ها، شرط merge باشد. این ساده‌ترین و پرسودترین تصمیم زیرساختی TypeScript است (فصل ۳۱).

زیر کاپوت — چرا این‌طور است؟

چرا «ترجمه» و «بررسی» دو کار جدا هستند؟ پاسخ در خط تولید خود کامپایلر است. tsc چهار فاز دارد: **parse** (متن → درخت syntax)، **bind** (اتصال شناسه‌ها به اعلان‌ها)، **check** (استنتاج و مقایسه نوع‌ها — فاز گران) و **emit** (تولید JS). نکته کلیدی: برای تولید خروجی فقط parse و emit لازم است — حذف annotation ها کاری syntax ای است و به «معنا» نیاز ندارد. esbuild و swc و type stripping نودی دقیقاً همین میان‌بر را می‌زنند: فایل‌به‌فایل، بدون bind و check، ده‌ها برابر سریع‌تر — و به همین دلیل **هیچ خطای نوعی نمی‌بینند**.

اما این میان‌بر یک پیش‌شرط پنهان دارد: ترجمه تک‌فایلی فقط وقتی درست است که معنای هیچ syntax ای به «نوع‌های فایل دیگر» وابسته نباشد. سه ویژگی قدیمی TypeScript این قاعده را می‌شکنند: `const enum` (مقدارش باید از فایل دیگر inline شود)، namespace های merge شونده، و re-export نوع‌ها بدون نشانه‌گذاری (مترجم تک‌فایلی نمی‌داند `export { X }` نوع است که باید حذف شود یا مقدار که باید بماند). گزینه‌های `isolatedModules` و

`verbatimModuleSyntax` دقیقاً برای همین وجود دارند: کد شما را در «زیرمجموعه تک‌فایل‌پذیر» زبان نگه می‌دارند — قرارداد سازگاری با کل اکوسیستم ابزار مدرن، نه سلیقه.

حالا تصویر کامل است: در توسعه، ترجمه سریعِ بدون بررسی (حلقه feedback کوتاه)؛ در CI، بررسی کاملِ بدون ترجمه (`tsc --noEmit`) — حقیقت نهایی). دو ابزار، دو فاز متفاوت از یک خط تولید — نه رقیب، که مکمل. تیمی که این تفکیک را نفهمد، یا dev سرور کندی دارد یا CI دروغگو.

تمرین

مقصر را پیدا کن

در هر سناریو، کدام جزء زنجیره مقصر است؟

۱. Vite پروژه را بدون اعتراض build می‌کند، اما `tsc --noEmit` چهل خطا می‌دهد.

۲. در VS Code خطایی نیست، همکارتان در ترمینال خطا می‌گیرد.

۳. کد کامپایل می‌شود اما در مرورگر `ReferenceError` می‌دهد.

پاسخ: ۱. هیچ‌کس — رفتار طبیعی است؛ esbuild نوع بررسی نمی‌کند؛ typecheck را به pipeline اضافه کنید. ۲. اختلاف نسخه tsserver ادیتور با typescript پروژه، یا tsconfig متفاوت. ۳. مشکل runtime است نه نوع — معمولاً ماژول/باندل یا ترتیب بارگذاری؛ سراغ فصل ۲۸ بروید.

پرسش‌های مصاحبه این فصل

۱. build با Vite/esbuild در CI سبز است — یعنی خطای نوع نداریم؟

نه. transpiler ها نوع‌ها را بدون فهمیدن حذف می‌کنند؛ «build سبز» فقط یعنی syntax قابل ترجمه بود. بررسی نوع، job جداگانه `tsc --noEmit` است که باید در CI اجباری باشد. تیمی که این job را ندارد، TypeScript را عملاً خاموش کرده است.

۲. Node 24 فایل .ts را مستقیم اجرا می‌کند؛ پس tsc به چه دردی می‌خورد؟

type stripping فقط حذف syntax نوع است — هیچ بررسی‌ای انجام نمی‌دهد و enum/namespace/parameter properties را هم به‌طور پیش‌فرض رد می‌کند. اجرای مستقیم برای حلقه توسعه عالی است؛ صحت نوع همچنان فقط از tsc (یا سرویس native نسخه ۷) می‌آید.

tsconfig.json — طراحی آگاهانه پیکربندی

strict و خانواده‌اش، target/module و گزینه‌هایی که معنای برنامه را عوض می‌کنند

اهداف یادگیری این فصل

- گزینه‌های حیاتی tsconfig را بر اساس «چه خطایی را می‌گیرد» انتخاب کنی
- تفاوت گزینه‌های type-checking با گزینه‌های emit را بدانی
- یک پیکربندی پایه قابل دفاع برای پروژه جدید بنویسی

`tsconfig.json` پرکپی‌شده‌ترین فایل دنیای TypeScript است — و این دقیقاً مشکل است. گزینه‌هایی که معنای برنامه شما را عوض می‌کنند، نباید از یک gist ناشناس بیایند. این فصل گزینه‌ها را بر اساس یک سؤال مرتب می‌کند: **این گزینه چه خطایی را می‌گیرد و چه هزینه‌ای دارد؟**

سه خانواده گزینه

گزینه‌های tsconfig سه کار کاملاً متفاوت انجام می‌دهند؛ قاطی کردنشان منشأ بیشتر سردرگمی‌هاست:

۱. **بررسی نوع** — چه چیزی خطا حساب شود: `strict` و خانواده‌اش.

۲. **Emit** — چه JavaScriptای تولید شود: `target` ، `outDir` ، `declaration` ، `sourceMap` .

۳. **Resolution** — ماژول‌ها و فایل‌ها چطور پیدا شوند: `module` , `moduleResolution` , `paths` , `include` .

خانواده سوم آن‌قدر مهم و پردردسر است که فصل ۲۸ کاملاً به آن اختصاص دارد. این‌جا روی خانواده اول و یک پایه سالم تمرکز می‌کنیم.

یک پایه قابل دفاع برای پروژه جدید

tsconfig.json

```
{
  "compilerOptions": {
   */---   بررسی نوع--  /*
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "noImplicitOverride": true,

   */---   ماژول و خروجی  (برای Node ۲۸ فصل ؛مدرن}--  /*
    "module": "nodenext",
    "target": "es2023",
    "outDir": "dist",
    "sourceMap": true,
```

```
    */--- سلامت پروژه ---  /*
    "verbatimModuleSyntax": true,
    "skipLibCheck": true
  },
  "include": ["src"]
}
```

هر خط این فایل یک تصمیم است، نه یک طلسم. مهم‌ترین‌ها:

strict — نه یک گزینه، بلکه یک چتر

`strict: true` مجموعه‌ای از بررسی‌ها را روشن می‌کند. سه عضو پرتأثیر:

گزینه	بدون آن	با آن
`strictNullChecks`	`null` به هر نوعی قابل انتساب است؛ کامپایلر درباره غیبت مقدار سکوت می‌کند	`string` و `string | null` دو نوع متفاوت‌اند؛ دسترسی بدون بررسی خطاست (فصل ۷)
`noImplicitAny`	پارامتر بدون annotation بی‌صدا `any` می‌شود	کامپایلر می‌پرسد: «این چیست؟»
`strictFunctionTypes`	پارامترهای توابع با تسامح bivariant مقایسه می‌شوند	مقایسه درست contravariant (فصل ۲۴)

> **در عمل — strict برای پروژه جدید قابل مذاکره نیست**
>
> هزینه strict در پروژه نو تقریباً صفر است و سودش تا آخر عمر پروژه ادامه دارد. تنها جای بحث واقعی، مهاجرت codebase قدیمی است — آن‌جا هم راه درست، روشن کردن تدریجی گزینه‌هاست (فصل ۳۰)، نه چشم‌پوشی دائمی. تمام مثال‌های این کتاب با `strict: true` بررسی شده‌اند.

دو گزینه ارزشمند که حتی داخل `strict` نیستند:

- **`noUncheckedIndexedAccess`** — دسترسی ایندکسی `arr[i]` و `obj[key]` را به‌جای `T`، از نوع `T | undefined` می‌کند. واقعیت را می‌گوید: هیچ تضمینی نیست که ایندکس موجود باشد. هزینه‌اش بررسی‌های بیشتر است؛ سودش حذف یک دسته کامل باگ runtime (فصل ۲۷ با مثال).
- **`verbatimModuleSyntax`** — مرز import نوع و مقدار را شفاف می‌کند و برای ابزارهای تک‌فایلی (esbuild، type stripping در Node) ضروری است؛ importهای «فقط نوع» باید صریحاً `import type` باشند.

target و lib — دو چیز متفاوت

`target` مشخص می‌کند syntax خروجی چقدر قدیمی باشد (مثلاً آیا `?.` به کد معادل تبدیل شود). `lib` مشخص می‌کند **اعلان کدام APIها** در دسترس نوع‌هاست (مثلاً `Array.prototype.at` یا DOM). اگر `lib` را دستی ننویسید، از `target` مشتق می‌شود. خطای کلاسیک:

پیام کامپایلر

```
error TS2550: Property 'at' does not exist on type 'number[]'.
Do you need to change your target library? Try changing the
'lib' compiler option to 'es2022' or later.
```

معنایش ساده است: به APIای اشاره کرده‌اید که در `lib` فعلی اعلام نشده. راه‌حل: `lib` / `target` را بالا ببرید — به شرطی که runtime واقعی شما آن API را داشته باشد. **نوع‌ها runtime را تغییر نمی‌دهند**؛ `lib: es2023` روی Node قدیمی فقط دروغ را قشنگ‌تر می‌کند.

مدل ذهنی — tsconfig قرارداد سه‌طرفه است

پیکربندی شما قراردادی است بین **کد** («چه می‌نویسم»)، **کامپایلر** («چه چیزی را خطا بگیرم») و **محیط اجرا** («چه چیزی واقعاً وجود دارد»). هر وقت گزینه‌ای را تغییر می‌دهید بپرسید: کدام ضلع قرارداد را دارم عوض می‌کنم؟ ناسازگاری اضلاع، دیر یا زود خودش را در production نشان می‌دهد.

گزینه‌هایی که باید درباره‌شان شک کنید

- `skipLibCheck: true` — بررسی نوعِ فایل‌های `.d.ts` کتابخانه‌ها را خاموش می‌کند. عملاً استاندارد شده (سرعت و ناسازگاری‌های بین کتابخانه‌ها)، اما بدانید چه می‌خرید: اگر دو پکیج اعلان‌های متناقض بدهند، شما نمی‌فهمید.
- `esModuleInterop` — با `module: nodenext` به‌طور ضمنی فعال است؛ در پیکربندی‌های bundler خودتان روشنش کنید تا import پکیج‌های CommonJS طبیعی باشد.
- هر گزینه‌ای که فقط برای «ساکت کردن خطا» روشن می‌شود — مثل `strictNullChecks: false` — قرض فنی با بهره مرکب است.

در نسخه ۶، تعدادی گزینه قدیمی deprecated شدند و در نسخه ۷ حذف شده‌اند؛ اگر پروژه کهنه‌ای را ارث برده‌اید فصل ۳۳ نقشه گذار را می‌دهد. مرجع تصمیم‌گیری کامل گزینه‌ها نیز در پیوست C است.

زیر کاپوت — چرا این‌طور است؟

tsconfig را بهتر می‌فهمید اگر بدانید tsc با آن چه می‌سازد: یک **Program** — مجموعه بسته‌ای از فایل‌ها + گزینه‌ها که بررسی نوع درونش انجام می‌شود. سه گزینه‌ای که «جهانِ» این Program را تعریف می‌کنند، همان‌هایی‌اند که بیشترین سوءتفاهم را می‌سازند:

`lib` — جهان اعلان‌ها. نکته‌ای که کمتر گفته می‌شود: `toUpperCase` و `Promise` و `document` هیچ جایگاه ویژه‌ای در کامپایلر ندارند؛ همگی در فایل‌های `lib.*.d.ts` معمولیِ همراه TypeScript اعلان شده‌اند و lib تعیین می‌کند کدام‌ها بار شوند. وقتی خطای «این متد وجود ندارد» روی API جدید می‌گیرید، کامپایلر خراب نیست — جهانی که به او داده‌اید قدیمی است. و چون این‌ها اعلان معمولی‌اند، کتابخانه‌ها هم با همان سازوکار می‌توانند جهان را گسترش دهند (فصل ۲۵ و ۲۹).

`target` — **جهان syntax خروجی.** جدایی‌اش از lib عمدی است: می‌توانید برای runtime قدیمی emit کنید ولی با polyfill، APIهای جدید داشته باشید — دو محور مستقل یک تصمیم‌اند.

`moduleResolution` / `module` — **جهان اتصال فایل‌ها.** این دو می‌گویند «وقتی import دیدی، مثل چه کسی فکر کن»: مثل Node با قواعد سخت ESM، یا مثل bundler سخاوتمند. اگر این شبیه‌سازی با اجراکننده واقعی فرق کند، بررسی نوع دارد دنیایی را تأیید می‌کند که وجود خارجی ندارد — ریشه کل فصل ۲۸ همین جمله است.

و درباره خود `strict` : چتری با معنای «در حرکت» است — نسخه‌های آینده می‌توانند بررسی جدیدی زیرش بیاورند. عمدی است: تیم TypeScript می‌خواهد «بهترین پیش‌فرض» تک‌کلیدی بماند؛ بهایش این‌که ارتقای نسخه گاهی خطاهای «درستِ» جدید رو می‌کند.

تمرین

پیکربندی را نقد کن

این پیکربندی واقعی را از یک پروژه قدیمی تحویل گرفته‌اید. سه مشکل اصلی‌اش را نام ببرید:

legacy-tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "strict": false,
    "noImplicitAny": true,
    "skipLibCheck": true
  }
}
```

پاسخ: ۱. `strict: false` یعنی از جمله strictNullChecks خاموش است — بزرگ‌ترین منبع باگ گرفته نمی‌شود؛ noImplicitAny به‌تنهایی کافی نیست. ۲. `target: es5` برای محیط‌های امروزی تقریباً همیشه بی‌دلیل است: خروجی حجیم‌تر و کندتر، و lib قدیمی APIهای مدرن را نمی‌شناسد. ۳. `include` و `module` مشخص نیستند — رفتار resolution به پیش‌فرض‌های مبهم تکیه دارد؛ صریح بنویسید.

پرسش‌های مصاحبه این فصل

۱. برای پروژه جدید، کدام گزینه‌های tsconfig را غیرقابل مذاکره می‌دانید؟

`strict: true` ، `noUncheckedIndexedAccess` (خارج از چتر strict است و جدا روشن می‌شود)، `verbatimModuleSyntax` برای سازگاری با ابزارهای تک‌فایلی، و module/target ای که آینه runtime واقعی باشد. بقیه قابل بحث‌اند؛ این‌ها نه.

۲. «tsconfig باید آینه runtime باشد» یعنی چه؟ یک مثال شکست بدهید.

یعنی گزینه‌های ماژول/target باید همان چیزی را توصیف کنند که اجراکننده نهایی می‌فهمد. مثال: `moduleResolution:` `"bundler"` برای کدی که Node خام اجرایش می‌کند — import بدون پسوند محلی سبز است و در deploy با `ERR_MODULE_NOT_FOUND` می‌شکند.

استنتاج نوع و Contextual Typing

کامپایلر از کجا نوع را حدس می‌زند — و کی باید خودت annotation بنویسی

اهداف یادگیری این فصل

- پیش‌بینی کنی کامپایلر برای یک عبارت چه نوعی استنتاج می‌کند
- جهت جریان اطلاعات نوع (از مقدار یا از زمینه) را تشخیص دهی
- بدانی کجا annotation ارزش دارد و کجا نویز است

نوشتن annotation برای هر متغیر، نشانه تسلط بر TypeScript نیست — برعکس، نشانه بی‌اعتمادی به استنتاج است. کامپایلر در بیشتر موارد نوع را بهتر و دقیق‌تر از ما تشخیص می‌دهد. مهارت واقعی این است که بدانید **اطلاعات نوع از کجا می‌آید و به کدام سمت جریان دارد**. این فصل همان مهارت است.

دو جهت جریان اطلاعات نوع

هر استنتاجی در TypeScript یکی از این دو جهت را دارد:

جهت	نام رسمی	مثال
از مقدار به متغیر (پایین به بالا)	Type Inference	`let n = 3` ← نوع `number`
از زمینه به عبارت (بالا به پایین)	Contextual Typing	پارامتر callback در `map`

جهت اول را همه می‌شناسند: کامپایلر به مقدار سمت راست نگاه می‌کند و نوع متغیر را نتیجه می‌گیرد:

inference-basic.ts — type-checks ✓

```ts
let count = 42;              // number
const name = "Sara";         // "Sara"  ← literal، چون const (فصل ۵) است
let scores = [90, 85, 100];  // number[]
let mixed = [1, "a", true];  // (string | number | boolean)[]

let user = { id: 7, active: true };
// { id: number; active: boolean }
```

جهت دوم ظریف‌تر و مهم‌تر است. وقتی عبارتی در «جایگاهی با نوع مشخص» قرار می‌گیرد، نوع آن جایگاه به داخل عبارت نشت می‌کند:

contextual.ts

type-checks ✓

```ts
const names = ["Ali", "Sara", "Reza"];

// n از کجا می‌داند string از امضای Array.prototype.map است؟
// map<U>(callbackfn: (value: string, ...) => U): U[]
const upper = names.map(n => n.toUpperCase());  // n: string — بدون annotation

//  همین مکانیزم برای event handlerها:
window.addEventListener("click", e => {
  e.clientX; // e: MouseEvent — از امضای overload مربوط به "click"
});
```

مدل ذهنی — دو جهت، یک قاعده

از خودتان بپرسید: «نوع این عبارت را چه کسی تعیین می‌کند؟» اگر **مقدارش**، استنتاج معمولی است. اگر **جایی که قرار گرفته** (پارامتر تابع، annotation متغیر، نوع بازگشتی اعلام‌شده)، contextual typing است. تقریباً تمام رفتارهای «جادویی» TypeScript با همین سؤال قابل توضیح‌اند.

وقتی زمینه وجود ندارد

جدا کردن تابع از جایگاهش، زمینه را از بین می‌برد — یکی از رایج‌ترین سردرگمی‌های تازه‌کارها:

no-context.ts

× خطای عمدی

```ts
//  زمینه داریم — map داخل کار می‌کند،:
["1", "2"].map(s => s.length);

//  اما همان lambda را بیرون بکشیم، زمینه‌ای نیست:
const getLen = s => s.length;    // ×
```

پیام کامپایلر

```
no-context.ts:5:16 - error TS7006: Parameter 's' implicitly
has an 'any' type.
```

کامپایلر نمی‌داند `s` چیست، چون این تابع هنوز به هیچ جایگاهی وصل نشده است. دو راه‌حل دارید: یا پارامتر را annotate کنید (`(s: string) => s.length`)، یا نوع خود متغیر را بنویسید تا زمینه بسازد:

make-context.ts

type-checks ✓

```ts
const getLen: (s: string) => number = s => s.length; // s از annotation می‌آید
```

استنتاج نوع بازگشتی و Best Common Type

کامپایلر نوع بازگشتی را از تمام مسیرهای `return` جمع می‌زند و در آرایه‌ها «بهترین نوع مشترک» را می‌سازد:

return-infer.ts — type-checks ✓

```ts
function pickFirst(xs: number[], fallback: string) {
  if (xs.length > 0) return xs[0];  // number
  return fallback;                  // string
}
//  نوع بازگشتی استنتاج‌شده: string | number

class Animal { name = "" }
class Dog extends Animal { bark() {} }
class Cat extends Animal { meow() {} }

const pets = [new Dog(), new Cat()]; // (Dog | Cat)[] — نه Animal[]
```

نکته مهم: کامپایلر برای `pets` به دنبال والد مشترک **نمی‌گردد**؛ union اعضا را می‌سازد. اگر `Animal[]` می‌خواهید باید خودتان بگویید: `const pets: Animal[] = [...]` .

کجا annotation بنویسیم؟ قاعده تیمی

در عمل — قاعده‌ای که در code review قابل دفاع است

- **بنویس:** امضای توابع عمومی (پارامترها همیشه، خروجی در APIهای عمومی) — اینجا annotation «قرارداد» است نه نویز، و خطا را به محل درستش منتقل می‌کند.
- **بنویس:** جایی که می‌خواهی زمینه بسازی — متغیری که بعداً مقدار می‌گیرد، آرایه خالی، `useState` های union-دار.
- **ننویس:** متغیرهای محلی با مقدار اولیه روشن. `const n: number = 3` فقط نویز است و در refactor دروغ می‌گوید.

چرا annotation خروجی توابعِ عمومی مهم است؟ چون **محل گزارش خطا** را عوض می‌کند. بدون آن، اگر پیاده‌سازی را خراب کنید، خطا در «محل استفاده» ظاهر می‌شود — جایی دور از علت واقعی. با آن، خطا همان‌جا در بدنه تابع گزارش می‌شود.

زیر کاپوت — چرا این‌طور است؟

استنتاج را با یک سؤال محک بزنید: «کامپایلر از کجا شروع می‌کند؟» پاسخ: از **جایگاه‌های استنتاج**. در `const x = expr` نوع از برگ‌های عبارت به بالا محاسبه می‌شود (literal → عملگر → فراخوانی). در فراخوانی تابع جنریک، جایگاه‌های حاوی پارامتر نوع با نوع آرگومان‌ها «جفت» می‌شوند، از هر جفت کاندیدایی جمع می‌شود و بهترین انتخاب می‌شود. برای literal چندعضوی ناهمگون، الگوریتم «بهترین نوع مشترک» union می‌سازد. جادویی نیست — جمع‌آوری کاندیدا و انتخاب.

سؤال عمیق‌تر: چرا TypeScript مثل زبان‌های ML استنتاج «سراسری» نمی‌کند که امضای توابع را هم خودش بفهمد؟ سه دلیل مهندسی: **اول، محل خطا.** در استنتاج سراسری، ناسازگاری ممکن است صدها خط دورتر از علت گزارش شود —

همه‌چیز به همه‌چیز گره خورده. امضای صریح «مرز سرزنش» می‌سازد: خطا همان‌جا ظاهر می‌شود که قرارداد نقض شده. **دوم، کارایی.** استنتاج محلی cache پذیر و موازی‌پذیر است؛ سراسری یعنی هر تغییر کوچک، بازمحاسبه زنجیره‌ای. **سوم، تدریجی بودن.** سیستمی که با any و کد JS تایپ‌نشده همزیستی می‌کند، نمی‌تواند به استنتاج تمام‌عیار تکیه کند.

این سه دلیل، توصیه عملی فصل را «اثبات» می‌کنند: درون تابع به استنتاج اعتماد کنید (محلی و دقیق)، روی مرزهای عمومی annotation بنویسید (مرز سرزنش + قفل قرارداد). قاعده از معماری خود کامپایلر درمی‌آید، نه سلیقه.

و contextual typing؟ همان الگوریتم در جهت معکوس: جایگاهِ دارای «نوع مورد انتظار»، انتظارش را به درون عبارت نشت می‌دهد — پارامترهای callback از امضای هدف پر می‌شوند. دو جهتِ یک جریان. هر جا استنتاج «کار نکرد»، بپرسید: کدام جهت قطع شده؟

تمرین

پیش‌بینی کن، بعد اجرا کن

بدون اجرای کامپایلر، نوع استنتاج‌شده هر مورد را بنویسید؛ سپس با hover در ادیتور بررسی کنید:

quiz.ts — type-checks ✓

```ts
const a = [1, 2, null];
let b = { tags: [] };
const c = (flag: boolean) => flag ? "yes" : 0;
const d = Promise.resolve([true]);
```

پاسخ: `a: (number | null)[]` · `b: { tags: never[] }` — آرایه خالی بدون زمینه `never[]` می‌شود و این دقیقاً جایی است که annotation لازم است · `c: (flag: boolean) => "yes" | 0` · `d: Promise<boolean[]>` .

پرسش‌های مصاحبه این فصل

۱. با وجود استنتاج قوی، کجا annotation را اجباری می‌دانید؟

امضای توابع صادراتی و API عمومی: قرارداد قفل می‌شود، خطا به محل علت می‌رود نه محل مصرف، و برای declaration emit تک‌فایلی لازم است. به‌علاوه جاهایی که استنتاج ضعیف است: آرایه خالی (`never[]`)، و پارامترهای بدون زمینه.

۲. contextual typing چیست و چرا جهتش «برعکس» استنتاج معمولی است؟

استنتاج معمولی از مقدار به نوع می‌رود؛ contextual typing از جایگاهِ هدف به عبارت برمی‌گردد: پارامتر callback در `map` یا `addEventListener` نوعش را از امضای مقصد می‌گیرد. برای همین callback درون‌خط annotation نمی‌خواهد ولی همان تابع جدا شده، می‌خواهد.

نوع‌های اولیه، Literal Types و Widening

چرا "let x = "a نوع string می‌گیرد اما const x نوع "a"

اهداف یادگیری این فصل

- رفتار widening را در let/const و propertyها پیش‌بینی کنی
- از literal typeها برای مدل‌سازی مقادیر مجاز استفاده کنی
- تفاوت string با String و دام‌های wrapper typeها را بشناسی

چرا `const x = "on"` نوع `"on"` می‌گیرد اما `let x = "on"` نوع `string` ؟ پاسخ این سؤال — سازوکار **widening** — کلید فهم نیمی از رفتارهای «عجیب» TypeScript است، از خطاهای Redux action تا جادوی `as const` .

Literal Types — نوع‌هایی به اندازه یک مقدار

در TypeScript هر مقدارِ ثابت می‌تواند خودش یک نوع باشد: `"on"` ، `42` ، `true` . به‌تنهایی کم‌فایده‌اند، اما در union، ابزار اصلی مدل‌سازی «مقادیر مجاز» می‌شوند:

literals.ts — type-checks ✓

```ts
type Direction = "up" | "down" | "left" | "right";
type DiceRoll  = 1 | 2 | 3 | 4 | 5 | 6;

function move(dir: Direction, steps: DiceRoll) { /* ... */ }

move("up", 3);      // ✓
```

literals-err.ts — × خطای عمدی

```ts
type Direction = "up" | "down" | "left" | "right";
declare function move(dir: Direction): void;

move("Up"); //  × کامپایلر — حروف بزرگ typo را می‌گیرد
```

پیام کامپایلر

```
literals-err.ts:4:6 - error TS2345: Argument of type '"Up"' is not
assignable to parameter of type 'Direction'.
```

این همان جایی است که TypeScript از «رشته‌های جادویی» JavaScript انتقام می‌گیرد: مجموعه مقادیر مجاز، بخشی از قرارداد تابع می‌شود — با تکمیل خودکار رایگان در ادیتور.

Widening — کامپایلر محافظه‌کار می‌شود

وقتی مقدار literal به متغیر **قابل تغییر** نسبت داده می‌شود، کامپایلر نوع را «پهن» می‌کند، چون فرض می‌کند بعداً مقدار دیگری از همان جنس خواهد گرفت:

widening.ts — type-checks ✓

```ts
const a = "on";  //  نوع: "on"      ← const تغییر نمی‌کند، literal می‌ماند
let   b = "on";  //  نوع: string    ← widened

const obj = { mode: "on" };
//  نوع: { mode: string }  ← property حتی در قابل تغییر است، const!
```

خط آخر مهم‌ترین دام این فصل است: `const` فقط **خودِ binding** را ثابت می‌کند، نه propertyهای شیء را. برای همین `obj.mode` به `string` پهن می‌شود — و بعد این اتفاق می‌افتد:

widening-trap.ts — × خطای عمدی

```ts
type Mode = "on" | "off";
declare function setMode(m: Mode): void;

const config = { mode: "on" };   // mode: string
setMode(config.mode);            // ×
```

پیام کامپایلر

```
widening-trap.ts:5:9 - error TS2345: Argument of type 'string' is
not assignable to parameter of type 'Mode'.
```

سه راه‌حل، از موضعی تا ساختاری:

widening-fixes.ts — type-checks ✓

```ts
type Mode = "on" | "off";
declare function setMode(m: Mode): void;

//  ۱) annotation (فصل ۴) روی ساختار — زمینه می‌سازد
const c1: { mode: Mode } = { mode: "on" };

//  ۲) as const — کل literal می‌کند «منجمد» را
const c2 = { mode: "on" } as const;   // { readonly mode: "on" }

//  ۳) satisfies — بررسی بدون widening (فصل ۲۱، بهترین ابزار مدرن)
const c3 = { mode: "on" } satisfies { mode: Mode };

setMode(c1.mode); setMode(c2.mode); setMode(c3.mode); //  ✓ هر سه
```

مدل ذهنی — پهن‌شدن یعنی «احتمال تغییر»

کامپایلر از خودش می‌پرسد: «آیا این جایگاه ممکن است بعداً مقدار دیگری بگیرد؟» اگر بله (`let` ، property معمولی)، literal به نوع عمومی‌اش پهن می‌شود. اگر نه (`const` ، `readonly` ، `as const`)، literal می‌ماند. widening نه باگ است نه جادو — مدیریت ریسکِ تغییر است.

نوع‌های اولیه — چند نکته مهندسی

هفت نوع اولیه (`string` , `number` , `boolean` , `bigint` , `symbol` , `null` , `undefined`) را می‌شناسید. نکته‌هایی که در عمل اهمیت دارند:

- `string` **بنویسید، نه** `String` . نوع‌های wrapper (با حرف بزرگ) نوع آبجکت‌های بسته‌بندی‌اند؛ `String` تقریباً همیشه اشتباه است و باگ سازگاری می‌سازد.
- `number` **همان دقت IEEE 754 است.** نوع، شما را از `0.1 + 0.2 === 0.3` نجات نمی‌دهد؛ برای پول از integer سنت‌محور یا `bigint` استفاده کنید.
- `unique symbol` پایه الگوی branding در فصل ۲۳ است.

در عمل — literal union به‌جای enum

در TypeScript مدرن، برای بیشتر سناریوها `type Status = "active" | "blocked"` بر `enum` ترجیح دارد: خروجی runtime ندارد (erasure کامل)، با JSON و APIها مستقیم جور است و با ابزارهای تک‌فایلی (فصل ۲ و ۳۳) مشکل ندارد. اگر به مقادیر در runtime هم نیاز دارید، الگوی `as const` + استخراج نوع را به‌کار ببرید (فصل ۱۶ و ۲۱).

زیر کاپوت — چرا این‌طور است؟

برای فهم کامل widening باید مفهوم پنهانش را بشناسید: **تازگی (freshness)**. کامپایلر برای هر literal ابتدا دقیق‌ترین نوع ممکن را می‌سازد — `"on"` ، نه string. سرنوشت این نوعِ «تازه» در لحظه عبور از مرز تصمیم‌گیری تعیین می‌شود:

اگر جایگاه مقصد «نوع مورد انتظار» داشته باشد (annotation، پارامتر تایپ‌دار، satisfies)، literal با همان دقت کامل با انتظار مقایسه می‌شود — widening اصلاً رخ نمی‌دهد. برای همین `setMode("on")` مستقیم کار می‌کند ولی عبور همان مقدار از متغیر واسطِ بدون annotation می‌شکند: متغیر واسط «جایگاه بدون انتظار» است.

اگر جایگاه مقصد قابل‌تغییر باشد (let، property معمولی، عنصر آرایه)، نوع به والد عمومی پهن می‌شود. معیار دقیقاً «امکان نوشتن مجدد» است: کامپایلر باید نوعی بدهد که مقادیر آینده را هم پوشش دهد، و چون آینده را نمی‌داند، به جنس عمومی پناه می‌برد. طراحی جایگزین را تصور کنید — literal ماندن let: آن‌گاه `let s = "on"; s = "off"` خطا می‌شد و زبان غیرقابل استفاده. پس widening «از دست دادن اطلاعات» نیست؛ معامله آگاهانه دقت با آزادی تغییر است.

حالا ابزارهای فصل جای واقعی‌شان را پیدا می‌کنند: `as const` یعنی «freshness را برای همیشه قفل کن» (readonly + literal، بازگشتی روی کل ساختار)؛ annotation یعنی «جایگاه را صاحب انتظار کن»؛ satisfies یعنی «انتظار را تحمیل کن ولی نوع تازه را نگه دار». سه اهرم روی یک سازوکار واحد — نه سه ترفند جدا. با این مدل، دیگر لازم نیست «کی literal می‌ماند» را حفظ کنید؛ هر مورد را می‌توانید استدلال کنید.

تمرین

نوع هر عبارت را پیش‌بینی کن

quiz5.ts — type-checks ✓

```ts
const a = 404;
let   b = 404;
const user = { role: "admin", verified: true };
const point = [10, 20] as const;
```

پاسخ: `a: 404` · `b: number` · `user: { role: string; verified: boolean }` — هر دو property پهن شده‌اند · `point: readonly [10, 20]` — as const آرایه را به tuple فقط‌خواندنی با literalها تبدیل می‌کند.

پرسش‌های مصاحبه این فصل

۱. چرا propertyهای یک `const` شیئی، literal نمی‌مانند؟ سه راه حفظشان؟

چون خود property قابل‌تغییر است و widening اعمال می‌شود — const فقط binding را قفل می‌کند نه محتوا را. راه‌ها: annotation صریح به نوع literal دار، `as const` برای freeze کامل، یا `satisfies` وقتی بررسی قرارداد هم می‌خواهید.

۲. به‌جای `status: string` چه پیشنهاد می‌دهید و چه چیزی می‌خرید؟

union of literals: `"idle" | "loading" | "done"`. می‌خرید: جلوگیری از typo در کامپایل، autocomplete، و امکان exhaustive check. رشته باز یعنی قرارداد «هر چیزی» — مصرف‌کننده هیچ کمکی نمی‌گیرد.

نوع‌های شیئی و Structural Typing

سازگاری بر اساس شکل، نه نام — و استثنای Excess Property Check

اهداف یادگیری این فصل

- قاعده سازگاری ساختاری را روی مثال واقعی اجرا کنی
- بدانی چرا object literal تازه سخت‌گیرانه‌تر بررسی می‌شود
- پیامدهای structural بودن را در طراحی API بشناسی

در بسیاری از زبان‌ها (C، Java#)، سازگاری نوع‌ها **اسمی** است: `A` به `B` قابل انتساب است اگر رسماً از آن ارث برده باشد. TypeScript راه دیگری رفته: سازگاری **ساختاری**. این تصمیم، هم دلیل انعطاف فوق‌العاده TypeScript است و هم منشأ چند رفتار غیرمنتظره که مهندس جدی باید دقیق بشناسد.

قاعده طلایی: شکل مهم است، نه نام

structural.ts — type-checks ✓

```ts
interface Point { x: number; y: number }

class MousePosition {
  constructor(public x: number, public y: number) {}
}

function distance(p: Point): number {
  return Math.hypot(p.x, p.y);
}

distance(new MousePosition(3, 4));      //  ✓ هیچ implements لازم نیست
distance({ x: 1, y: 2 });               // ✓ object literal
distance({ x: 1, y: 2, z: 3 } as { x: number; y: number; z: number });//  ✓ عضو اضافه مانعی نیست
```

`MousePosition` هرگز نگفته «من Point هستم» — اما شکلش می‌خواند، پس هست. قاعده دقیق:

مدل ذهنی — سازگاری یعنی «حداقل همین‌قدر»

`S` به `T` قابل انتساب است اگر `S` **حداقل** تمام اعضای `T` را با نوع‌های سازگار داشته باشد. عضو اضافه مشکلی نیست — نوع‌ها «مجموعه مقادیر»اند و هر چه عضو بیشتر، مجموعه کوچک‌تر و خاص‌تر. نوع خاص‌تر همیشه جای عمومی‌تر می‌نشیند، نه برعکس.

استثنای مهم: Excess Property Check

اگر عضو اضافه مشکلی نیست، چرا این خطا می‌گیرد؟

excess.ts — خطای عمدی

```ts
interface Options { title: string; darkMode?: boolean }
declare function init(o: Options): void;

init({ title: "App", darkmode: true }); //  ✗ حرف m کوچک!
```

پیام کامپایلر

```
excess.ts:4:21 - error TS2353: Object literal may only specify known
properties, but 'darkmode' does not exist in type 'Options'.
Did you mean to write 'darkMode'?
```

توضیح: وقتی یک **object literal تازه** مستقیماً به جایگاهی با نوع مشخص داده می‌شود، کامپایلر بررسی سخت‌گیرانه‌تری اجرا می‌کند — چون عضو اضافه در literal تازه هیچ مصرف‌کننده دیگری ندارد و تقریباً همیشه یعنی typo. همین مقدار، اگر اول در متغیری ذخیره شود، از بررسی عبور می‌کند:

excess-indirect.ts — type-checks ✓

```ts
interface Options { title: string; darkMode?: boolean }
declare function init(o: Options): void;

const opts = { title: "App", darkmode: true }; //  نوع خودش را دارد
init(opts); //  ✓ سازگاری ساختاری عادی — عبور می‌کند
```

این تفاوت را جدی بگیرید: excess property check یک **سرویس خطایابی روی literalها** است، نه بخشی از قاعده سازگاری. اگر به آن به چشم «جلوگیری از property اضافه» نگاه کنید، معماری‌تان را روی تضمینی بنا کرده‌اید که وجود ندارد.

پیامدهای مهندسی structural بودن

۱. دو نوع هم‌شکل، از نظر کامپایلر یکی هستند:

same-shape.ts — type-checks ✓

```ts
type UserId    = string;
type ProductId = string;

declare function getUser(id: UserId): void;
const pid: ProductId = "p_123";
getUser(pid);  //  ✓ و این شاید باگ باشد — کامپایل می‌شود!
```

alias فقط نام مستعار است. اگر می‌خواهید `UserId` و `ProductId` واقعاً غیرقابل تعویض باشند، به نوع‌گذاری اسمی شبیه‌سازی‌شده نیاز دارید — الگوی brand در فصل ۲۳.

۲. API عمومی شما همان «شکل» است: هر شیئی که شکل interface شما را داشته باشد، مشتری قانونی API شماست. برای کتابخانه‌ها این یعنی تست‌پذیری عالی (mock بدون کلاس واقعی)، اما یعنی اضافه کردن عضو جدید به interface، یک تغییر شکننده است (فصل ۴۰).

۳. مقایسه بازگشتی و lazy است: سازگاری اشیاء تودرتو عضوبه‌عضو و تا عمق لازم بررسی می‌شود؛ برای نوع‌های بازگشتی، کامپایلر با فرض سازگاری جلو می‌رود تا به تناقض برسد — برای همین نوع‌های بازگشتی اصلاً کار می‌کنند (فصل ۲۶).

> **هشدار — کلاس‌های با عضو private استثنا هستند**
>
> اگر کلاسی عضو `private` یا `protected` داشته باشد، سازگاری آن عملاً اسمی می‌شود: فقط همان کلاس یا زیرکلاس‌هایش سازگارند، حتی اگر شیء دیگری دقیقاً هم‌شکل باشد. این ترفندی رسمی برای ساختن نوع‌های «واقعاً خصوصی» است — و گاهی منبع خطاهای گیج‌کننده بین دو نسخه از یک کتابخانه در node_modules.

زیر کاپوت — چرا این‌طور است؟

چرا excess property check فقط روی literal «تازه» اجرا می‌شود و نه همیشه؟ این سؤال ما را به قلب طراحی ساختاری می‌برد: **width subtyping**. در سیستم ساختاری، شیء با اعضای بیشتر، زیرنوعِ شیء با اعضای کمتر است — `{ name, age }` هر جا `{ name }` خواسته شود پذیرفتنی است. این قاعده اختیاری نیست؛ ستون فقرات ترکیب‌پذیری است: بدون آن هیچ تابعی نمی‌توانست شیء «غنی‌تر» از امضایش بپذیرد و هر interface باید فهرست بسته کائنات می‌بود.

اما width subtyping یک عارضه دارد: typo در نام property خطا نیست — «عضو اضافه» است! `{ naem: "Ali" }` از دید ساختاری فقط شیئی است با یک عضو اضافه و یک عضو کم. تیم TypeScript این‌جا راه‌حل جراحی انتخاب کرد، نه عمومی: فقط وقتی literal «همان لحظه، همان‌جا» ساخته می‌شود، عضو اضافه تقریباً قطعاً اشتباه انسانی است — نویسنده داشت برای همین جایگاه می‌نوشت و هیچ مصرف‌کننده دیگری برای آن عضو وجود ندارد. اما مقداری که از متغیر می‌آید «تاریخ» دارد: شاید جای دیگری از اعضای اضافه‌اش استفاده شود؛ سخت‌گیری بر آن یعنی شکستن width subtyping و کل ترکیب‌پذیری.

پس excess check «استثنای آگاهانه بر قاعده» است — و حالا می‌فهمید چرا با متغیر واسط دور زده می‌شود: این دور زدن باگ نیست؛ خودِ قاعده ساختاری است که دوباره جاری شده. نتیجه مهندسی: به آن به چشم «typo یاب» نگاه کنید، نه «مهر exact بودن». اگر واقعاً exact لازم دارید (DTO سخت‌گیر، پیکربندی حساس)، آن تضمین را جای دیگری بگیرید: validation مرزی فصل ۳۴، یا تکنیک‌های نوعی‌ای که عضو ناشناس را never می‌کنند — با همه هزینه‌هایشان.

تمرین

قبول یا رد؟

بدون کامپایلر پیش‌بینی کنید کدام خطوط خطا می‌گیرند:

quiz6.ts

× خطای عمدی

```ts
interface Named { name: string }

const a = { name: "Ali", age: 30 };
const b: Named = a;                          //  ۱
const c: Named = { name: "Ali", age: 30 };   //  ۲
const d: Named = { name: undefined! };       //  ۳
```

پاسخ: ۱. ✓ — سازگاری ساختاری عادی؛ عضو اضافه از مسیر غیرمستقیم مانعی ندارد. ۲. ✗ — literal تازه است؛ excess property check اعتراض می‌کند (TS2353). ۳. ✓ کامپایل می‌شود — `undefined!` با non-null assertion به `never` ... دقیق‌تر: `!` فقط null/undefined را از نوع حذف می‌کند و اینجا ادعای دروغ است؛ کامپایلر قبول می‌کند و باگ به runtime می‌رود. درس: assertion یعنی مسئولیت با شما.

پرسش‌های مصاحبه این فصل

۱. دو interface هم‌شکل از دو ماژول متفاوت — قابل تعویض‌اند؟ اگر نخواهیم چه؟

بله — سازگاری ساختاری فقط شکل را می‌بیند، نه نام و محل تعریف را. اگر تفکیک لازم است (مثلاً `UserId` و `OrderId` روی string)، باید اسمی‌سازی دستی کنید: branded type با مهر فقط-نوعی (فصل ۲۳).

۲. کی `as` قابل دفاع است؟

جاهای معدودِ مستند: دروازه brand بعد از validation واقعی، بعضی مرزهای DOM/`Object.keys` که exactness ندارند، و cast متمرکزِ خروجی mapped که حلقه runtime نمی‌تواند اثباتش کند — همیشه با کامنتِ «چرا امن است». هر as دیگر در review باید سؤال‌برانگیز باشد.

null و undefined — پایان «خطای میلیارد دلاری»

strictNullChecks، اپراتورهای ?. و ?? و الگوهای درست مقدار غایب

اهداف یادگیری این فصل

- با strictNullChecks مقدارهای غایب را در سطح نوع مدل کنی
- تفاوت معنایی null و undefined و optional را توضیح دهی
- از non-null assertion فقط در جای قابل دفاع استفاده کنی

تونی هور، مخترع `null`، آن را «خطای میلیارد دلاری» نامید. سهم JavaScript از این خطا دو برابر است: هم `null` دارد هم `undefined`. مهم‌ترین اتفاق TypeScript برای کیفیت کد — مهم‌تر از جنریک‌ها و همه ابزارهای پیشرفته — `strictNullChecks` است: غیبتِ مقدار را از «سورپرایز runtime» به «عضو رسمی سیستم نوع» تبدیل می‌کند.

قبل و بعد از strictNullChecks

بدون این گزینه، `null` و `undefined` به **هر نوعی** قابل انتساب‌اند — یعنی نوع‌ها درباره رایج‌ترین باگ JavaScript هیچ حرفی ندارند. با آن:

strict-null.ts — خطای عمدی

```ts
function len(s: string | null): number {
  return s.length;    // × s ممکن است null باشد
}
```

پیام کامپایلر

```
strict-null.ts:2:10 - error TS18047: 's' is possibly 'null'.
```

کامپایلر شما را مجبور می‌کند قبل از استفاده، غیبت را **رد کنید** — و این دقیقاً همان‌جایی است که narrowing (فصل ۱۰) وارد می‌شود:

null-narrowing.ts — type-checks ✓

```ts
function len(s: string | null): number {
  if (s === null) return 0;
  return s.length;   //  ✓ اینجا s: string — کامپایلر if را فهمیده
}
```

جعبه‌ابزار کار با مقدار غایب

toolbox.ts

type-checks ✓

```ts
interface User {
  name: string;
  nickname?: string;          // optional: ممکن است اصلاً نباشد
  avatarUrl: string | null;   // صریح: هست، ولی شاید مقدار نداشته باشد
}

function describe(u: User): string {
  // ?. — دسترسی امن: اگر وسط زنجیره nullish بود، کل عبارت undefined
  const firstChar = u.nickname?.charAt(0);

  // ?? — پیش‌فرض فقط برای null/undefined (نه "" و 0!)
  const display = u.nickname ?? u.name;

  // ترکیب رایج:
  return `${display} (${u.avatarUrl ?? "بدون تصویر"}) ${firstChar ?? ""}`;
}
```

هشدار — || جای ?? نیست

`value || fallback` برای هر مقدار falsy (رشته خالی، صفر، false) هم fallback را برمی‌گرداند. برای «پیش‌فرض فقط وقتی مقدار نیست» همیشه `??` . باگ کلاسیک: `port || 3000` که پورت صفر را بی‌صدا به ۳۰۰۰ تبدیل می‌کند.

null یا undefined؟ یک قرارداد تیمی

زبان هر دو را دارد و حذف هیچ‌کدام ممکن نیست؛ مهم، قرارداد سازگار است. قرارداد رایج و قابل دفاع:

- `undefined` یعنی «تنظیم نشده / وجود ندارد» — پیش‌فرض طبیعی زبان (پارامتر و property اختیاری).
- `null` یعنی «صریحاً خالی» — تصمیمی آگاهانه، مثلاً «کاربر آواتارش را پاک کرد»؛ و در JSON فقط `null` وجود دارد.
- در API عمومی، نوع را صریح بنویسید (`string | null`) و در داخل برنامه تا می‌توانید از optional (`?`) استفاده کنید.

Non-null assertion — چاقوی بدون دسته

عملگر `!` به کامپایلر می‌گوید «مطمئنم null نیست» — بدون هیچ بررسی runtime:

non-null.ts

type-checks ✓

```ts
const el = document.getElementById("app");  // HTMLElement | null

el!.focus();          //  اگر null باشد، crash در runtime؛ کامپایل می‌شود

//  جایگزین‌های بهتر، به ترتیب ترجیح:
if (el) el.focus();                       //  ۱) narrowing واقعی
el?.focus();                              //  ۲) دسترسی امن
if (!el) throw new Error("#app missing");//  ۳) شکست زودهنگام و صریح
el.focus();                               //     اینجا el: HTMLElement
```

گزینه ۳ را دست‌کم نگیرید: اگر نبودن عنصر واقعاً «غیرممکن» است، تبدیلش به خطای صریح با پیام خوب، از `!` بی‌صدا شریف‌تر است — وقتی فرض اشتباه از آب درآمد، می‌فهمید کجا و چرا. نسخه ساختاریافته همین ایده، assertion functionها هستند (فصل ۲۳).

> **مدل ذهنی — غیبت را به مرز هل بده**
>
> هر `| null` که در عمق برنامه جریان دارد، در هر نقطه مصرف یک «if» طلب می‌کند. طراحی پخته، غیبت را **در مرز** حل می‌کند: ورودی را همان اول بررسی کن، مقدار پیش‌فرض بده یا خطا بینداز — و به بقیه برنامه نوعِ تمیزِ بدون null تحویل بده. تعداد `?.` های یک فایل، سنجه خوبی برای «نشت غیبت به عمق» است.

زیر کاپوت — چرا این‌طور است؟

عبارت «خطای میلیارد دلاری» از خود مخترع null است — Tony Hoare، ۱۹۶۵، در ALGOL — که دهه‌ها بعد اعتراف کرد صرفاً «چون پیاده‌سازی‌اش آسان بود» اضافه‌اش کرده. JavaScript این خطا را دو بار مرتکب شد: `undefined` (جایگاه مقداردهی‌نشده) و `null` (غیبت عمدی) — تمایزی ظریف ولی واقعی: undefined یعنی «کسی چیزی نگذاشته»، null یعنی «کسی آگاهانه گفت هیچ».

بخش عمیق ماجرا این‌جاست: `strictNullChecks` فقط «یک بررسی بیشتر» نیست؛ **جایگاه null و undefined در کل مدل نوع را عوض می‌کند**. بدون آن، این دو مقدار عضو خاموشِ «همه» نوع‌ها هستند — string یعنی «string یا null یا undefined» و کامپایلر این را پنهان می‌کند؛ دقیقاً همان طراحی‌ای که Hoare بابتش عذرخواهی کرد. با آن، null و undefined نوع‌های مستقل و ناسازگار می‌شوند و «امکان غیبت» باید صریح در امضا بیاید: `User | null`. تفاوت فلسفی است: غیبت از «واقعیت پنهانِ همه‌جا» به «بخش قابل‌مشاهده قرارداد» تبدیل می‌شود.

چرا این مدل کار می‌کند؟ چون تحلیل جریان کنترل (فصل ۱۰) پشتش ایستاده: هر if و return و `?.` به کامپایلر «اثبات» می‌دهد که در این شاخه غیبت منتفی است — و نوع همان‌جا باریک می‌شود. Kotlin و Swift و Rust همین معماری را دارند (nullable types / Option)؛ TypeScript آن را بدون تغییر حتی یک بایت از runtime جاوااسکریپت پیاده کرد — فقط با لایه ایستا. این شاید زیباترین نمایش قدرت «نوع بدون runtime» در کل زبان باشد: یک کلاس کامل از باگ‌های پنجاه‌ساله، با صفر هزینه اجرایی، خطای کامپایل شد.

تمرین

بازنویسی بدون !

این تابع را طوری بازنویسی کنید که بدون هیچ `!` کامپایل شود و رفتار معقول داشته باشد:

quiz7.ts — type-checks ✓

```ts
function getDomain(email: string | undefined): string {
  return email!.split("@")[1]!.toLowerCase();
}
```

یک پاسخ خوب:

quiz7-solution.ts — type-checks ✓

```ts
function getDomain(email: string | undefined): string | undefined {
  const domain = email?.split("@")[1];
  return domain?.toLowerCase();
}
```

توجه کنید ایندکس `[1]` با `noUncheckedIndexedAccess` خودش `string | undefined` است (فصل ۲۷) — پس `?.` دوم لازم و صادقانه است. اگر قرارداد شما «ایمیل همیشه معتبر است» باشد، راه درست اعتبارسنجی در مرز است، نه `!` در عمق.

پرسش‌های مصاحبه این فصل

۱. برای رفع «possibly undefined» چه ترتیبی را درست می‌دانید؟

نردبان: اول narrowing واقعی (early return/if)، بعد `?.` وقتی ادامه‌ندادن منطقی است، بعد `??` برای پیش‌فرض معنادار، و throw صریح با پیام وقتی غیبت یعنی باگ. `!` آخرین پله است و فقط با اثباتی که در کامنت بیاید.

۲. `config.port || 3000` چه باگی دارد؟

پورت `0` (یعنی «پورت تصادفی بده») falsy است و بلعیده می‌شود؛ همین‌طور رشته خالی در تنظیمات متنی. برای پیش‌فرض‌گیری فقط `??` درست است که تنها null/undefined را جایگزین می‌کند.

void و any، unknown، never

چهار نوع مرزی که کیفیت یک codebase را لو می‌دهند

اهداف یادگیری این فصل

- بدانی any دقیقاً چه چیزی را خاموش می‌کند و چگونه منتشر می‌شود
- unknown را به‌عنوان جایگزین امن any به‌کار ببری
- نقش never در exhaustiveness و نوع‌های غیرممکن را بفهمی

چهار نوعِ این فصل، «مرزهای» سیستم نوع‌اند: بالاترین نقطه، پایین‌ترین نقطه، دریچه فرار و علامت سکوت. کیفیت یک codebase را اغلب می‌توان از نحوه برخوردش با همین چهار نوع حدس زد — مخصوصاً اولی.

any — خاموش کردن کامپایلر

`any` یک نوع نیست؛ **قرارداد عدم بررسی** است. هر عملیاتی روی آن مجاز است و بدتر: به هر نوعی قابل انتساب است و هر نوعی به آن.

any-virus.ts — type-checks ✓

```ts
function parsePayload(json: string) {
  const data = JSON.parse(json);    // data: any
  const id = data.user.id;          // id: any — هیچ بررسی‌ای نشد
  const name: string = data.nmae;   // typo! ولی کامپایلر ساکت است
  return { id, name };              //   آلودگی به خروجی منتقل شد
}
```

به این پدیده «ویروسی بودن any» می‌گویند: از نقطه ورود (اینجا `JSON.parse`)، در تمام مسیر داده منتشر می‌شود و هر جا برسد، بررسی را خاموش می‌کند. `noImplicitAny` فقط anyهای ضمنی را می‌گیرد؛ anyهای صریح و anyهای برگشتی از APIها (مثل `JSON.parse` و `res.json()`) با شما می‌مانند.

unknown — همان انعطاف، با حفظ بررسی

`unknown` هم مثل `any` هر مقداری را می‌پذیرد — اما برعکسِ آن، **هیچ عملیاتی** روی آن مجاز نیست تا وقتی نوع را اثبات کنید:

unknown-strict.ts — × خطای عمدی

```ts
function handle(value: unknown) {
  value.toUpperCase();   //  × اول ثابت کن رشته است
}
```

پیام کامپایلر

```
unknown-strict.ts:2:3 - error TS18046: 'value' is of type 'unknown'.
```

unknown-proof.ts — type-checks ✓

```ts
function handle(value: unknown): string {
  if (typeof value === "string") return value.toUpperCase(); // ✓ narrowing
  if (value instanceof Error)    return value.message;       // ✓
  return JSON.stringify(value);
}
```

قاعده مهندسی: **هر جا any می‌نویسید، بپرسید چرا unknown نه.** تقریباً تنها پاسخ‌های قابل قبول: کد موقت مهاجرت (فصل ۳۰)، و نقاط خاصی از نوع‌های جنریک که خود کامپایلر `any` را به‌عنوان جوکر لازم دارد (مثل `(...args: any[])` یا `any ⇒` در constraintها).

never — نوعِ «هرگز»

`never` مجموعه خالی است: هیچ مقداری از نوع `never` وجود ندارد. دو کاربرد اصلی:

۱. خروجی توابعی که هرگز برنمی‌گردند:

never-return.ts — type-checks ✓

```ts
function fail(msg: string): never { throw new Error(msg); }
function loop(): never { while (true) {} }

function readPort(env: Record<string, string | undefined>): number {
  const raw = env.PORT ?? fail("PORT is required");
  return Number(raw);   // raw: string — چون شاخه fail هرگز برنمی‌گردد
}
```

دقت کنید `fail` چطور در narrowing شرکت کرد: چون نوع بازگشتی‌اش `never` است، کامپایلر می‌داند بعد از `??` فقط شاخه `string` زنده می‌ماند.

۲. بررسی exhaustive — مهم‌ترین کاربرد: وقتی همه اعضای یک union را پوشش داده باشید، باقیمانده `never` است. این حقیقت را به یک «آلارم refactoring» تبدیل می‌کنیم:

never-exhaustive.ts

type-checks ✓

```ts
type Shape =
  | { kind: "circle"; r: number }
  | { kind: "square"; side: number };

function area(s: Shape): number {
  switch (s.kind) {
    case "circle": return Math.PI * s.r ** 2;
    case "square": return s.side ** 2;
    default: {
      const unreachable: never = s;  //  ✓ الان s واقعاً never است
      throw new Error(`Unhandled shape: ${JSON.stringify(unreachable)}`);
    }
  }
}
```

فردا که عضو `{ kind: "triangle"; ... }` به `Shape` اضافه شود، خط `const unreachable: never = s` خطا می‌گیرد — کامپایلر تک‌تک switchهای جامانده را برایتان پیدا می‌کند. فصل ۲۲ این الگو را کامل می‌کند.

void — «خروجی مهم نیست»

`void` یعنی «به مقدار بازگشتی تکیه نکن». ظرافت مهمش در callbackهاست: تابعی که چیزی برمی‌گرداند، در جایگاه `() ⇒ void` پذیرفته می‌شود:

void-callback.ts

type-checks ✓

```ts
const numbers = [1, 2, 3];
const callbacks: Array<() ⇒ void> = [];

// push خروجی number دارد؛ اما در جایگاه void پذیرفته می‌شود:
callbacks.push(() ⇒ numbers.push(4));  //  ✓ عمداً مجاز است
```

این تسامح عمدی است و درست: معنای `() ⇒ void` این است که صداکننده از خروجی استفاده نمی‌کند، نه این‌که callback حق ندارد خروجی داشته باشد. اگر غیر از این بود، `forEach(x ⇒ arr.push(x))` خطا می‌گرفت.

مدل ذهنی — سلسله‌مراتب مجموعه‌ها

به نوع‌ها به چشم مجموعه نگاه کنید: `unknown` مجموعه **همه** مقادیر است (بالای سلسله‌مراتب)، `never` مجموعه **خالی** (پایین). هر نوعی به unknown قابل انتساب است؛ never به هر نوعی. و `any`؟ اصلاً در این سلسله‌مراتب نیست — خارج از قواعد بازی می‌ایستد و هر دو جهت را می‌شکند. برای همین «نوع» نامیدنش سخاوت است.

در عمل — سیاست تیمی برای any

تیم‌های جدی سه خط دفاعی دارند: `noImplicitAny` در tsconfig، قانون `no-explicit-any` در ESLint (با استثنای آگاهانه و کامنت‌دار)، و بازبینی مرزهای ورود داده — چون `JSON.parse` و `fetch` منابع اصلی نشت any هستند. در فصل ۳۴ مرزها را با Zod می‌بندیم تا any اصلاً وارد نشود.

زیر کاپوت — چرا این‌طور است؟

چهار نوع این فصل را یک نظم ریاضی ساده کنار هم می‌نشاند: **نوع یعنی مجموعه مقادیر**. با این عینک دوباره نگاه کنید:

`unknown` مجموعه «همه مقادیر» است — بالاترین نقطه سلسله‌مراتب (top type). هر نوعی زیرمجموعه آن است، پس هر مقداری به آن قابل انتساب است؛ اما چون درباره اعضایش هیچ نمی‌دانید، هیچ عملیاتی مجاز نیست — تا با narrowing «دانش» اضافه کنید. سالم و منطقی.

`never` مجموعه تهی است — پایین‌ترین نقطه (bottom type). هیچ مقداری عضوش نیست (هیچ‌چیز به آن قابل انتساب نیست)، اما خودش زیرمجموعه همه است (به همه‌چیز قابل انتساب است — تابعی که throw می‌کند جای هر خروجی می‌نشیند). حالا exhaustiveness هم «قضیه» می‌شود نه ترفند: از union همه اعضا را کم کنید، به مجموعه تهی می‌رسید — باقی ماندن چیزی جز never یعنی عضوی جا مانده، و این دقیقاً همان است که انتسابِ به never گزارش می‌کند.

و `any`؟ در این نظم ریاضی **جایی ندارد** — و مشکلش دقیقاً همین است. any هم‌زمان مثل top رفتار می‌کند (همه‌چیز به آن قابل انتساب) و مثل bottom (به همه‌چیز قابل انتساب) — چنین مجموعه‌ای وجود ندارد. any نوع نیست؛ دستور «بررسی را برای این مقدار خاموش کن» است که لباس نوع پوشیده. ویروسی بودنش هم از همین است: هر عبارتی که any در آن دست داشته باشد خودش any می‌شود و خاموشی منتشر می‌شود.

این عینک مجموعه‌ای را نگه دارید — فصل ۹ union/intersection دقیقاً اجتماع و اشتراک از آب درمی‌آیند، و فصل ۱۷ حتی توزیع conditional ها هم با همین منطق بدیهی می‌شود.

تمرین

جای خالی را با یکی از چهار نوع پر کن

۱. خروجی تابعی که همیشه `process.exit(1)` صدا می‌زند: _____

۲. پارامتر تابعی که هر JSON decode شده‌ای را لاگ می‌کند: _____

۳. خروجی event handler که کسی به مقدارش کار ندارد: _____

۴. نوع عبارت `x` در `default` یک switch کامل روی union: _____

پاسخ: ۱. `never` — هرگز برنمی‌گردد. ۲. `unknown` — می‌پذیرد ولی قبل از استفاده اثبات می‌خواهد. ۳. `void` . ۴. `never` — همه اعضا حذف شده‌اند و همین پایه بررسی exhaustive است.

پرسش‌های مصاحبه این فصل

۱. در بلوک catch چه نوعی دارید و با آن چه می‌کنید؟

زیر strict، خطای گرفته‌شده `unknown` است — چون در JS هر چیزی قابل throw است. قبل از استفاده باید اثبات کنید: `instanceof Error` برای پیام/stack، یا بررسی‌های خاص‌تر. تایپ کردنش به `any` یا `as Error` همان دروغ همیشگی است.

۲. فرق عملی any و unknown را در یک جمله برای تیم چه می‌گویید؟

هر دو «هر چیزی» می‌پذیرند؛ اما any اجازه «هر کاری» هم می‌دهد و ویروسی پخش می‌شود، در حالی که unknown قبل از هر استفاده، اثبات (narrowing) می‌خواهد — یعنی ایمنی در نقطه مصرف حفظ است.

Union و Intersection

جبرِ نوع‌ها: «یا»، «و»، و رفتار دسترسی به اعضا

اهداف یادگیری این فصل

- بدانی چرا روی union فقط اعضای مشترک در دسترس‌اند
- intersection را برای ترکیب قراردادها به‌کار ببری
- دام‌های intersection (مثل never شدن) را تشخیص دهی

اگر نوع‌ها مجموعه مقادیرند، باید بتوان با آن‌ها جبر انجام داد: اجتماع و اشتراک. `|` و `&` دقیقاً همین‌اند — و فهم «مجموعه‌ای» آن‌ها، دو ابهام رایج را یک‌جا حل می‌کند: چرا روی union فقط اعضای مشترک در دسترس‌اند، و چرا intersection دو primitive می‌شود `never`.

Union — این یا آن

union-basics.ts — type-checks ✓

```ts
type Id = number | string;

function normalizeId(id: Id): string {
  // روی union فقط عملیات مشترک هر دو عضو مجاز است:
  return id.toString();  // ✓ هم number دارد هم string
}
```

union-error.ts — × خطای عمدی

```ts
function shout(id: number | string) {
  return id.toUpperCase();   // × این متد را ندارد number
}
```

پیام کامپایلر

```
union-error.ts:2:13 - error TS2339: Property 'toUpperCase' does not
exist on type 'string | number'.
  Property 'toUpperCase' does not exist on type 'number'.
```

چرا این محدودیت؟ چون مقدارِ واقعی در runtime **فقط یکی** از اعضاست و کامپایلر نمی‌داند کدام. تا وقتی با narrowing (فصل ۱۰) مشخص نکنید، فقط به وجه مشترک اجازه می‌دهد — این سخت‌گیری نیست، صداقت است.

مدل ذهنی — union بزرگ‌تر است، نه کوچک‌تر

شهود اشتباه رایج: «`A | B` یعنی چیزی بین A و B». درست: **اجتماع** مجموعه‌هاست — مقادیر بیشتری می‌پذیرد، پس درباره هر مقدارش کمتر می‌دانیم و عملیات کمتری مجاز است. قرینه‌اش: `A & B` اشتراک است — مقادیر کمتر، دانش بیشتر، عملیات بیشتر. «بزرگیِ مجموعه» و «قدرت عملیات» همیشه عکس هم‌اند.

Intersection — هم این، هم آن

کاربرد اصلی `&` : ترکیب قراردادهای شیئی.

intersection.ts — type-checks ✓

```ts
type Timestamps = { createdAt: Date; updatedAt: Date };
type SoftDelete = { deletedAt: Date | null };

type UserRow = { id: number; email: string } & Timestamps & SoftDelete;

const row: UserRow = {
  id: 1, email: "a@b.dev",
  createdAt: new Date(), updatedAt: new Date(), deletedAt: null,
}; //  ✓ باید همه اعضای هر سه را داشته باشد
```

مقدارِ از نوع `A & B` باید هم‌زمان عضو هر دو مجموعه باشد. برای اشیاء یعنی «همه propertyهای هر دو». اما برای primitiveها:

intersection-never.ts — type-checks ✓

```ts
type Impossible = string & number;    // never — هیچ مقداری هم‌زمان هر دو نیست

//  و دام واقعی‌تر: propertyهای هم‌نام با نوع‌های ناسازگار
type A = { status: "ok" };
type B = { status: "error" };
type AB = A & B;          // status: "ok" & "error" → never
// AB عملاً غیرقابل ساخت است — و کامپایلر فقط موقع ساختن مقدار شکایت می‌کند
```

هشدار — intersection ادغام نیست

`&` اعضای هم‌نام را «merge» نمی‌کند؛ نوع‌هایشان را intersect می‌کند و نتیجه ناسازگار، بی‌سروصدا `never` می‌شود. اگر خواستید نوعی را «override» کنید، الگوی درست `Omit<A, "k"> & { k: New }` است (فصل ۲۰). هر وقت مقداری «به هیچ شکلی قابل ساخت نبود»، اول دنبال never پنهان در intersection بگردید.

union اشیاء ≠ شیءِ unionها

تفاوتی که در طراحی state بسیار مهم است:

union-vs-object.ts

type-checks ✓

```ts
// ❌ ترکیب‌های بی‌معنا مجازند — شیءِ unionها:
type LoadingBad = {
  status: "loading" | "success" | "error";
  data: string | null;
  error: Error | null;
};
const weird: LoadingBad = { status: "success", data: null, error: new Error("?") };
// ✓ کامپایل می‌شود ولی منطقاً مزخرف است!

// ✅ فقط ترکیب‌های معنادار قابل ساخت‌اند — اشیاءِ union:
type Loading =
  | { status: "loading" }
  | { status: "success"; data: string }
  | { status: "error";   error: Error };
```

در نسخه دوم، «success بدون data» و «error بدون error» اصلاً **قابل بیان نیستند**. این هسته فلسفه «state ناممکن را غیرقابل‌ساخت کن» است که فصل ۲۲ و ۳۶ بر آن بنا می‌شوند — و دلیل این‌که union را باید ابزار مدل‌سازی دید، نه فقط «یا»ی ساده.

چند نکته ترکیبی

- union خودش را مسطح می‌کند: `(A | B) | C` همان `A | B | C` است؛ اعضای تکراری و اعضای زیرمجموعه جذب می‌شوند (`"a" | string` → `string`).
- `&` روی union توزیع می‌شود: `A & (B | C)` برابر `(A & B) | (A & C)` — همان جبر مدرسه.
- ترتیب اعضای union تضمین معنایی ندارد؛ هرگز به ترتیب آن‌ها در خطاها یا tooltipها تکیه نکنید (و declaration emit در نسخه ۶/۷ با `stableTypeOrdering` پایدارتر شده — فصل ۳۳).

زیر کاپوت — چرا این‌طور است؟

با عینک مجموعه‌ای فصل ۸، دو رفتار «عجیب» این فصل کاملاً منطقی می‌شوند:

چرا روی union فقط اعضای مشترک مجازند؟ `A | B` یعنی «مقداری از اجتماع دو مجموعه» — شاید A، شاید B؛ کامپایلر نمی‌داند کدام. عملیات مجاز باید در «هر دو جهان ممکن» سالم باشد، پس فقط اشتراک قابلیت‌ها در دسترس است. سخت‌گیری نیست؛ تنها پاسخ صادقانه به عدم قطعیت است. narrowing (فصل ۱۰) دقیقاً سازوکار «حذف جهان‌های ممکن» است — هر بررسی یکی از شاخه‌های تردید را می‌بندد تا قابلیت‌های اختصاصی آزاد شوند.

چرا intersection دو primitive بی‌صدا never می‌شود؟ `string & number` اشتراک دو مجموعه بی‌تقاطع است — تهی، یعنی never. ریاضی‌اش بی‌نقص است؛ مشکل «بی‌صدا» بودن است: خطا در محل تعریف گزارش نمی‌شود، بلکه بعداً در نقطه استفاده ظاهر می‌شود و شما را کیلومترها دورتر از علت رها می‌کند. برای اشیاء، intersection عضوبه‌عضو فرو می‌رود: عضو هم‌نام، اشتراکِ نوع‌هایش را می‌گیرد — و اگر ناسازگار باشند، never به «درون» یک عضو نشت می‌کند؛ یافتنش

سخت‌تر. دقیقاً به همین دلیل برای قراردادهای شیئی `interface extends` توصیه می‌شود: extends تضاد را در «محل تعریف» با پیام صریح گزارش می‌کند، چون قاعده‌اش «سازگاری» است نه «اشتراک ریاضی».

جمع‌بندی سازوکاری: union و intersection عملگرهای مجموعه‌ای خالص‌اند و استثنا ندارند — همه شگفتی‌ها از این است که شهود ما گاهی «ترکیب ساختاری» انتظار دارد، جایی که زبان «عملیات مجموعه‌ای» انجام می‌دهد. هر جا گیج شدید، سؤال درست: «مجموعه مقادیر این نوع دقیقاً چیست؟»

تمرین

جبر نوع‌ها

نتیجه هر عبارت نوعی را ساده کنید:

quiz9.ts — type-checks ✓

```ts
type T1 = (string | number) & (number | boolean);
type T2 = "a" | "b" | string;
type T3 = { x: 1 } & { y: 2 };
type T4 = keyof ({ a: 1 } | { b: 2 });  //  پیش‌نمایش فصل ۱۶
```

پاسخ: `T1 = number` — توزیع و حذف ترکیب‌های never · `T2 = string` — literalها جذب supertype می‌شوند · `T3 = { x: 1; y: 2 }` (به‌صورت intersection دو شیء سازگار) · `T4 = never` — keyof روی union فقط کلیدهای **مشترک** را می‌دهد و این دو شیء کلید مشترکی ندارند.

پرسش‌های مصاحبه این فصل

۱. intersection دو شیء با عضو هم‌نامِ ناسازگار چه می‌شود؟

عضو هم‌نام، intersection نوع‌هایش را می‌گیرد؛ اگر ناسازگار باشند (مثل `string & number`) بی‌صدا `never` می‌شود و خطا جای عجیبی ظاهر می‌شود. برای قراردادهای شیئی، `interface extends` بهتر است چون تضاد را همان‌جا صریح گزارش می‌کند.

۲. تابعی که `string | number` برمی‌گرداند طراحی خوبی است؟

بستگی دارد بار narrowing را چه کسی باید بکشد. اگر خروجی واقعاً دو حالت دامنه‌ای دارد، union صادقانه است — ترجیحاً discriminated. اما اگر union فقط حاصل تنبلی پیاده‌سازی است، با overload یا جنریک رابطه ورودی→خروجی را دقیق کنید تا مصرف‌کننده if نزند.

Narrowing و تحلیل جریان کنترل

کامپایلر کد تو را خط‌به‌خط می‌خواند — typeof، in، instanceof و discriminant

اهداف یادگیری این فصل

- انواع type guard داخلی و محل اثرشان را بشناسی
- نوع یک متغیر را در هر نقطه از تابع پیش‌بینی کنی
- بدانی چرا narrowing گاهی «از دست می‌رود» (callback، mutation)

تا این‌جا گفتیم روی union فقط اعضای مشترک در دسترس‌اند. اما کد واقعی پر از `if` است — و TypeScript این `if`ها را **می‌خواند**. به این سازوکار، تحلیل جریان کنترل (control-flow analysis) می‌گویند: نوع هر متغیر، در هر نقطه از کد، بر اساس بررسی‌هایی که تا آن نقطه از آن عبور کرده، باریک می‌شود. این «هوشمندترین» بخش کامپایلر و ستون کار روزمره با TypeScript است.

جعبه‌ابزار narrowing داخلی

guards.ts — type-checks ✓

```
function fmt(v: string | number | Date | null): string {
  if (v === null) return "—";            // حذف null با equality
  if (typeof v === "string") return v;     // typeof برای primitiveها
  if (typeof v === "number") return v.toFixed(2);
  return v.toISOString();                  // فقط Date مانده — خود کامپایلر می‌داند
}
```

ابزارها و محدوده اثرشان:

Guard	چه چیزی را تشخیص می‌دهد	نکته
`typeof x === "..."`	هفت primitive	`typeof null === "object"` — دام قدیمی JS
`x === null`، `x == null`	غیبت	`== null` هر دو null/undefined را می‌گیرد
`x instanceof C`	نمونه کلاس	برای اشیاء plain کار نمی‌کند
`"k" in x`	وجود property	برای تفکیک شکل اشیاء
`Array.isArray(x)`	آرایه	بهتر از instanceof برای آرایه
truthiness (`if (x)`)	حذف falsy	مراقب "" و 0 باشید!
`x.kind === "..."`	discriminant	قهرمان فصل ۲۲

و دو رفتار ظریف‌تر که حرفه‌ای‌ها را از بقیه جدا می‌کند:

flow-subtle.ts — type-checks ✓

```ts
function firstUpper(items: string[] | null) {
  // narrowing بعد از این خط، نوع دقیق‌تر دنبال می‌شود :با انتساب
  const list = items ?? [];

  // narrowing در عبارات منطقی:
  return list.length > 0 && list[0] !== undefined
    ? list[0].toUpperCase()
    : "";
}

function process(input: string | number) {
  if (typeof input === "number") {
    input = String(input); //  انتساب، نوع جریان را عوض می‌کند
  }
  return input.trim();     //  ✓ اینجا input فقط string است
}
```

جایی که narrowing «گم می‌شود»

دو موقعیت که همه به آن می‌خورند. **اول: مرز تابع.** تحلیل جریان، درونِ یک تابع است؛ callback مرز جدیدی است:

lost-callback.ts — type-checks ✓

```ts
function send(msg: string | null) {
  if (msg !== null) {
    setTimeout(() => {
      console.log(msg.toUpperCase()); //  در TS چرا مجاز شد؟ — ✓ :مدرن
    }, 100);
  }
}
```

نسخه‌های مدرن TypeScript این مورد خاص را هوشمندانه حل می‌کنند (چون `msg` پارامتر است و بعد از بررسی مقدار نمی‌گیرد)، اما قاعده عمومی مهم است: اگر متغیر، `let` باشد یا جایی mutate شود، کامپایلر نمی‌تواند تضمین کند بین «بررسی» و «اجرای callback» مقدار عوض نشده — و narrowing را کنار می‌گذارد. راه‌حل همیشگی و مطمئن:

capture-const.ts — type-checks ✓

```ts
function send(msg: string | null) {
  if (msg === null) return;
  const safeMsg = msg;              // const snapshot — نوع: string
  setTimeout(() => console.log(safeMsg.toUpperCase()), 100); //  ✓ همیشه
}
```

دوم: property اشیاء. narrowing روی `obj.prop` ضعیف‌تر از متغیر محلی است، چون property می‌تواند getter باشد یا از بیرون تغییر کند. الگوی حرفه‌ای: اول در متغیر محلی بریزید، بعد بررسی کنید.

> **مدل ذهنی — کامپایلر به‌عنوان خواننده بدبین**
>
> کامپایلر کد شما را مثل یک reviewer بدبین می‌خواند: فقط چیزی را قبول می‌کند که **در همین مسیر اجرا اثبات شده و تا نقطه مصرف نمی‌تواند عوض شده باشد.** هر وقت narrowing «کار نکرد»، بپرسید: از دید یک خواننده بدبین، بین بررسی و استفاده چه اتفاقی می‌تواند بیفتد؟ پاسخ تقریباً همیشه mutation یا مرز تابع است.

شکست زودهنگام — narrowing معماری می‌سازد

الگوی early-return فقط سلیقه نیست؛ با تحلیل جریان هم‌افزاست:

early-return.ts — type-checks ✓

```ts
interface Session { user: { id: string; role: "admin" | "member" } | null }

function requireAdmin(session: Session) {
  if (!session.user) throw new Error("Not authenticated");
  if (session.user.role !== "admin") throw new Error("Forbidden");

  // از این‌جا به بعد، بدون هیچ بررسی اضافه:
  return session.user;   // { id: string; role: "admin" }
}
```

هر `return` / `throw` زودهنگام، فضای حالت باقیمانده را کوچک‌تر می‌کند و کدِ پایین تابع در «دنیای تمیزشده» زندگی می‌کند. مقایسه کنید با if‌های تودرتو که در هر شاخه باید دوباره همه چیز را ثابت کنید.

> **در عمل — بگذار کامپایلر تایپ‌نگهدار باشد**
>
> وقتی narrowing درست چیده شود، تقریباً هیچ annotation میانی لازم نیست — نوع‌ها «جاری» می‌شوند. اگر دیدید وسط تابع مرتب `as` می‌نویسید، معمولاً یعنی ترتیب بررسی‌ها بد است یا داده‌تان discriminant ندارد (فصل ۲۲)، نه این‌که TypeScript «کم‌هوش» است.

زیر کاپوت — چرا این‌طور است؟

تحلیل جریان کنترل (CFA) را بهتر می‌فهمید اگر بدانید کامپایلر واقعاً چه می‌سازد: روی هر تابع یک **گراف جریان** — گره‌ها نقاط برنامه، یال‌ها مسیرهای ممکن اجرا. برای هر «ارجاع» (متغیر یا زنجیره property) در هر گره، مجموعه‌ای از «حقایق اثبات‌شده» نگه داشته می‌شود: «از انشعابِ `typeof x === "string"` رد شدیم، پس در این شاخه x از stringها فراتر نیست.» نوع باریک‌شده، حاصل اعمال همین حقایق بر نوع اعلام‌شده است.

نکته‌ای که همه رفتارهای «گم شدن narrowing» را توضیح می‌دهد: **حقیقت فقط تا وقتی معتبر است که مقدار نتوانسته باشد عوض شود.** هر رویدادی که احتمال تغییر بیاورد، حقایق مربوط را باطل می‌کند:

- **انتساب مجدد** — بدیهی‌ترین باطل‌کننده؛ نوع از راستِ انتساب دوباره شروع می‌شود (و خودِ انتساب هم narrowing جدید می‌سازد).
- **مرز تابع** — callback شاید دیرتر اجرا شود؛ بین «بررسی» و «اجرا» هر اتفاقی ممکن است. استثنای مدرن: اگر متغیر const یا پارامترِ بدونِ انتساب مجدد باشد، از TS 5.4 حقیقت درون closure حفظ می‌شود — تغییرناپذیری، اعتبار ابدی می‌دهد.
- **فراخوانی تابع یا await بین بررسی و استفاده — برای propertyها.** `obj.prop` ممکن است getter باشد یا کسی از بیرون شیء را دست‌کاری کند؛ کامپایلر «خلوص» توابع میانی را نمی‌تواند اثبات کند، پس بدبینانه حقایق property را دور می‌ریزد.

حالا الگوی «snapshot در const» از ترفند به قضیه تبدیل می‌شود: متغیر محلی تغییرناپذیر، حقیقتش را هیچ رویدادی باطل نمی‌کند — نه closure، نه await، نه فراخوانی. با یک خط، «ارجاع فرّار» را «ارجاع اثبات‌پذیر» کرده‌اید. قاعده کلی دیباگ narrowing: هر وقت باریک‌سازی غیب شد، دنبال «رویداد باطل‌کننده» بین بررسی و استفاده بگردید — همیشه یکی از همین سه‌تاست.

تمرین

نوع را در هر نقطه بگو

نوع `x` را در نقاط ۱ تا ۳ مشخص کنید:

quiz10.ts — type-checks ✓

```ts
function f(x: string | number | string[] | null) {
  if (x == null) return;
  // ۱
  if (typeof x === "object") {
    // ۲
    return x.length;
  }
  // ۳
}
```

پاسخ: ۱. `string | number | string[]` — `== null` فقط null/undefined را حذف کرد. ۲. `string[]` — تنها عضو «object» باقیمانده. ۳. `string | number` — شاخه object با return خارج شد.

پرسش‌های مصاحبه این فصل

۱. چرا بعد از `await` گاهی narrowing قبلی از بین می‌رود؟

تحلیل جریان به mutation بدبین است: روی property اشیاء، بین بررسی و استفاده اگر مرز تابع/await باشد، کامپایلر نمی‌تواند تضمین کند مقدار عوض نشده. راه‌حل استاندارد: snapshot در متغیر محلی const قبل از بررسی — هم برای کامپایلر اثبات است هم برای خواننده.

۲. عملگر `in` را کجا برای narrowing ترجیح می‌دهید؟

برای تفکیک اشیائی که tag مشترک ندارند و نمی‌توانید تغییرشان دهید (داده‌های خارجی، APIهای قدیمی). اگر مالک نوع‌ها خودتان هستید، discriminated union با tag صریح همیشه انتخاب اول است — narrowing تمیزتر و exhaustiveness رایگان.

توابع — قرارداد ورودی و خروجی

نوع تابع، پارامتر اختیاری، rest، this و نوع بازگشتی

اهداف یادگیری این فصل

- نوع تابع را به هر دو شکل alias و signature بنویسی
- رفتار bivariance پارامترها در method را بشناسی
- تصمیم بگیری نوع بازگشتی را صریح بنویسی یا به استنتاج بسپاری

تابع، واحد اصلی قرارداد در هر برنامه است: «این را بده، آن را بگیر». این فصل زبانِ دقیق نوشتن این قرارداد را کامل می‌کند — از syntax پایه تا ظرافت‌هایی مثل پارامترهای rest و نوع `this` که در کتابخانه‌نویسی حیاتی می‌شوند.

دو شکل نوشتن نوع تابع

fn-types.ts — type-checks ✓

```ts
// ۱) امضای inline — نوع پارامترها و خروجی سر جایشان:
function priceAfterTax(price: number, rate: number): number {
  return price * (1 + rate);
}

// ۲) type alias وقتی قرارداد مستقل از پیاده‌سازی است — برای نوع تابع:
type Formatter = (value: number, unit: string) => string;

const fmtFa: Formatter = (v, u) => `${v.toLocaleString("fa-IR")} ${u}`;
//            ^^^^^^^^^^ پارامترها زمینه! annotation (فصل ۴) نمی‌خواهند
```

شکل دوم را دست‌کم نگیرید: وقتی نوع تابع اسم دارد، می‌توانید آن را بین فایل‌ها به اشتراک بگذارید، در props کامپوننت‌ها بگذارید و مستندش کنید. در کتابخانه‌ها، «نوعِ callback» بخشی از API عمومی است.

پارامترهای اختیاری، پیش‌فرض و rest

params.ts — type-checks ✓

```ts
function buildUrl(
  base: string,
  path = "/",                       //  نوع ⇐ پیش‌فرض string، اختیاری
  query?: Record<string, string>,   //  اختیاری ⇐ string | undefined... نه!
  ...segments: string[]             // rest — صفر یا بیشتر
): string {
  const q = query ? "?" + new URLSearchParams(query) : "";
  return [base, path, ...segments].join("") + q;
}

buildUrl("https://api.dev");
buildUrl("https://api.dev", "/v2", { page: "1" }, "users", "42");
```

دو نکته دقیق:

- نوع `query` درونِ تابع `Record<string, string> | undefined` است — علامت `?` یعنی «می‌تواند جا بماند»، و جاماندن یعنی `undefined`. پس قبل از استفاده باید بررسی شود.
- پارامتر اختیاری باید بعد از اجباری‌ها بیاید. اگر «وسطی» اختیاری است، به‌جای جابه‌جایی، یک **options object** بدهید — خواناتر و توسعه‌پذیرتر:

options-object.ts — type-checks ✓

```ts
interface RetryOptions { attempts?: number; delayMs?: number; signal?: AbortSignal }

function retry<T>(task: () ⇒ Promise<T>, opts: RetryOptions = {}): Promise<T> {
  const { attempts = 3, delayMs = 200 } = opts;   //  پیش‌فرض‌ها یکجا و خوانا
  void delayMs;
  return task(); //  (پیاده‌سازی کامل در فصل ۳۵)
}
void (retry<number> satisfies (t: () ⇒ Promise<number>, o?: RetryOptions) ⇒
Promise<number>);
```

نوع بازگشتی — صریح یا استنتاجی؟

قاعده فصل ۴ را حالا دقیق‌تر کنیم:

- **توابع عمومی/صادراتی:** نوع بازگشتی را بنویسید. قرارداد را قفل می‌کند، خطا را به محل علت می‌برد و سرعت کامپایل در پروژه‌های بزرگ را هم بهتر می‌کند (کامپایلر لازم نیست بدنه را برای مصرف‌کننده‌ها دوباره استنتاج کند — فصل ۳۲).
- **توابع محلی کوچک:** استنتاج کافی و تمیزتر است.
- **توابع async:** خروجی همیشه `Promise<T>` است؛ ننوشتنش مجاز است اما در API عمومی بنویسید تا قرارداد در امضا دیده شود.

نوع this — پارامتر نامرئی

در JavaScript، `this` عملاً یک پارامتر پنهان است؛ TypeScript اجازه می‌دهد صریحش کنید:

this-param.ts — type-checks ✓

```ts
interface Counter { count: number }

function increment(this: Counter, by: number) {
  this.count += by;    // ✓ this تایپ شده
}

const c: Counter & { increment: typeof increment } = {
  count: 0,
  increment,
};
c.increment(5);        // ✓ this ≡ c
```

this-err.ts — × خطای عمدی

```ts
function standalone(this: { count: number }) { this.count++; }
standalone();    // × this تامین نشده
```

پیام کامپایلر

```
this-err.ts:2:1 - error TS2684: The 'this' context of type 'void' is
not assignable to method's 'this' of type '{ count: number; }'.
```

پارامتر `this` در خروجی JS حذف می‌شود (erasure) — فقط قرارداد است. در کد مدرن با arrow functionها کمتر لازمش دارید، اما برای تایپ کردن کتابخانه‌های قدیمی‌تر و event handlerهای DOM ضروری است.

مدل ذهنی — امضا یعنی «همه چیزهایی که صدازننده باید بداند»

امضای خوب، خودکفاست: ورودی‌ها، خروجی، this، و خطاهای ممکن (که در TS در نوع نمی‌آیند — فصل ۳۵ راه‌حل Result را می‌دهد). اگر برای استفاده درست از تابعی باید بدنه‌اش را خواند، امضایش ناقص است. این سنجه را در code review به‌کار ببرید.

هشدار — تعداد پارامتر کمتر، همیشه سازگار است

در TypeScript تابعی با پارامترهای کمتر، جانشین قانونی تابعی با پارامترهای بیشتر است: `arr.map(x ⇒ x)` درست کار می‌کند با این‌که callback رسمی map سه پارامتر دارد. این برای callbackها عالی است، اما یعنی «فراموش کردن پارامتر» خطا نیست! اگر اجبار می‌خواهید، آن را در نوع داده مدل کنید نه در تعداد پارامتر.

زیر کاپوت — چرا این‌طور است؟

سه قاعده «عجیب» سازگاری توابع، همگی از یک اصل مشتق می‌شوند: **جایگزینی امن در نقطه فراخوانی**. تابع B وقتی می‌تواند جای تابع A بنشیند که هر فراخوانیِ نوشته‌شده برای A، با B هم سالم بماند. حالا هر سه را استدلال کنید:

چرا تابعِ با پارامتر کمتر جایگزین‌پذیر است؟ فراخواننده‌ای که برای `(a, b) ⇒ void` کد نوشته، همیشه دو آرگومان می‌فرستد. تابعی که فقط a را می‌خواند، آرگومان دوم را بی‌خطر نادیده می‌گیرد — در JavaScript آرگومان اضافه هرگز خطا نبوده. برعکسش فاجعه است: تابعِ پرتوقع‌تر، پارامتر سومی می‌خواند که فراخواننده هرگز نفرستاده — undefined غیرمنتظره. پس قاعده «کمتر، آری؛ بیشتر، هرگز» دقیقاً همان چیزی است که callback نویسی روزمره (`arr.map(x ⇒ x * 2)` با callback سه‌پارامتری) را هم ممکن کرده.

چرا تابع مقداربرگردان جای `() ⇒ void` می‌نشیند؟ امضای void یعنی «فراخواننده قول داده به خروجی نگاه نکند» — قراردادی درباره مصرف‌کننده است، نه تولیدکننده. مقدارِ برگشتی که کسی نمی‌خواند، بی‌ضرر است. اگر این قاعده نبود، `arr.forEach(x ⇒ results.push(x))` خطا می‌شد چون push عدد برمی‌گرداند — زبان با الگوهای واقعی JS می‌جنگید.

چرا ترتیب overload مهم است؟ چون resolution «اولین امضای سازگار» را برمی‌دارد و متوقف می‌شود — نه «بهترین» را. انتخابِ ساده‌تر و قابل پیش‌بینی‌تر، به بهای مسئولیت شما: خاص‌ترین امضا باید اول بیاید، وگرنه امضای عام همیشه می‌برد و دقت overload های بعدی هرگز دیده نمی‌شود. فصل ۲۵ این را عمیق‌تر می‌شکافد.

تمرین

امضای بهتر طراحی کن

این امضا سه مشکل طراحی دارد؛ پیدا و اصلاحشان کنید:

quiz11.ts — type-checks ✓

```ts
function createUser(name: string, admin?: boolean, email?: string,
                    sendWelcome?: boolean, locale?: string) { /* ... */ }
```

پاسخ: ۱. چهار پارامتر اختیاریِ پشت‌سرهم، در محل فراخوانی ناخواناست: `createUser("x", undefined, "a@b.c", true)`. ۲. دو boolean کنار هم مستعد جابه‌جایی بی‌صداست. ۳. توسعه آینده یعنی شکستن همه فراخوانی‌ها. اصلاح: پارامتر اول اجباری، بقیه در options object با نام:

quiz11-solution.ts — type-checks ✓

```ts
interface CreateUserOptions {
  email?: string; role?: "admin" | "member";
  sendWelcome?: boolean; locale?: "fa" | "en";
}
function createUser(name: string, options: CreateUserOptions = {}) { void name; void
options; }
```

بونوس: `role` به‌جای boolean — توسعه‌پذیر و خوانا (فصل ۵).

پرسش‌های مصاحبه این فصل

۱. چرا callback با نوع `() ⇒ void` می‌تواند مقدار برگرداند و این سالم است؟

void در جایگاه خروجی callback یعنی «خروجی‌ات نادیده گرفته می‌شود»، نه «چیزی برنگردان». برای همین `arr.forEach(x ⇒ list.push(x))` مجاز است با این‌که push عدد برمی‌گرداند. ناسالم نیست چون هیچ‌کس به آن خروجی تکیه نمی‌کند — قرارداد همین را می‌گوید.

۲. پارامتر اختیاری یا پارامتر با پیش‌فرض — تفاوت قراردادی؟

`x?: T` یعنی caller می‌تواند ندهد و شما با `T | undefined` طرفید؛ `x: T = def` یعنی caller می‌تواند ندهد ولی بدنه شما همیشه T تمیز دارد. برای منطق داخلی، پیش‌فرض تقریباً همیشه بهتر است — undefined را همان مرز امضا حل می‌کند.

آرایه، Tuple، readonly — و هنر خواندن خطای کامپایلر

ساختارهای ترتیبی + روش سیستماتیک تفسیر پیام‌های خطا

اهداف یادگیری این فصل

- tuple را برای ساختارهای با طول ثابت و نقش‌دار به‌کار ببری
- تفاوت readonly T[] با T[] را در سازگاری بدانی
- هر خطای کامپایلر را با الگوریتم چهارمرحله‌ای تفسیر کنی

این فصل دو مهارت را تمام می‌کند که بخش یکم را به سطح «کار روزمره روان» می‌رسانند: مدل‌سازی ساختارهای ترتیبی (آرایه، tuple، readonly) و — شاید مهم‌تر — روش سیستماتیک خواندن خطاهای کامپایلر، که بقیه کتاب مدام به آن تکیه خواهد کرد.

آرایه و دو واقعیت ناخوشایند

arrays.ts — type-checks ✓

```
const scores: number[] = [85, 92, 78];
//  معادل: Array<number> — سلیقه‌ای، ولی در تیم یکدست کنید

//  واقعیت ۱: ایندکس ممکن است وجود نداشته باشد
const first = scores[0];
//  بدون noUncheckedIndexedAccess: number (خوش‌بینانه!)
//  با آن: number | undefined (صادقانه — فصل ۲۷)

//  واقعیت ۲: متدهای mutate کننده در نوع دیده نمی‌شوند
function highest(xs: number[]): number | undefined {
  return [...xs].sort((a, b) => b - a)[0]; //  کپی؛ آرایه ورودی را خراب نکند sort
}
```

readonly — قرارداد «دست نمی‌زنم»

readonly.ts — × خطای عمدی

```
function sum(xs: readonly number[]): number {
  xs.push(0);  //  × متدهای mutate کننده روی readonly وجود ندارند
  return xs.reduce((a, b) => a + b, 0);
}
```

پیام کامپایلر

```
readonly.ts:2:6 - error TS2339: Property 'push' does not exist on
type 'readonly number[]'.
```

قاعده سازگاری مهم: `number[]` به `readonly number[]` قابل انتساب است (قول کمتر، آزادی بیشتر برای صدازننده)، اما برعکس نه. نتیجه عملی: **پارامترهای آرایه‌ای که تغییر نمی‌دهید را readonly بنویسید** — ورودی‌های بیشتری می‌پذیرید و قراردادتان صادق‌تر است. همین منطق برای `ReadonlyMap` ، `ReadonlySet` و `readonly` روی propertyها برقرار است.

Tuple — طول ثابت، نقش ثابت

وقتی «موقعیت» معنا دارد، tuple بنویسید نه آرایه:

tuples.ts — type-checks ✓

```ts
type LatLng = [lat: number, lng: number];      //  عناصر نام‌دار — فقط مستندات
const tehran: LatLng = [35.6892, 51.3890];

// tuple با عضو اختیاری و rest:
type Entry = [key: string, value: string, ...flags: boolean[]];

//  کاربرد آشنا: useState دقیقا یک tuple برمی‌گرداند
function useToggle(init: boolean): [boolean, () => void] {
  let v = init;
  return [v, () => { v = !v; }];
}
const [on, toggle] = useToggle(false);  //  نوع‌ها: boolean و () => void
void on; void toggle;
```

tuple دقیقاً همان‌جایی ارزش دارد که destructuring معنادار است. اگر عناصر هم‌جنس و هم‌نقش‌اند (`number[]` نمره‌ها)، آرایه بمانید؛ اگر «عنصر اول یک چیز است و دومی چیز دیگر»، tuple قرارداد را رسمی می‌کند.

هنر خواندن خطای کامپایلر

خطاهای TypeScript طولانی‌اند چون **کامل**‌اند: کل زنجیره استدلال کامپایلر را گزارش می‌کنند. الگوریتم چهار مرحله‌ای برای هر خطا:

۱. **آخرین سطر تورفته را بخوانید، نه اولین را.** خطاهای تودرتو از کل به جزء می‌روند؛ علت واقعی در عمیق‌ترین سطح است.

۲. **دو نوع دعوا را پیدا کنید.** تقریباً هر خطا شکل «X is not assignable to Y» دارد — X چیست؟ Y چیست؟ کدام طرف اشتباه است؟ (نصف اوقات Y — یعنی annotation شما — غلط است!)

۳. **کد خطا را جدی بگیرید.** TS2345 (آرگومان ناسازگار)، TS2339 (property غایب)، TS2322 (انتساب ناسازگار)، TS18047 (possibly null) — هر کد، یک کلاس مشخص از مشکل است. پیوست D جدول کامل را دارد.

۴. **کوچک‌ترین بازتولید را بسازید.** اگر هنوز مبهم است، عبارت را به متغیرهای میانی با annotation بشکنید — خطا دقیقاً روی قطعه مقصر می‌نشیند (فصل ۳۲ این تکنیک را گسترش می‌دهد).

روی یک مثال واقعی:

read-error.ts — × خطای عمدی

```ts
interface Order { items: { sku: string; qty: number }[] }

function totalQty(order: Order): number {
  return order.items.reduce((sum, it) => sum + it.quantity, 0);  // ×
}
```

پیام کامپایلر

```
read-error.ts:4:52 - error TS2339: Property 'quantity' does not exist
on type '{ sku: string; qty: number; }'. Did you mean 'qty'?
```

مرحله ۱: یک سطح بیشتر ندارد. مرحله ۲: دعوای «quantity روی نوع عضو items». مرحله ۳: TS2339 یعنی عضو غایب — و پیشنهاد `Did you mean` معمولاً درست است. کل مسیر: ده ثانیه. همین الگوریتم روی خطای چهل‌خطی جنریک هم کار می‌کند — فقط مرحله ۱ مهم‌تر می‌شود.

مدل ذهنی — خطا یک stack trace استدلال است

به خطای کامپایلر مثل stack trace نگاه کنید: سطرهای بیرونی «از کجا شروع کردم» هستند و سطر درونی «کجا گیر کردم». با همین نگاه، خطای بزرگ ترسناک نیست — فقط مفصل است. عادت «فقط سطر اول را خواندن و حدس زدن»، پرهزینه‌ترین عادت کار با TypeScript است.

در عمل — pretty print و ابزار کمکی

خروجی رنگی و تورفته `tsc --pretty` (پیش‌فرض ترمینال) خوانش را ساده می‌کند؛ در ادیتور هم افزونه‌هایی مثل Pretty TypeScript Errors پیام‌ها را فرمت می‌کنند. اما ابزار جای الگوریتم را نمی‌گیرد: «عمیق‌ترین سطح، دو نوع دعوا، کد خطا، بازتولید کوچک».

زیر کاپوت — چرا این‌طور است؟

دو تصمیم طراحی این فصل، پشت‌صحنه روشنی دارند:

چرا `readonly T[]` ابرنوعِ `T[]` است و نه برعکس؟ با منطق «قابلیت» نگاه کنید: readonly آرایه‌ای است که فقط «قول خواندن» می‌دهد — زیرمجموعه‌ای از قابلیت‌های آرایه کامل. هر آرایه معمولی این قول را به‌راحتی می‌دهد (چیزی که می‌توانی بنویسی، قطعاً می‌توانی نخوانی‌نوشت)، پس انتساب T[] → readonly T[] امن و مجاز است. عکسش یعنی «ادعای توان نوشتن» برای مقداری که فقط قول خواندن داده — ناامن و ممنوع. از این نامتقارنی یک قاعده API درمی‌آید که ارزان‌ترین ارتقای ایمنی کل کتاب است: پارامترهای ورودی را readonly بگیرید؛ هم آرایه معمولی می‌پذیرید هم تعهد

می‌دهید دست نمی‌زنید — بدون ذره‌ای هزینه runtime، چون readonly هم مثل بقیه نوع‌ها پاک می‌شود و فقط «قرارداد کامپایلی» است.

چرا tuple این‌قدر با آرایه فرق دارد، وقتی در runtime هر دو یک Array اند؟ چون تفاوتشان فقط در «دانش ایستا» است: tuple نوعی است که **طول** و **نوعِ هر جایگاه** را به‌صورت literal می‌داند — `length` آن نوع `2` دارد، نه number. همین دانش اضافه است که destructuring دقیق، پرچم‌گذاری عناصر اختیاری و variadic tuples را ممکن می‌کند، و در فصل‌های ۱۴ و ۲۶ به ابزار محاسبات type-level تبدیل می‌شود. tuple یعنی «آرایه + اثبات ساختار» — اثباتی که فقط کامپایلر می‌بیند.

نکته آخرِ ظریف: `noUncheckedIndexedAccess` در واقع اعتراف صادقانه سیستم نوع به یک واقعیت runtime است — ایندکس دلخواه آرایه ممکن است بیرون محدوده باشد. نوع `T | undefined` برای `arr[i]` «حقیقت» است؛ پیش‌فرضِ بدون این پرچم، «خوش‌بینی تاریخی» است که به‌خاطر سازگاری باقی مانده.

تمرین

خطا را تا ته بخوان

بدون اجرا، بگویید این کد چه خطایی می‌گیرد و چرا:

quiz12.ts — × خطای عمدی

```ts
const pair: [string, number] = ["a", 1];
const nums: number[] = [1, 2, 3];
const x: [string, number] = ["a", 1, 2] as [string, number, 2];
const y: readonly number[] = nums;
const z: number[] = y;
```

پاسخ: خط ۳: tuple سه‌عضوی به دوعضوی قابل انتساب نیست (طول بخشی از نوع است) — TS2322. خط ۵: `readonly number[]` به `number[]` قابل انتساب نیست — جهت سازگاری readonly یک‌طرفه است؛ خط ۴ درست است چون جهت درست همان است.

پرسش‌های مصاحبه این فصل

۱. خطای چهل‌خطی — اولین حرکت شما؟

عمیق‌ترین سطرِ تورفته را می‌خوانم: WeasyPrint-وار از کلی به جزئی چیده شده و علت واقعی تهِ تورفتگی است. بعد دو نوع دعوا را مشخص می‌کنم و می‌پرسم کدام طرف غلط است — نیمی از موارد annotation خوش‌بینانه خودم است، نه کد.

۲. `readonly T[]` در پارامتر تابع چه پیامی می‌دهد؟

دو پیام: «من تغییرت نمی‌دهم» (تعهد قابل اتکا برای caller) و «آرایه‌های readonly را هم بپذیر» — چون `readonly T[]` به `T[]` قابل انتساب است ولی برعکس نه. پارامترهای فقط-خواندنی را همیشه readonly بگیرید؛ API شما پذیرنده‌تر و صادق‌تر می‌شود.

PART 02

۲

طراحی نوع پیشرفته

اینجا TypeScript از «حاشیه‌نویسی کد» به «زبان طراحی» تبدیل می‌شود. جنریک‌ها، conditional و mapped typeها، template literalها و الگوهای برندسازی نوع — همراه با معیار همیشگی این کتاب: هر abstraction باید هزینه خوانایی و نگه‌داری‌اش را توجیه کند.

---

13 interface و type — یک انتخاب مهندسی

14 جنریک‌ها — پارامتر نوع، Constraint و Default

15 جنریک در عمل — API و کامپوننت قابل‌استفاده مجدد

16 keyof، typeof و Indexed Access

17 Conditional Types و infer

18 Mapped Types و Key Remapping

19 Template Literal Types

20 Utility Types — استاندارد و سفارشی

21 as const، satisfies و Const Type Parameters

22 Discriminated Union و Exhaustiveness

23 Type Predicate، Assertion Function و Branded Types

24 Variance و سازگاری توابع

25 Declaration Merging، Overload و Module Augmentation

26 برنامه‌نویسی Type-Level — قدرت، هزینه و مرز عقل

interface و type — یک انتخاب مهندسی

تفاوت‌های واقعی، declaration merging و قاعده انتخاب در تیم

اهداف یادگیری این فصل

- تفاوت‌های عملی interface و type alias را فهرست کنی
- بدانی merging کجا مفید و کجا خطرناک است
- یک قاعده سازگار برای codebase انتخاب کنی

«interface یا type؟» — پرتکرارترین سؤال مصاحبه و code review تایپ‌اسکریپتی. پاسخ درست نه «فرقی ندارد» است و نه فهرست حفظی تفاوت‌ها؛ پاسخ درست، فهم دو تفاوت واقعی و بعد **انتخاب یک قاعده تیمی** است.

آنچه هر دو به یک اندازه خوب انجام می‌دهند

شکل شیء، امضای تابع، extend کردن، implement شدن توسط کلاس — هر دو ابزار از پس همه این‌ها برمی‌آیند:

both-work.ts — type-checks ✓

```ts
interface UserI { id: number; name: string }
type      UserT = { id: number; name: string };

interface AdminI extends UserI { scope: string }
type      AdminT = UserT & { scope: string };

class Member implements UserI { id = 0; name = "" }
class MemberT implements AdminT { id = 0; name = ""; scope = "" }
```

برای ۸۰٪ کد روزمره، این دو واقعاً هم‌ارزند. تفاوت‌ها از این‌جا شروع می‌شوند:

تفاوت ۱ — فقط type: union، tuple، نوع‌های محاسبه‌شده

`interface` فقط «شکل شیء/تابع» را بیان می‌کند. هر چیزی غیر از آن — mapped، conditional، tuple، union، template literal — قلمرو انحصاری `type` است:

type-only.ts — type-checks ✓

```ts
type Status   = "idle" | "loading" | "done";      // union — interface نمی‌تواند
type Pair     = [string, number];                  // tuple
type Nullable<T> = T | null;                       // جنریک روی union
type Keys     = `user:${string}`;                  // template literal (فصل ۱۹)
```

نتیجه عملی: در کدی که با الگوهای فصل‌های ۱۷ تا ۲۲ مدل‌سازی می‌کند، `type` ابزار اصلی است — نه به دلیل سلیقه، بلکه به دلیل توانایی.

تفاوت ۲ — فقط interface: declaration merging

دو اعلان interface هم‌نام، **ادغام** می‌شوند؛ دو `type` هم‌نام، خطای تعریف تکراری:

merging.ts

type-checks ✓

```
interface WindowConfig { theme: string }
interface WindowConfig { locale: string }

//  نتیجه: { theme: string; locale: string }
const cfg: WindowConfig = { theme: "dark", locale: "fa" };
```

این قابلیت در کد اپلیکیشن معمولاً **ضدقابلیت** است — نوعی که هر فایلی می‌تواند بی‌صدا گسترشش دهد، قابل اعتماد نیست. اما دقیقاً همین سازوکار است که اجازه می‌دهد نوع کتابخانه‌های دیگران را گسترش دهید: اضافه کردن property به `Window`، تزریق فیلد به `Request` در Express، یا تایپ کردن `process.env` — فصل ۲۵ با جزئیات.

نکته — تفاوت‌های کوچک‌تری که گاهی مهم می‌شوند

- خطاهای interface معمولاً با نام خودش گزارش می‌شوند و خطاهای type گاهی به شکل بازشده — در ساختارهای بزرگ، خوانایی خطا با interface کمی بهتر است.
- implements و extends روی interface مرسوم‌تر و پیام‌های خطایش تمیزتر است.
- کارایی کامپایل: interfaceها کمی cache پذیرترند؛ در مقیاس خیلی بزرگ محسوس می‌شود (فصل ۳۲).

قاعده پیشنهادی برای تیم

در عمل — قاعده‌ای که این کتاب دنبال می‌کند

- **شکل اشیاء عمومی و قراردادهای API:** `interface` — به‌خاطر خوانایی خطا، extends طبیعی و امکان augmentation برای مصرف‌کننده‌ها (در کتابخانه‌ها مزیت واقعی است).
- **همه نوع‌های دیگر:** `type` — tuple، union، مشتق‌ها و ترکیب‌ها.
- **داخل اپلیکیشن:** هر کدام که تیم انتخاب کرد — فقط یکدست. دعوای interface/type ارزش وقت review ندارد؛ ناسازگاری‌اش چرا.

یک هشدار درباره extends در برابر &: نتیجه معمولاً یکی است، اما وقتی propertyهای هم‌نام ناسازگار باشند رفتار فرق می‌کند — `extends` همان لحظه خطای واضح می‌دهد، در حالی که `&` بی‌صدا `never` می‌سازد و خطا را به محل استفاده عقب می‌اندازد (فصل ۹). یک دلیل دیگر برای ترجیح interface در سلسله‌مراتب اشیاء.

extends-vs-and.ts

× خطای عمدی

```ts
interface A { id: string }
interface B extends A { id: number }  //  × شفاف — خطا همین‌جا
```

پیام کامپایلر

```
extends-vs-and.ts:2:11 - error TS2430: Interface 'B' incorrectly
extends interface 'A'.
  Types of property 'id' are incompatible.
    Type 'number' is not assignable to type 'string'.
```

زیر کاپوت — چرا این‌طور است؟

تفاوت interface و type alias از تفاوت «هویت» آن‌ها در دل کامپایلر می‌آید — و وقتی این را ببینید، همه تفاوت‌های سطحی توضیح پیدا می‌کنند.

interface یک «اعلانِ نام‌دار و باز» است. کامپایلر برای آن یک symbol واقعی می‌سازد که اعضایش می‌توانند در چند جا تعریف شوند — این همان declaration merging است. باز بودن یک تصمیم عمدی و اکوسیستمی است: بدون آن، نه `Window` قابل گسترش بود، نه ماژول augmentation فصل ۲۵ معنا داشت، نه کتابخانه‌ها می‌توانستند به interface های یکدیگر عضو اضافه کنند. بهای این قدرت: merge شدن «از راه دور» می‌تواند غافلگیرکننده باشد — interface در فایل شما ممکن است جای دیگری عضو گرفته باشد.

type alias یک «نامِ مستعار برای یک عبارت نوعی» است. بسته و تغییرناپذیر — همین بستگی است که به آن آزادی می‌دهد به هر چیزی اشاره کند: mapped type، conditional، tuple، union. interface نمی‌تواند union باشد چون «بدنه قابل merge» باید شکل عضو-محور داشته باشد؛ union بدنه ندارد.

تفاوت عملکردی ظریف‌تر: interface ها **تنبل (lazy)** بررسی می‌شوند و همیشه با نامشان در خطاها و cache کامپایلر ظاهر می‌شوند؛ alias های پیچیده گاهی به شکل بازشده در پیام خطا پهن می‌شوند. برای قراردادهای شیئی بزرگ و پرارجاع، interface پیام خطای خواناتر و رفتار cache بهتری می‌دهد — این پایه توصیه رایج «شیء؟ interface؛ بقیه؟ type» است.

جمع‌بندی تصمیم‌گیری: سؤال درست «کدام مُد است؟» نیست؛ «آیا این قرارداد باید باز و قابل‌گسترش باشد (interface) یا عبارتی محاسباتی و بسته (type)؟» — پاسخ این سؤال، انتخاب را تقریباً همیشه قطعی می‌کند.

تمرین

انتخاب ابزار درست

برای هر مورد، interface یا type؟ و چرا؟

۱. مدل `Product` در پکیج عمومی SDK شما.

۲. `ApiResponse = Success | Failure` .

۳. گسترش `ProcessEnv` برای متغیرهای محیطی پروژه.

۴. نوع کمکی `WithTimestamps<T>` که به هر مدلی دو فیلد اضافه می‌کند.

پاسخ: ۱. interface — قرارداد عمومی شیئی، قابل augment توسط مصرف‌کننده. ۲. union — type است؛ interface نمی‌تواند. ۳. declaration merging — interface + تنها راه استاندارد (فصل ۲۵). ۴. type — ترکیب جنریک با &: `type WithTimestamps<T> = T & { createdAt: Date; updatedAt: Date }`.

پرسش‌های مصاحبه این فصل

۱. قاعده تیمی‌تان برای interface در برابر type چیست و چرا؟

قراردادهای شیئی عمومی (props، مدل‌ها، پارامترهای سرویس) — interface به‌خاطر extends تمیز و خطای صریح تضاد؛ هر چیز ترکیبی (union، tuple، mapped، conditional) — type چون interface اصلاً نمی‌تواند. مهم‌تر از خودِ انتخاب، یکدستی در codebase است.

۲. declaration merging کی نعمت است و کی مصیبت؟

نعمت: augmentation تایپ کتابخانه‌ها (`req.user`، `ProcessEnv`) — اصلاً برای همین طراحی شده. مصیبت: merge تصادفی دو interface هم‌نام در scope مشترک که بی‌صدا اعضایشان جمع می‌شود — یکی از دلایلی که در فایل‌های global/script محتاط باشید.

جنریک‌ها — پارامتر نوع، Constraint و Default

متغیرِ نوع: همان منطق پارامتر تابع، یک طبقه بالاتر

اهداف یادگیری این فصل

- جنریک را به‌عنوان «تابع در سطح نوع» توضیح دهی
- با extends دامنه پارامتر نوع را محدود کنی
- بدانی کامپایلر آرگومان نوع را از کجا استنتاج می‌کند

تابع، «مقدار» را پارامتری می‌کند؛ جنریک، «نوع» را. همین. اگر این جمله را عمیق بفهمید، جنریک‌ها از «syntax ترسناک با براکت» به ابزاری بدیهی تبدیل می‌شوند — و این فصل برای همین نوشته شده است.

مسئله‌ای که جنریک حل می‌کند

بدون جنریک، بین دو بدی گیر می‌کنید:

why-generics.ts — type-checks ✓

```ts
// بدی ۱: برای هر نوع یک کپی — تکرار
function firstOfStrings(xs: string[]): string | undefined { return xs[0]; }
function firstOfNumbers(xs: number[]): number | undefined { return xs[0]; }

// بدی ۲: any — از دست دادن اطلاعات
function firstAny(xs: any[]): any { return xs[0]; }
const n = firstAny([1, 2, 3]);   // n: any — رابطه ورودی/خروجی گم شد

// راه‌حل: نوع را پارامتر کن
function first<T>(xs: readonly T[]): T | undefined { return xs[0]; }

const a = first([1, 2, 3]);        // T = number     → a: number | undefined
const b = first(["x", "y"]);       // T = string     → b: string | undefined
const c = first([{ id: 1 }]);      // T = { id: number }
void a; void b; void c;
```

`<T>` اعلامِ یک **متغیر نوع** است: «این تابع با نوعی کار می‌کند که هنگام استفاده مشخص می‌شود». قدرت جنریک در حفظ **رابطه‌ها** است: «خروجی از جنس عنصر ورودی است» — چیزی که `any` نابود می‌کند و کپی‌کاری چندبرابر می‌کند.

استنتاج آرگومان نوع — از کجا؟

تقریباً هیچ‌وقت `first<number>([1,2])` نمی‌نویسید؛ کامپایلر `T` را از **آرگومان‌ها** استنتاج می‌کند. قاعده: کامپایلر جای ظاهر شدن `T` در امضا را با نوع آرگومان واقعی تطبیق می‌دهد:

inference-sites.ts

type-checks ✓

```ts
declare function map<T, U>(xs: T[], f: (x: T) => U): U[];

const lens = map(["a", "bb"], s => s.length);
//  گام ۱: xs = string[] تطبیق با T[] ⇒ T = string
//  حالا :۲ گام f باید (x: string) ≥  U باشد ⇒ s: string (contextual!)
//  گام ۳: s.length ⇒ U = number ⇒ خروجی: number[]
```

به ترتیب گام‌ها دقت کنید: استنتاج `T` از آرگومان اول، به پارامتر callback «زمینه» داد (فصل ۴). این هم‌افزایی استنتاج و contextual typing است که جنریک‌های خوب را بی‌annotation می‌کند. آرگومان نوع صریح برای وقتی است که **هیچ آرگومان مقداری برای استنتاج نیست:** `const cache = new Map<string, User>()` .

Constraint — با extends دامنه را ببند

بدون محدودیت، درونِ بدنه فقط عملیاتی مجاز است که روی «هر نوعی» کار کند. constraint می‌گوید `T` حداقل چه شکلی دارد:

constraints.ts

type-checks ✓

```ts
interface HasId { id: string | number }

function indexById<T extends HasId>(items: readonly T[]): Map<T["id"], T> {
  const m = new Map<T["id"], T>();
  for (const it of items) m.set(it.id, it);    // ✓ it.id قول دادیم — وجود دارد
  return m;
}

const users = indexById([{ id: 1, name: "Ali" }, { id: 2, name: "Sara" }]);
// T = { id: number; name: string } — نه فقط HasId! جزئیات حفظ شد
const ali = users.get(1);    // { id: number; name: string } | undefined
void ali;
```

نکته ظریف و مهم: `T` به `HasId` **تبدیل نمی‌شود**؛ فقط ملزم است آن را شامل باشد. برای همین `users.get(1)` نوع کاملِ عضو را برمی‌گرداند. این تفاوتِ جنریکِ constraint دار با پارامترِ ساده `HasId` است:

generic-vs-plain.ts

type-checks ✓

```ts
interface HasId { id: number }
// بدون جنریک — جزئیات ورودی در خروجی گم می‌شود:
function pickPlain(x: HasId): HasId { return x; }
const input = { id: 1, name: "Ali" };
const p = pickPlain(input);
// p: HasId — نام از دید نوع‌ها پاک شد!

// با جنریک — جزئیات عبور می‌کند:
function pickGeneric<T extends HasId>(x: T): T { return x; }
const g = pickGeneric({ id: 1, name: "Ali" });
// g: { id: number; name: string }
void p; void g;
```

مدل ذهنی — جنریک یعنی «رابطه»، constraint یعنی «حداقل»

هر وقت خواستید جنریک بنویسید بپرسید: «چه **رابطه‌ای** را می‌خواهم حفظ کنم؟» (ورودی↔خروجی، کلید↔مقدار، آرگومان↔callback). اگر هیچ رابطه‌ای نیست، جنریک لازم نیست. constraint هم فقط «حداقل لازم برای بدنه» است — نه بیشتر؛ constraint چاق، ورودی‌های قانونی را بی‌دلیل رد می‌کند.

Default — پیش‌فرض پارامتر نوع

defaults.ts

type-checks ✓

```ts
interface ApiError { code: string; message: string }

interface Paginated<T, E = ApiError> {
  items: T[];
  total: number;
  error?: E;
}

type UsersPage = Paginated<{ id: number }>;          // E = ApiError
type CustomPage = Paginated<string, { reason: string }>;
```

default برای «نقطه شروع معقول با امکان سفارشی‌سازی» است — پرکاربرد در کتابخانه‌ها (مثلاً `Component<P = {}, S = {}>` در React قدیم). ترتیب مهم است: پارامترهای default دار بعد از بقیه.

هشدار — جنریکِ بی‌رابطه ننویس

امضای `function log<T>(x: T): void` هیچ رابطه‌ای حفظ نمی‌کند — `unknown` کافی و صادق‌تر است. قاعده سرانگشتی رسمی: پارامتر نوعی که فقط **یک بار** در امضا ظاهر می‌شود، تقریباً همیشه حذف‌شدنی است. جنریک اضافی، API را ترسناک‌تر و خطاها را بدتر می‌کند بدون هیچ سودی.

زیر کاپوت — چرا این‌طور است؟

دو حقیقت زیربنایی generics را اگر نگه دارید، بقیه فصل «نتیجه» می‌شود:

۱) پارامتر نوع در runtime هیچ است. `identity<string>` و `identity<number>` یک تابع‌اند — erasure کامل. پس جنریک «قالب تولید کد» نیست (برخلاف template های C++ که برای هر نوع، کد جدا می‌سازند)؛ صرفاً «رابطه بین ورودی و خروجی» است که کامپایلر ثبت و اعمال می‌کند. از همین‌جا یک آزمون طلایی برای طراحی API درمی‌آید: **پارامتر نوعی که فقط یک‌بار در امضا ظاهر می‌شود، تقریباً همیشه اشتباه است** — رابطه‌ای برقرار نمی‌کند، فقط any پنهان است. جنریک خوب حداقل دو نقطه را به هم گره می‌زند: ورودی به خروجی، آرگومان به آرگومان، property به متد.

۲) استنتاج آرگومان نوع، «جمع کاندیدا و انتخاب» است. کامپایلر جایگاه‌های حاوی T در امضا را با نوع آرگومان‌های واقعی جفت می‌کند؛ از هر جفت، کاندیدایی برای T استخراج می‌شود؛ سپس از کاندیداها بهترین (معمولاً عمومی‌ترین کاندیدای سازگار) انتخاب و در کل امضا جایگذاری می‌شود. وقتی استنتاج «اشتباه» می‌کند — مثلاً T دقیق‌تر یا عام‌تر از انتظار درمی‌آید — تقریباً همیشه یکی از این دوتاست: کاندیداهای ناسازگار از دو جایگاه مختلف آمده‌اند، یا literal در جایگاه بدون انتظار widen شده (فصل ۵). آرگومان صریح `fn<Type>(…)` یعنی «استنتاج را کنار بگذار» — ابزار درستش همین لحظه‌هاست، نه همه‌جا.

و constraint؟ `T extends X` دو کار هم‌زمان می‌کند: به بدنه تابع «حداقل قابلیت‌های T» را می‌دهد (بدون آن، T مثل unknown است و هیچ عملیاتی مجاز نیست)، و به استنتاج «سقف جستجو» می‌دهد. constraint را حداقلی نگه دارید: هر چه تنگ‌تر، فراخواننده‌های کمتری را می‌پذیرید — قاعده «سخاوت در ورودی» این‌جا هم جاری است.

تمرین

استنتاج را دنبال کن

نوع نتیجه و مقدار هر پارامتر نوع را مشخص کنید:

quiz14.ts — × خطای عمدی

```ts
declare function zip<A, B>(a: A[], b: B[]): [A, B][];
declare function wrap<T extends { length: number }>(x: T): { value: T; size: number };

const z = zip([1, 2], ["a", "b"]);
const w = wrap("hello");
const bad = wrap(42);
```

پاسخ: `z: [number, string][]` با A=number و B=string · `w: { value: string; size: number }` — رشته "literal "hello") T = string به string پهن می‌شود چون جایگاه جنریک معمولی است) · `bad` : خطای TS2345 — `number` محدودیت `{ length: number }` را برآورده نمی‌کند.

پرسش‌های مصاحبه این فصل

۱. چه زمانی جنریک «نباید» نوشت؟

وقتی رابطه‌ای برای حفظ کردن نیست. `<T>(x: T) ⇒ number` که فقط طول می‌گیرد، جنریک بیهوده است — `{ length: number }` کافی است. جنریک وقتی می‌ارزد که نوع خروجی/اعضا به نوع ورودی «وابسته» باشد و بخواهید آن وابستگی از دست نرود.

۲. constraint یعنی «دقیقاً این نوع»؟

نه — `T extends HasId` یعنی «حداقل این شکل»؛ آرگومان کامل‌تر، با همه جزئیاتش در T حفظ می‌شود. همین تفاوتِ با پارامتر ساده `HasId` است که همه‌چیز را به حداقل فرومی‌کاهد. در مصاحبه این تمایز را با مثال خروجی بازگشتی نشان دهید.

جنریک در عمل — API و کامپوننت قابل‌استفاده مجدد

از تابع جنریک تا کلاس، interface جنریک و الگوی Builder

اهداف یادگیری این فصل

- API جنریک طراحی کنی که در نقطه استفاده annotation نخواهد
- خطای رایج «جنریک بی‌فایده» را تشخیص دهی و حذف کنی
- یک abstraction داده‌محور جنریک (repository) بسازی

فصل قبل مکانیک جنریک را داد؛ این فصل مهندسی‌اش را: چطور APIهایی طراحی کنیم که در نقطه استفاده «خودشان کار کنند» — بدون annotation، بدون as، با خطاهای قابل فهم. معیار کیفیت یک API جنریک، تجربه مصرف‌کننده‌اش است، نه هوشمندی پیاده‌سازی‌اش.

الگوی ۱ — تابع جنریک با قرارداد رفت‌وبرگشتی

کلاسیک‌ترین الگو: ساختار نگه‌دارنده که نوع محتوا را حفظ می‌کند.

result-helpers.ts — type-checks ✓

```ts
type Result<T, E = Error> =
  | { ok: true;  value: T }
  | { ok: false; error: E };

const ok  = <T,>(value: T): Result<T, never> => ({ ok: true, value });
const err = <E,>(error: E): Result<never, E> => ({ ok: false, error });

function tryParse(json: string): Result<unknown, SyntaxError> {
  try { return ok(JSON.parse(json)); }
  catch (e) { return err(e as SyntaxError); }
}

const r = tryParse('{"a":1}');
if (r.ok) console.log(r.value);    // unknown — صادقانه؛ فصل ۳۴ کاملش می‌کند
else      console.error(r.error.message);  // SyntaxError
```

به `Result<T, never>` در خروجی `ok` دقت کنید: چون `never` به همه چیز قابل انتساب است، `ok(5)` در هر جا می‌گیرد. استفاده آگاهانه از never در جنریک‌ها، امضای «حداکثر سازگار» می‌سازد. `Result<number, AnyError>`

الگوی ۲ — interface جنریک به‌عنوان قرارداد لایه

repository.ts

type-checks ✓

```ts
interface Entity { id: string }

interface Repository<T extends Entity> {
  findById(id: string): Promise<T | null>;
  findAll(filter?: Partial<T>): Promise<T[]>;
  save(entity: T): Promise<T>;
  delete(id: string): Promise<boolean>;
}

//  یک پیاده‌سازی عمومی — برای تست و نمونه‌سازی سریع
class InMemoryRepository<T extends Entity> implements Repository<T> {
  private store = new Map<string, T>();

  async findById(id: string) { return this.store.get(id) ?? null; }
  async findAll(filter?: Partial<T>) {
    const all = [...this.store.values()];
    if (!filter) return all;
    return all.filter(e =>
      (Object.keys(filter) as (keyof T)[]).every(k => e[k] === filter[k]));
  }
  async save(entity: T) { this.store.set(entity.id, entity); return entity; }
  async delete(id: string) { return this.store.delete(id); }
}

interface Product extends Entity { name: string; price: number }
const products: Repository<Product> = new InMemoryRepository<Product>();

await products.save({ id: "p1", name: "Keyboard", price: 1200 });
const cheap = await products.findAll({ price: 1200 });  // Product[]
void cheap;
export {};
```

این **پروژه ۳** کتاب در نسخه فشرده است (نسخه کامل با نسخه Postgres فرضی و تست در `examples/repository`).

نکته‌های طراحی:

- `T extends Entity` حداقلِ لازم است — repository فقط به `id` نیاز دارد.
- `Partial<T>` برای فیلتر، بدون هیچ نوع جدیدی، «زیرمجموعه‌ای از فیلدها» را بیان می‌کند (فصل ۲۰).
- cast `as (keyof T)[]` روی `Object.keys` ضروری است — چون `Object.keys` عمداً `string[]` برمی‌گرداند (شیء ممکن است عضو اضافه ساختاری داشته باشد؛ فصل ۶). این یکی از جاهای معدودی است که cast قابل دفاع است و باید کامنت داشته باشد.

الگوی ۳ — جای annotation، استنتاج از ساختار

API خوب، نوع را از **داده پیکربندی** استنتاج می‌کند، نه از پارامتر صریح:

form-builder.ts

type-checks ✓

```ts
interface FieldSpec<T> { parse: (raw: string) => T; label: string }

// استنتاج می‌شود spec کل فرم از روی:
function buildForm<S extends Record<string, FieldSpec<unknown>>>(spec: S) {
  return {
    parse(raw: Record<keyof S, string>): { [K in keyof S]: ReturnType<S[K]["parse"]> } {
      const out: Record<string, unknown> = {};
      for (const [key, field] of Object.entries(spec)) {
        out[key] = field.parse(raw[key as keyof S]);
      }
      // متمرکز cast — نمی‌تواند به کامپایلر اثبات کند runtime را حلقه mapped قرارداد:
      return out as { [K in keyof S]: ReturnType<S[K]["parse"]> };
    },
  };
}

const form = buildForm({
  age:   { label: "سن",    parse: raw => Number(raw) },
  email: { label: "ایمیل", parse: raw => raw.trim().toLowerCase() },
});

const values = form.parse({ age: "30", email: " A@B.dev " });
// values: { age: number; email: string } — از spec مشتق‌شده!
void values;
```

مصرف‌کننده فقط داده داد؛ همه نوع‌ها خودشان آمدند. این الگوی «configuration-first» است که کتابخانه‌های محبوب (Zod، TanStack) را دوست‌داشتنی می‌کند — و ترکیبی است از mapped type، constraint (فصل ۱۸) و `ReturnType` (فصل ۲۰).

مدل ذهنی — جنریک خوب، نامرئی است

مصرف‌کننده API جنریک خوب، هرگز براکت نمی‌بیند: نه هنگام فراخوانی annotation می‌دهد، نه در خطاها با `T` و `K` سردرگم می‌شود. اگر کاربران API شما مجبورند آرگومان نوع صریح بدهند، معمولاً جای استنتاج را اشتباه طراحی کرده‌اید — نوع را از آرگومانی بگیرید که کاربر به‌هرحال می‌دهد.

هشدار — abstraction جنریک را زود نسازید

قانون سه‌بار: تا وقتی سه مورد استفاده واقعی ندیده‌اید، نسخه جنریک نسازید. جنریکی که برای یک مصرف‌کننده طراحی شود، شکل همان یک مصرف‌کننده را می‌گیرد و برای دومی باید شکسته شود — با هزینه مهاجرت. کپی موقت، از abstraction زودرس ارزان‌تر است.

زیر کاپوت — چرا این‌طور است؟

چرا TypeScript برای «بهبود استنتاج» این‌همه ابزار ظریف دارد — `NoInfer`، `const` type parameter، ترتیب آرگومان‌ها؟ چون استنتاج یک الگوریتم مکانیکی با نقاط تصمیم مشخص است و این ابزارها «شیر تنظیم» همان نقاطند. سه نقطه تصمیم را بشناسید تا ابزارها بدیهی شوند:

نقطه ۱: کدام جایگاه‌ها در استنتاج «رأی» دارند؟ پیش‌فرض: همه جایگاه‌های حاوی T. مشکل کلاسیک: در `pick(animals, default)` می‌خواهید T فقط از آرایه بیاید، نه از default — وگرنه مقدار پیش‌فرضِ خارج از مجموعه، union را گشاد می‌کند و خطای مطلوب گم می‌شود. `NoInfer<T>` دقیقاً «حق رأی این جایگاه را بگیر» است: آن پارامتر فقط «بررسی» می‌شود، در «انتخاب» شرکت نمی‌کند.

نقطه ۲: literal ها widen بشوند یا نه؟ پیش‌فرض استنتاج، literal های تازه را به‌سمت نوع عمومی می‌برد (همان منطق فصل ۵). وقتی API شما به literal ها نیاز دارد — مسیرهای route، نام event ها — `const` type parameter به کامپایلر می‌گوید: «آرگومان‌های این جایگاه را مثل as const نگه دار». پیش از TS 5.0 این کار به دوش فراخواننده بود (as const دستی)؛ حالا طراح API می‌تواند تضمینش کند.

نقطه ۳: استنتاج به چه ترتیبی جاری می‌شود؟ چپ به راست: آرگومان‌های اول، کاندیداهای T را می‌سازند و contextual typing آرگومان‌های بعدی (مثلاً callback) از همان کاندیداها تغذیه می‌شود. برای همین «آرایه اول، callback دوم» الگوی استاندارد است — برعکسش، callback بدون context می‌ماند و پارامترهایش implicit any می‌شوند.

درس طراحی: وقتی مصرف‌کننده API شما مجبور است آرگومان نوع را دستی بنویسد، معمولاً یکی از این سه نقطه را درست تنظیم نکرده‌اید — امضا را تغییر دهید، نه مصرف‌کننده را.

تمرین

API را قابل استنتاج کن

این API کاربر را مجبور به annotation می‌کند. بازطراحی کنید تا نوع از آرگومان استنتاج شود:

quiz15.ts — type-checks ✓

```
declare function createStore<State>(): {
  get(): State; set(s: State): void;
};
// ✓ استفاده فعلی: createStore<{ count: number }>()  ← براکت اجباری
```

پاسخ: نوع را از مقدار اولیه بگیرید — آرگومانی که کاربر به‌هرحال باید بدهد:

quiz15-solution.ts

type-checks ✓

```ts
function createStore<State>(initial: State) {
  let state = initial;
  return { get: () => state, set: (s: State) => { state = s; } };
}
const store = createStore({ count: 0 });  // State = { count: number } — استنتاج
store.set({ count: 5 });
```

پرسش‌های مصاحبه این فصل

۱. «جنریک خوب» از دید مصرف‌کننده چه شکلی است؟

نامرئی. مصرف‌کننده آرگومان می‌دهد و همه‌چیز استنتاج می‌شود — هیچ براکت صریحی لازم نیست. اگر کاربران API شما مرتب `<Foo>` می‌نویسند، طراحی را بازبینی کنید: معمولاً جای پارامتر نوع در امضا اشتباه است و استنتاج راه ندارد.

۲. در طراحی factory چطور کاری می‌کنید که نوع از «استفاده» بیاید نه annotation؟

قرارداد را روی پارامترهای ورودی سوار می‌کنم تا استنتاج از آرگومان واقعی جاری شود — مثل `buildForm(spec)` که شکل کل فرم را از خود spec می‌گیرد، یا `defineConfig(cfg)` . مصرف‌کننده داده می‌دهد؛ نوع، محصول جانبی رایگان است.

keyof، typeof و Indexed Access

سه اپراتور کوچک که اتصال نوع‌ها به داده واقعی را ممکن می‌کنند

اهداف یادگیری این فصل

- با keyof و indexed access نوع را از ساختار داده مشتق کنی
- با typeof از مقدار runtime نوع بسازی و تکرار را حذف کنی
- الگوی «منبع حقیقت واحد» را با این اپراتورها پیاده کنی

سه اپراتور این فصل کوچک‌اند اما استراتژیک: پل بین «داده واقعی برنامه» و «نوع‌ها». با آن‌ها به‌جای تعریف نوع‌های موازی با داده — که دیر یا زود از هم فاصله می‌گیرند — نوع را **از خود داده مشتق می‌کنید**: یک منبع حقیقت، صفر ناهمگامی.

keyof — کلیدها به‌عنوان union

keyof-basics.ts — type-checks ✓

```ts
interface User { id: number; name: string; email: string }

type UserKey = keyof User;   // "id" | "name" | "email"

// کاربرد کلاسیک: getter امن که کلید و خروجی را به هم گره می‌زند
function getProp<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const u: User = { id: 1, name: "Ali", email: "a@b.dev" };
const n = getProp(u, "name");   // string — دقیق
const i = getProp(u, "id");     // number — دقیق
void n; void i;
```

keyof-err.ts — × خطای عمدی

```ts
interface User { id: number; name: string }
declare function getProp<T, K extends keyof T>(obj: T, key: K): T[K];
declare const u: User;

getProp(u, "age");  // × کلید ناموجود — typo در compile-time گرفته شد
```

پیام کامپایلر

```
keyof-err.ts:5:12 - error TS2345: Argument of type '"age"' is not
assignable to parameter of type 'keyof User'.
```

امضای `getProp` را حفظ کنید — سه ایده در یک خط: `K extends keyof T` (کلید معتبر)، `T[K]` (نوع همان کلید)، و رابطه بین دو پارامتر نوع. این امضای واقعی هزاران کتابخانه است، از lodash تا فرم‌سازها.

Indexed Access — T[K] نوعِ «مقدارِ آن کلید»

indexed.ts — type-checks ✓

```ts
interface ApiConfig {
  endpoints: { users: string; orders: string };
  retry: { attempts: number; delayMs: number };
}

type RetryConfig = ApiConfig["retry"];          // { attempts: number; delayMs: number }
type Endpoint    = ApiConfig["endpoints"]["users"];  // string

//  ایندکس کنید — همه مقادیر با union:
type AnySection  = ApiConfig[keyof ApiConfig];  // endpoints | retry

//  عنصر آرایه — الگوی بسیار پرکاربرد:
declare const roles: readonly ["admin", "editor", "viewer"];
type Role = (typeof roles)[number];             // "admin" | "editor" | "viewer"
```

`T[number]` را به خاطر بسپارید: «نوع عناصر آرایه/tuple». در ترکیب با `as const`، ستون الگوی بعدی است.

typeof — از دنیای مقدار به دنیای نوع

`typeof` تایپ‌اسکریپتی (در جایگاه نوع) نوعِ یک **مقدار موجود** را می‌دهد — پل رسمی دو فضای فصل ۱:

single-source.ts — type-checks ✓

```ts
//  منبع حقیقت: یک مقدار runtime واقعی
const PLANS = {
  free:    { maxProjects: 1,  price: 0 },
  pro:     { maxProjects: 20, price: 15 },
  team:    { maxProjects: 99, price: 45 },
} as const;

//  همه نوع‌ها مشتق می‌شوند:
type Plans    = typeof PLANS;
type PlanName = keyof Plans;                  // "free" | "pro" | "team"
type Plan     = Plans[PlanName];              // union سه شیء readonly

function price(name: PlanName): number {
  return PLANS[name].price;   //  ✓ بدون هیچ cast اضافه و بررسی
}
```

```
price("pro");
```

single-source-err.ts — خطای عمدی

```
declare function price(name: "free" | "pro" | "team"): number;
price("enterprise");  //  ✕ اضافه کن PLANS طرح جدید؟ اول به
```

پیام کامپایلر

```
single-source-err.ts:2:7 - error TS2345: Argument of type
'"enterprise"' is not assignable to parameter of type
'"free" | "pro" | "team"'.
```

گردش کار را ببینید: فردا `enterprise` به `PLANS` اضافه می‌شود — و `PlanName`، اعتبارسنجی `price`، تکمیل خودکار ادیتور و همه مصرف‌کننده‌ها **خودکار** به‌روز می‌شوند. هیچ enum موازی، هیچ لیست دستی، هیچ ناهمگامی.

مدل ذهنی — داده بالادست، نوع پایین‌دست

هر وقت نوعی را دست‌نویس کردید که «باید با فلان شیء/آرایه هماهنگ بماند»، علامت خطر است: دو منبع حقیقت دارید. جهت درست: مقدار را با `as const` دقیق نگه دارید و نوع را با `typeof` + `keyof` + `[number]` مشتق کنید. ناهمگامی از «ممکن» به «غیرقابل بیان» تبدیل می‌شود.

هشدار — Object.keys عمداً string[] است

انتظار دارید `Object.keys(user)` نوع `(keyof User)[]` بدهد؛ اما `string[]` می‌دهد — و این عمدی است: به دلیل سازگاری ساختاری (فصل ۶)، شیء ممکن است در runtime کلیدهای اضافه داشته باشد. اگر مطمئنید، cast صریح با کامنت بنویسید؛ cast را داخل یک helper کوچک (`typedKeys`) کپسوله کنید تا در کد پخش نشود.

زیر کاپوت — چرا این‌طور است؟

چرا `Object.keys` در lib.d.ts خروجی `string[]` دارد و نه `(keyof T)[]` ؟ این «محدودیت» نیست — **درستی** است، و دلیلش همان width subtyping فصل ۶ است: مقداری که نوعش T است، در runtime مجاز است propertyهای اضافه‌ای داشته باشد که در T نیستند. اگر keys نوع `(keyof T)[]` برمی‌گرداند، کلیدهای واقعاً موجودِ اضافه، با نوعی برچسب می‌خوردند که عضوشان نیست — دروغ ایستا با پیامد runtime. تیم TypeScript بین «راحتی» و «صداقت» صداقت را انتخاب کرد؛ و الگوی عملی‌اش این شد که هر جا «مالک کامل» شیء هستید (literal خودتان، نه ورودی بیرونی)، باریک‌سازی با `as (keyof T)[]` را خودتان و آگاهانه امضا کنید — مسئولیت دقیقاً همان‌جا که دانش هست.

خود `keyof` را هم مکانیکی ببینید: عملگری که از «جدول اعضای» یک نوع، union کلیدهای literal می‌سازد — پل میان دنیای ساختار و دنیای union. اهمیتش معماری است: همه الگوهای «کلید امن» (`T[K]` ، mapped types فصل ۱۸، getter جنریک) از همین پل عبور می‌کنند. و چون union است، همه جبر فصل ۹ رویش کار می‌کند: `keyof (A & B) = keyof A | keyof B` (اعضای بیشتر، کلیدهای بیشتر) و قرینه‌اش `keyof (A | B) = keyof A & keyof B` (فقط

کلیدهای مشترک قطعی‌اند) — دو اتحاد که در نگاه اول وارونه به‌نظر می‌رسند ولی دقیقاً منطق «در همه جهان‌های ممکن امن» فصل ۹ هستند.

typeof (نسخه type-level) هم قطعه سوم پازل است: «حقیقت از مقدار به نوع». ترکیب `keyof typeof obj` یعنی «کلیدهای واقعیِ همین شیء موجود» — الگوی single source of truth که config و ثابت‌ها را بدون تکرارِ دستیِ نوع، منبع نوع می‌کند. هر بار نوع و مقدار را جدا نگه داشتید و ناهمگام شدند، این پل را فراموش کرده بودید.

تمرین

مشتق‌سازی به‌جای تکرار

بدون نوشتن هیچ union دستی، نوع‌های خواسته‌شده را از این داده مشتق کنید:

quiz16.ts — type-checks ✓

```ts
const ROUTES = {
  home:    { path: "/", armed: false },
  login:   { path: "/login", armed: false },
  admin:   { path: "/admin", armed: true },
} as const;
//  خواسته ۱: RouteName = "home" | "login" | "admin"
//  خواسته ۲: RoutePath = "/" | "/login" | "/admin"
```

پاسخ:

quiz16-solution.ts — type-checks ✓

```ts
const ROUTES = {
  home:  { path: "/", armed: false },
  login: { path: "/login", armed: false },
  admin: { path: "/admin", armed: true },
} as const;

type RouteName = keyof typeof ROUTES;
type RoutePath = (typeof ROUTES)[RouteName]["path"];
```

زنجیره خواسته ۲ را باز کنید: `typeof ROUTES` کل شیء، ایندکس با union نام‌ها ⇒ union سه شیء، ایندکس با "path" ⇒ union سه literal مسیر. همین زنجیره سه‌حلقه‌ای، ۹۰٪ مشتق‌سازی‌های دنیای واقعی است.

پرسش‌های مصاحبه این فصل

۱. چرا `Object.keys` خروجی `string[]` دارد نه `(keyof T)[]`؟ راه حرفه‌ای؟

چون نوع‌ها exact نیستند — مقدار ممکن است در runtime کلیدهایی بیش از نوع ایستا داشته باشد و `(keyof T)[]` دروغ شود. راه حرفه‌ای: helper متمرکز `typedKeys` با cast مستند، فقط برای اشیائی که مالک ساختشان هستید — نه داده بیرونی.

۲. الگوی «منبع حقیقت واحد» با keyof/typeof را توضیح دهید.

داده را یک بار `as const` تعریف می‌کنید و نوع‌ها را از آن «مشتق» می‌کنید: `type Role = keyof typeof ROLES`. افزودن عضو جدید به داده، همه نوع‌ها و exhaustive check ها را خودکار به‌روز می‌کند — هیچ لیست موازی‌ای که از sync خارج شود وجود ندارد.

Conditional Types و infer

if/else در سطح نوع — و استخراج نوع از دل ساختارها

اهداف یادگیری این فصل

- یک conditional type را مرحله‌به‌مرحله ارزیابی کنی
- رفتار توزیعی (distributive) روی union را پیش‌بینی کنی
- با infer نوع را از تابع، آرایه یا Promise استخراج کنی

تا این‌جا نوع‌ها «ایستا» بودند: تعریف می‌شدند و همان می‌ماندند. conditional typeها به سیستم نوع، **عبارت شرطی** می‌دهند: نوعی که بر اساس یک پرسش assignability، بین دو شاخه انتخاب می‌کند. این سازوکارِ پشت `ReturnType`، `Awaited`، `Exclude` و تقریباً تمام utilityهای جدی است.

نحو و معنای دقیق

conditional-syntax.ts — type-checks ✓

```ts
type IsString<T> = T extends string ? true : false;

type A = IsString<"hello">;  // true
type B = IsString<42>;       // false
```

عبارت `T extends U ? X : Y` یک پرسش دقیق است: «آیا هر مقدارِ از نوع `T`، قابل انتساب به `U` است؟» اگر بله، نتیجه `X`؛ وگرنه `Y`. دقت کنید این همان `extends` ارث‌بری کلاس‌ها نیست — این **پرسش سازگاری** است، با همان قواعد ساختاری فصل ۶.

رفتار توزیعی روی Union — مهم‌ترین قاعده این فصل

وقتی `T` یک **پارامتر نوع برهنه** (naked) باشد و union بگیرد، شرط روی تک‌تک اعضا اجرا و نتیجه‌ها union می‌شوند:

ورودی

```ts
type ToArray<T> = T extends unknown ? T[] : never;
type R = ToArray<string | number>;
```

گام ۱ — توزیع روی اعضای union

```ts
ToArray<string> | ToArray<number>
```

گام ۲ — ارزیابی هر شاخه

```
string[] | number[]
```

نتیجه – و نکته

```
R = string[] | number[]   // نه (string | number)[] !
```

اگر توزیع را **نخواهید**، پارامتر را از «برهنگی» درآورید – متداول‌ترین ترفند: tuple کردن دو طرف:

no-distribute.ts — type-checks ✓

```
type ToArrayAll<T> = [T] extends [unknown] ? T[] : never;
type R2 = ToArrayAll<string | number>;  // (string | number)[]
```

هشدار – never عضو خاموش union است

توزیع روی `never` (union خالی) هیچ شاخه‌ای تولید نمی‌کند؛ پس `IsString<never>` برابر `never` است، نه `false`. اگر تست نوع می‌نویسید و با never نتیجه عجیب می‌گیرید، اول همین را چک کنید – الگوی `[T] extends [never]` راه استاندارد تشخیص never است.

infer – استخراج نوع از دل ساختار

`infer` در شاخه شرط، یک متغیر نوع جدید معرفی می‌کند و از کامپایلر می‌خواهد آن را از ساختار `T` **پیدا کند**:

infer-basics.ts — type-checks ✓

```
type ElementOf<T>  = T extends (infer E)[] ? E : never;
type UnwrapPromise<T> = T extends Promise<infer V> ? V : T;
type FirstArg<F>   = F extends (a: infer A, ...rest: any[]) => any ? A : never;

type E1 = ElementOf<number[]>;                  // number
type V1 = UnwrapPromise<Promise<string>>;       // string
type V2 = UnwrapPromise<boolean>;               // boolean – شاخه else
type A1 = FirstArg<(id: string, n: number) => void>; // string
```

این دقیقاً سازوکار `ReturnType` استاندارد است:

returntype-impl.ts — type-checks ✓

```
type MyReturnType<T extends (...args: any) => any> =
  T extends (...args: any) => infer R ? R : never;

type R3 = MyReturnType<typeof JSON.parse>;  // any – چون امضای parse چنین است
type R4 = MyReturnType<() => Date>;         // Date
```

مدل ذهنی — infer یعنی pattern matching

به شاخه شرط مثل یک «الگو» نگاه کنید: `Promise<infer V>` یعنی «اگر T با الگوی Promise از چیزی جور شد، آن چیز را V بنام». کامپایلر matching را انجام می‌دهد؛ شما فقط جای خالی را نام‌گذاری می‌کنید — همان destructuring، در سطح نوع.

یک مشتق واقعی: نوع پاسخ از نوع route

ترکیب conditional و infer جایی می‌درخشد که بخواهید از یک قرارداد موجود، نوع مشتق کنید:

api-derive.ts — type-checks ✓

```ts
type Routes = {
  "GET /users":     { response: { id: number; name: string }[] };
  "GET /users/:id": { response: { id: number; name: string } };
  "POST /users":    { body: { name: string }; response: { id: number } };
};

type ResponseOf<R extends keyof Routes> =
  Routes[R] extends { response: infer Res } ? Res : never;

type BodyOf<R extends keyof Routes> =
  Routes[R] extends { body: infer B } ? B : undefined;

declare function call<R extends keyof Routes>(
  route: R,
  body: BodyOf<R>
): Promise<ResponseOf<R>>;

const users = await call("GET /users", undefined);
//    ^? { id: number; name: string }[]  — مشتق‌شده، نه دست‌نویس
```

قرارداد `Routes` **منبع حقیقت واحد** است؛ بقیه نوع‌ها از آن مشتق می‌شوند و با هر تغییر route، همه مصرف‌کننده‌ها هم‌زمان به‌روز — یا در صورت ناسازگاری، قرمز — می‌شوند. فصل ۳۵ همین الگو را به یک کلاینت کامل تبدیل می‌کند.

در عمل — کی conditional type ننویسیم

اگر شرط فقط دو حالت ثابت دارد و در یک‌جا مصرف می‌شود، دو type alias ساده بنویسید. conditional type وقتی ارزش دارد که **قاعده** باشد نه **مورد خاص** — قاعده‌ای که روی ورودی‌های متنوع اعمال می‌شود. خطای چهل‌خطی ناشی از conditionalهای تودرتو، هزینه‌ای واقعی در تیم است (فصل ۲۶).

زیر کاپوت — چرا این‌طور است؟

conditional type را «if نوعی» معرفی کردیم، اما رفتار واقعی‌اش دو سازوکار دارد که بدون فهمشان، در اولین خروجی غیرمنتظره گم می‌شوید:

۱) توزیع = map روی اعضای union. وقتی پارامتر نوعِ «برهنه» (بدون پوشش) در سمت چپ extends بیاید و union بگیرد، شرط یک‌بار ارزیابی نمی‌شود — روی تک‌تک اعضا اجرا و نتایج union می‌شوند. چرا این پیش‌فرض است؟ به معنای مجموعه‌ای برگردید: union یعنی «شاید این، شاید آن». سؤالِ «آیا A | B شرط را می‌گذراند؟» برای هر عضو پاسخ جدایی دارد؛ پاسخ صادقانه، اجتماعِ پاسخ‌هاست. `Exclude` و `Extract` و NonNullable همگی فقط همین توزیع‌اند. قاعده «برهنگی» هم مرز خروج است: پوشاندن دو طرف در tuple — `[T] extends [U]` — یعنی «union را یکجا، به‌عنوان یک مجموعه واحد، بسنج». هر دو رفتار گاهی درستند؛ زبان هر دو را در اختیارتان گذاشته و پیش‌فرض را روی پرکاربردتر گذاشته است.

۲) تعویق (deferral). اگر T هنوز آزاد باشد (داخل بدنه تابع جنریک)، کامپایلر شرط را ارزیابی نمی‌کند — نمی‌تواند؛ T هنوز مقدار ندارد. نوعِ «معلق» با هیچ شاخه مشخصی قابل انتساب نیست و این منبع خطاهای گیج‌کننده «T is not assignable to Conditional» درون توابع جنریک است. راه‌حل‌های سالم: overload، یا جابه‌جایی شرط به بیرون امضا. این محدودیت تصادفی نیست — ارزیابی زودهنگام شرطِ وابسته به T آزاد، ناسالم می‌بود.

و `infer`؟ آن را «تطبیق الگو با متغیر گیرنده» ببینید: در جایگاه infer R، کامپایلر همان ماشینِ جمع کاندیدای فصل ۱۴ را اجرا می‌کند — چند جایگاه infer هم‌نام، کاندیداهایشان ادغام می‌شود. ReturnType و Parameters و Awaited چیزی جز همین الگوبرداری نیستند؛ Awaited فقط بازگشتی‌اش کرده است.

تمرین

سه مشتق — با قلم و کاغذ

نتیجه هر کدام را مرحله‌به‌مرحله (مثل جدول مشتق بالا) به‌دست آورید:

quiz17.ts — type-checks ✓

```ts
type X1 = Exclude<"a" | "b" | "c", "a">;
type X2 = NonNullable<string | null | undefined>;
type X3 = Awaited<Promise<Promise<number>>>;
```

پاسخ: `X1 = "b" | "c"` — توزیع، شاخه‌های برابر با "a" به never می‌روند و از union حذف می‌شوند · `X2 = string` — همان سازوکار برای null/undefined · `X3 = number` — Awaited بازگشتی است و لایه‌های تودرتو را باز می‌کند.

پرسش‌های مصاحبه این فصل

۱. conditional توزیعی کی به نفع شماست و چطور خاموشش می‌کنید؟

به نفع: وقتی می‌خواهید تبدیل، عضوبه‌عضو روی union اعمال شود — مثل `Exclude`. مزاحم: وقتی می‌خواهید union را «یک‌جا» بسنجید. خاموشی: دو طرف را tuple کنید — `[T] extends [U]`. و حواستان به حالت مرزی never باشد که توزیعش نتیجه را never می‌کند.

۲. `infer` کجا مجاز است و پرکاربردترین الگوهایش؟

فقط در سمت extends یک conditional. الگوهای نان‌وآب‌دار: استخراج خروجی تابع (ReturnType)، عنصر آرایه (`T extends (infer E)[]`)، مقدار Promise (`Awaited` ساده)، و آرگومان‌ها (Parameters). ذهنیت درست: pattern matching سطح نوع.

Mapped Types و Key Remapping

حلقه for روی کلیدها: ساخت نوع از نوع

اهداف یادگیری این فصل

- mapped type بنویسی که propertyها را تغییر شکل دهد
- با as کلیدها را بازنویسی یا فیلتر کنی
- بفهمی utility typeهای استاندارد چگونه ساخته شده‌اند

اگر «conditional type if» دنیای نوع‌ها بود، mapped type حلقه «for» آن است: روی کلیدهای یک نوع قدم می‌زند و برای هر کلید، عضوی در نوع جدید می‌سازد. با همین یک ابزار، `Partial`، `Readonly`، `Pick` و بی‌شمار تبدیل سفارشی ساخته شده‌اند — و بعد از این فصل، دیگر هیچ‌کدامشان جادو نیستند.

آناتومی یک mapped type

mapped-anatomy.ts — type-checks ✓

```ts
interface User { id: number; name: string; email: string }

// برای هر K در کلیدهای User، همان property با همان نوع:
type Clone<T> = { [K in keyof T]: T[K] };

// حالا دستکاری: همه اختیاری
type MyPartial<T> = { [K in keyof T]?: T[K] };

// همه readonly و همه required:
type MyReadonly<T> = { readonly [K in keyof T]: T[K] };
type Draft = MyPartial<User>;    // { id?: number; name?: string; email?: string }
```

اجزای قابل تنظیم: کلیدها (`K in ...`)، نوع مقدار (`T[K]` یا هر عبارتی)، و دو modifier: `?` و `readonly`. modifierها را می‌توان **حذف** هم کرد — با پیشوند `-`:

modifiers.ts — type-checks ✓

```ts
type Mutable<T>  = { -readonly [K in keyof T]: T[K] }; // حذف readonly
type Required2<T> = { [K in keyof T]-?: T[K] };        // حذف اختیاری بودن

interface Config { readonly host?: string; readonly port?: number }
type EditableConfig = Mutable<Required2<Config>>;
// { host: string; port: number }
```

نکته حفظی مهم: mapped type روی `keyof T`، modifierهای موجود (`?` و `readonly`) را **به ارث می‌برد** مگر صریحاً تغییرشان دهید — برای همین `Clone<T>` واقعاً clone است، نه نسخه بی‌خاصیت.

Key Remapping با as — فیلتر و تغییر نام

از TypeScript 4.1، می‌توانید در حین پیمایش، نام کلید را بازنویسی کنید — یا با `never` حذفش کنید:

remapping.ts — type-checks ✓

```ts
interface User { id: number; name: string; save(): void }

//  ۱) تغییر نام: getter برای هر فیلد (با template literal فصل ۱۹)
type Getters<T> = {
  [K in keyof T & string as `get${Capitalize<K>}`]: () => T[K];
};
type UserGetters = Getters<User>;
// { getId: () => number; getName: () => string; getSave: () => () => void }

//  ۲) فیلتر: فقط فیلدهای داده‌ای، بدون متدها
type DataOnly<T> = {
  [K in keyof T as T[K] extends (...a: never[]) => unknown ? never : K]: T[K];
};
type UserData = DataOnly<User>;   // { id: number; name: string }
```

الگوی «as + شرط + never» را به خاطر بسپارید: **فیلتر کلید بر اساس نوع مقدار**. این پاسخ سؤال پرتکرار «چطور Pick بر اساس نوع بنویسم؟» است:

pick-by-type.ts — type-checks ✓

```ts
type PickByType<T, V> = {
  [K in keyof T as T[K] extends V ? K : never]: T[K];
};

interface Form { name: string; age: number; bio: string; score: number }
type StringFields = PickByType<Form, string>;   // { name: string; bio: string }
```

بازسازی utilityهای استاندارد — تست فهم

بهترین تست این فصل: پیاده‌سازی‌های واقعی (ساده‌شده از `lib.es5.d.ts`) را بفهمید:

stdlib-impl.ts

type-checks ✓

```ts
type MyPick<T, K extends keyof T> = { [P in K]: T[P] };
type MyRecord<K extends keyof any, V> = { [P in K]: V };
type MyOmit<T, K extends keyof any> = MyPick<T, Exclude<keyof T, K>>;
// Exclude خودش conditional ۱۷ فصل — توزیعی است!

type Settings = MyRecord<"theme" | "lang", string>;
// { theme: string; lang: string }
type NoId = MyOmit<{ id: number; name: string }, "id">;  // { name: string }
```

دقت کنید `in` لازم نیست حتماً روی `keyof T` بچرخد — هر union از stringها/numberها/symbolها قابل پیمایش است (`MyRecord`). و ترکیب‌پذیری را ببینید: `Omit = Pick + Exclude` — ابزارهای کوچک، ترکیب بزرگ.

مدل ذهنی — کارخانه property، نه ویرایش درجا

mapped type نوع ورودی را «ویرایش» نمی‌کند؛ نوع **جدیدی** می‌سازد: به ازای هر کلید ورودی، یک ایستگاه تصمیم — چه نامی (as)، چه نوعی (عبارت مقدار)، چه modifierهایی. اگر خروجی عجیب بود، همین سه ایستگاه را جدا بررسی کنید؛ معمولاً یکی‌شان چیزی نیست که فکر می‌کردید.

هشدار — mapped type روی union کلیدها homomorphic نمی‌ماند

وقتی از الگوی `[K in keyof T]` فاصله می‌گیرید (مثلاً `[K in MyUnion]`)، وراثت modifierها و رفتار ویژه tuple/آرایه‌ها را از دست می‌دهید. `Partial<string[]>` هنوز آرایه می‌ماند (`(string | undefined)[]`) چون Partial homomorphic است؛ نسخه دست‌ساز غیر homomorphic ممکن است شیئی با کلیدهای عددی بسازد. اگر ورودی‌تان آرایه/tuple هم می‌تواند باشد، تست بنویسید (فصل ۳۲).

زیر کاپوت — چرا این‌طور است؟

mapped type را «حلقه for روی کلیدها» معرفی کردیم؛ حالا دو جزئیات سازوکاری که رفتارهای ظریفش را توضیح می‌دهد:

۱) همریختی (homomorphism) — چرا Partial، tuple را خراب نمی‌کند؟ وقتی نگاشت دقیقاً از الگوی `[K in keyof T]` پیروی کند، کامپایلر آن را «همریخت» علامت می‌زند: نگاشتی که ساختار T را بازتولید می‌کند. برای نگاشت همریخت، modifier های اصلی (optional و readonly هر عضو) از T کپی می‌شوند، و مهم‌تر: اگر T آرایه یا tuple باشد، نتیجه آرایه/tuple می‌ماند — `Partial<[string, number]>` یک tuple با عناصر اختیاری است، نه شیئی با کلیدهای "0" و "1" و length و push. اگر کلیدها را از جای دیگری بگیرید — `[K in Keys]` — همریختی از بین می‌رود و همه این رفتارهای حفظی هم با آن. وقتی mapped type شما «ویژگی‌های عجیب آرایه» تولید کرد، همریختی را شکسته‌اید.

۲) بازنگاشت کلید با as — و حذف با never. `[K in keyof T as NewName]` برای هر کلید، نام جدید را محاسبه می‌کند — معمولاً با template literal فصل ۱۹ (`on${Capitalize<K>}`). قاعده ویژه: اگر محاسبه نام به never برسد، آن عضو به‌کلی حذف می‌شود. این قرارداد عمدی است و «فیلتر» را به mapped type اضافه می‌کند: ترکیب conditional + as + never یعنی «نگه‌دار اگر شرط برقرار است» — مثلاً فقط اعضای تابعی یک interface. map و filter، هر دو در یک سازه.

جمع‌بندی معماری: mapped type ها «ضرب‌المثل DRY» سیستم نوع‌اند — یک منبع حقیقت (شکل T)، بی‌نهایت مشتق هم‌گام (getter، Readonly، Partial ها، event map ها). هر جا دو نوع را دستی هم‌شکل نگه می‌دارید، یک mapped type بدهکارید؛ ناهمگامیِ دیر-کشف‌شده، بهره آن بدهی است.

تمرین

دو utility سفارشی بساز

۱. `Nullable<T>` : همه propertyها همان نوع یا null.

۲. `MethodsOf<T>` : فقط متدها، با همان امضا.

پاسخ:

quiz18-solution.ts — type-checks ✓

```ts
type Nullable<T> = { [K in keyof T]: T[K] | null };

type MethodsOf<T> = {
  [K in keyof T as T[K] extends (...a: never[]) => unknown ? K : never]: T[K];
};

interface Svc { url: string; fetch(): Promise<void>; retry(n: number): void }
type SvcMethods = MethodsOf<Svc>;  // { fetch(): Promise<void>; retry(n: number): void }
type SvcNullable = Nullable<Svc>;
```

چرا `(...a: never[]) => unknown` و نه `(...a: any[]) => any` ؟ با never[] هر تابعی match می‌شود (پارامتر contravariant است — فصل ۲۴) بدون این‌که any وارد کد کنید؛ هر دو این‌جا کار می‌کنند، اما عادت به نسخه بدون any بهتر است.

پرسش‌های مصاحبه این فصل

۱. در mapped type چطور یک کلید را «حذف» می‌کنید؟

با key remapping به never: `{ [K in keyof T as K extends Secret ? never : K]: T[K] }` — نگاشتِ کلید به never یعنی آن عضو اصلاً ساخته نشود. همین ترفند پایه فیلترهایی مثل PickByType است که بر اساس «نوع مقدار» صافی می‌کنند.

۲. mapped type هومومورفیک چه چیزی را مجانی حفظ می‌کند؟

وقتی روی `keyof T` نگاشت می‌کنید، modifier های optional و readonly هر عضو از منبع منتقل می‌شوند — برای همین `Partial` و `Readonly` این‌قدر کوتاه‌اند. با `-?` و `-readonly` هم می‌شود همان modifier ها را آگاهانه برداشت (مثل Required).

Template Literal Types

رشته‌ها در سطح نوع — از event name تا route تایپ‌شده

اهداف یادگیری این فصل

- الگوهای رشته‌ای را در سطح نوع مدل کنی
- ترکیب template literal با union و infer را به‌کار ببری
- بدانی کجا این تکنیک از کنترل خارج می‌شود

رشته‌ها همه‌جای برنامه‌های واقعی‌اند: نام رویدادها، مسیرهای route، کلیدهای ترجمه، کوئری‌ها. template literal typeها همان قدرت template string جاوااسکریپت را به سطح نوع می‌آورند: الگوی رشته به‌عنوان نوع، با ترکیب و تجزیه. نتیجه: رشته‌هایی که دیگر «جادویی» نیستند — قرارداد دارند.

ساخت: الگو × union = همه ترکیب‌ها

tlt-basics.ts — type-checks ✓

```ts
type Locale   = "fa" | "en" | "de";
type Page     = "home" | "pricing";

type RouteKey = `${Locale}/${Page}`;
// "fa/home" | "fa/pricing" | "en/home" | "en/pricing" | "de/home" | "de/pricing"
//  ضرب دکارتی — خودکار

type CssVar   = `--${string}`;        //  هر رشته‌ای که با -- شروع شود
type Handler  = `on${Capitalize<"click" | "focus">}`;  // "onClick" | "onFocus"

const v: CssVar = "--primary-color";   // ✓
```

tlt-err.ts — × خطای عمدی

```ts
type CssVar = `--${string}`;
const bad: CssVar = "primary-color";  //  × پیشوند ندارد
```

پیام کامپایلر

```
tlt-err.ts:2:7 - error TS2322: Type '"primary-color"' is not
assignable to type '`--${string}`'.
```

چهار utility رشته‌ای داخلی — `Uppercase` ، `Lowercase` ، `Capitalize` ، `Uncapitalize` — مستقیماً در کامپایلر پیاده شده‌اند و با template literalها ترکیب می‌شوند.

تجزیه: infer داخل الگو

همان‌طور که فصل ۱۷ از ساختارها نوع استخراج کرد، این‌جا از **رشته‌ها** استخراج می‌کنیم:

tlt-parse.ts — type-checks ✓

```ts
// استخراج نام پارامترهای یک مسیر — مانند Express:
type PathParams<P extends string> =
  P extends `${string}:${infer Param}/${infer Rest}`
    ? Param | PathParams<`/${Rest}`>
    : P extends `${string}:${infer Param}`
      ? Param
      : never;

type P1 = PathParams<"/users/:userId/posts/:postId">;  // "userId" | "postId"
type P2 = PathParams<"/health">;                       // never

// و از آن، شیء پارامترها:
type ParamsObject<P extends string> = { [K in PathParams<P>]: string };

declare function get<P extends string>(
  path: P,
  handler: (params: ParamsObject<P>) => void
): void;

get("/users/:userId/posts/:postId", (params) => {
  params.userId;   // ✓ string
  params.postId;   // ✓ string
});
```

tlt-parse-err.ts — × خطای عمدی

```ts
type PathParams<P extends string> =
  P extends `${string}:${infer Param}/${infer Rest}` ? Param | PathParams<`/${Rest}`>
  : P extends `${string}:${infer Param}` ? Param : never;
declare function get<P extends string>(
  path: P, handler: (params: { [K in PathParams<P>]: string }) => void): void;

get("/users/:userId", (params) => { params.postId; });  // ×
```

پیام کامپایلر

```
tlt-parse-err.ts:7:44 - error TS2339: Property 'postId' does not
exist on type '{ userId: string; }'.
```

این دقیقاً تجربه‌ای است که کتابخانه‌های مدرن routing و ORM می‌فروشند: مسیر را تایپ کن، پارامترها را مجانی بگیر. حالا می‌دانید زیر کاپوت چیست: template literal + infer بازگشتی + mapped type.

الگوی event map — پرکاربردترین استفاده عملی

event-map.ts — type-checks ✓

```ts
type EntityEvents<E extends string> =
  | `${E}:created` | `${E}:updated` | `${E}:deleted`;

type AppEvent = EntityEvents<"user" | "order">;
// "user:created" | "user:updated" | "user:deleted"
// "order:created" | "order:updated" | "order:deleted"

declare function track(event: AppEvent, payload: Record<string, unknown>): void;

track("order:created", { id: 1 });  //  ✓ + تکمیل خودکار همه ۶ حالت
```

event-map-err.ts — × خطای عمدی

```ts
type AppEvent = `${"user" | "order"}:${"created" | "updated" | "deleted"}`;
declare function track(event: AppEvent): void;
track("order:update");   // × typo — بدون template literal هرگز گرفته نمی‌شد
```

پیام کامپایلر

```
event-map-err.ts:3:7 - error TS2345: Argument of type '"order:update"'
is not assignable to parameter of type ... | "order:updated" | ...
```

مدل ذهنی — regex خیلی ساده، در سطح نوع

به template literal type مثل یک «الگوی تطبیق» نگاه کنید — چیزی بین string literal و regex: ثابت‌ها باید عیناً باشند، `${string}` یعنی «هر چیزی»، union یعنی «یکی از این‌ها»، و infer یعنی «این قسمت را به من بده». هر چه الگو دقیق‌تر، اعتبارسنجی قوی‌تر — و نوع نهایی هم بزرگ‌تر.

هشدار — انفجار ترکیبی واقعی است

ضرب دکارتی union ها سریع بزرگ می‌شود: چهار union ده‌عضوی داخل یک الگو یعنی ۱۰٬۰۰۰ عضو. کامپایلر برای unionهای خیلی بزرگ (مرز داخلی حدود ۱۰۰٬۰۰۰) خطای «expression produces a union type that is too complex to represent» می‌دهد — و قبل از آن، ادیتورتان کند شده است. برای شناسه‌های واقعاً آزاد، `` `prefix${string}` `` کافی است؛ لیست کامل لازم نیست (فصل ۲۶).

زیر کاپوت — چرا این‌طور است؟

template literal type ها فقط «چسباندن رشته» نیستند — دو سازوکار پشتشان است که هم قدرتشان را توضیح می‌دهد هم خطرشان را:

۱) ضرب دکارتی. وقتی جایگاه‌های یک الگو union بگیرند، نتیجه همه ترکیب‌هاست: `${A}-${B}` با A|=۳| و B|=۴| دوازده عضو می‌سازد. این «ضرب» است، نه «جمع» — و نمایی رشد می‌کند: چهار جایگاهِ ده‌عضوی یعنی ده‌هزار literal که کامپایلر باید بسازد و در هر مقایسه مرور کند. وقتی union سازی از الگوها کند شد یا به سقف خورد، علت همین انفجار ترکیبی است؛ درمان: الگو را باریک کنید یا به‌جای برشمردن همه ترکیب‌ها، با خودِ الگوی template (که «مجموعه بی‌نهایت با قاعده» است) تطبیق دهید — الگو لازم نیست باز شود تا بسنجد.

۲) تطبیق الگو با regex — سبکِ infer type-level. `S extends ${infer A}-${infer B}` رشته را بر اولین تطبیق می‌شکافد و پایه همه parser های type-level است: استخراج پارامترهای مسیر `/users/:id`، شکافتن "a.b.c" برای کلیدهای تو در تو، تبدیل case. با بازگشت (فصل ۲۶) این تطبیق به پیمایش کامل رشته تبدیل می‌شود.

درباره intrinsic ها (`Uppercase` و خویشاوندانش) یک حقیقت صادقانه: این چهار نوع در خود کامپایلر پیاده شده‌اند — تنها نوع‌های کل زبان که با سازه‌های خود زبان قابل ساخت نیستند (تبدیل حروف، قاعده نوشتنی ندارد). استثنای عمل‌گرایانه، نه الگویی برای تقلید.

و مرز معماری را دوباره ببینید: این‌ها «اعتبارسنج شکل رشته‌های ایستا» هستند — رشته‌ای که در کد نوشته شده. رشته آمده از کاربر و شبکه، type نیست و فقط validation فصل ۳۴ می‌تواند درباره‌اش حرف بزند. template literal ها دقیق‌ترین جایی است که مرز compile-time/runtime باید جلوی وسوسه را بگیرد.

تمرین

کلید ترجمه تایپ‌شده

نوع `TranslationKey` را طوری بسازید که فقط ترکیب‌های `بخش.زیرکلید` از این ساختار معتبر باشند:

quiz19.ts — type-checks ✓

```ts
const fa = {
  auth:  { login: "ورود", logout: "خروج" },
  cart:  { empty: "سبد خالی است" },
} as const;
//  هدف: "auth.login" | "auth.logout" | "cart.empty"
```

پاسخ: پیمایش دو سطح با mapped + template:

quiz19-solution.ts

type-checks ✓

```ts
const fa = {
  auth: { login: "ورود", logout: "خروج" },
  cart: { empty: "سبد خالی است" },
} as const;

type Messages = typeof fa;
type TranslationKey = {
  [S in keyof Messages & string]: `${S}.${keyof Messages[S] & string}`;
}[keyof Messages & string];
// "auth.login" | "auth.logout" | "cart.empty"
```

ترفند پایانی `{...}[keyof ...]` («ایندکس به خود») همه مقادیر mapped type را به یک union تخت تبدیل می‌کند — الگویی که در فصل ۲۰ هم دوباره می‌بینید.

پرسش‌های مصاحبه این فصل

۱. دو کاربرد واقعی template literal types که به production رسانده‌اید؟

استخراج پارامترهای مسیر از رشته route (`PathParams<"/users/:id">`) و کلیدهای ترجمه/رویداد با namespace (`cart:$`) — هر دو APIهای «stringly-typed» را compile-time می‌کنند: typo در مسیر یا کلید، خطای ساخت است نه باگ شب جمعه.

۲. خطر اصلی این ابزار چیست؟

انفجار ترکیبی: چند بخش قالب که هر کدام union بزرگ‌اند در هم ضرب می‌شوند و به سقف‌های کامپایلر می‌خورند (TS2590). قاعده فصل ۲۶: قدرت را جایی خرج کنید که دامنه کوچک و ارزش بالاست؛ برای دامنه باز، همان string با validation مرزی.

Utility Types — استاندارد و سفارشی

Partial تا Omit: ابزارهای آماده، و ساخت تبدیل‌های خودت

اهداف یادگیری این فصل

- utility typeهای پرکاربرد را با پیاده‌سازی‌شان بشناسی
- تبدیل سفارشی (مثل DeepReadonly) بسازی
- بدانی کی utility سفارشی ارزش نگه‌داری دارد

با فصل‌های ۱۷ و ۱۸، حالا می‌توانید هر utility type را بخوانید و بسازید. این فصل دو کار می‌کند: utilityهای استاندارد را بر اساس «مسئله‌ای که حل می‌کنند» مرتب می‌کند، و معیار مهندسی برای ساخت (یا نساختن) utility سفارشی می‌دهد.

جعبه‌ابزار استاندارد — براساس مسئله

مسئله	ابزار	مثال
«همین شکل، ولی همه اختیاری» — ورودی update	`Partial<T>`	`update(id, patch: Partial<User>)`
«همه اجباری» — بعد از پر کردن پیش‌فرض‌ها	`Required<T>`	`Required<Config>`
«غیرقابل تغییر» — state و props	`Readonly<T>`	`Readonly<State>`
«فقط این فیلدها» — DTO خروجی	`Pick<T, K>`	`Pick<User, "id" | "name">`
«همه جز این‌ها» — حذف فیلد حساس	`Omit<T, K>`	`Omit<User, "passwordHash">`
«این کلیدها، آن مقدار» — دیکشنری	`Record<K, V>`	`Record<Locale, string>`
«از union حذف کن / نگه دار»	`Extract` / `Exclude`	`Exclude<Status, "deleted">`
«null و undefined را بریز دور»	`NonNullable<T>`	`NonNullable<Found>`
«نوع خروجی / پارامترهای تابع»	`ReturnType` / `Parameters`	`ReturnType<typeof load>`
«نتیجه نهایی await»	`Awaited<T>`	`Awaited<ReturnType<typeof fetchUser>>`

دو الگوی ترکیبی که بیش از بقیه در کد production دیده می‌شوند:

combos.ts

type-checks ✓

```ts
// ۱: الگوی export نکرده — نوع از تابع موجود — وقتی کتابخانه نوع را
async function loadDashboard() {
  return { widgets: [{ id: 1, title: "Sales" }], refreshedAt: new Date() };
}
type Dashboard = Awaited<ReturnType<typeof loadDashboard>>;
// { widgets: { id: number; title: string }[]; refreshedAt: Date }

// ۲: الگوی override — کردن نوع یک فیلد — Omit بعد intersection
interface ApiUser { id: number; createdAt: string }  // API تاریخ به شکل رشته از
type User = Omit<ApiUser, "createdAt"> & { createdAt: Date }; // داخل برنامه: Date
```

الگوی ۲ را با `&` خام مقایسه کنید: `ApiUser & { createdAt: Date }` به `createdAt: string & Date` یعنی never می‌رسید (فصل ۹). **اول Omit، بعد اضافه** — این را به حافظه عضلانی تبدیل کنید.

ساخت utility سفارشی — سه نمونه محک‌خورده

custom-utils.ts

type-checks ✓

```ts
// ۱) DeepReadonly — بازگشتی (مرزها: فصل ۲۶)
type DeepReadonly<T> = T extends (...a: never[]) => unknown ? T
  : T extends object ? { readonly [K in keyof T]: DeepReadonly<T[K]> }
  : T;

// ۲) RequireAtLeastOne — حداقل یکی از این کلیدها
type RequireAtLeastOne<T, K extends keyof T = keyof T> =
  Omit<T, K> & { [P in K]: Required<Pick<T, P>> & Partial<Pick<T, Exclude<K, P>>> }[K];

interface Notify { email?: string; sms?: string; push?: string }
type ValidNotify = RequireAtLeastOne<Notify>;

const n1: ValidNotify = { email: "a@b.dev" };            // ✓
// const n0: ValidNotify = {};                            // × خطا — هیچ‌کدام

// ۳) Prettify — صاف کردن intersection برای نمایش بهتر
type Prettify<T> = { [K in keyof T]: T[K] } & {};
type Ugly = { a: 1 } & { b: 2 } & { c: 3 };
type Nice = Prettify<Ugly>;   // hover: { a: 1; b: 2; c: 3 }
void (0 as unknown as Nice); void (0 as unknown as DeepReadonly<Notify>);
const n2: ValidNotify = { sms: "0912", push: "token" };   // ✓
void n1; void n2;
```

`Prettify` هیچ معنایی عوض نمی‌کند — فقط tooltip و پیام خطا را انسانی می‌کند. در APIهای عمومی که از ترکیب چند intersection ساخته می‌شوند، همین «هیچ‌کار» کوچک، کیفیت تجربه مصرف‌کننده را محسوس بالا می‌برد (فصل ۲۶).

مدل ذهنی — utility یعنی «قاعده با اسم»

هر utility سفارشی باید یک قاعده قابل بیان به زبان آدمیزاد باشد: «حداقل یکی از کلیدها اجباری»، «همه‌چیز تا عمق آخر readonly». اگر نمی‌توانید قاعده را در یک جمله بگویید، آن نوع نباید ساخته شود — چون شش ماه بعد خودتان هم نمی‌توانید بخوانیدش. اسمِ خوب روی قاعده، نصف مستندسازی است.

در عمل — معیار «بساز یا نساز»

- **بساز** اگر: قاعده در ۳+ جا لازم است، به زبان ساده بیان‌پذیر است، و تست نوع برایش می‌نویسید (فصل ۳۲).
- **نساز** اگر: فقط برای کوتاه‌تر شدن یک annotation است — صریح بنویسید؛ یا اگر پیاده‌سازی‌اش به چند conditional تودرتو نیاز دارد که فقط نویسنده‌اش می‌فهمد — هزینه نگه‌داری از سود آن بیشتر می‌شود.
- قبل از ساختن، یک جست‌وجو در `type-fest` (کتابخانه معتبر utilityها) بزنید؛ احتمالاً ساخته و تست شده است.

زیر کاپوت — چرا این‌طور است؟

مهم‌ترین حقیقت درباره utility type ها: **هیچ‌کدام جادوی کامپایلر نیستند.** Partial و Required و Pick و Record و ReturnType همگی در lib.d.ts با همان سازه‌هایی تعریف شده‌اند که در سه فصل گذشته یاد گرفتید — mapped type، keyof، infer، conditional. این یعنی دو چیز: اولاً می‌توانید تعریفشان را بخوانید (در editor روی نام Go to Definition بزنید — بهترین درس type-level programming همان‌جاست)؛ ثانیاً هر جا کم آوردند، می‌توانید هم‌خانواده‌شان را خودتان بسازید — DeepPartial و Mutable و RequireAtLeastOne همگی ترکیب همان آجرها هستند.

یک تصمیم طراحی ظریف که کاربران را غافلگیر می‌کند: **Omit توزیع‌شونده نیست.** `Omit<A | B, K>` روی تک‌تک اعضای union اجرا نمی‌شود — چون Omit با `Pick<T, Exclude<keyof T, K>>` تعریف شده و `keyof (A | B)` طبق اتحاد فصل ۱۶ فقط کلیدهای مشترک را می‌دهد؛ نتیجه، ساختاری است که هویت تک‌تک اعضا را گم کرده — برای discriminated union فاجعه است چون discriminant غیرمشترک حذف می‌شود. درمان استاندارد: نسخه توزیعی خودتان با conditional برهنه — `T extends unknown ? Omit<T, K> : never` — که قاعده توزیع فصل ۱۷ را عمداً فعال می‌کند. این مثال، درس بزرگ‌تری دارد: رفتار هر utility را تعریفش تعیین می‌کند، نه نامش؛ نام‌ها شهود می‌سازند و شهود، جای خواندن تعریف را نمی‌گیرد.

قاعده سبک پایانی هم از همین‌جا می‌آید: utility ها «عملگر مشتق‌سازی»اند — وقتی درخشانند که یک منبع حقیقت را به اشکال موردنیاز بازتاب دهند. زنجیره‌های تو در توی چندلایه (`Partial<Omit<Pick<… >>>`) علامت این است که منبع حقیقت اشتباه انتخاب شده: به‌جای مشتق گرفتنِ پیچیده از نوعِ بزرگ، نوع کوچکِ پایه را تعریف کنید و بزرگ را از آن بسازید.

تمرین

سه تبدیل، یک خط هر کدام

با ترکیب utilityهای استاندارد (بدون mapped type دستی) بنویسید:

۱. `PublicUser` : همه فیلدهای `User` جز `passwordHash` و `mfaSecret` ، همگی readonly.

۲. `UpdatePayload` : فیلدهای `Product` به‌جز `id` ، همه اختیاری.

۳. `HandlerMap` : به ازای هر عضو `"click" | "hover"` ، تابعی از `Event` به void.

پاسخ:

quiz20-solution.ts — type-checks ✓

```ts
interface User { id: number; name: string; passwordHash: string; mfaSecret: string }
interface Product { id: string; title: string; price: number }

type PublicUser    = Readonly<Omit<User, "passwordHash" | "mfaSecret">>;
type UpdatePayload = Partial<Omit<Product, "id">>;
type HandlerMap    = Record<"click" | "hover", (e: Event) => void>;
```

پرسش‌های مصاحبه این فصل

۱. چرا `Partial<User>` برای ورودی update معمولاً اشتباه است؟

چون همه‌چیز را اختیاری می‌کند — از جمله id و فیلدهای هویتی/invariant دار که نباید patchable باشند. الگوی درست: `Partial<Omit<User, "id" | "createdAt">>` — اول تعیین کنید چه چیزی «اصلاً» قابل تغییر است، بعد اختیاری‌اش کنید.

۲. `Omit` روی discriminated union چه رفتاری دارد؟

خراب: توزیع نمی‌کند و روی «اعضای مشترک» union کار می‌کند، نتیجه اغلب چیزی نیست که انتظار دارید. چاره، نسخه توزیعی دست‌ساز است: `T extends unknown ? Omit<T, K> : never` — و این یکی از سؤال‌های محبوب غربالگری عمق دانش utility هاست.

Const Type Parameters و as const، satisfies

سه ابزار برای نگه‌داشتن دقیق‌ترین نوع بدون از دست دادن بررسی

اهداف یادگیری این فصل

- تفاوت as const با annotation و با as را دقیق بدانی
- با satisfies بررسی بدون widening انجام دهی
- const type parameter را در API جنریک به‌کار ببری

سه ابزار این فصل یک مسئله مشترک را حل می‌کنند که در فصل ۵ دیدیم: کشمکش بین «دقیق نگه داشتن نوع» و «بررسی شدن مقدار». هر سه جوان‌اند (satisfies از 4.9، const type parameter از 5.0) و هر سه در کد مدرن باکیفیت، امضای سبک نویسنده‌اند.

as const — منجمد کن

as-const.ts — type-checks ✓

```ts
const config = {
  env: "production",
  ports: [8080, 8443],
  features: { darkMode: true },
} as const;
/*  نوع:
 {
   readonly env: "production";
   readonly ports: readonly [8080, 8443];   ← tuple نه، number[]
   readonly features: { readonly darkMode: true };
 } */
```

`as const` عمیق عمل می‌کند: همه propertyها readonly، همه literalها بدون widening، همه آرایه‌ها tuple خواندنی. بهترین دوستِ الگوی «منبع حقیقت واحد» فصل ۱۶. محدودیتش: هیچ **بررسی** انجام نمی‌دهد — فقط شکل استنتاج را عوض می‌کند. اگر typo داشته باشید، typo با جزئیات کامل منجمد می‌شود!

satisfies — بررسی کن، ولی پهن نکن

مسئله کلاسیک annotation را ببینید:

satisfies-why.ts

type-checks ✓

```ts
type Theme = Record<string, string | [string, string]>;

// ⚠ راه قدیمی: annotation — بررسی می‌شود اما نوع «پهن» می‌شود
const theme1: Theme = {
  primary: "#3178C6",
  surface: ["#FFFFFF", "#0B1222"],
};
// theme1.surface: string | [string, string]  ← دقت از دست رفت
// theme1.surface[0] ⇒ خطا! اول باید narrowing کنید

// ✅ satisfies: همان بررسی، بدون قربانی کردن استنتاج
const theme2 = {
  primary: "#3178C6",
  surface: ["#FFFFFF", "#0B1222"],
} satisfies Theme;

theme2.surface[0];  //  ✓ نوع surface دقیقاً [string, string] ماند
theme2.primary.toUpperCase();  //  ✓ دقیقاً string
```

satisfies-catch.ts

× خطای عمدی

```ts
type Theme = Record<string, string | [string, string]>;
const broken = {
  primary: "#3178C6",
  surface: 0xFFFFFF,    //  × عدد — همین‌جا گرفته می‌شود
} satisfies Theme;
```

پیام کامپایلر

```
satisfies-catch.ts:4:3 - error TS2322: Type 'number' is not
assignable to type 'string | [string, string]'.
```

قاعده انتخاب بین سه گزینه:

ابزار	بررسی سازگاری؟	نوع نهایی	کی استفاده کنیم
`: Type` (annotation)	✓	همان Type (پهن)	وقتی قرارداد مهم‌تر از جزئیات است: امضای عمومی، متغیری که بعداً مقدارهای دیگر می‌گیرد
`satisfies Type`	✓	استنتاج دقیق	پیکربندی و ثابت‌ها: هم بررسی، هم حفظ literalها
`as const`	✗	دقیق‌ترین ممکن	وقتی منبع حقیقت خودِ مقدار است و قراردادی در کار نیست
`as Type` (assertion)	تقریباً ✗	همان Type	تقریباً هیچ‌وقت — فرار از بررسی است، نه ابزار طراحی

ترکیب هم ممکن و رایج است: `as const satisfies Theme` — منجمدِ بررسی‌شده. برای جدول‌های پیکربندی، بهترینِ هر دو جهان.

const type parameter — as const خودکار در API

وقتی API شما لیست یا شیء literal می‌گیرد، کاربرانتان دائم باید `as const` بنویسند — مگر این‌که پارامتر نوع را `const` کنید:

const-type-param.ts — type-checks ✓

```ts
// ❌ بدون const: استنتاج معمولی پهن می‌کند
function defineRoutes<T extends readonly string[]>(routes: T): T { return routes; }
const r1 = defineRoutes(["/home", "/about"]);
// r1: string[] — literalها از دست رفتند

// ✅ با const: انگار کاربر as const نوشته
function defineRoutesC<const T extends readonly string[]>(routes: T): T { return routes; }
const r2 = defineRoutesC(["/home", "/about"]);
// r2: readonly ["/home", "/about"] — دقیق!

type FirstRoute = (typeof r2)[0];   // "/home"
void r1; void (0 as unknown as FirstRoute);
```

کتابخانه‌هایی که از پیکربندی literal نوع می‌سازند (routerها، فرم‌سازها، state machineها — فصل ۳۶) بدون این قابلیت یا دقت را از دست می‌دادند یا کاربر را مجبور به `as const` . حالا API خودش دقت را نگه می‌دارد.

> **مدل ذهنی — سه سؤال، سه ابزار**
>
> «می‌خواهم این مقدار **بررسی** شود؟» اگر نه ⇒ as const. اگر بله: «می‌خواهم نوعش بعد از بررسی **دقیق** بماند؟» اگر بله ⇒ satisfies؛ اگر قرارداد عمومی کافی است ⇒ annotation. و «as» معمولی؟ آن پاسخ هیچ سؤالی نیست — اعتراف است که نمی‌خواهید کامپایلر نظر بدهد.

در عمل — جاهایی که satisfies بلافاصله کد را بهتر می‌کند

- شیءهای پیکربندی (theme، feature flagها، جدول routeها) — بررسی + حفظ کلیدها.
- پیاده‌سازی interfaceهای Record-مانند که نمی‌خواهید کلیدهایشان گم شود.
- داده تست که باید با نوع API سازگار بماند ولی literal هم بماند.
- exhaustive map: `const labels = {...} satisfies Record<Status, string>` — عضو جا بیفتد، خطا؛ عضو اضافه بنویسید، خطا.

زیر کاپوت — چرا این‌طور است؟

`as const` و `satisfies` را وقتی عمیق می‌فهمید که هر دو را روی محور واحدِ «freshness فصل ۵» ببینید:

as const = انجماد freshness. literal تازه، به‌طور طبیعی در اولین مرزِ بدون انتظار widen می‌شود. as const این سرنوشت را لغو می‌کند: به‌صورت بازگشتی، هر literal در عمق ساختار همان literal می‌ماند، هر شیء و آرایه readonly می‌شود (آرایه، tuple می‌شود). چرا readonly اجباری است؟ چون literal ماندن و قابل‌نوشتن بودن متناقض‌اند — جایگاهی که فردا می‌تواند مقدار دیگری بگیرد، نمی‌تواند امروز به یک literal سنجاق شود؛ همان معیار «قابلیت تغییر» فصل ۵. پس as const یک بسته غیرقابل تفکیک است: دقت حداکثری + تغییرناپذیری.

satisfies = بررسی بدون اعلام. مشکل annotation این است که دو کار را هم‌زمان می‌کند: «بررسی کن» و «از این به بعد، نوع من این باشد» — و دومی اطلاعات استنتاج‌شده دقیق‌تر را دور می‌ریزد: با `Record<string, string[]>` دیگر نمی‌دانید کدام کلیدها واقعاً وجود دارند. satisfies این دو کار را از هم جدا می‌کند: سازگاری با نوع هدف همین‌جا بررسی می‌شود (با excess check کامل، چون literal هنوز تازه است)، ولی نوعِ اعلامی عبارت، همان استنتاج دقیق می‌ماند. به زبان فصل ۵: annotation «جایگاه را صاحب انتظار می‌کند و انتظار را جایگزین استنتاج می‌کند»؛ satisfies «انتظار را تحمیل می‌کند و استنتاج را نگه می‌دارد».

جدول تصمیم نهایی با این مدل بدیهی می‌شود: literal ها باید دقیق بمانند و منبع نوع شوند؟ as const. ساختار باید با قراردادی چک شود ولی دقتش حفظ شود؟ satisfies. جایگاه باید قراردادِ خودش را داشته باشد و جزئیات مقدار بی‌اهمیت است (مرز عمومی API)؟ annotation. هر سه، شیرهای تنظیم یک ماشین‌اند — ماشین freshness/widening — و ترکیب `satisfies X as const` … ممنوع نیست، فقط معمولاً علامت این است که هنوز نمی‌دانید کدام شیر را می‌خواستید.

تمرین

ابزار درست برای هر خط

برای هر مورد بهترین گزینه را انتخاب کنید (as const / satisfies / annotation / هیچ):

۱. جدول `STATUS_LABELS` که باید دقیقاً یک برچسب برای هر `Status` داشته باشد و کلیدهایش در تکمیل خودکار بیایند.

۲. خروجی تابع عمومی `getConfig(): AppConfig` .

۳. آرایه `WEEKDAYS` که فقط منبع مشتق‌گیری نوع است.

۴. متغیر محلی `total = items.length` .

پاسخ: ۱. `satisfies Record<Status, string>` — بررسی کامل بودن + حفظ literal کلیدها. ۲. annotation — قرارداد عمومی، جزئیات پیاده‌سازی نباید نشت کند. ۳. `as const` — بررسی لازم نیست، دقت لازم است. ۴. هیچ — استنتاج کافی است (فصل ۴).

پرسش‌های مصاحبه این فصل

۱. as const، satisfies، annotation — برای یک شیء config کدام و چرا؟

`as const satisfies ConfigShape` : annotation تنها، literal ها را به قرارداد پهن می‌کند و دقت گم می‌شود؛ as const تنها، هیچ بررسی‌ای نمی‌کند. ترکیب، هر دو را می‌دهد — مقادیر دقیق literal برای مشتق‌گیری، و بررسی کامل قرارداد در همان خط تعریف.

۲. فرق بنیادین `as` با `satisfies` در یک جمله؟

`as` به کامپایلر می‌گوید «بررسی نکن، حرف من است»؛ `satisfies` می‌گوید «کامل بررسی کن، ولی نوع استنتاجی دقیقم را عوض نکن». یکی ایمنی را کم می‌کند، دیگری اضافه — این‌که اسم‌هایشان شبیه کار می‌رود، منبع نیمی از سوءاستفاده‌هاست.

Discriminated Union و Exhaustiveness

مهم‌ترین الگوی مدل‌سازی در TypeScript واقعی

اهداف یادگیری این فصل

- state ناممکن را با discriminated union حذف کنی
- با never بررسی exhaustive بسازی که در refactor هشدار دهد
- این الگو را جایگزین boolean-های موازی کنی

اگر مجبور بودیم از کل این کتاب فقط یک الگو را نگه داریم، همین بود. discriminated union (یا tagged union) پاسخ TypeScript به قدیمی‌ترین مسئله مدل‌سازی است: **چطور داده‌ای را نمایش دهیم که چند حالت متفاوت دارد، طوری که حالت‌های بی‌معنا اصلاً قابل ساخت نباشند؟**

آناتومی الگو

سه جزء: چند عضو شیئی، یک property مشترک literal (تگ/discriminant)، و مقادیر متمایز برای آن:

request-state.ts — type-checks ✓

```ts
type RequestState<T> =
  | { status: "idle" }
  | { status: "loading"; startedAt: number }
  | { status: "success"; data: T; loadedAt: number }
  | { status: "error";   error: Error; retryCount: number };

function render(state: RequestState<string[]>): string {
  switch (state.status) {
    case "idle":    return "آماده";
    case "loading": return `در حال بارگذاری از ${state.startedAt}`;
    case "success": return state.data.join(", ");    // ✓ data فقط اینجا هست
    case "error":   return state.error.message;      // ✓ error فقط اینجا
  }
}
```

کامپایلر با دیدن `state.status === "success"` کل عضو مربوط را انتخاب می‌کند — نه فقط نوع `status` را. این narrowing ساختاری (فصل ۱۰) در بهترین حالتش است: داخل هر case، دقیقاً فیلدهای همان حالت در دسترس‌اند، نه بیشتر نه کمتر.

مقایسه کنید با مدل «پرچم‌های موازی» که این الگو جایگزینش می‌شود:

flags-bad.ts

type-checks ✓

```ts
// ۱۶ ترکیب قابل بیان، فقط ۴ تا معنادار:
interface RequestStateBad<T> {
  isLoading: boolean;
  data?: T;
  error?: Error;
  startedAt?: number;
}
// همزمان؟ کامپایلر ساکت است data و error همزمان؟ isLoading=true و error
// و هر مصرف‌کننده باید «قرارداد نانوشته» را حدس بزند
declare const s: RequestStateBad<string>;
if (!s.isLoading && !s.error) { s.data?.toUpperCase(); } //  ? اجباری — چون نوع نمی‌داند
```

در مدل بد، درستی برنامه به نظم ذهنی همه اعضای تیم بسته است. در discriminated union، به کامپایلر. در refactor بعدی تفاوت را حس می‌کنید.

Exhaustiveness — سوئیچی که آینده را می‌بیند

الگو با بررسی exhaustive کامل می‌شود (از فصل ۸ به یاد دارید):

exhaustive.ts

type-checks ✓

```ts
type PaymentMethod =
  | { kind: "card"; last4: string }
  | { kind: "wallet"; provider: "apple" | "google" }
  | { kind: "bank"; iban: string };

function assertNever(x: never): never {
  throw new Error(`Unhandled case: ${JSON.stringify(x)}`);
}

function label(pm: PaymentMethod): string {
  switch (pm.kind) {
    case "card":   return `کارت •••• ${pm.last4}`;
    case "wallet": return pm.provider === "apple" ? "Apple Pay" : "Google Pay";
    case "bank":   return `حساب ${pm.iban.slice(0, 8)}…`;
    default:       return assertNever(pm);
  }
}
```

exhaustive-err.ts

× خطای عمدی

```ts
type PM = { kind: "card" } | { kind: "wallet" } | { kind: "crypto" };
declare function assertNever(x: never): never;
function label(pm: PM) {
  switch (pm.kind) {
    case "card":   return "کارت";
    case "wallet": return "کیف پول";
    default:       return assertNever(pm);   // × crypto جا مانده!
  }
}
```

پیام کامپایلر

```
exhaustive-err.ts:7:35 - error TS2345: Argument of type
'{ kind: "crypto"; }' is not assignable to parameter of type 'never'.
```

این خطا بهترین نوع خطاست: وقتی عضو جدیدی به union اضافه می‌شود، **تک‌تک نقاط تصمیم‌گیری فراموش‌شده** در سراسر codebase قرمز می‌شوند — با آدرس دقیق. تیم‌هایی که این الگو را جدی می‌گیرند، «اضافه کردن حالت جدید» را از پروژه ترسناک به کار نیم‌ساعته تبدیل کرده‌اند.

انتخاب discriminant خوب

- **literal رشته‌ای** متداول‌ترین و خواناترین است؛ عدد و boolean هم مجازند (`success: true/false` در مدل Result — فصل ۳۵).
- نامش مرسوم است `kind`، `type`، `status`، `tag` — در هر codebase یکی را استاندارد کنید.
- discriminant باید در **همه اعضا** حاضر و literal باشد؛ یک عضو بدون تگ، narrowing همه را ضعیف می‌کند.
- تگ تودرتو هم کار می‌کند (`event.payload.op === "move"`) اما تگ سطح بالا تجربه بهتری می‌دهد.

مدل ذهنی — فضای حالت را مدل کن، نه فیلدها را

طراحی نوع را از این سؤال شروع کنید: «این داده واقعاً در چند حالت می‌تواند باشد و در هر حالت، چه چیزهایی **حتماً** وجود دارند؟» بعد همان را عیناً بنویسید: یک عضو per حالت. فیلد optional فقط وقتی مجاز است که در همان حالت واقعاً گاهی هست و گاهی نیست — نه برای چسباندن چند حالت به هم.

در عمل — همه‌جا هست، وقتی چشمتان باز شود

پیام‌های WebSocket، action‌های Redux، نتیجه عملیات (Result)، رویدادهای دامنه، واریانت‌های کامپوننت UI، مراحل یک wizard، وضعیت sync آفلاین/آنلاین — همه ذاتاً «چند حالت با داده‌های متفاوت»اند. اگر در مدل فعلی‌تان optional و boolean موج می‌زند، تقریباً همیشه یک discriminated union پنهان آن زیر است. فصل ۳۶ همین را به ماشین حالت کامل گسترش می‌دهد.

زیر کاپوت — چرا این‌طور است؟

قدرت discriminated union را می‌شود با «حساب فضای حالت» اثبات کرد — ریاضیِ ساده‌ای که کل فصل را از توصیه به قضیه تبدیل می‌کند.

شیءِ پرچم‌محورِ کلاسیک — `{ loading: boolean; data?: T; error?: string }` — یک **نوع ضربی** است: فضای حالتش حاصل‌ضرب دامنه اعضاست: ۲ × ۲ × ۲ = ۸ ترکیب. اما دامنه واقعی مسئله فقط ۳ یا ۴ حالت معنادار دارد: idle، loading، success، error. پنج ترکیب باقی‌مانده — مثل «loading درحالی‌که data هست و error هم هست» — **حالت‌های ناممکنی‌اند که نوعِ شما مجازشان کرده**. هر کدام یک if دفاعی در هر مصرف‌کننده، یک شاخه تست‌نشده و یک باگ آینده است.

discriminated union همین فضا را **جمعی** می‌سازد: اجتماعِ دقیقاً همان ۴ حالت معنادار، هر یک فقط با اعضای مرتبط خودش. ناممکن‌ها نه «بررسی‌نشده» بلکه «بیان‌نشدنی» می‌شوند — compile error در لحظه ساخت، نه runtime bug در لحظه مصرف. این همان شعار «make illegal states unrepresentable» است، با عدد و رقم.

سازوکار narrowing اش هم ترکیب دو قطعه آشناست: discriminant یک literal type است (فصل ۵) و CFA (فصل ۱۰) با مقایسه آن، اعضای ناسازگار union را حذف می‌کند — هیچ قاعده جدیدی لازم نبود؛ زبان از آجرهای موجودش این الگو را «مجانی» گرفت. برای همین discriminant باید literal باشد: boolean و string عمومی چیزی برای مقایسه حذفی ندارند.

و exhaustiveness با never؟ قضیه فصل ۸: حذف همه اعضا از union = مجموعه تهی = never. اگر بعد از همه caseها چیزی جز never ماند، عضوی از union پوشش داده نشده — و انتساب به never دقیقاً همین را گزارش می‌کند. افزودن حالت پنجم به union، همه switchهای ناقص را «خودش» پیدا می‌کند: این یعنی نگهداری‌پذیری اثبات‌شده، نه امید به دیسیپلین تیم.

تمرین

بازطراحی یک مدل واقعی

این مدل را به discriminated union تبدیل کنید و یک ترکیب بی‌معنا را که الان قابل ساخت است نام ببرید:

quiz22.ts — type-checks ✓

```ts
interface Subscription {
  active: boolean;
  trialEndsAt?: Date;
  canceledAt?: Date;
  plan?: "pro" | "team";
}
```

پاسخ (یک طراحی خوب): ترکیب بی‌معنای نمونه: `{ active: true, canceledAt: new Date() }` — فعال و لغوشده هم‌زمان.

quiz22-solution.ts

type-checks ✓

```ts
type Subscription =
  | { status: "trial";    plan: "pro" | "team"; trialEndsAt: Date }
  | { status: "active";   plan: "pro" | "team"; renewsAt: Date }
  | { status: "canceled"; plan: "pro" | "team"; canceledAt: Date; accessUntil: Date }
  | { status: "none" };
```

دقت کنید چند فیلد «کشف» شد (`renewsAt` ، `accessUntil`) — مدل‌سازی حالت‌محور معمولاً نیازمندی‌های پنهان را هم بیرون می‌کشد. این قدرت واقعی الگوست: نوع، ابزار فکر کردن می‌شود.

پرسش‌های مصاحبه این فصل

۱. در طراحی فیلد tag چه معیارهایی دارید؟

literal رشته‌ای (در پیام خطا و دیباگ خوانا)، حاضر در «همه» اعضا، نام مرسوم و ثابت در کل codebase (`kind` یا `status` — یکی را استاندارد کنید)، و مقادیرش از جنس دامنه نه پیاده‌سازی. tag تودرتو هم کار می‌کند ولی تجربه ساده‌تر با tag سطح بالاست.

۲. exhaustive switch چطور refactor را «کار کامپایلر» می‌کند؟

با انتساب default به never (یا assertNever)، افزودن عضو جدید به union، تک‌تک switch های جامانده در سراسر پروژه را خطای کامپایل می‌کند — فهرست کامل «جاهایی که باید تصمیم بگیری» را خود کامپایلر می‌دهد. بدون آن، refactor یعنی grep و دعا.

Branded Types و Type Predicate، Assertion Function

گسترش narrowing با توابع خودت — و نوع اسمی روی زیرساخت ساختاری

اهداف یادگیری این فصل

- type predicate درست و امن بنویسی و خطر دروغ‌گویی‌اش را بدانی
- assertion function را برای اعتبارسنجی ورودی به‌کار ببری
- با brand از اختلاط Id-های هم‌شکل جلوگیری کنی

فصل ۱۰ نشان داد کامپایلر چطور با `typeof` و `in` نوع را باریک می‌کند. اما وقتی منطق تشخیص پیچیده‌تر است — «آیا این شیء، یک User معتبر است؟» — به ابزاری نیاز دارید که narrowing را **به توابع خودتان** گسترش دهد. و در انتها، الگوی branding، که مشکل «همه stringها شبیه هم‌اند» فصل ۶ را حل می‌کند.

Type Predicate — تابعی که نوع را ثابت می‌کند

predicates.ts — type-checks ✓

```ts
interface User { id: number; name: string }

function isUser(value: unknown): value is User {
  return (
    typeof value === "object" && value !== null &&
    "id" in value   && typeof (value as Record<string, unknown>).id === "number" &&
    "name" in value && typeof (value as Record<string, unknown>).name === "string"
  );
}

function greet(input: unknown) {
  if (isUser(input)) {
    console.log(`سلام ${input.name}`);    // ✓ input: User — narrowing سفارشی
  }
}
greet({ id: 1, name: "Ali" });
```

امضای `value is User` به کامپایلر می‌گوید: «اگر true برگرداندم، value از نوع User است». از آن به بعد، این تابع در تحلیل جریان کنترل مثل `typeof` عمل می‌کند — در `if`، در `filter`، همه‌جا:

filter-predicate.ts

type-checks ✓

```ts
const inputs: (string | null | undefined)[] = ["a", null, "b", undefined];

// بدون predicate: نوع خروجی همان union می‌ماند
const bad = inputs.filter(x => x != null);        // (string | null | undefined)[] در TS<5.5

// predicate صریح — در همه نسخه‌ها دقیق:
const good = inputs.filter((x): x is string => x != null);  // string[]
void bad; void good;
```

نکته به‌روز: از TypeScript 5.5 کامپایلر برای توابع بولی ساده، predicate را **خودش استنتاج می‌کند** — نسخه `bad` هم در کامپایلرهای جدید `string[]` می‌شود. اما predicate صریح همچنان برای منطق پیچیده و قراردادهای عمومی لازم است.

هشدار — predicate می‌تواند دروغ بگوید

کامپایلر **بدنه predicate را با ادعایش مقایسه نمی‌کند**. اگر `value is User` بنویسید و فقط `"id" in value` را چک کنید، کامپایلر باور می‌کند — و باگ، تایپ‌امن به نظر می‌رسد: بدترین نوع باگ. قاعده: predicate دست‌نویس = تست واحد اجباری؛ یا بهتر، برای داده بیرونی به‌جای predicate دست‌نویس از schema استفاده کنید (فصل ۳۴) که بررسی را از روی تعریف تولید می‌کند.

Assertion Function — بررسی یا پرتاب

گاهی به‌جای شاخه true/false می‌خواهید «اگر نبود، همان‌جا بمیر». امضای `asserts` :

assertions.ts

type-checks ✓

```ts
function assertDefined<T>(value: T, name: string): asserts value is NonNullable<T> {
  if (value === null || value === undefined) {
    throw new Error(`${name} must be defined`);
  }
}

function assertIsAdmin(user: { role: string }): asserts user is { role: "admin" } {
  if (user.role !== "admin") throw new Error("Forbidden");
}

function deleteProject(projectId: string | undefined, user: { role: string }) {
  assertDefined(projectId, "projectId");
  assertIsAdmin(user);
  // از اینجا: projectId: string و user.role: "admin" — بدون if اضافه
  console.log(`${user.role} deletes ${projectId}`);
}
```

تفاوت با predicate: اثر narrowing **بعد از فراخوانی** و در همان scope اعمال می‌شود، نه داخل شاخه if. این دقیقاً ساختار «گارد در ابتدای تابع» است که فصل ۱۰ توصیه کرد — حالا با نام و قابل استفاده مجدد. توجه: assertion functionها باید اعلان نوع بازگشتی صریح داشته باشند و با arrow function ذخیره‌شده در const، annotation نوع متغیر لازم دارند — محدودیت رسمی زبان.

Branded Types — نوع اسمی روی زیرساخت ساختاری

مشکل فصل ۶: `UserId` و `OrderId` هر دو `string` اند و قابل تعویض — کامپایلر باگِ «آیدی اشتباه به تابع اشتباه» را نمی‌گیرد. راه‌حل: یک «مهر» غیرساختاری به نوع اضافه کنیم:

branding.ts — type-checks ✓

```ts
declare const brand: unique symbol;
type Brand<T, B extends string> = T & { readonly [brand]: B };

type UserId  = Brand<string, "UserId">;
type OrderId = Brand<string, "OrderId">;

// تنها دروازه ساخت: توابع سازنده (همراه با اعتبارسنجی!)
function toUserId(raw: string): UserId {
  if (!/^u_[a-z0-9]{8}$/.test(raw)) throw new Error(`Invalid user id: ${raw}`);
  return raw as UserId;  // تنها as در دروازه — مجاز
}

declare function getOrders(id: UserId): Promise<OrderId[]>;

const uid = toUserId("u_ab12cd34");
await getOrders(uid);                    // ✓
export {};
```

branding-err.ts — × خطای عمدی

```ts
declare const brand: unique symbol;
type UserId  = string & { readonly [brand]: "UserId" };
declare function getOrders(id: UserId): void;

getOrders("just-any-string");   // × string خام پذیرفته نمی‌شود
```

پیام کامپایلر

```
branding-err.ts:5:11 - error TS2345: Argument of type 'string' is not
assignable to parameter of type 'UserId'.
```

مهر `[brand]` فقط در سطح نوع وجود دارد — erasure کامل، صفر هزینه runtime. و چون ساخت فقط از دروازه `toUserId` ممکن است، **هر UserId در برنامه، از اعتبارسنجی عبور کرده است**. این ترکیب branding + validation، الگوی

«parse, don't validate» است: به‌جای بررسی مکرر در عمق، یک بار در مرز تبدیل کنید و از آن به بعد نوع، گواهی اعتبار است.

مدل ذهنی — نوع به‌عنوان گواهی

assertion ،predicate و brand هر سه یک ایده‌اند: **تبدیل دانش runtime به گواهی compile-time**. تفاوتشان فقط شکل تحویل گواهی است: شاخه if (predicate)، ادامه خط (assertion)، یا خود نوع برای همیشه (brand). هر چه گواهی زودتر و بالاتر در مرز صادر شود، کد پایین‌دست تمیزتر.

زیر کاپوت — چرا این‌طور است؟

type guard سفارشی قوی‌ترین و خطرناک‌ترین ابزار این کتاب تا این‌جاست — و دلیلش را باید دقیق بدانید: **کامپایلر بدنه guard را با ادعایش تطبیق نمی‌دهد.** امضای `x is User` یک قرارداد یک‌طرفه است: شما اعلام می‌کنید «true من یعنی User»، و CFA بدون هیچ راستی‌آزمایی، در شاخه true نوع را باریک می‌کند. اگر بدنه فقط `"id" in x` را بسنجد ولی ادعا User کامل باشد، یک دروغ ایستا ساخته‌اید که از هر as خطرناک‌تر است — چون as در محل استفاده دیده می‌شود، ولی guard دروغگو «قابل اعتماد» به نظر می‌رسد و در ده‌ها نقطه مصرف می‌شود.

چرا TypeScript بدنه را بررسی نمی‌کند؟ تاریخ و توان: اثبات معادل بودنِ «هر بدنه دلخواه» با یک گزاره نوعی، در حالت عمومی ممکن نیست. اما از TS 5.5 نیمه راه آمده: برای توابع بولی ساده با الگوهای قابل تحلیل، predicate را **خودش استنتاج می‌کند** — `x ⇒ typeof x === "string"` در filter دیگر annotation نمی‌خواهد. قاعده عملی به‌روز: اول بگذارید استنتاج کار کند؛ annotation دستی `x is T` را برای بدنه‌های پیچیده نگه دارید و با آن مثل as رفتار کنید — کامنت، تست، و بازبینی در هر تغییر T. به‌خصوص مراقب «فاصله افتادن» باشید: عضو جدیدی به T اضافه می‌شود و guard قدیمی هنوز true می‌گوید.

branding هم عمق خودش را دارد: مشکل این است که سیستم ساختاری، `UserId` و `OrderId` هر دو string را یکی می‌بیند — ساختارشان یکی است. برند، یک عضو «شبح» به نوع اضافه می‌کند (`string & { readonly __brand: "UserId" }`) که فقط در دنیای نوع وجود دارد: هیچ مقداری واقعاً آن عضو را ندارد و هیچ بایتی در runtime تغییر نمی‌کند. این یعنی «نوع اسمی شبیه‌سازی‌شده روی سیستم ساختاری» — ورود به برند فقط از دروازه‌ای که شما می‌سازید (تابع سازنده یا guard) ممکن است، و همین دروازه است که اعتبارسنجی را به تنها مسیر تولید مقدار تبدیل می‌کند. برند پاک می‌شود؛ قراردادش می‌ماند.

تمرین

دروازه بساز

نوع branded `Email` و تابع `toEmail` بنویسید؛ سپس امضای `sendWelcome` را طوری تغییر دهید که با string خام اصلاً صدازدنی نباشد.

پاسخ:

quiz23-solution.ts

type-checks ✓

```ts
declare const brand: unique symbol;
type Email = string & { readonly [brand]: "Email" };

function toEmail(raw: string): Email {
  const v = raw.trim().toLowerCase();
  if (!/^[^@\s]+@[^@\s]+\.[^@\s]+$/.test(v)) throw new Error(`Invalid email: ${raw}`);
  return v as Email;
}

function sendWelcome(to: Email): void { void to; }

sendWelcome(toEmail("  Ali@Example.COM ")); //  ✓ تنها مسیر ممکن
```

به اثر جانبی مثبت دقت کنید: normalize (trim/lowercase) هم فقط یک‌جا اتفاق می‌افتد — دروازه، محل تجمیع همه قواعد ورودی است.

پرسش‌های مصاحبه این فصل

۱. type predicate دروغ بگوید چه می‌شود و دفاع شما چیست؟

کامپایلر بدنه را با ادعای `x is T` مقایسه نمی‌کند — predicate دروغگو باگ تایپ‌امن‌نما تولید می‌کند که از هر any بدتر است چون اعتماد جلب می‌کند. دفاع: predicate فقط برای تفکیک‌های داخلی ساده؛ برای داده بیرونی، schema که خودش خروجی تایپ‌دار می‌دهد.

۲. branding بدون هزینه runtime چطور کار می‌کند؟

مهر، عضوی فقط-نوعی است (`unique symbol` در declare) که هیچ بازتابی در JS ندارد؛ ساخت مقدار مهردار فقط از دروازه‌ای ممکن است که validate می‌کند و با یک `as` مستند مهر می‌زند. از آن به بعد، هر `UserId` در امضاها یعنی «از دروازه رد شده» — گواهی compile-time با هزینه صفر.

Variance و سازگاری توابع

چرا (x: Animal) => void به (x: Cat) => void قابل انتساب است — و برعکس نه

اهداف یادگیری این فصل

- چهار حالت variance را با مثال TypeScript توضیح دهی
- تفاوت سخت‌گیری method و property function را بدانی
- خطاهای واقعی ناشی از variance را تفسیر کنی

این فصل سخت‌ترین سؤال نظری سیستم نوع را عملی می‌کند: وقتی `A` زیرنوع `B` است، تکلیف `Container<A>` نسبت به `Container<B>` چیست؟ و چرا توابع برعکس رفتار می‌کنند؟ اگر این فصل بنشیند، دسته‌ای از خطاهای «بی‌منطق» برایتان منطقی می‌شوند — و در مصاحبه‌های senior حرفی برای گفتن دارید.

چهار حالت variance

فرض: `Dog` زیرنوع `Animal` است. رابطه `F<Dog>` با `F<Animal>` چهار حالت دارد:

حالت	رابطه	مثال در TypeScript
Covariant (هم‌جهت)	`F<Dog>` → `F<Animal>`	خروجی تابع، `readonly T[]`، propertyهای خواندنی
Contravariant (خلاف‌جهت)	`F<Animal>` → `F<Dog>`	**پارامترهای تابع** (با strictFunctionTypes)
Invariant (هیچ‌کدام)	فقط دقیقاً همان	property قابل نوشتن، `T[]` در تئوری*
Bivariant (هر دو — ناسالم)	هر دو جهت	پارامترهای **method** — تسامح عمدی

*TypeScript آرایه قابل‌نوشتن را عملاً covariant فرض می‌کند — تسامحی عمدی و ناسالم برای کاربردپذیری؛ در ادامه می‌بینیم.

چرا پارامتر تابع خلاف‌جهت است؟

شهود را با یک مثال واقعی بسازید — نه با فرمول:

contravariance.ts

type-checks ✓

```ts
type Animal = { name: string };
type Dog    = Animal & { bark(): void };

// «مصرف‌کننده‌ای که هر حیوانی را هندل می‌کند»:
type AnimalHandler = (a: Animal) => void;
// «مصرف‌کننده‌ای که فقط سگ هندل می‌کند»:
type DogHandler = (d: Dog) => void;

declare function onDogAdopted(handler: DogHandler): void;

const generalHandler: AnimalHandler = a => console.log(a.name);
onDogAdopted(generalHandler);  // ✓ هر که Animal را هندل کند، Dog را هم می‌تواند
```

contravariance-err.ts

× خطای عمدی

```ts
type Animal = { name: string };
type Dog = Animal & { bark(): void };
declare function onAnimalSeen(handler: (a: Animal) => void): void;

const dogOnly = (d: Dog) => d.bark();
onAnimalSeen(dogOnly);  // × بیاید، bark ندارد — crash
```

پیام کامپایلر

```
contravariance-err.ts:6:14 - error TS2345: Argument of type
'(d: Dog) => void' is not assignable to parameter of type
'(a: Animal) => void'.
  Types of parameters 'd' and 'a' are incompatible.
    Property 'bark' is missing in type '{ name: string; }'.
```

قاعده را حالا می‌توان گفت: تابع جانشین باید **کمتر یا مساوی طلب کند** (پارامتر: عمومی‌تر مجاز) و **بیشتر یا مساوی تحویل دهد** (خروجی: خاص‌تر مجاز). به انگلیسی حفظش کنید — در منابع همین را می‌بینید: be liberal in what you accept, conservative in what you return.

method bivariance — تسامح عمدی

bivariance.ts

type-checks ✓

```ts
interface EventListeners {
  // syntax متدی — bivariant بررسی می‌شود (حتی با strict):
  onEvent(e: MouseEvent): void;
}
interface EventListenersStrict {
  // syntax property ای — contravariant دقیق:
  onEvent: (e: MouseEvent) => void;
}

//  چرا؟ سازگاری با الگوهای رایج DOM مثل:
// addEventListener("click", (e: MouseEvent) => ...) روی امضای Event-پذیر
declare const a: EventListeners; declare const b: EventListenersStrict;
void a; void b;
```

نتیجه مهندسی: **در interfaceهای خودتان، callbackها را به شکل property بنویسید** (`onEvent: (e) => void`) نه method — بررسی دقیق‌تری می‌گیرید. شکل متدی را برای متدهای واقعی نگه دارید.

آرایه‌ها — سوراخ آگاهانه

array-hole.ts

type-checks ✓

```ts
type Animal = { name: string };
type Dog = Animal & { bark(): void };

const dogs: Dog[] = [{ name: "Rex", bark: () => {} }];
const animals: Animal[] = dogs;  //  ✓ کامپایل می‌شود (covariance ناسالم)

animals.push({ name: "Whiskers" }); //  گربه داخل آرایه سگ‌ها!
dogs[0]?.bark();                    //  هنوز OK...
dogs[1]?.bark();                     // 💥 runtime: bark is not a function
```

TypeScript این را عمداً اجازه می‌دهد — invariant کردن آرایه‌ها نیمی از کد دنیا را می‌شکست. دفاع شما: جایی که آرایه را فقط می‌خوانید، `readonly T[]` بگیرید؛ readonly چون متد نوشتن ندارد، covariance آن کاملاً سالم است. (و این دلیل عمیق‌ترِ توصیه فصل ۱۲ است.)

مدل ذهنی — جهت جریان داده، جهت variance

به جای حفظ جدول: بپرسید داده از کدام طرف **وارد** می‌شود و از کدام طرف **خارج**. خروجی‌ها (داده بیرون می‌آید) هم‌جهت‌اند؛ ورودی‌ها (داده تو می‌رود) خلاف‌جهت؛ جایی که هر دو اتفاق می‌افتد (property قابل نوشتن) هیچ‌جهت. تمام جدول بالا از همین یک جمله درمی‌آید.

در عمل — کجا این دانش به دردتان می‌خورد

- خطاهای «Types of parameters ... are incompatible» — حالا می‌دانید جهت مقایسه پارامترها برعکس است و پیام را درست می‌خوانید.
- طراحی interfaceهای listener/plugin — شکل property برای callbackها.
- in/out annotation روی پارامترهای جنریک (TS 4.7+): `interface Producer<out T>` — بیشتر برای صراحت و سرعت کامپایل در نوع‌های حلقوی بزرگ؛ رفتار پیش‌فرض استنتاجی معمولاً درست است.

زیر کاپوت — چرا این‌طور است؟

variance را از یک سؤال ساده استخراج کنید — «جایگزینی امن است؟» — و دیگر هرگز جدولش را حفظ نخواهید کرد.

`Box<Dog>` کی جای `Box<Animal>` امن است؟ به جهت جریان داده نگاه کنید. اگر Box فقط **تولیدکننده** است (فقط می‌خوانید)، هر چه از Box<Dog> بیرون بیاید Dog است و Dog یک Animal است — امن؛ این **covariance** است: جهت رابطه حفظ می‌شود. اگر Box فقط **مصرف‌کننده** است (فقط می‌نویسید)، قضیه برعکس: جایی که Animal می‌پذیرد، هر Dog ای را هم می‌پذیرد، پس `Sink<Animal>` جای `Sink<Dog>` می‌نشیند — **contravariance**: جهت رابطه وارونه. و اگر هم می‌خواند هم می‌نویسد، هیچ‌کدام از دو جهت امن نیست — **invariance**. همین. پارامترهای تابع مصرف‌کننده‌اند → contravariant؛ خروجی تولیدکننده است → covariant. همه «قواعد» فصل، مشتق همین استدلال‌اند.

حالا دو حفره آگاهانه TypeScript: **آرایه‌ها covariant اند** درحالی‌که خواندنی-نوشتنی‌اند و باید invariant می‌بودند — `dogs` را به `animals` بدهید، کسی گربه pushمی‌کند و آرایه Dog شما آلوده می‌شود، بی‌خطای کامپایل. چرا پذیرفتند؟ چون invariance آرایه، حجم عظیمی از کد JS بی‌خطر واقعی را رد می‌کرد (اکثر توابع فقط می‌خوانند). راه فرار اصولی را فصل ۱۲ داده بود: پارامتر `readonly T[]` بگیرید — فقط-خواندنی، covariance اش واقعاً سالم است.

متدها bivariant اند (هر دو جهت!) مگر این‌که با syntax property تعریف شوند — میراث دوران پیش از `strictFunctionTypes`، که عمداً برای متدها نگه داشته شد تا الگوهای رایج DOM و event نشکنند. درس طراحی: در interface ها callback را به شکل property بنویسید (`on: (e: E) ⇒ void` نه `on(e: E): void`) تا بررسی سخت‌گیرانه contravariance فعال بماند. حفره‌ها را زبان به شما تحمیل نکرده — فقط باید بدانید درها کجا باز مانده‌اند.

تمرین

قبول یا رد — با دلیل variance

با فرض `Dog extends Animal`، کدام انتساب‌ها مجازند؟

quiz24.ts

× خطای عمدی

```ts
interface Animal { name: string }
interface Dog extends Animal { bark(): void }

declare let f1: () => Dog;
declare let f2: () => Animal;
declare let g1: (d: Dog) => void;
declare let g2: (a: Animal) => void;
declare let r1: readonly Dog[];
declare let r2: readonly Animal[];

f2 = f1; //  ۱
f1 = f2; //  ۲
g1 = g2; //  ۳
g2 = g1; //  ۴
r2 = r1; //  ۵
```

پاسخ: ۱. ✓ خروجی covariant (سگ تحویل می‌دهد، حیوان خواسته بودند). ۲. ✗. ۳. ✓ پارامتر contravariant (هندلر عمومی‌تر جانشین خاص‌تر). ۴. ✗ — همان مثال گربه/readonly. ۵. ✓ bark آرایه covariant سالم است.

پرسش‌های مصاحبه این فصل

۱. چرا پارامترهای method به‌صورت shorthand ملایم‌تر بررسی می‌شوند؟

method shorthand عمداً bivariant است — میراثی برای کاربردپذیری (وگرنه APIهای DOM/آرایه سخت می‌شدند). ریسک: جانشینِ ناسالم رد نمی‌شود. برای interface هایی که callback قرارداد می‌کنند، عضو را به شکل property تابعی بنویسید (`on: (e: E) => void`) تا strictFunctionTypes کامل اعمال شود.

۲. چرا covariance آرایه قابل‌نوشتن ناسالم است ولی `readonly T[]` سالم؟

ناسالمی از «مسیر نوشتن» می‌آید: Dog[] را Animal[] بگیر، گربه push کن — آرایه سگ‌ها آلوده شد. readonly مسیر نوشتن را حذف می‌کند، پس خواندنِ Dog به‌عنوان Animal همیشه درست است. این دقیقاً همان قاعده عمومی variance است: خروجی/خواندن covariant، ورودی/نوشتن contravariant.

Overload، Declaration Merging و Module Augmentation

چند امضا برای یک تابع، و گسترش نوع‌های دیگران

اهداف یادگیری این فصل

- overload بنویسی و ترتیب و خطرات آن را بدانی
- با module augmentation نوع کتابخانه‌ها را گسترش دهی
- بدانی کی جنریک یا union جایگزین بهتر overload است

سه سازوکار این فصل یک وجه مشترک دارند: چند اعلان، یک موجودیت. overload چند امضا به یک تابع می‌دهد؛ merging چند اعلان را یک نوع می‌کند؛ augmentation این ادغام را به ماژول‌های دیگران می‌برد. هر سه قدرتمند، هر سه با دام‌های مخصوص خودشان.

Function Overload — چند قرارداد، یک پیاده‌سازی

وقتی رابطه ورودی→خروجی «شرطی» است و جنریک ساده بیانش نمی‌کند:

overloads.ts — type-checks ✓

```ts
// امضاهای عمومی — چیزی که مصرف‌کننده می‌بیند (از بالا به پایین امتحان می‌شوند):
function parseInput(input: string): string[];
function parseInput(input: string, asNumber: true): number[];
// امضای پیاده‌سازی — عمومی‌ترین؛ از بیرون نامرئی است:
function parseInput(input: string, asNumber?: boolean): string[] | number[] {
  const parts = input.split(",").map(s => s.trim());
  return asNumber ? parts.map(Number) : parts;
}

const tags = parseInput("a, b, c");        // string[] — دقیق
const ids  = parseInput("1, 2, 3", true);  // number[] — دقیق
void tags; void ids;
```

بدون overload، خروجی `string[] | number[]` می‌شد و هر مصرف‌کننده باید narrowing می‌کرد. قواعد حیاتی:

- **ترتیب مهم است:** کامپایلر اولین امضای سازگار را انتخاب می‌کند — خاص‌ترین را بالاتر بگذارید.
- **امضای پیاده‌سازی بررسی سختی با امضاهای عمومی ندارد** — می‌تواند با آن‌ها ناسازگارِ خطرناک باشد و کامپایلر فقط سازگاری حداقلی را چک می‌کند. امضاهای عمومی را تست کنید (فصل ۳۲).
- قبل از overload بپرسید: آیا جنریک + conditional type همین را تمیزتر نمی‌گوید؟ برای دو حالت ثابت، overload خواناتر است؛ برای رابطه قاعده‌مند، conditional (فصل ۱۷).

Declaration Merging — جمع اعلان‌های هم‌نام

از فصل ۱۳ می‌دانید interfaceهای هم‌نام merge می‌شوند. این در سه سطح کار می‌کند: interface + interface، namespace + function (الگوی قدیمی jQuery)، و — مهم‌تر از همه برای شما — **در فایل‌های اعلان، روی ماژول‌های دیگران**.

Module Augmentation — گسترش نوع کتابخانه‌ها

سناریوی واقعی: middleware شما به `Request` اکسپرس فیلد `user` اضافه می‌کند. بدون augmentation، در هر handler باید `(req as any).user` بنویسید. با آن:

types/express.d.ts — type-checks ✓

```ts
// در — فایل اعلان پروژه include قرار دارد
import type { AuthUser } from "../src/auth";

declare module "express-serve-static-core" {
  interface Request {
    user?: AuthUser;    // merge با Request اصلی
  }
}
```

handler.ts — type-checks ✓

```ts
import express from "express";    // types/express.d.ts بالا در پروژه حاضر است

const app = express();
app.get("/me", (req, res) => {
  if (!req.user) return res.status(401).end();
  res.json({ id: req.user.id });  //  ✓ تایپ‌شده، بدون cast
});
```

و پرکاربردترین augmentation دنیا — متغیرهای محیطی:

types/env.d.ts — type-checks ✓

```ts
declare global {
  namespace NodeJS {
    interface ProcessEnv {
      DATABASE_URL: string;
      REDIS_URL?: string;
      NODE_ENV: "development" | "production" | "test";
    }
  }
}
export {};  // فایل را ماژول نگه می‌دارد تا declare global معتبر باشد
```

از این به بعد `process.env.DATABASE_URL` از نوع `string` است و `process.env.DATABASE_URl` (typo) خطای property ناشناخته می‌دهد.

> **هشدار — augmentation ادعاست، نه تضمین**
>
> اعلانِ `DATABASE_URL: string` باعث نمی‌شود متغیر در runtime وجود داشته باشد — فقط کامپایلر را قانع می‌کند. ادعای نوع را با اعتبارسنجی واقعی هنگام boot جفت کنید (مثلاً schema روی `process.env` با Zod — فصل ۳۴). قاعده کلی: **هر declare، یک بدهی اثبات در runtime است.**

قواعد فنی augmentation که وقت دیباگ جانتان را نجات می‌دهند:

- فایل حاوی `import` / `export` سطح بالا «ماژول» است؛ برای دست زدن به scope سراسری داخل ماژول، `declare global` لازم است.
- نام ماژول در `declare module "..."` باید دقیقاً همان specifier واقعی باشد — برای Express، نوع‌های Request واقعاً در `express-serve-static-core` زندگی می‌کنند؛ augment کردن `"express"` خالی اثر نمی‌کند. (نام را از خود `.d.ts` کتابخانه پیدا کنید.)
- فایل augmentation باید در `include` پروژه باشد وگرنه بی‌صدا نادیده گرفته می‌شود — رایج‌ترین علت «augmentation کار نمی‌کند».

> **مدل ذهنی — merging یعنی «نوع‌ها open-world اند»**
>
> در TypeScript، interface و module بسته نیستند؛ هر فایلِ در scope می‌تواند عضوی اضافه کند. این برای اکوسیستم پلاگین‌محور (Express، Vue، Fastify) حیاتی است — و دقیقاً به همین دلیل در کد اپلیکیشن باید به‌ندرت استفاده شود: نوعی که همه‌جا باز است، هیچ‌جا قابل اعتماد نیست. augmentation را فقط در پوشه `types/` با نام‌گذاری روشن نگه دارید.

زیر کاپوت — چرا این‌طور است؟

دو سازوکار این فصل، هر دو «قرارداد اجتماعی اکوسیستم» هستند و جزئیاتشان از همین منظر قابل فهم است:

چرا «اولین تطبیق» است، نه «بهترین»؟ overload resolution الگوریتم بهترین-تطبیق (مثل C# و Java) باید برای هر فراخوانی، همه امضاها را بسنجد و با قواعد پیچیده «خاص‌تر بودن» برنده را انتخاب کند — قواعدی که در حضور union و structural typing و استنتاج جنریک، هم کند می‌شد هم غیرقابل پیش‌بینی. انتخاب TypeScript ساده و قطعی است: از بالا به پایین، اولین امضای سازگار می‌برد و جستجو تمام. پیامد مستقیم: **ترتیب، بخشی از API شماست.** امضای خاص بعد از امضای عام، کد مرده است — عام همیشه زودتر تطبیق می‌یابد. و چون بدنه پیاده‌سازی با امضاهای عمومی فقط «سازگاری سست» دارد (بدنه باید همه حالت‌ها را پوشش دهد ولی خودش overload نیست)، دروغ گفتن در امضاها ممکن است — overload را مثل قرارداد عمومی، آزموده و حداقلی نگه دارید. و همان توصیه همیشگی: اگر union ساده کار را درمی‌آورد، overload نسازید؛ overload برای وقتی است که «رابطه ورودی-خروجی» بین امضاها عوض می‌شود.

چرا اصلاً وجود دارند؟ declaration merging و module augmentation چون TypeScript باید اکوسیستمی را تایپ کند که خودش مالکش نیست. JS دهه‌ها فرهنگ «گسترش از بیرون» داشته: پلاگین‌های Express که به Request عضو اضافه می‌کنند، middleware هایی که config را غنی می‌کنند. سیستمِ نوعِ بسته، این فرهنگ را یا رد می‌کرد یا به any حواله

می‌داد. interface باز (فصل ۱۳) + augmentation یعنی: هر بسته‌ای می‌تواند «سهم نوعی» خودش را به قرارداد مشترک اضافه کند و کامپایلر همه سهم‌ها را merge می‌کند. بهای این قدرت، عمل از راه دور است: نوعی که می‌بینید ممکن است حاصل جمعِ چند فایل ناآشنا باشد. قاعده بهداشتی: augmentation فقط برای نوع‌هایی که «واقعاً» گسترششان داده‌اید (middleware ای که واقعاً آن عضو را می‌سازد)، همیشه در فایل‌های اختصاصی با نام روشن، و هرگز برای دور زدن نقص نوع‌های خودتان.

تمرین

overload یا conditional؟

تابع `createElement` با رشته `"a"` باید `HTMLAnchorElement` و با `"div"` باید `HTMLDivElement` برگرداند — برای همه تگ‌های HTML. overload می‌نویسید یا راه بهتری هست؟

پاسخ: صدها overload لازم می‌شد! رابطه «قاعده‌مند» است و نگاشت آماده در lib وجود دارد — جنریک + indexed access:

quiz25-solution.ts — type-checks ✓

```ts
function create<K extends keyof HTMLElementTagNameMap>(
  tag: K
): HTMLElementTagNameMap[K] {
  return document.createElement(tag);
}
const link = create("a");    // HTMLAnchorElement
const box  = create("div");  // HTMLDivElement
void link; void box;
```

درس کلی: overload برای چند حالتِ **بی‌قاعده**؛ جنریک + نگاشت برای حالت‌های **قاعده‌مند**. خود lib.dom هم دقیقاً همین الگو را دارد.

پرسش‌های مصاحبه این فصل

۱. ترتیب overload ها چرا مهم است؟

کامپایلر از بالا اولین امضای match را برمی‌دارد؛ اگر امضای عمومی بالاتر از خاص باشد، خاص هرگز انتخاب نمی‌شود و خروجی‌ها گشاد می‌شوند. قاعده: خاص‌ترین اول. و یادتان باشد امضای پیاده‌سازی از بیرون نامرئی است و سازگاری‌اش با تک‌تک overload ها فقط حداقلی بررسی می‌شود — تست لازم است.

۲. module augmentation چه چیزی را «ثابت نمی‌کند»؟

هیچ چیز runtime را. `declare module` فقط ادعاست — این‌که `req.user` واقعاً پر می‌شود کار middleware است و این‌که `process.env.X` واقعاً هست کار schema در boot. الگوی سالم: هر augmentation جفت شود با کدی که ادعا را در runtime برقرار می‌کند.

برنامه‌نویسی Type-Level — قدرت، هزینه و مرز عقل

نوع‌های بازگشتی، کارایی کامپایلر و طراحی API عمومی خوانا

اهداف یادگیری این فصل

- نوع بازگشتی بنویسی و محدودیت عمق آن را بدانی
- هزینه کامپایل نوع‌های پیچیده را تشخیص و کاهش دهی
- برای API عمومی بین دقت و خوانایی تعادل برقرار کنی

بخش دوم به شما قدرت زیادی داده است — کافی است توییتر تایپ‌اسکریپتی‌ها را ببینید: پیاده‌سازی بازی شطرنج در سطح نوع، parser کامل SQL در جنریک‌ها. این فصل درباره مرز است: نوع بازگشتی تا کجا؟ هوشمندی تا کجا؟ و چه چیزی API عمومی خوب می‌سازد؟ پاسخ‌ها مهندسی‌اند، نه ایدئولوژیک.

نوع‌های بازگشتی — ابزار مشروع، با سقف

recursive.ts — type-checks ✓

```ts
// نمونه کلاسیک بازگشت سالم — مدل JSON:
type Json = string | number | boolean | null | Json[] | { [key: string]: Json };

const valid: Json = { users: [{ name: "Ali", tags: ["a"], active: true }] };

// بازگشت با template literal — مسیرهای تودرتوی یک شیء:
type Paths<T> = T extends object
  ? { [K in keyof T & string]: T[K] extends object ? K | `${K}.${Paths<T[K]>}` : K }
[keyof T & string]
  : never;

interface Settings { ui: { theme: string; font: { size: number } }; lang: string }
type SettingKey = Paths<Settings>;
// "ui" | "lang" | "ui.theme" | "ui.font" | "ui.font.size"
void valid; void (0 as unknown as SettingKey);
```

`Paths` در کتابخانه‌های فرم (react-hook-form) واقعاً همین‌طور پیاده شده و تجربه فوق‌العاده‌ای می‌دهد. اما بدانید سقف کجاست: کامپایلر عمق نمونه‌سازی نوع را محدود می‌کند (خطاهای «Type instantiation is excessively deep and possibly infinite», TS2589) — روی ساختارهای عمیق یا جنریک‌های تودرتو، این الگوها ناگهان می‌شکنند یا ادیتور را می‌خزانند.

هزینه واقعی نوع‌های پیچیده

هزینه در سه‌جا پرداخت می‌شود، و فقط یکی‌اش فنی است:

۱. **کارایی کامپایل و ادیتور:** هر instantiation جنریک، کار واقعی کامپایلر است. conditional تودرتو × union بزرگ × بازگشت = تاخیر محسوس تکمیل خودکار برای کل تیم، هر روز. (ابزار اندازه‌گیری: فصل ۳۲.)

۲. **پیام‌های خطا:** مصرف‌کننده‌ای که آرگومان اشتباه بدهد، به‌جای «string بده»، چهل خط بازشده mapped/conditional دریافت می‌کند. API که خطاهایش قابل خواندن نیست، در عمل مستند هم نیست.

۳. **نگه‌داری:** نوع هوشمندانه‌ای که فقط نویسنده‌اش می‌فهمد، یک bus factor یک‌نفره در قلب قرارداد تیم است.

cost-example.ts

type-checks ✓

```ts
// «هوشمند» — camelCase کردن کلیدها در سطح نوع:
type CamelCase<S extends string> =
  S extends `${infer H}_${infer T}` ? `${H}${Capitalize<CamelCase<T>>}` : S;
type CamelKeys<T> = { [K in keyof T & string as CamelCase<K>]: T[K] };

// سؤال مهندسی: آیا این پیچیدگی لازم است؟
// اگر داده از API snake_case می‌آید، تبدیل واقعی در runtime اتفاق می‌افتد —
// پس نوع خروجیِ «تابع تبدیل» باید این باشد، و آنجا ارزشش را دارد.
// اگر فقط برای زیبایی mock — نه — هاست.
type Row = { user_id: number; created_at: string };
type ApiRow = CamelKeys<Row>;   // { userId: number; createdAt: string }
void (0 as unknown as ApiRow);
```

مدل ذهنی — بودجه پیچیدگی

برای هر نوع پیچیده بپرسید: «چند مصرف‌کننده × چند بار در روز، از این دقت سود می‌برند؟» و «چند نفر می‌توانند نگهش دارند؟» نوع پیچیده در **هسته یک کتابخانه پرمصرف** (مثل Zod) سرمایه‌گذاری است؛ همان نوع در یک util اپلیکیشنی، بدهی. پیچیدگی را جایی خرج کنید که ضریب استفاده بالاست.

طراحی API عمومی خوانا — چهار تکنیک

۱. نام بدهید، باز نکنید. به‌جای این‌که مصرف‌کننده در tooltip با `Omit<Pick<...>> & {...}` روبه‌رو شود، نوع میانی نام‌دار بسازید و `Prettify` (فصل ۲۰) را روی خروجی‌های عمومی اعمال کنید.

۲. مرز هوشمندی را داخل نگه دارید. امضای عمومی ساده؛ جادو در internalها:

boundary.ts

type-checks ✓

```ts
// عمومی — ساده و قابل مستندسازی:
export interface QueryResult<Row> { rows: Row[]; total: number }
export declare function query<Row>(sql: string): Promise<QueryResult<Row>>;

// داخلی — هر چقدر لازم است پیچیده (و آزاد برای تغییر):
type ParseColumns<SQL extends string> =
  SQL extends `SELECT ${infer Cols} FROM ${string}` ? Cols : never;
type X = ParseColumns<"SELECT id, name FROM users">;  // "id, name"
void (0 as unknown as X);
```

۳. خطا را طراحی کنید. constraint با نام معنادار بنویسید تا پیام خطا راهنما شود؛ حتی تکنیک «نوع خطای سفارشی» — برگرداندن literal توضیحی از conditional — در کتابخانه‌های جدی رایج است.

۴. تست نوع بنویسید. API عمومی بدون تست نوع (فصل ۳۲)، با هر refactor داخلی ممکن است برای مصرف‌کننده‌ها عوض شود — بی‌صدا.

در عمل — چک‌لیست قبل از merge یک نوع پیچیده

- آیا قاعده را می‌توان در یک جمله گفت؟ (فصل ۲۰)
- آیا tooltip نتیجه برای مصرف‌کننده خوانا است؟ (با hover واقعی چک کنید)
- آیا یک خطای عمدی، پیام قابل فهم می‌دهد؟ (بشکنیدش و ببینید)
- آیا تست نوع دارد؟
- آیا جایگزین ساده‌تر — overload، چند alias صریح، یا کمی تکرار — واقعاً بدتر است؟

زیر کاپوت — چرا این‌طور است؟

type-level programming واقعی است چون سیستم نوع TypeScript — با infer + conditional + بازگشت — از نظر محاسباتی کامل است: کامپایلر در زمان check، واقعاً «برنامه شما را اجرا می‌کند». این حقیقت، هم قدرت فصل را توضیح می‌دهد هم محدودیت‌هایش را:

چرا سقف عمق بازگشت وجود دارد؟ چون ارزیابی نوعِ بازگشتی، واقعاً بازگشتِ در کامپایلر است — بدون سقف، یک نوع بدپیمان می‌توانست tsc را در حلقه بی‌نهایت یا stack overflow بیندازد (و editor شما هر بار با هر کلید، همین ارزیابی را اجرا می‌کند!). سقف حدود ۵۰ برای بازگشت عادی است، اما TypeScript برای الگوی خاصی ارفاق بزرگی دارد: اگر فراخوانی بازگشتی، «کل نتیجه» باشد (tail-recursive — مثل الگوی accumulator)، کامپایلر آن را به حلقه تبدیل می‌کند و سقف به حدود هزار می‌رسد. برای همین نسخه انباشتگر `Reverse<S, Acc>` رشته‌های بلند را درمی‌آورد و نسخه ساده نه — دقیقاً همان بهینه‌سازی tail-call زبان‌های تابعی، این‌بار در ارزیاب نوع.

چرا کارایی type-level مهم است؟ چون این «برنامه» در هر check اجرا می‌شود — در CI یک‌بار، در editor مدام. سه هزینه اصلی: union های بزرگ (ضرب دکارتی فصل ۱۹)، بازگشت عمیق، و نوع‌های جنریکی که cache نمی‌شوند چون هر بار با آرگومان تازه instantiate می‌شوند. کامپایلر نتایج instantiation را cache می‌کند — ولی فقط برای ترکیب آرگومان‌های تکراری؛ نوعی که ورودی‌هایش همیشه تازه‌اند، همیشه از نو محاسبه می‌شود.

مرز مهندسی کجاست؟ معیار پیشنهادی این کتاب: نوع پیچیده وقتی موجه است که «قراردادی را اثبات کند که مصرف‌کننده‌های زیادی از آن سود می‌برند» — مسیرهای type-safe یک router، کلیدهای یک event bus. وقتی پیچیدگی برای یک نقطه مصرف است، یا پیام خطایش برای همکارتان قابل رمزگشایی نیست، همان قرارداد را با کد ساده‌تر + تست بیان کنید. نوع‌ها هم کدند: هزینه نگهداری دارند.

تمرین

بازبینی مهندسی

همکارتان این نوع را برای «گرفتن مقدار با مسیر نقطه‌دار» نوشته و می‌خواهد merge کند. دو سؤال فنی و یک پیشنهاد جایگزین مطرح کنید:

quiz26.ts — type-checks ✓

```ts
type Get<T, P extends string> =
  P extends `${infer K}.${infer Rest}`
    ? K extends keyof T ? Get<T[K], Rest> : never
    : P extends keyof T ? T[P] : never;
declare function get<T, P extends string>(obj: T, path: P): Get<T, P>;
```

پاسخ نمونه: سؤال ۱: رفتار با آرایه‌ها و ایندکس عددی (`"items.0.name"`) چیست؟ الان never می‌شود — یعنی نیمی از مسیرهای واقعی. سؤال ۲: وقتی مسیر غلط است never برمی‌گردد و خطا به نقطه مصرفِ دورتر منتقل می‌شود — بهتر نیست P به `Paths<T>` محدود شود تا خطا در محل فراخوانی و با پیشنهاد تکمیل خودکار باشد؟ پیشنهاد: اگر فقط ۲-۳ مصرف‌کننده دارد، امضای ساده‌تر `get(obj, ["items", 0, "name"] as const)` با tuple، هم ساده‌تر پیاده می‌شود هم خطاهای بهتری می‌دهد.

پرسش‌های مصاحبه این فصل

۱. TS2589 گرفتید — چه می‌کنید؟

یعنی instantiation بی‌نهایت یا خیلی عمیق: نوع بازگشتی بدون شرط توقف درست، یا ورودی عمیق‌تر از سقف. اول شرط پایان را چک می‌کنم (primitive ها و تابع‌ها باید خارج شوند)، بعد اگر عمق واقعاً لازم است شمارنده عمق tuple-based می‌گذارم، و در نهایت می‌پرسم: آیا این پیچیدگی اصلاً باید type-level باشد؟

۲. مرز «پیچیدگی type-level می‌ارزد» کجاست؟

سه هزینه را با ضریب استفاده مقایسه کنید: کندی کامپایل/ادیتور، پیام خطای غیرقابل خواندن برای مصرف‌کننده، و bus factor. هسته کتابخانه با صدها مصرف‌کننده: بله، با تست نوع و امضای عمومی ساده. util داخلی یک فیچر: تقریباً هرگز — همان overload یا چند امضای صریح.

PART 03

۳

کامپایلر، ابزار و معماری

TypeScript فقط یک زبان نیست؛ یک زنجیره ابزار است. این بخش ماژول‌ها و resolution، فایل‌های اعلان، مهاجرت تدریجی، تست نوع، CI و گذار تاریخی TypeScript 6 به 7 (کامپایلر native) را پوشش می‌دهد — با معیار انتخاب، نه فهرست گزینه‌ها.

---

27 خانواده strict — طراحی سطح سخت‌گیری

28 ماژول‌ها — CommonJS، ESM و Module Resolution

29 فایل‌های اعلان و @types

30 مهاجرت از JavaScript — تدریجی و قابل برگشت

31 Monorepo، Project References و CI

32 تست نوع، دیباگ خطاهای پیچیده و کارایی کامپایلر

33 گذار TypeScript 6 به 7 — کامپایلر Native

خانواده strict — طراحی سطح سخت‌گیری

از noImplicitAny تا exactOptionalPropertyTypes و noUncheckedIndexedAccess

اهداف یادگیری این فصل

- برای هر گزینه strict بدانی چه باگی را می‌گیرد و چه هزینه‌ای دارد
- دو گزینه فراتر از strict را آگاهانه ارزیابی کنی
- سیاست سخت‌گیری تدریجی برای تیم تعریف کنی

فصل ۳ گفت `strict: true` نقطه شروع است، نه خط پایان. این فصل عمیق می‌شود: اعضای چتر strict دقیقاً چه می‌کنند، دو گزینه مهمِ خارج از چتر چه چیزی اضافه می‌کنند، و چطور برای یک تیم واقعی «سیاست سخت‌گیری» طراحی کنیم که مقاومت ایجاد نکند.

اعضای چتر strict — هرکدام علیه یک کلاس باگ

گزینه	کلاس باگی که حذف می‌کند
`strictNullChecks`	دسترسی به مقدار غایب — فصل ۷
`noImplicitAny`	نشت بی‌صدای any از پارامترهای بی‌annotation
`strictFunctionTypes`	جانشینی ناسالم توابع — فصل ۲۴
`strictBindCallApply`	bind/call/apply بدون بررسی آرگومان
`strictPropertyInitialization`	property کلاس که هیچ‌جا مقدار نگرفته
`noImplicitThis`	this ناشناخته در توابع آزاد
`useUnknownInCatchVariables`	متغیر catch به‌جای any، unknown — خطای واقعی ممکن است Error نباشد
`alwaysStrict`	emit با "use strict" و پارس strict

`strict: true` یعنی همه این‌ها، به‌علاوه اعضای آینده — ارتقای نسخه ممکن است بررسی جدید بیاورد؛ این feature است، نه باگ: ارتقا = سخت‌گیری رایگان.

فراتر از strict — دو گزینه که باید جدی بگیرید

noUncheckedIndexedAccess — صادق‌ترین گزینه کامپایلر:

nuia.ts

type-checks ✓

```ts
const limits: Record<string, number> = { free: 1, pro: 20 };

//  بدون گزینه: limit: number — خوش‌بینی بی‌پشتوانه
//  با گزینه:  limit: number | undefined — واقعیت
const limit = limits["enterprise"];

//  کد مجبور می‌شود تصمیم صریح بگیرد:
const effective = limit ?? 0;

//  آرایه‌ها هم همین‌طور:
const parts = "a,b".split(",");
const second = parts[1];          // string | undefined — درست است!
void effective; void second;
```

هزینه: در کدی که با ایندکس‌های «مطمئن» کار می‌کند (حلقه‌های ایندکسی، الگوریتم‌ها) بررسی‌های اضافه می‌خواهد — الگوهای جایگزین: `for...of`، `entries()`، `.at()` با بررسی، یا در موارد اثبات‌پذیر `!` با کامنت. توصیه این کتاب: در پروژه جدید روشن کنید؛ ارزش حذف یک کلاس کامل `undefined is not a function` را دارد.

exactOptionalPropertyTypes — تفاوت «نبودن» با «undefined بودن»:

eopt.ts

× خطای عمدی

```ts
interface Profile { nickname?: string }

const p: Profile = {};                // ✓ نبودن
const q: Profile = { nickname: undefined }; //  × با گزینه فعال
```

پیام کامپایلر

```
eopt.ts:4:20 - error TS2375: Type '{ nickname: undefined; }' is not
assignable to type 'Profile' with 'exactOptionalPropertyTypes: true'.
Consider adding 'undefined' to the types of the target's properties.
```

چرا مهم است؟ چون `"nickname" in q` برابر true است و `Object.keys` آن را می‌شمارد؛ «حاضرِ undefined» و «غایب» رفتار runtime متفاوت دارند (patch، JSON.stringify، spreadهای دیتابیس). با این گزینه، `?` یعنی فقط «می‌تواند غایب باشد»؛ اگر undefined صریح هم می‌خواهید، بنویسید `nickname?: string | undefined`. هزینه‌اش: برخی کتابخانه‌ها هنوز با آن تمیز نیستند؛ در کد خودتان ارزشمند است، در مرز کتابخانه‌ها گاهی اصطکاک دارد.

گزینه‌های سلامت کد — مکمل‌های ارزان

health-options.jsonc

```jsonc
{
  "compilerOptions": {
    "noImplicitOverride": true,       // override ؛خطا = بدون کلیدواژه refactor امن کلاس‌ها
    "noFallthroughCasesInSwitch": true, //  با بدنه = خطا breakسقوط بی
    "noImplicitReturns": true,        //  کنند یا هیچ‌کدام return همه مسیرها یا
    "noUnusedLocals": true,           //  متغیر مرده (در CIاست مزاحم توسعه هنگام ؛)
    "noUnusedParameters": true
  }
}
```

دو مورد آخر را خیلی تیم‌ها به ESLint می‌سپارند تا در ادیتور «هشدار» باشند نه «خطا» — تصمیم معقولی است؛ مهم این است که در CI یک‌جا بررسی شوند.

مدل ذهنی — سخت‌گیری یعنی جابه‌جایی هزینه

هیچ گزینه‌ای باگ را مجانی حذف نمی‌کند؛ هزینه را از **آینده مبهم** (دیباگ production ساعت ۲ شب) به **حال مشخص** (بررسی صریح هنگام نوشتن) منتقل می‌کند. سؤال درست هنگام ارزیابی هر گزینه: «آیا حاضرم این هزینه را الان، در هر فایل، بپردازم تا آن کلاس باگ ناممکن شود؟» برای strictNullChecks پاسخ تقریباً همیشه بله است؛ برای exactOptionalPropertyTypes بستگی به دامنه دارد — و این تفاوتِ تصمیم مهندسی با کپی کردن config است.

در عمل — سیاست تدریجی برای تیم

- **پروژه جدید:** noUncheckedIndexedAccess + strict از روز اول؛ بقیه سلامت‌سنج‌ها در CI.
- **پروژه موجود:** هرگز «همه را یک‌جا روشن» نکنید — هزاران خطا، سرکوب انبوه با as و !، و بدتر از قبل. مسیر درست فصل ۳۰ است: گزینه‌به‌گزینه، با مرز «کد جدید strict، کد قدیمی به‌تدریج».
- **قانون طلایی تیمی:** هر `// @ts-expect-error` و `!` باید کامنتِ «چرا» داشته باشد؛ هر as باید در review توجیه شود. سرکوب بی‌دلیل، سخت‌گیری را به تئاتر تبدیل می‌کند.

زیر کاپوت — چرا این‌طور است؟

پرچم‌های strict را وقتی درست ارزیابی می‌کنید که بدانید هر کدام «مکانیکی» دقیقاً چه چیزی را در checker عوض می‌کند:

`strictNullChecks` عضویت null/undefined در همه نوع‌ها را لغو می‌کند — تغییر در خودِ جبر نوع‌ها (فصل ۷)، نه افزودن یک بررسی. سنگین‌ترین و ارزشمندترین پرچم. `strictFunctionTypes` بررسی پارامترهای تابع را از bivariant به contravariant سخت می‌کند — ولی فقط برای function type ها، نه متدها (استثنای عمدی فصل ۲۴). `noImplicitAny` سیاست «استنتاج شکست خورد → any خاموش» را به «استنتاج شکست خورد → خطا» تبدیل می‌کند؛ بدون آن، any بی‌صدا در کد نشت می‌کند و ویروس فصل ۸ پخش می‌شود. `strictPropertyInitialization` اثبات مقداردهی قطعی propertyهای کلاس را می‌خواهد — جلوی undefined ای که annotation پنهانش کرده.

`useUnknownInCatchVariables` متغیر catch را از any به unknown می‌برد — صادقانه، چون JS اجازه throw هر چیزی را می‌دهد (فصل ۳۵).

دو نکته تصمیم‌گیری از این تصویر درمی‌آید. اول: پرچم‌های strict «سطح سخت‌گیری» نیستند؛ هر کدام کلاس باگ متفاوتی را هدف می‌گیرند — خاموش کردن یکی یعنی پذیرش آگاهانه همان کلاس باگ، و این تصمیم باید در tsconfig کامنت داشته باشد. دوم: پرچم‌های خارج از چتر strict — به‌خصوص `noUncheckedIndexedAccess` و `exactOptionalPropertyTypes` — بیرون مانده‌اند نه چون کم‌ارزش‌اند، بلکه چون نویز بالایی روی کدهای موجود دارند؛ در پروژه سبز (greenfield) هر دو را روشن کنید.

و برای migration، ترتیب درست از همین مکانیک درمی‌آید: اول noImplicitAny (جلوی نشت جدید را می‌گیرد)، بعد strictNullChecks (بزرگ‌ترین بازده)، بعد بقیه. روشن کردن همه با هم روی codebase قدیمی، هزاران خطای درهم می‌دهد که تفکیکشان سخت‌تر از حل تدریجی‌شان است.

تمرین

کدام گزینه این باگ را می‌گرفت؟

برای هر باگ production، گزینه‌ای که از آن جلوگیری می‌کرد:

۱. `config.servers[0].host` روی آرایه خالی crash کرد.

۲. `catch (e) { log(e.message) }` روی رشته پرتاب‌شده crash کرد.

۳. subclass متدی را override کرد؛ بعداً نام متد والد عوض شد و override یتیم ماند.

۴. switch روی status، حالت جدید را بی‌صدا رد کرد و به case بعدی افتاد.

پاسخ: ۱. noUncheckedIndexedAccess — ایندکس 0 هم undefined پذیر می‌شد. ۲. useUnknownInCatchVariables (داخل strict) — دسترسی به .message روی unknown خطا می‌داد و narrowing می‌خواست. ۳. noImplicitOverride — متد یتیم، دیگر override چیزی نبود و خطا می‌گرفت. ۴. noFallthroughCasesInSwitch، و در سطح مدل‌سازی: exhaustive check فصل ۲۲ که حالت جدید را در همه switchها گزارش می‌کرد.

پرسش‌های مصاحبه این فصل

۱. خارج از چتر strict، کدام گزینه را حتماً روشن می‌کنید و چرا؟

`noUncheckedIndexedAccess`: ایندکس آرایه و Record را صادقانه `T | undefined` می‌کند — همان جایی که strict ساکت است و باگ‌های واقعی (آرایه خالی، کلید غایب) زندگی می‌کنند. کنارش `noImplicitOverride` که تقریباً مجانی است.

۲. سیاست سرکوب خطا در تیم شما چیست؟

فقط `@ts-expect-error` (نه ts-ignore — چون وقتی خطا رفع شود، خودش خطا می‌شود و پاک‌سازی را اجباری می‌کند)، همیشه با کامنت «چرا» و شناسه issue، و شمارش ترند سرکوب‌ها در CI. سرکوب بدون توضیح در review رد می‌شود — بدهی بی‌سند، بدهی فراموش‌شده است.

ماژول‌ها — CommonJS، ESM و Module Resolution

module، lib، target و nodenext؛ جایی که بیشتر پروژه‌ها زمین می‌خورند

اهداف یادگیری این فصل

- نقش مستقل module، lib، target و moduleResolution را بدانی
- تفاوت ESM و CJS را در سطح نوع و runtime توضیح دهی
- پیکربندی درست برای Node مدرن و bundler انتخاب کنی

هیچ بخشی از TypeScript به اندازه ماژول‌ها «کار می‌کند تا وقتی نمی‌کند» نیست. ریشه سردرگمی یک چیز است: **دو دنیای ماژول واقعی در JavaScript** (CommonJS و ESM) و وظیفه TypeScript برای مدل کردن هر دو، بدون اجرا کردن هیچ‌کدام. این فصل نقشه‌ای می‌دهد که با آن هر خطای resolution را بتوانید ریشه‌یابی کنید.

چهار گزینه، چهار سؤال متفاوت

گزینه	سؤالی که پاسخ می‌دهد
`target`	خروجی با چه syntax ای emit شود؟ (es2017؟ es2023؟)
`lib`	کدام APIهای built-in در نوع‌ها «وجود دارند»؟
`module`	importها به چه سیستمی emit شوند و چه معنایی داشته باشند؟
`moduleResolution`	متن `"./x"` یا `"lodash"` چطور به فایل واقعی نگاشت شود؟

خطای کلاسیک: target را es5 گذاشتن «برای سازگاری» و تعجب از اینکه `lib` هم es5 شده و `Promise` ناشناخته است. این چهار گزینه مستقل‌اند اما پیش‌فرض همدیگر را تعیین می‌کنند — صریح بنویسیدشان.

CJS و ESM — تفاوت‌هایی که نوع‌ها نمی‌پوشانند

esm-vs-cjs.ts — type-checks ✓

```
// ESM — استاندارد زبان، static، قابل تحلیل:
import { readFile } from "node:fs/promises";
export const VERSION = "1.0.0";

// CommonJS — قرارداد Node قدیمی، دینامیک:
// const fs = require("fs");  module.exports = {...}
```

تفاوت‌ها runtime واقعی دارند: ESM live binding و oistingه دارد، `__dirname` ندارد (به‌جایش `import.meta.url`)، و مهم‌تر: **فایل ESM نمی‌تواند با `require` قدیمی بارگذاری شود** (در Node جدید، require کردن ESM همگام تحت

شرایطی ممکن شده، اما قاعده کلی مرز را جدی بگیرید). نوع فایل را در Node این‌ها تعیین می‌کنند: پسوند `.mts` / `.cts` صریح، یا فیلد `"type"` در نزدیک‌ترین package.json.

پیکربندی درست برای دو دنیای رایج

۱. کد Node (سرور، CLI، پکیج):

tsconfig.node.jsonc

```jsonc
{
  "compilerOptions": {
    "module": "nodenext",          // moduleResolution=nodenext را هم می‌آورد
    "target": "es2023",
    "verbatimModuleSyntax": true
  }
}
```

`nodenext` یعنی «دقیقاً قواعد خود Node»: احترام به `exports` در package.json، اجبار پسوند در importهای نسبی ESM (`./util.js` — بله، `.js` حتی در فایل ts!)، و تفکیک cjs/esm بر اساس type. این سخت‌گیری‌ها آزاردهنده به نظر می‌رسند اما آینه runtime واقعی‌اند — بدون آن‌ها، tsc چیزی را قبول می‌کند که Node شب deploy رد خواهد کرد.

۲. کدی که bundler می‌سازد (فرانت‌اند با Vite/Next):

tsconfig.web.jsonc

```jsonc
{
  "compilerOptions": {
    "module": "esnext",
    "moduleResolution": "bundler",  //  قواعد سخاوتمندانه‌تر bundlerها
    "target": "es2022",
    "lib": ["es2022", "dom", "dom.iterable"],
    "noEmit": true                  // emit با bundler ؛ است tsc فقط چک می‌کند
  }
}
```

`bundler` همان resolution مدرن است منهای اجبار پسوند — چون esbuild/Vite خودشان حلش می‌کنند. انتخاب بین این دو پیکربندی، تقریباً همیشه پاسخ سؤال «module چی بذارم؟» است: **اجراکننده نهایی کیست؟ Node خالص ⇒ nodenext؛ bundler ⇒ bundler.**

exports در package.json — دروازه رسمی پکیج

package-exports.jsonc

```jsonc
{
  "name": "@acme/toolkit",
  "type": "module",
  "exports": {
    ".":        { "types": "./dist/index.d.ts", "default": "./dist/index.js" },
    "./schema": { "types": "./dist/schema.d.ts", "default": "./dist/schema.js" }
  }
}
```

وقتی `exports` هست، فقط مسیرهای اعلام‌شده قابل import اند — `@acme/toolkit/dist/internal` دیگر از بیرون باز نمی‌شود (کپسوله‌سازی واقعی، برای اولین بار در تاریخ npm). شرط دیده شدن `types` توسط resolverهای مدرن: اول در هر entry باشد. خطای معروف TS2307 بعد از نصب پکیج سالم، معمولاً یعنی resolution شما قدیمی است (`node10`) و exports را نمی‌خواند — نه اینکه پکیج خراب است.

مدل ذهنی — tsc باید مترجمِ وفادار runtime باشد

پیکربندی ماژول درست، آن است که tsc دقیقاً همان چیزی را قبول/رد کند که runtime نهایی قبول/رد می‌کند. هر وقت خطای resolution گرفتید، نپرسید «چطور tsc را ساکت کنم» — بپرسید «اجراکننده واقعی این import را چطور حل می‌کند؟» بعد پیکربندی را با همان تنظیم کنید. ساکت کردن tsc با گزینه‌های سخاوتمندتر، فقط شکست را به runtime منتقل می‌کند.

هشدار — import type و verbatimModuleSyntax

وقتی importی فقط برای نوع است، `import type { User } from "./models.js"` بنویسید. بدون صراحت، transpilerهای تک‌فایلی (type stripping، SWC، esbuild، Node خود) نمی‌دانند آن import را باید حذف کنند یا نگه دارند — نتیجه: ماژولی که در runtime اجرا می‌شود فقط برای هیچ، یا برعکس، حذفی که side effect لازم را می‌برد.
`verbatimModuleSyntax` این ابهام را به خطای کامپایل تبدیل می‌کند؛ روشنش کنید.

زیر کاپوت — چرا این‌طور است؟

تمام دردهای ماژول و resolution از یک جمله ریشه می‌گیرند: **tsc کد شما را اجرا نمی‌کند؛ اجراکننده واقعی را «شبیه‌سازی» می‌کند** — و شبیه‌سازی فقط وقتی ارزش دارد که با واقعیت یکی باشد. `moduleResolution` دقیقاً «انتخاب شبیه‌سازی‌شونده» است:

`node10` دنیای قدیم CommonJS را شبیه‌سازی می‌کند: پسوند لازم نیست، index.js حدس زده می‌شود، main در package.json خوانده می‌شود. `nodenext` دنیای ESM نود مدرن را: پسوند صریح اجباری، فیلد exports در package.json به‌عنوان مرز رسمی بسته (نقشه «ورودی‌های مجاز» — مسیرهای خارج از آن، حتی اگر فایل موجود باشد، ممنوع)، و تمایز import/require. `bundler` واقع‌بینانه وسط ایستاده: exports را می‌فهمد ولی سخت‌گیری پسوند را ندارد — چون bundler ها واقعاً همین‌طورند. قاعده تصمیم همیشه یکی است: «کد را واقعاً چه کسی load می‌کند؟» همان را

شبیه‌سازی کنید. ناهماهنگی یعنی کامپایلری که دنیایی خیالی را سبز می‌کند — و خطای واقعی در شب deploy ظاهر می‌شود.

و معمای معروف «چرا باید import فایل `.ts` را با پسوند `.js` بنویسم؟» پاسخش یک اصل قدیمی TypeScript است: **کامپایلر هرگز متن specifier را بازنویسی نمی‌کند.** آنچه می‌نویسید، عیناً در خروجی می‌آید. چرا؟ چون بازنویسی امن نیست: tsc به‌صورت تک‌فایلی هم اجرا می‌شود (فصل ۲)، همیشه نمی‌داند specifier به چه فایلی خواهد رسید، و حدس زدن یعنی شکستن build های دیگر ابزارها. پس شما باید «مسیرِ پس از کامپایل» را بنویسید — فایل .ts شما .js خواهد شد، پس import هم .js. ناخوشایند ولی سازگار؛ و ابزارهای مدرن (bundler ها، allowImportingTsExtensions برای اجراکننده‌های مستقیم TS) هر کدام راه‌حل محیط خودشان را دارند.

`esModuleInterop` هم داستان مشابهی دارد: دو جهان CJS و ESM مدل «خروجی پیش‌فرض» ناسازگاری دارند؛ این پرچم، شیم کوچکی در خروجی می‌گذارد تا `import x from "cjs"` همان‌طوری کار کند که Node و bundler ها رفتار می‌کنند. خاموش بودنش در پروژه‌های قدیمی، منشأ کلاسیک «TS سبز، runtime قرمز» است.

تمرین

سه خطای resolution، سه ریشه

۱. `Relative import paths need explicit file extensions in ECMAScript imports... Did you mean './util.js'?`

۲. پکیجی تازه نصب کرده‌اید؛ `TS2307: Cannot find module '@acme/kit/schema'` — اما فایل در node_modules هست.

۳. در پروژه Vite همه‌چیز سبز است؛ همان کد را در Node می‌زنید: `ERR_MODULE_NOT_FOUND` .

پاسخ: ۱. module=nodenext و import نسبی بدون پسوند — `.js` اضافه کنید (در فایل ts هم مسیر خروجی را می‌نویسید). ۲. moduleResolution قدیمی که فیلد exports را نمی‌خواند — به nodenext/bundler ارتقا دهید. ۳. پیکربندی bundler سخاوتمندتر از Node است (پسوند، alias، …) — برای کد دو-محیطی، با قواعد سخت‌گیرترِ nodenext چک کنید.

پرسش‌های مصاحبه این فصل

۱. خطای TS2307 روی پکیجی که مطمئنید سالم است — کجا را می‌گردید؟

moduleResolution خودم را. پکیج‌های مدرن فقط از مسیر `exports` حل می‌شوند و resolution های قدیمی (node10) آن فیلد را نمی‌فهمند. ارتقا به nodenext/bundler معمولاً حلش می‌کند. بعدش: شرط types در exports پکیج و ابزار arethetypeswrong برای تشخیص دقیق.

۲. چرا در ESM نودی باید `./util.js` بنویسید وقتی فایل `util.ts` است؟

چون specifier متعلق به «خروجی اجراشونده» است، نه سورس — Node در runtime دنبال .js می‌گردد و import ها را بازنویسی نمی‌کند؛ tsc هم عمداً specifier را دست نمی‌زند. module: nodenext همین واقعیت را آینه می‌کند. ناراحت‌کننده است ولی صادقانه؛ bundler ها این الزام را حذف می‌کنند چون خودشان حل می‌کنند.

فایل‌های اعلان و @types

d.ts، DefinitelyTyped و نوشتن اعلان برای کد بدون نوع

اهداف یادگیری این فصل

- ساختار یک فایل .d.ts را بخوانی و بنویسی
- منبع نوع یک پکیج (built-in یا @types) را تشخیص دهی
- برای ماژول بدون نوع اعلان حداقلی امن بنویسی

وقتی `npm install lodash` می‌زنید و VS Code بلافاصله امضای `_.groupBy` را می‌داند، زنجیره‌ای از فایل‌های `.d.ts` کار کرده‌اند. این فصل آن زنجیره را باز می‌کند: نوع‌ها از کجا می‌آیند، وقتی نیستند چه کنیم، و چطور برای کد بدون نوع، اعلان امن بنویسیم.

d.ts — امضا بدون بدنه

فایل اعلان، «هدر» ماژول است: فقط نوع‌ها و امضاها، هیچ پیاده‌سازی:

math-utils.d.ts — type-checks ✓

```ts
// اعلان متناظر با math-utils.js
export declare function mean(xs: readonly number[]): number;
export declare function clamp(x: number, lo: number, hi: number): number;
export declare const EPSILON: number;

export interface Stats { mean: number; variance: number }
export declare function describe(xs: readonly number[]): Stats;
```

کلیدواژه `declare` یعنی «این موجود هست؛ تعریفش جای دیگری است». کامپایلر به آن اعتماد می‌کند — بدون هیچ بررسی یا پیاده‌سازی واقعی. همین «اعتماد» است که اعلان‌نویسی را هم قدرتمند می‌کند هم خطرناک.

نوع هر پکیج از کجا می‌آید؟ — ترتیب جست‌وجو

۱. **خود پکیج** — فیلد `types` / `typings` یا `exports.types` در package.json آن. استاندارد مدرن: کتابخانه نوع‌هایش را خودش حمل می‌کند.

۲. **`@types/نام‌پکیج`** — از مخزن DefinitelyTyped؛ برای کتابخانه‌هایی که خودشان نوع ندارند (مثلاً `@types/express`). این بسته‌ها را جامعه می‌نویسد: کیفیت معمولاً خوب اما **نسخه‌بندی جدا از خود پکیج** است — ناهمگامی ممکن است.

۳. **هیچ‌کدام** — خطای TS7016، و تصمیم با شما.

پیام کامپایلر

```
error TS7016: Could not find a declaration file for module
'legacy-sdk'. 'node_modules/legacy-sdk/index.js' implicitly
has an 'any' type.
  Try `npm i --save-dev @types/legacy-sdk` if it exists or add
  a new declaration (.d.ts) file containing `declare module 'legacy-sdk';`
```

وقتی نوعی وجود ندارد — نردبان سه‌پله

پله ۱ — اعلان حداقلی برای رفع انسداد (آگاهانه موقتی):

types/legacy-sdk.d.ts — type-checks ✓

```ts
declare module "legacy-sdk";  //  کل ماژول: any — فقط برای عبور
```

پله ۲ — اعلان سطح API مصرفی شما (نقطه تعادل معمول):

types/legacy-sdk-v2.d.ts — type-checks ✓

```ts
declare module "legacy-sdk" {
  export interface Client { track(event: string, data?: Record<string, unknown>): void }
  export function createClient(apiKey: string): Client;
}
```

فقط همان بخشی را تایپ کنید که مصرف می‌کنید — نه کل سطح کتابخانه. اعلان کوچکِ دقیق، از اعلان بزرگِ حدسی بسیار بهتر است؛ هر امضای حدسی، یک دروغ بالقوه با امضای شماست.

پله ۳ — مشارکت در DefinitelyTyped یا PR به خود کتابخانه — وقتی استفاده‌تان جدی و بلندمدت است.

اسکریپت‌های global و فایل‌های ambient

هر `.d.ts` که import/export سطح بالا نداشته باشد، «اسکریپت سراسری» است — اعلان‌هایش بدون import همه‌جا دیده می‌شوند. برای تایپ کردن چیزهایی که واقعاً global اند (اسکریپت آنالیتیکس تزریق‌شده، متغیرهای build-time):

types/globals.d.ts — type-checks ✓

```ts
declare const __BUILD_ID__: string;          //  تزریق‌شده توسط bundler

interface Window {
  analytics?: { track(event: string): void }; // merge با Window (فصل ۲۵)
}
```

هشدار — d.ts یعنی مسئولیت شما، نه کامپایلر

فایل اعلان هیچ‌وقت با پیاده‌سازی مقایسه نمی‌شود — اگر امضا دروغ بگوید، کامپایلر با اطمینان کامل کد غلط را تأیید می‌کند. سه قاعده ایمنی: (۱) اعلان دستی را کنار منبعش مستند کنید (لینک به docs کتابخانه)، (۲) فقط API مصرفی را تایپ کنید، (۳) اگر کتابخانه JS مال خودتان است، به‌جای d.ts دستی، از JSDoc + checkJs یا تبدیل واقعی استفاده کنید (فصل ۳۰) تا اعلان از کد تولید شود، نه از حافظه.

تولید d.ts برای پکیج خودتان

tsconfig.lib.jsonc

```
{
  "compilerOptions": {
    "declaration": true,          //  تولید d.ts کنار js
    "declarationMap": true,        // go-to-definition مصرف‌کننده به سورس شما می‌رسد
    "stripInternal": true         //  از اعلان حذف شوند @internal اعضای
  }
}
```

نکته حرفه‌ای که در نسخه‌های اخیر اهمیت یافته: ابزارهای emit تک‌فایلی و «isolated declarations» از شما می‌خواهند خروجی‌های عمومی annotation صریح داشته باشند تا d.ts بدون تحلیل کامل قابل تولید باشد — دلیل دیگری برای قاعده «امضای عمومی صریح» فصل ۱۱. ادامه این مسیر (انتشار و تست پکیج) در فصل ۴۰.

مدل ذهنی — d.ts مرز اعتماد است

درون پروژه، نوع‌ها **اثبات** می‌شوند (کامپایلر از روی کد می‌سازد)؛ در مرز node_modules، نوع‌ها **ادعا** هستند (از d.ts خوانده می‌شوند). کیفیت تایپ کل برنامه شما سقفی دارد: صحت اعلان‌های مرزی. برای همین پکیج بدون نوع یا با نوع مشکوک، فقط یک ناراحتی DX نیست — حفره در دیوار اعتماد است و مثل داده بیرونی (فصل ۳۴) باید با احتیاط مصرف شود.

زیر کاپوت — چرا این‌طور است؟

رابطه TypeScript با کتابخانه‌ها بر یک زنجیره جستجوی قطعی استوار است — وقتی «نوع پیدا نشد» یا «نوع اشتباه آمد»، همیشه یکی از حلقه‌های همین زنجیره است: برای `import "pkg"`، کامپایلر اول فیلد `typings` / `types` در package.json بسته را می‌بیند (و در دنیای exports، شرط `types` داخل همان نقشه exports — که باید کنار هر entry point تکرار شود؛ جاافتادنش در subpath ها منشأ کلاسیک «نوع فقط برای ورودی اصلی» است)، بعد `index.d.ts` کنار main، بعد `node_modules/@types/pkg` . اولین یافته برنده است — بسته‌ای که خودش types دارد، @types اش نادیده گرفته می‌شود؛ نصب هم‌زمان هر دو با نسخه‌های ناهمگام، تعارض‌های گیج‌کننده می‌سازد.

جایگاه DefinitelyTyped را هم درست ببینید: مخزنی جمعی که «قرارداد نوعی» هزاران بسته JS را از بیرون نگه می‌دارد — عظیم و حیاتی، ولی ذاتاً «ترجمه دست‌نویسِ غیرمالک»: نسخه @types می‌تواند از کتابخانه عقب بماند و دقتش به همت نگهدارنده داوطلب بسته است. وقتی رفتار واقعی کتابخانه با نوعش نمی‌خواند، گزینه‌هایتان به ترتیب سلامت: augmentation محلی (فصل ۲۵)، پچ در فایل d.ts پروژه، و فقط در آخرین چاره any موضعیِ کامنت‌دار.

و چرا `isolatedDeclarations` (+TS 5.5) اصلاً ساخته شد؟ تولید d.ts در حالت عادی به checker کامل نیاز دارد — نوعِ استنتاج‌شده خروجی‌ها باید «محاسبه» شود، و این یعنی تولید اعلان‌های یک بسته در monorepo به check کامل همه وابستگی‌هایش گره می‌خورد: گلوگاه موازی‌سازی. قاعده isolatedDeclarations ساده است: «هر چیزِ export شده باید نوع صریح داشته باشد» — آن‌وقت d.ts با parse تنها، فایل‌به‌فایل و موازی تولید می‌شود؛ همان منطق isolatedModules فصل ۲، این‌بار برای اعلان‌ها. بهایش annotation بیشتر روی مرزهای عمومی است — که فصل ۴ به‌هرحال توصیه‌اش کرده بود: مرز صریح، هم برای انسان سند است هم برای ابزار آزادی.

تمرین

اعلان بنویس

کتابخانه فرضی `tiny-queue` (بدون نوع) این‌طور مصرف می‌شود:

usage.js — type-checks ✓

```
const { createQueue } = require("tiny-queue");
const q = createQueue({ concurrency: 2 });
q.push(async () => { /* job */ });
q.onIdle(() => console.log("done"));
```

اعلان مناسب بنویسید.

پاسخ:

types/tiny-queue.d.ts — type-checks ✓

```
declare module "tiny-queue" {
  export interface QueueOptions { concurrency?: number }
  export interface Queue {
    push(job: () => Promise<void>): void;
    onIdle(cb: () => void): void;
  }
  export function createQueue(options?: QueueOptions): Queue;
}
```

به تصمیم‌ها دقت کنید: `concurrency` اختیاری (در مثال مقدار دارد اما منطقاً پیش‌فرض‌پذیر است — از docs چک کنید)، job حتماً Promise برگرداند؟ اگر مطمئن نیستید `() => void | Promise<void>` صادق‌تر است. اعلان‌نویسی یعنی همین قضاوت‌های کوچک.

پرسش‌های مصاحبه این فصل

۱. وابستگی JS بدون نوع — نردبان برخوردتان؟

اول `@types/x` در DefinitelyTyped؛ نبود، declaration محلی «حداقلی» فقط برای سطحی که مصرف می‌کنم (نه کل کتابخانه)؛ و آخرین چاره `declare module "x"` با any صریح + issue برای تکمیل. پله آخر را بدون تاریخ انقضا نگذارید — any پشت declaration از any معمولی نامرئی‌تر است.

۲. `declarationMap` چرا برای monorepo حیاتی است؟

بدون آن، go-to-definition مصرف‌کننده‌ها به فایل d.ts تولیدی می‌رسد — مستندات مرده. با آن، به سورس واقعی TypeScript می‌روید و refactor بین‌پکیجی و code review قابل زندگی می‌شود. هزینه‌اش چند فایل map کوچک است؛ خاموش بودنش در monorepo تقریباً همیشه سهو است.

مهاجرت از JavaScript — تدریجی و قابل برگشت

checkJs، JSDoc و استراتژی مهاجرت ماژول‌به‌ماژول

اهداف یادگیری این فصل

- با JSDoc و checkJs بدون تغییر پسوند فایل، بررسی نوع بگیری
- نقشه مهاجرت تدریجی با مرزهای صریح طراحی کنی
- از دام «مهاجرت همه‌جا any» دوری کنی

«از دوشنبه همه‌چیز TypeScript» — این جمله بیشتر مهاجرت‌ها را کشته است. مهاجرت موفق، پروژه بازنویسی نیست؛ **فرایند تدریجیِ قابل برگشت** است که از روز اول ارزش بدهد و هیچ روزی تیم را از deploy بیندازد. TypeScript برای همین سناریو ابزارهای درجه‌یک دارد که کمتر کسی کامل می‌شناسد.

فاز صفر — بررسی نوع بدون تغییر حتی یک پسوند

JavaScript شما همین امروز می‌تواند type-check شود — با JSDoc:

pricing.js

type-checks ✓

```
// @ts-check   ← فعال‌سازی per-file؛ یا checkJs در tsconfig

/**
 * @param {number} price
 * @param {{ rate: number, includeShipping?: boolean }} options
 * @returns {number}
 */
export function finalPrice(price, options) {
  const shipping = options.includeShipping ? 50 : 0;
  return price * (1 + options.rate) + shipping;
}

finalPrice(100, { rate: 0.09 });          // ✓
// finalPrice("100", { rate: 0.09 });     // × قرمز می‌شود در ادیتور و CI
```

تقریباً تمام قدرت سیستم نوع — جنریک‌ها، نوع‌های import شده از d.ts، حتی cast — در JSDoc در دسترس است:

jsdoc-advanced.js

type-checks ✓

```
/** @template T
 *  @param {T[]} xs
 *  @returns {T | undefined} */
export const last = xs => xs[xs.length - 1];

/** @typedef {import("./types").Order} Order */
/** @type {Map<string, Order>} */
const cache = new Map();
```

پیکربندی فاز صفر:

tsconfig.migration.jsonc

```
{
  "compilerOptions": {
    "allowJs": true,          // js ها عضو پروژه‌اند
    "checkJs": true,          //  و بررسی می‌شوند
    "noEmit": true,           //  فقط چک — build فعلی دست‌نخورده
    "strict": false,          //  فعلاً؛ پله‌پله بالا می‌رود
    "noImplicitAny": false
  },
  "include": ["src"]
}
```

همین پیکربندی + اسکریپت `typecheck` در CI = مهاجرت رسماً شروع شده، با صفر تغییر در کد و build.

استراتژی ماژول‌به‌ماژول — نقشه و ترتیب

قاعده انتخاب ترتیب: **از برگ‌ها به ریشه** در گراف وابستگی، با اولویت دو دسته:

۱. **ماژول‌های زیربنایی پرمصرف** (types/domain، utils) — نوعشان به همه‌جا «نشت مثبت» می‌کند.

۲. **فایل‌های پرحادثه** — جایی که بیشترین باگ‌ها گزارش شده؛ بیشترین بازده فوری.

برای هر ماژول: تغییر پسوند به `.ts`، رفع خطاها (واقعی، نه با any!)، مرج. کوچک، مکرر، قابل review. دو ابزار کمکی:

- `// @ts-expect-error` با کامنت توضیحی برای خطاهایی که الان نمی‌ارزند — برتری‌اش بر `@ts-ignore`: وقتی خطا رفع شود، خودِ suppression خطا می‌شود و پاک‌سازی را یادآوری می‌کند.
- ابزارهای codemod (مثل ts-migrate برای شروع انبوه) — خروجی‌شان پر از any صریح است؛ به‌عنوان «فاز annotate خودکار» خوب‌اند، به شرط برنامه جدی برای پاک‌سازی.

دام بزرگ: مهاجرتِ any-محور

bad-migration.ts

type-checks ✓

```ts
// ❌ «مهاجرت» که فقط پسوند عوض کرده:
export function processOrder(order: any, user: any, options: any): any {
  return { ...order, by: user.id, express: options?.express ?? false };
}
```

این بدتر از JavaScript است: هزینه TypeScript را می‌دهید (build، ابزار، آموزش) و سودش را نمی‌گیرید، به‌علاوه **اعتماد کاذب** — فایل «تایپ‌شده» به نظر می‌رسد. سنجه سلامت مهاجرت، درصد فایل‌های ts نیست؛ روند تعداد any صریح + implicit است (ابزارهایی مثل type-coverage یا یک grep ساده در CI این عدد را ترند می‌کنند).

در عمل — نردبان سخت‌گیری در طول مهاجرت

گزینه‌ها را به این ترتیب بالا ببرید؛ هر پله وقتی «سبز» شد، قفلش کنید تا برنگردد:

- پله ۱: checkJs + noEmit (فاز صفر)
- پله ۲: noImplicitAny — دردناک‌ترین و پرسودترین پله
- پله ۳: strictNullChecks — نیمه دوم ارزش strict
- پله ۴: strict کامل + useUnknownInCatchVariables
- پله ۵ (دلخواه): noUncheckedIndexedAccess برای کد جدید

برای codebaseهای بزرگ، این نردبان را per-directory اجرا کنید (چند tsconfig با extends) — پوشه‌های مهاجرت‌کرده strict، بقیه در راه.

مدل ذهنی — مرز متحرک، نه جبهه کامل

در هر لحظه از مهاجرت، پروژه دو ناحیه دارد: **ناحیه تایپ‌شده قابل اعتماد** و **ناحیه legacy**. تمام تلاش مدیریتی باید صرف دو چیز شود: (۱) مرز بین دو ناحیه صریح و باریک بماند — ماژول‌های interface با اعلان دقیق، (۲) ناحیه اول فقط بزرگ شود. مهاجرتی که این مرز را مدیریت نکند، به «جزیره‌های TS در اقیانوس any» می‌رسد که سال‌ها همان‌جا می‌مانند.

زیر کاپوت — چرا این‌طور است؟

migration تدریجی JS→TS فقط «یک استراتژی از میان چند گزینه» نیست — تنها استراتژی سازگار با ساختار وابستگی کد است، و دلیلش را باید دقیق دید:

چرا برگ-اول (leaf-first)؟ نوع‌ها در جهت import جریان دارند: وقتی ماژول A از B import می‌کند، کیفیت نوع‌های A سقفش کیفیت نوع‌های B است — any از B به A نشت می‌کند (ویروس فصل ۸). اگر از ریشه شروع کنید، ماژول‌های تبدیل‌شده روی دریایی از وابستگی تایپ‌نشده شناورند و «TS بودن»شان نمایشی است: پر از any ضمنی از پایین. اگر از برگ‌ها شروع کنید — ماژول‌های بدون وابستگی داخلی: utils، ثابت‌ها، مدل‌های خالص — هر فایل تبدیل‌شده روی پایه‌ای کاملاً تایپ‌شده می‌ایستد و نوع‌هایش «واقعی»اند. هر گام، سقف کیفیت گام بعد را بالا می‌برد؛ جهت حرکت، همان جهت گراف وابستگی است.

چرا `allowJs + checkJs` پل میانی خوبی است؟ چون checker تمام‌عیار TypeScript را روی فایل‌های js. اجرا می‌کند — با استنتاج کامل و خواندن JSDoc به‌عنوان annotation — بدون هیچ تغییر syntax و هیچ تغییر در build. تیم می‌تواند ماه‌ها در این حالت زندگی کند: باگ‌های نوعی پیدا می‌شوند، JSDoc ها قرارداد می‌سازند، و تبدیل نهایی به ts. عمدتاً «ترجمه JSDoc به syntax» است، نه کشف دوباره.

چرا سخت‌گیری را مرحله‌ای بالا ببرید؟ چون هر پرچم strict کلاس خطای خودش را یکجا رو می‌کند (فصل ۲۷) و codebase در حال گذار، هم‌زمان ظرفیت رسیدگی به همه را ندارد. الگوی عملی: strict کامل برای فایل‌های جدید (با tsconfig لایه‌ای یا قاعده lint)، و بالا بردن تدریجی برای قدیمی‌ها — به‌علاوه خط قرمز «شمار any صریح هرگز بالا نرود» که ابزار ساده‌ای مثل یک grep شمارنده در CI نگهش می‌دارد. migration موفق، پروژه مهندسیِ قابل اندازه‌گیری است، نه جهاد یک‌شبه؛ معیار پیشرفتش هم دقیقاً همین عددهاست: درصد فایل‌های ts، شمار any، سطح پرچم‌ها.

تمرین

برنامه مهاجرت بنویس

پروژه‌ای با این مشخصات: ۴۰۰ فایل JS، تیم ۵ نفره، deploy روزانه، پوشه‌ها: `utils/` (کم‌تغییر، پرمصرف)، `models/`، `api/` (پرحادثه)، `legacy-admin/` (قرار است سال بعد بازنویسی شود). سه تصمیم اول شما؟

پاسخ نمونه: ۱. فاز صفر همین هفته: allowJs+checkJs+noEmit و typecheck در CI؛ ارزش فوری بدون ریسک. ۲. ترتیب: اول `utils/` و `models/` (برگ‌های پرمصرف — نوعشان همه‌جا اثر می‌کند)، بعد `api/` (پرحادثه، حالا با پایه تایپ‌شده)؛ هر PR حداکثر چند فایل. ۳. `legacy-admin/` را صریحاً exclude کنید و فقط یک اعلان مرزی برایش بنویسید — تایپ کردن کدی که عمرش معلوم است، سرمایه‌سوزی است. و یک سنجه: ترند any در داشبورد CI.

پرسش‌های مصاحبه این فصل

۱. موفقیت مهاجرت JS→TS را با چه عددی گزارش می‌کنید؟

ترند نزولی any و سرکوب‌ها (type-coverage یا شمارش ساده در CI)، نه «درصد فایل‌های ts». می‌شود صدِ درصد فایل‌ها را rename کرد و با any همان JS قبلی را داشت — بدتر از قبل، چون توهم پوشش می‌دهد. عدد دوم: تعداد خطای واقعی که فاز صفر (checkJs) از production بیرون کشید.

۲. چرا branch جداگانه «مهاجرت بزرگ» تقریباً همیشه شکست می‌خورد؟

چون codebase زیر پایتان حرکت می‌کند و merge debt نمایی رشد می‌کند؛ بعد از دو ماه، branch مهاجرت قابل ادغام نیست. راه درست: روی main، ماژول‌به‌ماژول از برگ‌های پرمصرف، با قفل تدریجی گزینه‌ها — هر commit کوچک، سبز و برگشت‌پذیر.

Project References، Monorepo و CI

بررسی نوع تکرارپذیر در مقیاس تیم

اهداف یادگیری این فصل

- project reference و composite را برای مرزهای داخلی به‌کار ببری
- pipeline بررسی نوع در CI را طراحی کنی
- بدانی incremental و cache کجا جواب می‌دهد و کجا نه

تا ده‌ها فایل، «همه‌چیز در یک tsc» جواب می‌دهد. اما در مقیاس تیم — چند پکیج، مرزهای مالکیت، CI که باید سریع بماند — به معماری بررسی نوع نیاز دارید. ابزارهای این فصل: project references، حالت incremental و طراحی pipeline ای که «سبز» آن قابل اعتماد باشد.

Project References — مرزهای رسمی داخل مخزن

به‌جای یک پروژه غول‌پیکر، چند پروژه کوچک با رابطه صریح:

monorepo-layout

```
acme/
├── packages/
│   ├── core/          ← util (وابسته به هیچ) و دامنه
│   │   └── tsconfig.json
│   ├── api-client/    ← وابسته به core
│   │   └── tsconfig.json
│   └── web/           ← وابسته به هر دو
│       └── tsconfig.json
└── tsconfig.json      ← solution: فقط ارجاع‌ها
```

packages/api-client/tsconfig.jsonc

```
{
  "compilerOptions": {
    "composite": true,          // الزام references: declaration + ...
    "rootDir": "src",
    "outDir": "dist"
  },
  "references": [{ "path": "../core" }],
  "include": ["src"]
}
```

tsconfig.solution.jsonc

```jsonc
{
  "files": [],
  "references": [
    { "path": "packages/core" },
    { "path": "packages/api-client" },
    { "path": "packages/web" }
  ]
}
```

build

```
tsc --build --verbose        # ترتیب topological، فقط پروژه‌های تغییرکرده
tsc --build --watch          # حالت توسعه
```

دو سود مستقل:

- **کارایی:** `tsc --build` گراف را می‌فهمد؛ تغییر در `web` باعث بازبررسی `core` نمی‌شود، و پروژه‌های upstream از `.d.ts` تولیدشده مصرف می‌شوند، نه سورس کامل — بررسی ارزان‌تر.
- **معماری:** مرز reference، مرز مجاز import است. `core` نمی‌تواند از `web` import کند حتی اگر developer بخواهد — لایه‌بندی، compile-enforced می‌شود.

incremental و cache — سریع ولی درست

ci-options.jsonc

```jsonc
{
  "compilerOptions": {
    "incremental": true,              //  ذخیره state در .tsbuildinfo
    "tsBuildInfoFile": "node_modules/.cache/tsc/app.tsbuildinfo"
  }
}
```

`incremental` بین اجراها فقط فایل‌های اثرپذیرفته را دوباره بررسی می‌کند. در CI فقط وقتی سود دارد که `.tsbuildinfo` بین اجراها cache شود (restore با کلید tsconfig + lockfile). دو قاعده احتیاط:

- cache خراب = خطاهای عجیب؛ همیشه مسیر «پاک‌سازی cache» در pipeline داشته باشید.
- ارتقای نسخه TypeScript، cache را باطل می‌کند — کلید cache باید نسخه را شامل شود.

pipeline مرجع برای CI

ci.yml

```yaml
steps:
  - run: npm ci                          # نصب قطعی از lockfile
  - run: npx tsc --build --pretty false  # بررسی نوع کل گراف
  - run: npx eslint . && npx vitest run  # سبک‌سنگین بعد از typecheck
```

نکته‌های تجربی:

- **همان نسخه، همه‌جا:** TypeScript فقط از lockfile — نه global، نه «آخرین». اختلاف نسخه ادیتور/CI، منبع کلاسیک «روی سیستم من سبز بود».
- **typecheck جدا از build:** حتی وقتی bundler خروجی می‌سازد، `tsc --noEmit` (یا `--build` با emit اعلان) مرحله مستقل CI باشد — تا سبزِ pipeline یعنی «نوع‌ها هم سالم‌اند».
- **monorepo با ابزار task-runner** (Nx/Turborepo): بگذارید آن‌ها cache بین‌ماشینی را مدیریت کنند؛ tsc --build داخل هر پکیج صدا می‌خورد و ابزار، گراف و cache سطح بالا را.

مدل ذهنی — واحد بررسی = واحد مالکیت

پروژه (به معنای tsconfig دار) را جایی ببُرید که مالکیت و نرخ تغییر عوض می‌شود: دامنه پایدار پرمصرف، جدا از اپ پرتغییر. مرز درست، هم کارایی می‌دهد (بازبررسی کمتر) هم کیفیت (وابستگی یک‌طرفه). مرز اشتباه — مثلاً بر اساس نوع فایل — فقط بوروکراسی اضافه می‌کند.

هشدار — skipLibCheck خطا را جابه‌جا می‌کند، نه حذف

در مقیاس monorepo وسوسه می‌شوید همه اصطکاک‌های بین‌پکیجی را با skipLibCheck ساکت کنید. به یاد داشته باشید چه خریده‌اید: ناسازگاری d.ts های تولیدی خودتان هم دیگر بررسی نمی‌شود. حداقل در پکیج‌های داخلی، تست‌های نوع (فصل ۳۲) را جایگزین این بررسی ازدست‌رفته کنید.

زیر کاپوت — چرا این‌طور است؟

چرا project references واقعاً سریع‌تر است؟ رازش در یک جایگزینی است: **مصرف d.ts به‌جای مصرف سورس.** در پروژه یکپارچه، هر check کامل یعنی بازخوانی و بازبررسی همه فایل‌ها — چون checker برای فهم نوع‌های ماژول A باید سورس همه وابستگی‌هایش را بفهمد. با references، هر پروژه composite خروجی d.ts تولید می‌کند و پروژه‌های بالادستی فقط همان اعلان‌ها را می‌خوانند: d.ts «رابط از پیش محاسبه‌شده» است — کوچک‌تر، بدون بدنه، بدون نیاز به استنتاج دوباره. بررسی پروژه بالادستی دیگر به «درون» وابستگی‌ها کاری ندارد؛ فقط به قراردادشان.

از این سازوکار، الزامات composite هم بدیهی می‌شوند: `declaration` اجباری است (محصول اصلی، همین d.ts هاست)، فهرست فایل‌ها باید صریح باشد (تا build incremental بداند مرز پروژه کجاست)، و پروژه‌ها نمی‌توانند به سورس یکدیگر دست‌درازی کنند — فقط از دروازه رسمی d.ts. و `tsc --build` در واقع یک build-orchestrator کوچک است: گراف references را مرتب توپولوژیک می‌کند، timestamp و امضای فایل‌ها را در `.tsbuildinfo` نگه می‌دارد و پروژه‌ای را

که نه خودش و نه قراردادِ وابستگی‌هایش عوض نشده، به‌کلی رد می‌کند. نکته ظریف: اگر تغییر وابستگی «فقط درون بدنه» باشد و d.ts اش ثابت بماند، بالادستی‌ها اصلاً rebuild نمی‌شوند — دقیقاً همان دلیلی که مرزِ قراردادیِ پایدار، در مقیاس، طلاست.

این معماری همان درس فصل ۲۹ در مقیاس monorepo است: مرز صریح = آزادی ابزار. و هزینه‌اش هم همان است: باید مرزها را طراحی کنید — پروژه‌بندی بد (حلقه وابستگی، مرزهای ریز بی‌معنا) سربار هماهنگی‌اش از سود incremental بیشتر می‌شود. references را وقتی بیاورید که دردش را دارید: زمان check چنده‌ثانیه‌ای، تیم‌های موازی روی بخش‌های جدا، یا نیاز به انتشار بسته‌های داخلی.

تمرین

مرزها را طراحی کن

یک مخزن با این اجزا دارید: مدل‌های دامنه (پایدار)، سرور API (روزانه تغییر)، اپ وب (روزانه)، ۲ تابع serverless کوچک، و یک پوشه scripts یک‌بارمصرف. چند پروژه می‌سازید و با چه ارجاع‌هایی؟

پاسخ نمونه: چهار پروژه composite: `domain` (بدون وابستگی)؛ `server` → domain؛ `web` → domain (نه server!). قرارداد از طریق نوع‌های API در domain یا پکیج contract جدا)؛ `functions` → domain. پوشه `scripts` پروژه نشود — یک tsconfig ساده با strict ملایم کافی است؛ بوروکراسی composite برای کد یک‌بارمصرف نمی‌ارزد. نکته کلیدی: جهت همه ارجاع‌ها به سمت domain است و هیچ ارجاعی بین مصرف‌کننده‌ها نیست.

پرسش‌های مصاحبه این فصل

۱. project references فراتر از سرعت چه می‌دهد؟

مرزبندی اجباری معماری: وابستگی‌ای که در references اعلام نشده، import ش خطای build است — «جهت مجاز وابستگی» از کنوانسیون به قاعده کامپایلری تبدیل می‌شود. به‌علاوه مصرف upstream از d.ts (ارزان‌تر) و بازبررسی فقط پروژه‌های اثرپذیرفته.

۲. cache ساخت افزایشی در CI را با چه کلیدی می‌زنید؟

نسخه دقیق TypeScript + هش tsconfig ها + هش lockfile؛ چون .tsbuildinfo به فرمت نسخه کامپایلر حساس است و با ارتقای TS باید باطل شود. کش کهنه با نسخه جدید، یا بی‌اثر است یا — بدتر — رفتار ناهمگون بین runner ها می‌سازد.

تست نوع، دیباگ خطاهای پیچیده و کارایی کامپایلر

جلوگیری از رگرسیون نوع و پروفایل‌گیری عملی

اهداف یادگیری این فصل

- برای نوع‌های عمومی تست رگرسیون بنویسی
- خطاهای چندصفحه‌ای را با تکنیک برش‌دادن دیباگ کنی
- گلوگاه کارایی type-check را پیدا و رفع کنی

تست می‌نویسیم تا رفتار runtime رگرس نکند — اما نوع‌ها هم رفتارند: utilityهای فصل ۲۰، امضاهای عمومی فصل ۲۶، قراردادهای پکیج فصل ۴۰. این فصل سه مهارت حرفه‌ای را پوشش می‌دهد: تست نوع، دیباگ سیستماتیک خطاهای غول‌پیکر، و پروفایل‌گیری وقتی کامپایل کند شده.

تست نوع — رگرسیون‌گیری از قراردادها

ساده‌ترین ابزار، در خود زبان است: `@ts-expect-error` و توابع کمکی سطح نوع:

type-tests.ts — type-checks ✓

```ts
// ابزارک استاندارد تست برابری نوع‌ها:
type Expect<T extends true> = T;
type Equal<A, B> =
  (<T>() => T extends A ? 1 : 2) extends (<T>() => T extends B ? 1 : 2)
    ? true : false;

// نوع تحت تست (از فصل ۲۰):
type PickByType<T, V> = { [K in keyof T as T[K] extends V ? K : never]: T[K] };

interface Form { name: string; age: number; bio: string }

// تست‌های مثبت — اگر رفتار عوض شود، این خط‌ها قرمز می‌شوند:
type _T1 = Expect<Equal<PickByType<Form, string>, { name: string; bio: string }>>;
type _T2 = Expect<Equal<PickByType<Form, boolean>, {}>>;

// تست منفی — این باید خطا بدهد؛ اگر روزی خطا ندهد، suppression خودش خطا می‌شود:
// @ts-expect-error — number به string قابل انتساب نیست
const bad: PickByType<Form, string> = { name: "x", bio: "y", age: 1 };
void bad;
export {};
```

ترفند `Equal` (مقایسه از طریق دو امضای جنریک) استانداردِ دوفاکتوی اکوسیستم است — تفاوت `any` / `unknown` / union را هم دقیق تشخیص می‌دهد، جایی که `A extends B` ساده شکست می‌خورد. برای پروژه‌های جدی‌تر،

کتابخانه‌هایی مثل `expect-type` (همان ایده با API روان‌تر: `expectTypeOf(x).toEqualTypeOf<Y>()`) و `tsd` این الگو را بسته‌بندی کرده‌اند؛ تست‌های نوع کنار تست‌های vitest زندگی می‌کنند و در همان `tsc --noEmit` اجرا (=بررسی) می‌شوند — سریع‌ترین suite دنیا.

کجا تست نوع واجب است؟ (۱) هر utility type صادراتی، (۲) امضاهای عمومی پکیج، (۳) هر جایی که باگ نوعی یک‌بار رخ داده — همان قاعده تست رفتاری.

دیباگ خطاهای چندصفحه‌ای — تکنیک برش

الگوریتم فصل ۱۲ (عمیق‌ترین سطر، دو نوع دعوا) را با سه تکنیک تکمیل کنید:

۱. متغیر میانی annotate شده — خطا را به مرز دقیق می‌کشاند:

bisect.ts

type-checks ✓

```
// به‌جای یک عبارت بزرگ که «جایی در آن» خطاست:
// const rows = data.map(normalize).filter(isValid).reduce(group, init);

// برش بزنید — خطا روی خطِ دقیقِ مقصر می‌نشیند:
declare const data: unknown[];
declare function normalize(x: unknown): { id: string };
declare function isValid(x: { id: string }): boolean;
const step1 = data.map(normalize);      // ✓
const step2 = step1.filter(isValid);    // ✓
// حالا اگر reduce خطا داد، فقط قرارداد reduce را بررسی می‌کنید
void step2;
```

۲. hover روی جنریک‌ها در ادیتور — قبل از خواندن خطا، ببینید کامپایلر برای `T` چه استنتاج کرده؛ معمولاً استنتاج غیرمنتظره (مثلاً `never` یا union پهن) خودِ مشکل است.

۳. حداقل بازتولید در Playground — نوع‌های پروژه را خالی‌سازی کنید تا مثال ۱۰خطی بسازید. نیمی از اوقات حین ساختن بازتولید، مشکل خودش را نشان می‌دهد؛ نیم دیگر، یک سؤال خوب برای issue/همکار دارید.

وقتی کامپایل کند شد — اندازه بگیر، حدس نزن

```profile
# آمار کلی — اولین نگاه:
npx tsc --noEmit --extendedDiagnostics
#   Types: 1,243,001        ← تعداد نوع‌های ساخته‌شده؛ عدد خیلی بزرگ = جایی انفجار است
#   Instantiations: 9,800,122
#   Check time: 41.2s

# ردیابی دقیق — کدام فایل، کدام نوع:
npx tsc --noEmit --generateTrace trace-dir
# بخوانید analyze-trace یا ابزار Perfetto خروجی را در:
npx @typescript/analyze-trace trace-dir
```

الگوهای مقصر تکرارشونده (از تجربه جمعی و مستند wiki کارایی TypeScript):

- union های غول‌پیکر (انفجار template literal — فصل ۱۹) و conditional های بازگشتی عمیق.
- نوع‌های استنتاجی بزرگ که در امضای عمومی صریح نشده‌اند — کامپایلر مجبور است بارها بازشان کند؛ annotation خروجی، آن‌ها را cache پذیر می‌کند.
- `skipLibCheck: false` روی node_modules سنگین — آگاهانه تصمیم بگیرید (فصل ۳ و ۳۱).
- یک کتابخانه با نوع‌های سنگین (نسخه‌های قدیمی برخی ORM/فرم‌ها) — با trace پیدا می‌شود، با ارتقا یا ایزوله‌سازی حل.

و راه‌حل ساختاری برای مقیاس: project references (فصل ۳۱) و — از ۲۰۲۶ — کامپایلر native نسخه ۷ که همین بررسی را حدود ده برابر سریع‌تر انجام می‌دهد (فصل ۳۳). اما توجه: کامپایلر سریع‌تر، نوعِ بد را خوب نمی‌کند — فقط صبر شما را بیشتر می‌کند.

> **مدل ذهنی — نوع‌ها هم کدند**
>
> با نوع‌ها همان رفتاری را بکنید که با کد: تست رگرسیون برای قراردادها، برش برای دیباگ، پروفایل قبل از بهینه‌سازی، و refactor وقتی پیچیدگی از کنترل خارج شد. تیمی که نوع‌ها را «حاشیه» می‌داند، در مقیاس، هم کندی می‌گیرد هم شکنندگی.

زیر کاپوت — چرا این‌طور است؟

تست نوع چرا به ترفندهای عجیبی مثل `Equal` نیاز دارد؟ چون «برابری دو نوع» در TypeScript مفهومی چندلایه است و ابزار ساده، لایه اشتباه را می‌سنجد.

سنجش بدیهی — «انتساب دوطرفه»: A به B قابل انتساب و B به A. برای اکثر موارد کافی است، اما سوراخ بزرگی دارد: `any` به همه‌چیز و همه‌چیز به any قابل انتساب است — پس any با «هر نوعی» برابرِ دوطرفه درمی‌آید، و تستی که باید رگرسیونِ «نوعم any شد!» را بگیرد، سبز می‌ماند؛ دقیقاً خطرناک‌ترین رگرسیون نوعی، نامرئی. ترفند `Equal` معروف — دو تابع جنریک شرطی که مقایسه را درون جایگاه «معوق» انجام می‌دهند — کامپایلر را وادار می‌کند به‌جای قاعده سست انتساب، از بررسی «این‌همانیِ» دو نوع استفاده کند: سخت‌گیرانه، any را از string جدا می‌کند و تفاوت‌های ظریف (optional بودن عضو،

readonly) را هم می‌بیند. لازم نیست جزئیات پیاده‌سازی‌اش را حفظ کنید — باید بدانید «چرا» به آن نیاز افتاد: هر جا تست برابری می‌نویسید، اول بپرسید «اگر این نوع any شود، تستم قرمز می‌شود؟»

چرا اصلاً تست نوع، و نه اتکا به اینکه «کد استفاده‌کننده خودش می‌شکند»؟ چون نوع‌های عمومی یک کتابخانه، API هستند با مصرف‌کننده‌های بیرونی که در CI شما حضور ندارند: شما refactor می‌کنید، استنتاجِ خروجی عوض می‌شود — مثلاً از `User[]` به `any[]` — کد خودتان سبز می‌ماند و مصرف‌کننده‌ها در نسخه بعد می‌شکنند. تست نوع، همان قرارداد را «در CI شما» میخ می‌کند. مکملش تست خطا است: `@ts-expect-error` اثبات می‌کند چیزی که باید رد شود، هنوز رد می‌شود — و اگر روزی خطا ندهد، خودِ این directive خطا می‌شود؛ تست دوطرفه کامل. جایگاهش در CI هم همان tsc است، نه test runner: تست نوع، «کدی است که فقط باید check شود».

تمرین

تست بنویس، بعد بشکن

برای `DeepReadonly` فصل ۲۰ سه تست نوع بنویسید: عمق دو سطح، آرایه، و تابع (که باید دست‌نخورده بماند).

پاسخ:

quiz32-solution.ts — type-checks ✓

```ts
type Expect<T extends true> = T;
type Equal<A, B> = (<T>() => T extends A ? 1 : 2) extends
  (<T>() => T extends B ? 1 : 2) ? true : false;
type DeepReadonly<T> = T extends (...a: never[]) => unknown ? T
  : T extends object ? { readonly [K in keyof T]: DeepReadonly<T[K]> } : T;

type _T1 = Expect<Equal<DeepReadonly<{ a: { b: number } }>,
                        { readonly a: { readonly b: number } }>>;
type _T2 = Expect<Equal<DeepReadonly<{ xs: string[] }>,
                        { readonly xs: readonly string[] }>>;
type _T3 = Expect<Equal<DeepReadonly<{ f: () => void }>,
                        { readonly f: () => void }>>;
export {};
```

حالا «بشکنید»: شرط تابع را از پیاده‌سازی حذف کنید — `_T3` فوراً قرمز می‌شود. این حس «تور ایمنی زیر نوع‌ها» دقیقاً هدف این فصل بود.

پرسش‌های مصاحبه این فصل

۱. ادعای «این utility type درست کار می‌کند» را چطور تبدیل به تست می‌کنید؟

ترفند Equal دوجنریکه + `Expect<Equal<Actual, Expected>>` برای حالت‌های مثبت، و `@ts-expect-error` برای حالت‌های منفی («این نباید کامپایل شود» — نیمه فراموش‌شده قرارداد). فایل‌های test-d در همان `tsc --noEmit` اجرا می‌شوند؛ ابزار آماده‌تر: expect-type.

۲. کامپایل ۹۰ ثانیه شده — سه قدم اول؟

۱) اندازه‌گیری، نه حدس: `--extendedDiagnostics` و نگاه به ۲ .Types/Instantiations) `--generateTrace` + analyze-trace برای یافتن فایل/نوع دقیق مقصر. ۳) الگوهای مقصر همیشگی: union/conditional منفجرشده، خروجی صادراتی بدون annotation، کتابخانه نوع‌سنگین — و راه‌حل ساختاری: references.

گذار TypeScript 6 به 7 — کامپایلر Native

وضعیت ۲۰۲۶: پورت Go، deprecationها و برنامه مهاجرت ابزارها

اهداف یادگیری این فصل

- نقشه نسخه‌های 5.9 → 6.0 → 7.0 و منطق آن را توضیح دهی
- گزینه‌های حذف‌شده در 7 و مسیر ignoreDeprecations را بدانی
- تصمیم مهاجرت تیم خودت را با معیار واقعی بگیری

سال ۲۰۲۶ مهم‌ترین سال تاریخ TypeScript بعد از ۲۰۱۲ است: کامپایلر از «TypeScript نوشته‌شده با TypeScript» به **پورت native با زبان Go** مهاجرت کرد. این فصل گزارش وضعیت است — بر اساس اعلان‌های رسمی تیم TypeScript تا اکتبر ۲۰۲۶ — و چارچوب تصمیم برای تیم شما. همه ادعاهای این فصل منبع‌دار در پیوست H آمده‌اند.

نقشه نسخه‌ها — چه شد و چرا

نسخه	تاریخ	نقش
5.9	پاییز ۲۰۲۵	آخرین نسخه خط ۵ (آخرین patch: 5.9.3)
6.0	مارس ۲۰۲۶	**پل** — آخرین نسخه مبتنی بر JS؛ deprecate کردن گزینه‌های قدیمی؛ هم‌راستاسازی رفتار با ۷
7.0	ژوئن ۲۰۲۶	**پورت native (Go)** — پروژه Corsa؛ حدود ۱۰ برابر سریع‌تر؛ هم‌ارزی رفتار type-checking با 6.0

منطق شماره‌گذاری: 6.0 و 7.0 جفت طراحی شده‌اند. 6.0 عمداً «آخرین پیش از پرش» است: هر گزینه‌ای که در 7 وجود ندارد، در 6 با هشدار deprecated علامت خورده و با `"ignoreDeprecations": "6.0"` موقتاً قابل سکوت است. ارتقای تمیز به 6.0 = آمادگی برای 7.0.

چه چیزهایی در ۷ حذف/عوض شد؟

- **گزینه‌های قدیمی**: اهداف ES5 و پایین‌تر، resolutionهای منسوخ (`node10`)، و پرچم‌های سازگاری عصر قدیم حذف شدند. اگر tsconfig شما فصل ۳ و ۲۸ را رعایت کند، چیزی برای تغییر ندارید.
- **API کامپایلر**: API برنامه‌نویسی نسخه ۷ (Corsa) با API قدیمی (Strada) سازگار نیست — این مهم‌ترین شکاف اکوسیستم است: ابزارهایی که مستقیم به API کامپایلر وصل‌اند (برخی پلاگین‌های lint، buildهای type-aware، کدجن‌ها) باید پورت شوند؛ دوره‌ای از ناهم‌زمانی ابزارها طبیعی است.
- **رفتار type-checking**: هدف رسمی، هم‌ارزی با 6.0 است؛ تفاوت‌های جزئی مستندشده (مثل جزئیات ترتیب در declaration emit — پرچم کمکی `stableTypeOrdering` در 6.0 برای هم‌راستاسازی خروجی) وجود دارد، اما «کد شما همان معنا را دارد» قاعده است.

- **سرعت**: ادعای رسمی ۱۰ برابر روی codebaseهای بزرگ، با موازی‌سازی واقعی و حافظه مشترک. برای تیم‌های بزرگ این یعنی ادیتور فوری و CI چند برابر ارزان‌تر — بزرگ‌ترین بهبود DX یک دهه اخیر.

چارچوب تصمیم برای تیم شما

```decision-tree
روی 5 هستید؟ شروع x.* →
  ├─ هشدارهای deprecation را «واقعاً» رفع کنید → بله → اول به 6.0 بروید؛
  │        بعد 7 را بسنجید → (دائمی ignoreDeprecations نه)
  └─ وابستگی حیاتی به API کامپایلر دارید؟ روی 6.0 هستید →
       ├─ بله (قدیمی type-aware پلاگین/کدجن/لینت)
       │     → 6 منتظر پورت آن ابزار بمانید یا جایگزین کنید؛ x.* امن است
       └─ خیر → typescript@latest، اجرای کامل CI، نصب 7.0:
                  در پکیج‌ها، سپس سوییچ ادیتور تیم d.ts مقایسه خروجی
```

نکته‌های عملی ارتقا که در گزارش‌های میدانی تکرار شده‌اند:

- **tsc همان tsc است**: فرمان‌ها و tsconfig یکسان؛ برای بیشتر پروژه‌های اپلیکیشنی، ارتقا «نصب و اجرا» بوده است.
- **پکیج‌نویس‌ها بیشتر مراقب باشند**: declaration emit را قبل/بعد diff کنید (فصل ۴۰) — مصرف‌کننده‌های شما به ثبات d.ts حساس‌اند.
- **ادیتور**: سرویس زبان native از مسیر افزونه رسمی فعال می‌شود؛ دوره‌ای که نیمی از تیم روی سرویس قدیم و نیمی روی جدید است را کوتاه کنید — اختلاف‌های جزئی quick-fix ها گیج‌کننده می‌شود.
- **عدد نسخه در وابستگی‌ها**: کتابخانه‌هایی که `"typescript": "^5"` را peer گذاشته‌اند هنوز دیده می‌شوند؛ معمولاً فقط قید نسخه عقب است نه ناسازگاری واقعی — issue بزنید، override آگاهانه بگذارید، و در CI با نسخه قفل‌شده تست کنید.

هشدار — «زبان» ارتقا یافت، نه «هر ابزار اطرافش»

جمله «TS 7 با TS 6 سازگار است» درباره **رفتار زبان** است. زنجیره ابزار شما — bundler، test runner، لینت type-aware، framework CLI — هرکدام جداگانه باید با ۷ سازگار شده باشند. قبل از مهاجرت تیمی، یک شاخه آزمایشی با کل pipeline (نه فقط tsc) سبز کنید. و ادعای سازگاری هر ابزار را از مستندات خودش بگیرید، نه از حدس.

مدل ذهنی — سرمایه شما زبان است، نه کامپایلر

همه‌چیز این کتاب — مدل ذهنی، الگوها، طراحی نوع — روی **زبان** بنا شده و زبان در این گذار ثابت ماند. کامپایلر پیاده‌سازی است: سریع‌تر شد و باز هم عوض خواهد شد. تیمی که دانشش روی زبان است، از هر گذار ابزاری فقط سود می‌گیرد؛ تیمی که به رفتارهای حاشیه‌ای ابزار تکیه کرده، در هر گذار می‌لرزد.

زیر کاپوت — چرا این‌طور است؟

چرا Microsoft بعد از یک دهه، کامپایلر را از TypeScript به Go بازنویسی کرد؟ پاسخ فنی دقیق، درس معماری مهمی دارد. کامپایلر قدیمی (کد Strada) خودش TS بود و روی Node اجرا می‌شد — سقف کارایی‌اش سقف JavaScript بود: تک‌رشته‌ای در عمل (checker به‌خاطر ساختارهای مشترک، موازی‌سازی واقعی نمی‌پذیرفت)، با GC و فشار حافظه JS برای گراف‌های عظیم نوع. ترجمه به Go (کد Corsa) دو قفل را هم‌زمان باز کرد: **هم‌ارزی ساختار** — Go اجازه داد کد منطقاً «همان» کد قبلی بماند (ترجمه تقریباً خط‌به‌خط، نه بازطراحی؛ ریسک ناسازگاری رفتاری حداقل شد) — و **موازی‌سازی حافظه-مشترک**: چند goroutine روی یک گراف نوع واحد، بدون کپی. ضریب سرعت اعلامی مایکروسافت (~۱۰×) از همین دو است: کد native بدون GC جاوااسکریپت + check موازی. عدد را به‌عنوان ادعای vendor نقل می‌کنیم؛ اندازه‌گیری معتبر، پروژه خودتان است.

دو پیامد عملی که باید بفهمید، نه حفظ کنید: **اول،** «رفتار زبان» تقریباً ثابت ماند — همان قواعد نوع، همان خطاها؛ چیزی که عوض شد پیاده‌سازی است. برای همین 6.0 (آخرین نسخه JSمحور) پل گذار تعریف شد: همان رفتار 7.0 با runtime قدیم، به‌علاوه deprecation گزینه‌های حذف‌شدنی. تمایز «زبان vs پیاده‌سازی» که از فصل ۱ تکرار کرده‌ایم، این‌جا نقد می‌شود. **دوم،** API برنامه‌نویسیِ کامپایلر شکست — ابزارهایی که به internals کامپایلر JS وصل بودند (برخی پلاگین‌ها، ابزارهای تحلیل سفارشی) باید به API جدید مهاجرت کنند؛ کاربر عادی tsc و editor فقط سرعت بیشتر می‌بیند. الگوی آشناست: قرارداد پایدار (زبان) مهاجرت را برای میلیون‌ها کاربر نامرئی کرد؛ هزینه فقط جایی پرداخت شد که کسی به «درونِ» پیاده‌سازی گره خورده بود. همان درس d.ts فصل ۳۱، در مقیاس خود کامپایلر.

تمرین

برنامه ارتقا بنویس

تیم شما: monorepo با ۳ پکیج منتشرشده npm و یک اپ Next.js، الان روی TS 5.9، یک codemod داخلی مبتنی بر API کامپایلر. ترتیب و ریسک‌های ارتقا؟

پاسخ نمونه: ۱. ارتقا به 6.0 در یک PR: رفع deprecationها (target/resolutionهای قدیمی)، CI سبز — ریسک کم. ۲. آزمایش 7.0 در شاخه: نصب، `tsc --build` ، diff کامل d.ts های ۳ پکیج؛ اپ Next را هم build کنید (سازگاری ابزار فریمورک را از مستندات نسخه فعلی آن تأیید کنید). ۳. ریسک اصلی: codemod داخلی روی API قدیمی — گزینه‌ها: ماندن codemod روی typescript@6 به‌عنوان devDependency جدا (دو نسخه هم‌زمان مجاز است)، پورت به API جدید، یا بازنویسی روی ابزار syntax-محور. ۴. سوییچ نهایی تیم + افزونه ادیتور، بعد از یک هفته سبز بودن شاخه.

پرسش‌های مصاحبه این فصل

۱. تیم روی 5.9 است؛ نقشه شما برای 7؟

دو پرش کنترل‌شده: اول 6.0 — deprecation ها را واقعاً رفع کنید (نه ignoreDeprecations دائمی) چون 7 حذفشان کرده؛ خروجی را با `--stableTypeOrdering` پایدار کنید. بعد 7 در branch: کل (lint/test/build) pipeline که به API کامپایلر تکیه دارند) + diff اعلان‌های تولیدی. رفتار type-checking هم‌ارز طراحی شده؛ ریسک در ابزارهاست.

۲. در گذار به کامپایلر native، چه چیزی «زبان» ماند و چه چیزی «پیاده‌سازی»؟

syntax و معنای نوع‌ها و رفتار بررسی: زبان — ثابت و سازگار. سرعت (۱۰× اعلامی)، معماری پردازش موازی، و API جاوااسکریپتی کامپایلر (Strada که دیگر پشتیبانی نمی‌شود): پیاده‌سازی. ابزارهایی که به API قدیم چسبیده‌اند، شکاف واقعی مهاجرت‌اند — نه کد شما.

PART 04

۴

TypeScript در سیستم‌های Production

بخش پایانی، همه ابزارهای کتاب را به مسئله‌های واقعی وصل می‌کند: مرز داده بیرونی و اعتبارسنجی runtime، کلاینت API با مدل خطای صریح، مدل‌سازی دامنه، React و Next.js، فرم‌ها و stateهای ناهمگام، و انتشار پکیج با تست نوع. هشت پروژه کامل این بخش، ستون عملی کتاب‌اند.

34 مرز داده بیرونی — اعتبارسنجی Runtime با Zod

35 کلاینت API تایپ‌شده با مدل خطای صریح

36 مدل‌سازی دامنه و ماشین حالت

37 React با TypeScript

38 Next.js — مرز سرور و کلاینت

39 فرم‌ها، State ناهمگام و Event Emitter تایپ‌شده

40 انتشار پکیج — اعلان عمومی و تست نوع

مرز داده بیرونی — اعتبارسنجی Runtime با Zod

پروژه ۱: نوع ایستا هیچ تضمینی درباره JSON واقعی نمی‌دهد

اهداف یادگیری این فصل

- تفاوت «اعلان نوع» و «اعتبارسنجی مقدار» را در مرز شبکه نشان دهی
- با Zod طرحواره بنویسی و نوع را از آن استنتاج کنی
- الگوی parse-at-the-boundary را در معماری جا بیندازی

این فصل مهم‌ترین مرز کتاب را می‌بندد: جایی که داده از دنیای بیرون وارد می‌شود. همه‌چیز از یک واقعیت شروع می‌شود که فصل ۱ کاشت: `as User` هیچ‌چیز را بررسی نمی‌کند. سرور می‌تواند هر چیزی برگرداند — نسخه جدید API، باگ، پاسخ خطای HTML به‌جای JSON. **پروژه ۱** این فصل: مرز ورودی را با Zod (نسخه ۴) واقعاً امن می‌کنیم.

مسئله را با چشم ببینیم

boundary-lie.ts — type-checks ✓

```ts
interface User { id: number; name: string; email: string }

async function getUser(id: number): Promise<User> {
  const res = await fetch(`/api/users/${id}`);
  return res.json() as Promise<User>;  //  ← دروغِ مؤدبانه
}

const user = await getUser(1);
user.email.toLowerCase();  //  را حذف کرده باشد email اگر سرور 
                           // TypeError: Cannot read properties of undefined
export {};
```

کامپایلر این کد را کامل تأیید می‌کند — چون ما با `as` قسم خورده‌ایم. خطای واقعی، دور از علت، در عمق UI منفجر می‌شود؛ و از همه بدتر: شاید اصلاً منفجر نشود و داده خراب بی‌صدا در دیتابیس بنشیند.

Schema: نوعی که در runtime هم زنده است

کتابخانه‌های schema validation (Zod، Valibot، ArkType — این کتاب Zod 4 را به‌کار می‌برد) یک ایده مشترک دارند: **تعریف واحد که هم بررسی runtime است هم منبع نوع ایستا.**

user-schema.ts

type-checks ✓

```ts
import { z } from "zod";

export const UserSchema = z.object({
  id: z.number().int().positive(),
  name: z.string().min(1),
  email: z.email(),                       // Zod 4: اعتبارسنج‌های top-level
  role: z.enum(["admin", "member"]).default("member"),
  createdAt: z.coerce.date(),             // "2026-01-05" → Date
});

// نوع ایستا از schema (فصل ۱۶!) مشتق می‌شود — یک منبع حقیقت:
export type User = z.infer<typeof UserSchema>;
// { id: number; name: string; email: string;
//   role: "admin" | "member"; createdAt: Date }
```

و مرز، این‌طور بسته می‌شود:

boundary-safe.ts

type-checks ✓

```ts
import { z } from "zod";
import { UserSchema, type User } from "./user-schema.js";

export async function getUser(id: number): Promise<User> {
  const res = await fetch(`/api/users/${id}`);
  if (!res.ok) throw new Error(`HTTP ${res.status}`);

  const parsed = UserSchema.safeParse(await res.json());
  if (!parsed.success) {
    // z.treeifyError — گزارش ساخت‌یافته خطاها (جایگزین format در Zod 4)
    console.error(z.treeifyError(parsed.error));
    throw new Error("پاسخ سرور با قرارداد نمی‌خواند");
  }
  return parsed.data;   // ✓ User — این بار واقعاً
}
```

از این نقطه به بعد، `User` در کل برنامه **گواهی** است (فصل ۲۳): هر مقداری با این نوع، از دروازه عبور کرده. خطای قرارداد، همان لحظه ورود و با پیام دقیق می‌ترکد — نه سه لایه پایین‌تر.

الگوی معماری: parse at the boundary

```
architecture

بیرون (unknown)          درون          مرز (typed)
─────────────────        ──────        ──────────────
fetch/res.json()    →    schema.parse  →   User, Order, ...
webhook payload     →    schema.parse  →   WebhookEvent
process.env         →    EnvSchema.parse → Env (در boot)
localStorage        →    schema.parse  →   SavedDraft | fallback
form input          →    schema.parse  →   ValidForm (فصل ۳۹)
```

قاعده: **داخل برنامه، هیچ `unknown` و هیچ validation پراکنده** — همه در لایه مرز. مزیت ترکیبی: schemaها همان‌جا transform و پیش‌فرض هم انجام می‌دهند (`default`، `coerce.date`)، پس «نرمال‌سازی» هم یک‌جا جمع می‌شود — همان درس دروازه فصل ۲۳.

env-schema.ts — type-checks ✓

```ts
import { z } from "zod";

const EnvSchema = z.object({
  DATABASE_URL: z.url(),
  PORT: z.coerce.number().int().default(3000),
  NODE_ENV: z.enum(["development", "production", "test"]),
});

// اگر env بالا نیاید، سرور اصلاً boot نقطه در ناقص است:
export const env = EnvSchema.parse(process.env);
// env: { DATABASE_URL: string; PORT: number; NODE_ENV: ... } — تمیز و قطعی
```

این دقیقاً «بدهی اثبات» فصل ۲۵ را تسویه می‌کند: augmentation ادعا بود؛ این، اثبات است.

مدل ذهنی — نوع ایستا برای درون، schema برای مرز

تقسیم کار را قاطع نگه دارید: **درون** برنامه، کامپایلر حاکم است — نوع‌ها اثبات می‌شوند و validation تکراری فقط نویز است. **مرز**، قلمرو runtime است — آن‌جا نوع ایستا هیچ قدرتی ندارد و schema حاکم است. بیشتر معماری‌های شکننده، این دو را قاطی می‌کنند: یا به نوع در مرز اعتماد می‌کنند (باگ بی‌صدا)، یا در عمق برنامه هی دوباره validate می‌کنند (نویز و کندی).

هشدار — schema را از نوع ننویسید؛ نوع را از schema بگیرید

جهت اشتباه: interface دست‌نویس + schema دست‌نویس «هماهنگ با آن» — دو منبع حقیقت که بی‌صدا فاصله می‌گیرند. جهت درست همیشه: schema تنها تعریف، `z.infer` تنها نوع. اگر نوع از قبل وجود دارد (مثلاً از OpenAPI)، یا codegen کنید که schema تولید کند، یا با `satisfies z.ZodType<User>` سازگاری schema با آن نوع را کامپایلری کنید.

کارگاه پروژه ۱ — ماژول کامل مرز API

ساختار پروژه (کد کامل قابل اجرا در `examples/boundary`):

project-structure

```
boundary/
├── src/
│   ├── schemas.ts      ← همه schemaهای ورودی + types استنتاجی
│   ├── api.ts          ← fetchJson جنریک با safeParse
│   └── main.ts         ← دمو: پاسخ سالم، پاسخ خراب، env
├── package.json        ← zod ^4
└── tsconfig.json       ← strict + nodenext
```

api.ts — type-checks ✓

```ts
import { z } from "zod";

export class ContractViolation extends Error {
  constructor(public url: string, public issues: z.core.$ZodIssue[]) {
    super(`Contract violation at ${url}`);
  }
}

export async function fetchJson<S extends z.ZodType>(
  url: string,
  schema: S,
  init?: RequestInit
): Promise<z.infer<S>> {
  const res = await fetch(url, init);
  if (!res.ok) throw new Error(`HTTP ${res.status} for ${url}`);
  const result = schema.safeParse(await res.json());
  if (!result.success) throw new ContractViolation(url, result.error.issues);
  return result.data;
}

// استفاده — نوع خروجی کاملاً استنتاجی:
// const user = await fetchJson(`/api/users/1`, UserSchema);
```

معیار پذیرش پروژه: (۱) `npm run typecheck` سبز، (۲) اجرای دمو با پاسخ سالم → خروجی تایپ‌شده، (۳) با پاسخ دستکاری‌شده → `ContractViolation` با گزارش دقیق فیلد خراب، نه TypeError مبهم. تمرین توسعه: schema ها را با `z.discriminatedUnion` برای پاسخ‌های چندحالته API گسترش دهید (پیوند مستقیم به فصل ۲۲ و ۳۵).

زیر کاپوت — چرا این‌طور است؟

این فصل بر یک مفهوم بنیادی استوار است که باید دقیق صورت‌بندی شود: **مرز اعتماد (trust boundary)**. درون برنامه، نوع‌ها «قضیه‌های اثبات‌شده»اند — کامپایلر از لحظه ساخت مقدار تا مصرفش، سازگاری را تضمین کرده. اما داده‌ای که از بیرون می‌آید (شبکه، فایل، localStorage، پیام postMessage، ستون دیتابیس) هیچ اثباتی همراهش نیست؛ نوع‌گذاری مستقیمش — `JSON.parse(x) as User` — یعنی واردکردن «قضیه اثبات‌نشده» به سیستم اثبات. از آن لحظه همه استدلال‌های کامپایلر روی پایه پوک بنا می‌شوند و خطا در دورترین نقطه از علت منفجر می‌شود — بدترین نوع باگ برای debug.

schema دقیقاً «اثبات‌ساز runtime» است: کدی که در لحظه عبور از مرز، شکل واقعی داده را می‌سنجد و فقط در صورت موفقیت، مقداری با نوع اثبات‌شده تحویل می‌دهد. و نبوغ ابزارهایی مثل Zod در حل مشکل دوگانگی است: به‌جای «نوع برای کامپایلر + validator برای runtime» که لاجرم ناهمگام می‌شوند، یک تعریف واحد (schema) هر دو را تولید می‌کند — `z.infer` نوع ایستا را از همان منبع استخراج می‌کند. single source of truth، این‌بار روی مرز.

شعار «parse, don't validate» هم تمایز مهمی دارد که سطحی خوانده می‌شود: validate یعنی «بررسی کن و همان ورودی خام را جلو بفرست» — دانشِ به‌دست‌آمده در جریان کنترل گم می‌شود و هر مصرف‌کننده باید دوباره بی‌اعتماد باشد. parse یعنی «بررسی کن و **مقداری با نوع غنی‌تر بساز**» — دانش، در خود نوع خروجی تجسد می‌یابد و تا ابد همراه مقدار سفر می‌کند؛ دیگر هیچ‌کس لازم نیست دوباره بپرسد. برند فصل ۲۳ حد نهایی همین ایده بود: نوعی که «فقط» از مسیر اعتبارسنجی قابل ساخت است.

و مرز را هوشمندانه انتخاب کنید: validation در «نقطه ورود» یک‌بار انجام می‌شود، نه در هر تابع. درونِ مرز، به نوع‌ها اعتماد کنید — کل ارزش این معماری همین است که بی‌اعتمادی را به لایه نازکی در لبه فشرده می‌کند.

پرسش‌های مصاحبه این فصل

۱. «parse, don't validate» را در یک جمله + یک پیامد معماری؟

به‌جای بررسی‌ای که فقط bool برمی‌گرداند و دانش را دور می‌ریزد، تجزیه‌ای که «مقدارِ تایپ‌دار» تحویل می‌دهد — safeParse خروجی `User` می‌دهد نه «احتمالاً درست است». پیامد: درون برنامه دیگر هیچ بررسی تکراری و دفاعی لازم نیست؛ نوع، گواهی عبور از مرز است.

۲. چرا نوع باید از schema مشتق شود، نه برعکس؟

دو تعریف موازی (interface دستی + schema دستی) دیر یا زود drift می‌کنند و خطرناک‌ترین حالت ممکن را می‌سازند: نوعی که دیگر با validation واقعی یکی نیست. `z.infer` جهت وابستگی را یک‌طرفه می‌کند — schema تنها منبع حقیقت، نوع محصولِ همیشه-همگام.

کلاینت API تایپ‌شده با مدل خطای صریح

پروژه ۲: Result به‌جای exception — خطا به‌عنوان داده

اهداف یادگیری این فصل

- مدل Result/Either را با discriminated union بسازی
- خطاهای شبکه، HTTP و validation را تفکیک و تایپ کنی
- کلاینت API کامل با retry و timeout تایپ‌شده بنویسی

exception در TypeScript یک نقطه کور بزرگ دارد: **در امضا دیده نمی‌شود**. `throw` هر نوعی، از هر عمقی، بدون هیچ ردی در نوع تابع. **پروژه ۲** این فصل، الگوی حرفه‌ای جایگزین را کامل پیاده می‌کند: خطا به‌عنوان داده، با `Result` — و یک کلاینت API واقعی با retry، timeout و مدل خطای تفکیک‌شده.

چرا exception برای خطاهای «منتظَر» ابزار بدی است

exception-blind.ts — type-checks ✓

```
// امضا هیچ‌چیز درباره شکست نمی‌گوید:
declare function chargeCard(amount: number): Promise<string>;
// آیا throw باید سورس را بخوانی — می‌کند؟ چه نوعی؟ شبکه؟ کارت رد شد؟
```

خطاها دو جنس‌اند: **باگ/غیرمنتظره** (قطع حافظه، نقض invariant — بگذارید بترکد و بالا گزارش شود) و **شکستِ قابل انتظار** (کارت رد شد، رکورد نبود، شبکه قطع بود — بخشی از منطق دامنه). جنس دوم باید در **نوع** حضور داشته باشد تا کامپایلر مصرف‌کننده را مجبور به تصمیم کند. ابزارش را فصل ۲۲ ساخته: discriminated union.

مدل Result

result.ts — type-checks ✓

```ts
export type Result<T, E> =
  | { ok: true;  value: T }
  | { ok: false; error: E };

export const ok  = <T,>(value: T): Result<T, never> => ({ ok: true, value });
export const err = <E,>(error: E): Result<never, E> => ({ ok: false, error });

//  چند ابزار ترکیبی که ۸۰٪ نیاز را می‌پوشانند:
export function map<T, U, E>(r: Result<T, E>, f: (v: T) => U): Result<U, E> {
  return r.ok ? ok(f(r.value)) : r;
}
export function unwrapOr<T, E>(r: Result<T, E>, fallback: T): T {
  return r.ok ? r.value : fallback;
}
```

مدل خطای تفکیک‌شده — قلب پروژه

هنر اصلی، طراحی `E` است. «یک Error عمومی» همان نقطه کور قبلی است با لباس جدید. خطاهای واقعی یک کلاینت HTTP سه جنس کاملاً متفاوت دارند:

api-error.ts — type-checks ✓

```ts
import type { z } from "zod";

export type ApiError =
  | { kind: "network";  cause: unknown; retryable: true }       // خودش شکست fetch
  | { kind: "http";     status: number; body: string;           //  سرور پاسخ بد داد
      retryable: boolean }
  | { kind: "contract"; issues: z.core.$ZodIssue[];             //  پاسخ با schema نخواند
      retryable: false };                                        //  فایده ندارد retry !
```

به فیلد `retryable` دقت کنید: تصمیم مهندسی در خود مدل. خطای قرارداد، retry پذیر نیست — پاسخ سرور خراب است، نه مسیر شبکه؛ تلاش مجدد فقط تاخیر و نویز اضافه می‌کند. این‌طور دانش عملیاتی، به‌جای کامنت، وارد نوع می‌شود.

کلاینت کامل — پروژه ۲

client.ts

type-checks ✓

```ts
import { z } from "zod";
import { type Result, ok, err } from "./result.js";
import type { ApiError } from "./api-error.js";

export interface ClientOptions { baseUrl: string; timeoutMs?: number; retries?: number }

export function createClient(opts: ClientOptions) {
  const { baseUrl, timeoutMs = 8000, retries = 2 } = opts;

  async function request<S extends z.ZodType>(
    path: string, schema: S, init?: RequestInit
  ): Promise<Result<z.infer<S>, ApiError>> {
    for (let attempt = 0; ; attempt++) {
      const result = await attemptOnce(path, schema, init);
      if (result.ok || !result.error.retryable || attempt >= retries) return result;
      await new Promise(r => setTimeout(r, 300 * 2 ** attempt));   // backoff
    }
  }

  async function attemptOnce<S extends z.ZodType>(
    path: string, schema: S, init?: RequestInit
  ): Promise<Result<z.infer<S>, ApiError>> {
    let res: Response;
    try {
      res = await fetch(baseUrl + path, {
        ...init,
        signal: AbortSignal.timeout(timeoutMs),  //  لغو تایپ‌شده — فصل ۳۹
      });
    } catch (cause) {
      return err({ kind: "network", cause, retryable: true });
    }
    if (!res.ok) {
      return err({ kind: "http", status: res.status, body: await res.text(),
                   retryable: res.status >= 500 || res.status === 429 });
    }
    const parsed = schema.safeParse(await res.json());
    return parsed.success
      ? ok(parsed.data)
      : err({ kind: "contract", issues: parsed.error.issues, retryable: false });
  }

  return { request };
}
```

و مصرف — جایی که همه فصل‌های قبل جمع می‌شوند:

consume.ts

type-checks ✓

```ts
import { z } from "zod";
import { createClient } from "./client.js";

const api = createClient({ baseUrl: "https://api.shop.dev" });
const ProductSchema = z.object({ id: z.string(), title: z.string(), price: z.number() });

const result = await api.request("/products/p1", ProductSchema);

if (!result.ok) {
  switch (result.error.kind) {                // exhaustive — فصل ۲۲
    case "network":  showToast("اتصال برقرار نشد؛ دوباره تلاش می‌کنیم"); break;
    case "http":     showToast(`خطای سرور (${result.error.status})`); break;
    case "contract": reportToSentry(result.error.issues); showToast("خطای داخلی"); break;
  }
} else {
  render(result.value);    // ✓ { id, title, price } — اثبات‌شده
}

declare function showToast(m: string): void;
declare function reportToSentry(x: unknown): void;
declare function render(p: { id: string; title: string; price: number }): void;
export {};
```

مقایسه کنید با نسخه try/catch: آن‌جا همه خطاها در یک `catch (e: unknown)` له می‌شدند و «کدام پیام به کاربر؟» حدس بود. این‌جا کامپایلر هر سه مسیر را طلب می‌کند — حالت جدید خطا اضافه کنید، همه مصرف‌کننده‌ها قرمز می‌شوند.

مدل ذهنی — امضای صادق

امضای `Promise<Result<Product, ApiError>>` کل داستان را می‌گوید: ناهمگام است، ممکن است شکست بخورد، شکستش این شکل‌هاست. قاعده تقسیم: **شکست قابل انتظار ⇒ در نوع (Result)؛ باگ ⇒ exception که بالا می‌رود**. کد میانی هم ساده می‌ماند: Result را پاس می‌دهد بالا، بدون try/catch تزئینی.

در عمل — کجا Result نبریم

همه‌چیز را Result نکنید: توابع خالص داخلی که شکست قابل انتظار ندارند، exception برای نقض invariant (fail fast فصل ۷)، و لایه‌هایی از فریمورک که قراردادشان exception است (error boundary های error.tsx، React در Next). Result مال **مرزهای عملیات‌های شکست‌پذیر** است — I/O، شبکه، parsing — نه هر تابع دو خطی. کتابخانه‌هایی مثل neverthrow هم نسخه بالغ همین الگو را می‌دهند اگر ابزار بیشتری خواستید.

پروژه کامل در `examples/api-client` — معیار پذیرش: typecheck سبز؛ دمو با سرور mock سه سناریو (موفق، 500 با retry و backoff، پاسخ خراب بدون retry) را با خروجی تفکیک‌شده نشان می‌دهد. تمرین توسعه: متد `requestPaginated` بنویسید که `Result` صفحات را با `AsyncGenerator` ترکیب کند، و خطای `kind: "aborted"` را برای لغو کاربر اضافه کنید (فصل ۳۹ لازمش دارد).

زیر کاپوت — چرا این‌طور است؟

چرا این فصل exception را «ناکافی» می‌داند و Result می‌سازد؟ ریشه مشکل یک غیبت است: **در TypeScript، نوعِ تابع هیچ چیزی درباره throw نمی‌گوید.** امضای `(id: string) ⇒ User` هیچ نشانی ندارد از این‌که این تابع ممکن است سه جور exception پرتاب کند — و کامپایلر نه فراخواننده را به catch وادار می‌کند، نه نوع خطای catch شده را می‌داند (متغیر catch حداکثر unknown است — فصل ۲۷ — چون JS اجازه throw «هر چیزی» را می‌دهد). خطاها از دید سیستم نوع نامرئی‌اند؛ مسیر شکست، تایپ‌نشده‌ترین مسیر کد شماست. (جاوا checked exception را امتحان کرد و به دلایل ارگونومیک شکست خورد؛ TypeScript اصلاً سمتش نرفت.)

الگوی Result یک جابه‌جایی ساده و عمیق است: **شکست را از «رویداد پرتاب‌شدنی» به «مقدار برگشتی» تبدیل کن.** مقدار، نوع دارد — پس شکست وارد امضا می‌شود: `Result<User, NotFound | Forbidden>` . حالا همه ماشین‌آلات کتاب روی مسیر شکست هم کار می‌کنند: discriminated union و narrowing (فصل ۲۲ و ۱۰) برای تفکیک حالت‌ها، exhaustiveness با never برای اطمینان از پوشش همه خطاها، و افزودن خطای جدید به union، همه مصرف‌کننده‌های ناقص را compile error می‌کند. مقایسه کنید با exception: حالت شکست جدید، بی‌صدا از همه catch های موجود عبور می‌کند.

مرزبندی مهندسی‌اش را هم دقیق بگیرید: Result برای **شکست‌های قابل انتظارِ دامنه** است — یافت‌نشدن، اعتبارسنجی ناموفق، تعارض — که مصرف‌کننده باید برایشان «تصمیم» بگیرد و بنابراین باید در قرارداد دیده شوند. exception را برای **نقص‌های برنامه و شرایط بازگشت‌ناپذیر** نگه دارید — assertion شکسته، پیکربندی غایب — که پاسخ درستشان crash زودهنگام است، نه تصمیم محلی. Result همه‌جایی هم نشانه بلوغ نیست: زنجیره‌های طولانی unwrap دستی روی مسیرهایی که هیچ تصمیمی ندارند، فقط نویز است. قاعده: جایی Result بدهید که فراخواننده «کاری» با شکست دارد.

پرسش‌های مصاحبه این فصل

۱. کدام خطاها را در «نوع خروجی» می‌آورید و کدام را throw می‌کنید؟

شکستِ قابل انتظار که مصرف‌کننده باید برایش تصمیم UI/retry بگیرد (شبکه، HTTP، نقض قرارداد، نبود رکورد): در Result — امضا مجبور به رسیدگی می‌کند. باگ و نقض invariant (ورود به شاخه ناممکن): throw تا به مانیتورینگ برسد. مخلوط کردن این دو، یا boilerplate می‌سازد یا خطای بلعیده.

۲. چرا خطای contract هرگز retryable نیست؟

retry برای شرایط «گذرا» است؛ پاسخ بدشکل، گذرا نیست — سرور همان را دوباره می‌فرستد و شما فقط load اضافه کرده‌اید و تأخیر کاربر را. تفکیک با فیلد `retryable` در خود نوع خطا، این تصمیم را از if های پراکنده به قرارداد منتقل می‌کند.

مدل‌سازی دامنه و ماشین حالت

پروژه ۳ و ۴: invariantهای کسب‌وکار در سطح نوع + state machine

اهداف یادگیری این فصل

- قاعده «state ناممکن را غیرقابل‌ساخت کن» را اجرا کنی
- ماشین حالت با گذارهای مجاز تایپ‌شده بسازی
- repository جنریک با قرارداد صریح پیاده کنی

فصل ۲۲ شعار داد: «state ناممکن را غیرقابل‌ساخت کن». این فصل آن را به سطح معماری می‌برد: مدل‌سازی قواعد کسب‌وکار طوری که نقض‌شان کامپایل نشود. دو پروژه کامل: **ماشین حالت سفارش** (پروژه ۳) و **repository جنریک** (پروژه ۴) که فصل ۱۵ وعده‌اش را داد.

invariant چیست و چرا جایش در نوع است

invariant یعنی قاعده‌ای که همیشه باید برقرار باشد: «سفارشِ ارسال‌شده حتماً آدرس دارد»، «فاکتور پرداخت‌شده قابل ویرایش نیست». روش سنتی: بررسی در runtime و امید به discipline تیم. روش این فصل: طوری مدل کنید که حالت ناقض، **قابل بیان نباشد**:

order-model.ts — type-checks ✓

```ts
//  ❌ مدل صاف — invariant ها فقط در ذهن تیم‌اند:
// interface Order { status: string; address?: Address; paidAt?: Date; ... }

//  ✅ هر حالت، دقیقاً با داده‌های تضمین‌شده‌اش:
export interface Address { city: string; street: string; postalCode: string }
export interface LineItem { sku: string; qty: number; unitPrice: number }

export type Order =
  | { status: "draft";     items: LineItem[] }
  | { status: "placed";    items: readonly LineItem[]; address: Address;
      placedAt: Date }
  | { status: "paid";      items: readonly LineItem[]; address: Address;
      placedAt: Date; paidAt: Date; txId: string }
  | { status: "shipped";   items: readonly LineItem[]; address: Address;
      placedAt: Date; paidAt: Date; txId: string; trackingCode: string }
  | { status: "canceled";  reason: string; canceledAt: Date };
```

بخوانیدش مثل مستندات: «ارسال‌شده یعنی حتماً trackingCode دارد و حتماً از paid عبور کرده (txId دارد)». و دو ظرافت: `items` بعد از draft به `readonly` تبدیل می‌شود — ویرایش سبد بعد از ثبت، خطای کامپایل است؛ و `canceled` اصلاً فیلدهای مالی ندارد.

ماشین حالت – گذارهای مجاز هم تایپ شوند

مدل بالا حالت‌ها را درست کرد؛ حالا **گذارها**:

transitions.ts — type-checks ✓

```ts
import type { Order, Address } from "./order-model.js";

// هر گذار: از حالتِ مشخص → به حالتِ مشخص. نه هیچ چیز دیگر.
export function place(
  order: Extract<Order, { status: "draft" }>,
  address: Address
): Extract<Order, { status: "placed" }> {
  if (order.items.length === 0) throw new Error("سبد خالی قابل ثبت نیست");
  return { status: "placed", items: order.items, address, placedAt: new Date() };
}

export function pay(
  order: Extract<Order, { status: "placed" }>,
  txId: string
): Extract<Order, { status: "paid" }> {
  return { ...order, status: "paid", paidAt: new Date(), txId };
}
```

transitions-err.ts — × خطای عمدی

```ts
import { pay } from "./transitions.js";
import type { Order } from "./order-model.js";

declare const draft: Extract<Order, { status: "draft" }>;

pay(draft, "tx_1");  // × کامپایل نمی‌شود – پرداختِ سفارش ثبت‌نشده
```

پیام کامپایلر

```
transitions-err.ts:4:5 - error TS2345: Argument of type
'{ status: "draft"; ... }' is not assignable to parameter of type
'{ status: "placed"; ... }'.
```

این را با نسخه رایج مقایسه کنید: `if (order.status !== "placed") throw` در runtime. آن بررسی خوب است اما **دیر** است – تست یا production باید مسیر اشتباه را اجرا کند تا بفهمید. این‌جا، IDE همان لحظه نوشتن قرمز می‌کند. گذار نامعتبر از «باگ ممکن» به «جمله غیرقابل بیان» تبدیل شد.

برای نمایش نقشه گذارها (مفید برای تست و مستندات):

machine-map.ts

type-checks ✓

```ts
import type { Order } from "./order-model.js";

const TRANSITIONS = {
  draft:    ["placed", "canceled"],
  placed:   ["paid", "canceled"],
  paid:     ["shipped", "canceled"],
  shipped:  [],
  canceled: [],
} as const satisfies Record<Order["status"], readonly Order["status"][]>;
// satisfies: (فصل ۲۱) خطا — اگر حالتی اضافه شود و اینجا جا بماند

export type CanTransition<S extends Order["status"]> = (typeof TRANSITIONS)[S][number];
type FromPaid = CanTransition<"paid">;   // "shipped" | "canceled"
```

نسخه کامل ماشین حالت — با گذارهای `ship` / `cancel`، تست‌های runtime و تست‌های type-level گذارهای ممنوع — در `examples/state-machine`.

پروژه ۴ — repository جنریک با قرارداد صریح

لایه داده‌ای که فصل ۱۵ طرحش را ریخت، حالا با قرارداد کامل و تفکیک خطا (فصل ۳۵):

repository-full.ts

type-checks ✓

```ts
import type { Result } from "./result.js";

export interface Entity { readonly id: string }

export type RepoError =
  | { kind: "not-found"; id: string }
  | { kind: "conflict";  id: string; detail: string }
  | { kind: "storage";   cause: unknown };

export interface Repository<T extends Entity> {
  findById(id: string): Promise<Result<T, RepoError>>;
  list(filter?: Partial<T>, page?: { limit: number; offset: number }):
    Promise<Result<readonly T[], RepoError>>;
  create(input: Omit<T, "id">): Promise<Result<T, RepoError>>;   // id را مخزن می‌سازد
  update(id: string, patch: Partial<Omit<T, "id">>): Promise<Result<T, RepoError>>;
  remove(id: string): Promise<Result<void, RepoError>>;
}
```

تصمیم‌های قرارداد را ببینید — هر کدام یک درس از فصل‌های قبل: `Omit<T, "id">` در create (شناسه را مصرف‌کننده نمی‌دهد)، `Partial<Omit<...>>` در update (پچ جزئی، ولی id هرگز)، `readonly T[]` در list (فصل ۱۲)، `Result` به‌جای throw (فصل ۳۵)، و `Entity.id` با `readonly` (هویت تغییرناپذیر). پیاده‌سازی `InMemoryRepository` و

تست‌ها در `examples/repository` — و چون قرارداد interface است، نسخه Postgres/Prisma بعداً بدون تغییر مصرف‌کننده‌ها جایگزین می‌شود.

مدل ذهنی — کامپایلر به‌عنوان نگهبان قواعد دامنه

هر قاعده کسب‌وکار را که می‌شنوید («فقط سفارش پرداخت‌شده ارسال می‌شود»)، بپرسید: این را می‌توان در **شکل داده** کد کرد؟ اگر بله — union حالت‌دار، brand، Omit، readonly — دیگر به حافظه تیم و code review وابسته نیستید؛ نگهبان، کامپایلر است و شیفتش ۲۴ساعته. فقط قواعدِ واقعاً runtime (مثل «سبد خالی نه») به بررسی اجرایی نیاز دارند — و آن‌ها هم در گذارها متمرکزند، نه پخش در برنامه.

هشدار — مدل دامنه ≠ مدل دیتابیس ≠ مدل API

وسوسه «یک interface برای همه‌جا» را پس بزنید. ردیف دیتابیس (flat، با نقض‌های تاریخی)، پاسخ API (نسخه‌بندی‌شده، serialize شده — فصل ۳۸) و مدل دامنه (invariant دار) سه موجودند با نرخ تغییر متفاوت. مرزشان همان schema های فصل ۳۴ است: `row → domain` و `domain → dto` توابع تبدیل صریح‌اند. هزینه‌اش چند تابع کوچک است؛ سودش این‌که تغییر schema دیتابیس، UI را گروگان نمی‌گیرد.

زیر کاپوت — چرا این‌طور است؟

state machine تایپ‌شده، همگرایی سه قضیه‌ای است که جداجدا اثبات کرده‌ایم — این فصل فقط آن‌ها را سوار هم می‌کند:

۱) حالت باید نوع جمعی باشد، نه ضربی (فصل ۲۲). شیء پرچم‌محور، حاصل‌ضرب دامنه‌هاست و ترکیب‌های ناممکن را مجاز می‌کند؛ union تمیز، فقط حالت‌های واقعی را بیان‌پذیر می‌کند و هر حالت فقط داده‌های خودش را حمل می‌کند (پرداختِ failed دلیلِ شکست دارد، paid شناسه تراکنش — نه هر دو optional روی یک شیء).

۲) گذارِ مجاز باید در امضا بیان شود، نه در کامنت. نکته ظریف پیاده‌سازی این‌جاست: تابع گذار اگر `State` بگیرد و درونش if بزند، قید «فقط از حالت cart می‌شود checkout کرد» runtime ای است. اما اگر ورودی را با `Extract<State, { status: "cart" }>` تایپ کنید، فراخوانی با حالت نادرست، compile error است — قاعده کسب‌وکار به قرارداد ایستا ارتقا یافته. مصرف‌کننده پیش از فراخوانی مجبور به narrowing است (فصل ۱۰) و این «اجبار به بررسی» دقیقاً همان رفتاری است که در code review همیشه آرزویش را داشتید — این‌بار خودکار.

۳) تغییر، باید همه نقاط ناسازگار را خودش پیدا کند. حالتِ جدید به union اضافه می‌شود → همه switch های exhaustive قرمز می‌شوند (قضیه never فصل ۸) → فهرست دقیقِ «جاهایی که باید تصمیم بگیرید» را کامپایلر تولید می‌کند. این خاصیت، ارزش اصلی الگو در عمر طولانی پروژه است: هزینه تغییر از «جستجوی انسانی خطاپذیر» به «فهرست مکانیکی کامل» تبدیل می‌شود.

و صادقانه، مرزش: آنچه ایستا تضمین شد «ترتیب مجاز گذارها در نوع‌هاست»؛ اگر حالت از بیرون بیاید (DB، URL)، در مرز باید parse شود (فصل ۳۴)، و اگر دو گذار هم‌زمان رخ دهند (race)، سیستم نوع حرفی ندارد — هم‌زمانی مسئله runtime است. ایمنی ایستا، جایگزین طراحی هم‌زمانی نیست؛ مکمل آن است.

تمرین

invariant را به نوع ببر

قاعده: «مقاله منتشرشده حتماً slug و تاریخ انتشار دارد؛ پیش‌نویس هیچ‌کدام را لازم ندارد؛ مقاله آرشیوشده همان منتشرشده + تاریخ آرشیو است.» مدل و امضای گذار `archive` را بنویسید.

پاسخ:

quiz36-solution.ts — type-checks ✓

```ts
type Article =
  | { state: "draft";     title: string; body: string }
  | { state: "published"; title: string; body: string;
      slug: string; publishedAt: Date }
  | { state: "archived";  title: string; body: string;
      slug: string; publishedAt: Date; archivedAt: Date };

function archive(
  a: Extract<Article, { state: "published" }>
): Extract<Article, { state: "archived" }> {
  return { ...a, state: "archived", archivedAt: new Date() };
}
```

آرشیوِ پیش‌نویس حالا غیرقابل بیان است — و دقت کنید spread همه تضمین‌های published (slug، publishedAt) را بدون بازنویسی منتقل کرد.

پرسش‌های مصاحبه این فصل

۱. «حالت ناممکن غیرقابل‌ساخت» را با یک مثال ۳۰ثانیه‌ای بگویید.

سفارش: به‌جای شیء صاف با optional های موازی، union حالت‌دار — `shipped` حتماً trackingCode و txId دارد، `canceled` اصلاً فیلد مالی ندارد. و گذارها تابع‌های حالت‌مشخص‌اند (`pay(order: Extract<Order, {status:"placed"}>)`) — پرداختِ سفارش ثبت‌نشده، جمله‌ای است که در این زبان نمی‌شود گفت.

۲. چرا مدل دامنه، DTO و ردیف دیتابیس را جدا نگه می‌دارید؟

سه نرخ تغییر و سه مالک متفاوت: schema دیتابیس با migration ها، قرارداد API با نسخه‌بندی بیرونی، invariant های دامنه با منطق کسب‌وکار. یک interface مشترک یعنی هر تغییر ستون، UI را گروگان بگیرد. هزینه جدایی چند تابع تبدیل صریح است — ارزان‌ترین بیمه معماری.

TypeScript با React

پروژه ۵: hook، propsها، reducer، context و کامپوننت جنریک

اهداف یادگیری این فصل

- props و event-ها را با الگوهای رایج React 19 تایپ کنی
- useReducer و Context را بدون as و بدون any مدل کنی
- کامپوننت جنریک (مثل جدول داده) طراحی کنی

React و TypeScript برای هم ساخته شده‌اند: props قرارداد است، state ماشین حالت است، و hookها توابع جنریک‌اند — همه ابزارهایش را دارید. این فصل الگوهای React 19 را مرور می‌کند و در **پروژه ۵**، یک فیچر کامل (جدول داده جنریک + hook داده‌گیری) می‌سازد. فرض: React را بلدید؛ تمرکز فقط روی نوع‌هاست.

Props — قرارداد کامپوننت

button.tsx — type-checks ✓

```tsx
import { type ReactNode, type ComponentPropsWithoutRef } from "react";

// الگوی حرفه‌ای: props دکمه + افزوده‌های خودتان
interface ButtonProps extends ComponentPropsWithoutRef<"button"> {
  variant?: "primary" | "ghost" | "danger";
  loading?: boolean;
  children: ReactNode;
}

export function Button({ variant = "primary", loading, children, ...rest }: ButtonProps) {
  return (
    <button data-variant={variant} disabled={loading || rest.disabled} {...rest}>
      {loading ? "..." : children}
    </button>
  );
}

// مصرف: onClick، type، aria-* — همه از ComponentPropsWithoutRef رایگان آمدند
<Button variant="ghost" onClick={e => console.log(e.currentTarget)}>ورود</Button>;
```

نکته‌های این قطعه، پاسخ ۹۰٪ سؤالات props اند: `ComponentPropsWithoutRef<"button">` به‌جای بازنویسی دستی propsهای بومی؛ union literal برای variant (فصل ۵)؛ `ReactNode` برای children. و در React 19، یک prop معمولی است — برای کامپوننت‌های خودتان `forwardRef` لازم نیست؛ فقط `ref?: Ref<HTMLButtonElement>` را در props بیاورید.

رویدادها و handlerها

events.tsx — type-checks ✓

```tsx
import { useState, type ChangeEvent, type FormEvent } from "react";

export function SearchForm({ onSearch }: { onSearch: (q: string) => void }) {
  const [query, setQuery] = useState("");  // استنتاج: string

 // تعریفش می‌کنید JSX جدا — وقتی بیرون handler نوع:
  const handleChange = (e: ChangeEvent<HTMLInputElement>) => setQuery(e.target.value);

  const handleSubmit = (e: FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    onSearch(query.trim());
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* inline نوع باشد، e خودش می‌آید — contextual typing ۴ فصل */}
      <input value={query} onChange={handleChange} />
    </form>
  );
}
```

قاعده عملی: handler درون JSX ⇒ بدون annotation (زمینه هست)؛ handler بیرون JSX ⇒ `ChangeEvent<HTMLInputElement>` و خویشاوندانش. اگر annotation ها تکراری شدند، نوع handler را alias کنید.

State پیچیده: useReducer + discriminated union

وقتی state چند حالت دارد، درس فصل ۲۲ مستقیم اعمال می‌شود — هم روی state و هم روی actionها:

reducer.tsx

type-checks ✓

```tsx
import { useReducer } from "react";

interface Item { id: string; title: string }
type FetchState =
  | { status: "idle" }
  | { status: "loading" }
  | { status: "success"; items: Item[] }
  | { status: "error"; message: string };

type FetchAction =
  | { type: "started" }
  | { type: "resolved"; items: Item[] }
  | { type: "rejected"; message: string };

function reducer(state: FetchState, action: FetchAction): FetchState {
  switch (action.type) {
    case "started":  return { status: "loading" };
    case "resolved": return { status: "success", items: action.items };
    case "rejected": return { status: "error", message: action.message };
  }
}

export function List() {
  const [state, dispatch] = useReducer(reducer, { status: "idle" } satisfies FetchState);
  void dispatch;
  // render با narrowing — های دفاعی"؟داده هست" و بدون ?. بدون:
  switch (state.status) {
    case "idle":    return <p>آماده</p>;
    case "loading": return <p>در حال بارگذاری…</p>;
    case "error":   return <p role="alert">{state.message}</p>;
    case "success": return <ul>{state.items.map(i => <li key={i.id}>{i.title}</li>)}</ul>;
  }
}
```

`dispatch({ type: "resolved" })` بدون items؟ خطای کامپایل. اضافه شدن حالت جدید؟ switch رندر قرمز می‌شود. این همان «UI به‌عنوان تابعی از ماشین حالت» است — با نگهبان کامپایلری.

کامپوننت جنریک — پروژه ۵

جدول داده‌ای که ستون‌هایش به نوع ردیف قفل‌اند:

data-table.tsx

type-checks ✓

```tsx
import { type ReactNode } from "react";

interface Column<Row> {
  header: string;
  // یا کلیدِ خود ردیف، یا رندر سفارشی — discriminated:
  cell: keyof Row | ((row: Row) => ReactNode);
}

interface DataTableProps<Row extends { id: string | number }> {
  rows: readonly Row[];
  columns: readonly Column<Row>[];
  onRowClick?: (row: Row) => void;
}

export function DataTable<Row extends { id: string | number }>(
  { rows, columns, onRowClick }: DataTableProps<Row>
) {
  return (
    <table>
      <thead><tr>{columns.map(c => <th key={c.header}>{c.header}</th>)}</tr></thead>
      <tbody>
        {rows.map(row => (
          <tr key={row.id} onClick={() => onRowClick?.(row)}>
            {columns.map(c => (
              <td key={c.header}>
                {typeof c.cell === "function" ? c.cell(row) : String(row[c.cell])}
              </td>
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  );
}

// استنتاج می‌شود، ستون‌ها قفل می‌شوند — Row مصرف:
interface User { id: number; name: string; email: string }
declare const users: User[];

<DataTable
  rows={users}
  columns={[
    { header: "نام", cell: "name" },
    { header: "ایمیل", cell: u => <a href={`mailto:${u.email}`}>{u.email}</a> },
    // { header: "سن", cell: "age" },   // ✗ خطا: "age" عضو keyof User نیست
  ]}
  onRowClick={u => console.log(u.email)}   // u: User — کامل استنتاج
/>;
```

کامپوننت جنریک، همان تابع جنریک است (فصل ۱۴) — `<Row,>` روی خود تابع. مصرف‌کننده هیچ براکتی نمی‌بیند: `rows` می‌دهد، بقیه قفل می‌شود. این معیار فصل ۱۵ («جنریک خوب نامرئی است») در UI است.

در عمل — Context بدون دروغ undefined

الگوی توصیه‌شده برای Context که مقدار اولیه معنادار ندارد: `createContext<Auth | null>(null)` + hook مصرفی که null را با خطای واضح رد می‌کند (`useAuth` که throw می‌کند). از `createContext<Auth>({} as Auth)` بپرهیزید — دروغی که خطای واقعی را به «undefined is not a function» در عمق تبدیل می‌کند. نسخه کامل در پروژه /examples react-feature.

هشدار — useState با union، آرگومان نوع می‌خواهد

`useState(null)` نوع `null` استنتاج می‌کند و ست کردن مقدار واقعی خطا می‌شود. هر جا state از چند نوع است، صریح بگویید: `useState<User | null>(null)` . همین برای آرایه خالی: `useState<Item[]>([])` — وگرنه `never[]` (فصل ۴، همان دام آرایه خالی).

پروژه کامل `examples/react-feature` : فیچر «لیست کاربران» با DataTable جنریک، hook داده‌گیری مبتنی بر Result فصل ۳۵، Context احراز هویت و تست‌های نوع. معیار پذیرش: `tsc --noEmit` سبز با strict کامل، صفر `as` و صفر `any` در کل فیچر. تمرین توسعه: ستون‌ها را sortable کنید طوری که فقط کلیدهای با مقدار `string | number` قابل sort باشند (ترکیب فصل ۱۸: `PickByType`).

زیر کاپوت — چرا این‌طور است؟

تایپ در React را وقتی عمیق می‌فهمید که ببینید «کامپوننت» برای کامپایلر چیست: **تابعی که props می‌گیرد و JSX برمی‌گرداند** — و JSX خودش فقط قند syntax برای فراخوانی تابع است. پس `<Button label="ok" />` دقیقاً یک فراخوانی معمولی است: شیء attribute ها به‌عنوان آرگومان props سنجیده می‌شود، با همه ماشین‌آلات آشنا — excess check روی literal تازه (فصل ۶: attribute اشتباه‌تایپ‌شده همان‌جا خطاست)، union و narrowing برای props شرطی، contextual typing برای callback ها (`onClick` بدون annotation نوع event را می‌داند — فصل ۴). هیچ قاعده جدیدی در کار نیست؛ JSX فقط «پوسته» است.

از این تصویر، دو الگوی مهم فصل استدلال‌پذیر می‌شوند:

props به‌مثابه discriminated union. کامپوننتی که «یا href یا onClick» می‌پذیرد، با props ضربی (هر دو optional) همان مشکل فصل ۲۲ را دارد: ترکیب‌های ناممکن مجازند. union تمیز با discriminant، استفاده نادرست را compile error می‌کند — API کامپوننت، قراردادی به دقت API سرور.

کامپوننت جنریک. `List<T>` که `items: T[]` و `renderItem: (item: T) ⇒ ReactNode` می‌گیرد، همان «گره زدن دو جایگاه» فصل ۱۴ است: T از items استنتاج می‌شود و به renderItem جاری می‌شود — مصرف‌کننده بدون نوشتن حتی یک callback، annotation کاملاً تایپ‌شده دارد. این الگو ستون کتابخانه‌های جدول/فرم/لیست حرفه‌ای است.

و یک چرای تاریخی که همه می‌پرسند: چرا `useState<User | null>(null)` آرگومان نوع لازم دارد؟ چون استنتاج فقط از «آرگومان موجود» کاندیدا می‌گیرد (فصل ۱۴) — از null فقط null درمی‌آید؛ هیچ جایگاهی به کامپایلر نگفته «بعداً User

هم می‌آید». آرگومان صریح، همان اطلاع غایب است. الگوی عمومی‌اش: هر جا state با مقدار «فقیرتر از دامنه واقعی» شروع می‌شود، union را صریح بدهید.

پرسش‌های مصاحبه این فصل

۱. props یک دکمه سفارشی که باید همه رفتار بومی را هم داشته باشد؟

`interface Props extends ComponentPropsWithoutRef<"button">` + فیلدهای خودی (variant با union literal) — onClick و disabled و aria-* مجانی می‌آیند و دستی بازنویسی نمی‌شوند. در React 19، ref هم prop معمولی است و forwardRef لازم نیست.

۲. `useState(null)` چه مشکلی دارد؟

نوع state به `null` استنتاج می‌شود و ست کردن مقدار واقعی خطاست. هر جا state چندنوعی است آرگومان صریح بدهید: `useState<User | null>(null)` ؛ و خواهر همین دام: `useState([])` که `never[]` می‌شود. قاعده: مقدار اولیه «فقیرتر» از دامنه واقعی = annotation بده.

Next.js — مرز سرور و کلاینت

پروژه ۶: route handlerها، serialization و قرارداد API

اهداف یادگیری این فصل

- مرز serialization بین سرور و کلاینت را تایپ و اعتبارسنجی کنی
- route handler با قرارداد ورودی/خروجی صریح بنویسی
- بدانی کدام تضمین‌ها در عبور از مرز از بین می‌روند

Next.js (نسخه ۱۶، App Router) برنامه شما را به دو دنیا تقسیم می‌کند: کدی که روی سرور اجرا می‌شود و کدی که در مرورگر. بین این دو، یک مرز serialization هست که **نوع‌ها از آن عبور می‌کنند ولی تضمین‌ها نه**. این فصل + **پروژه ۶**: قرارداد تایپ‌شده و اعتبارسنجی‌شده روی این مرز، از route handler تا کامپوننت.

مرز سرور/کلاینت — چه چیزی در عبور گم می‌شود

server-page.tsx — type-checks ✓

```
// app/orders/page.tsx — Server Component (پیش‌فرض)
import { db } from "@/lib/db";
import { OrderList } from "./order-list";

export default async function OrdersPage() {
  const orders = await db.orders.findRecent();  //  نوع کامل دامنه، سمت سرور
 //  عبور از مرز: props باید serializable باشند
  return <OrderList orders={orders.map(o => ({
    id: o.id,
    title: o.title,
    placedAt: o.placedAt.toISOString(),    // Date → string، صریح و آگاهانه
    total: o.total,
  }))} />;
}
```

قاعده‌ای که باید حفظ باشید: props عبوری از سرور به کلاینت، و خروجی Server Actionها، فقط مقادیر قابل serialize: primitives، آرایه/شیء ساده، و تعداد محدودی نوع خاص. `Date` عبور می‌کند اما `Map` ، `class instance` ، تابع و `undefined` در JSON معمولی خیر — و مهم‌تر: **نوع TypeScript شما این مرز را نمی‌فهمد**؛ interface با متد، بدون خطای کامپایل تا مرز می‌آید و در runtime چیز دیگری تحویل می‌شود. راه‌حل معماری: **DTO صریح برای مرز** (فصل ۳۶: مدل دامنه ≠ مدل API) — تبدیل آگاهانه، مثل `toISOString` بالا.

Route Handler — قرارداد ورودی/خروجی با Zod

پروژه ۶ — endpoint کامل با اعتبارسنجی دوطرفه:

app/api/orders/route.ts — type-checks ✓

```ts
import { NextResponse } from "next/server";
import { z } from "zod";

// ۱) قرارداد: منبع حقیقت مشترک سرور/کلاینت
export const CreateOrderInput = z.object({
  items: z.array(z.object({ sku: z.string(), qty: z.number().int().min(1) })).min(1),
  couponCode: z.string().optional(),
});
export const CreateOrderOutput = z.object({
  orderId: z.string(),
  total: z.number(),
  placedAt: z.iso.datetime(),      // DTO: رشته ISO — صادقانه تاریخ در
});
export type CreateOrderInputT  = z.infer<typeof CreateOrderInput>;
export type CreateOrderOutputT = z.infer<typeof CreateOrderOutput>;

// ۲) هندلر: مرز ورودی همین‌جا بسته می‌شود
export async function POST(req: Request) {
  const parsed = CreateOrderInput.safeParse(await req.json().catch(() => null));
  if (!parsed.success) {
    return NextResponse.json(
      { error: "invalid-input", issues: parsed.error.issues },
      { status: 400 }
    );
  }
  const order = await placeOrder(parsed.data);  // دامنه — فصل ۳۶
  const dto: CreateOrderOutputT = {
    orderId: order.id, total: order.total, placedAt: order.placedAt.toISOString(),
  };
  return NextResponse.json(dto, { status: 201 });
}

declare function placeOrder(input: CreateOrderInputT):
  Promise<{ id: string; total: number; placedAt: Date }>;
```

و سمت کلاینت، همان schema مصرف می‌شود — کلاینت فصل ۳۵ این‌جا به خانه می‌رسد:

lib/orders-client.ts

type-checks ✓

```ts
import { CreateOrderOutput, type CreateOrderInputT } from "@/app/api/orders/route";
import { createClient } from "./api-client";  //  پروژه ۲

const api = createClient({ baseUrl: "" });

export function createOrder(input: CreateOrderInputT) {
  return api.request("/api/orders", CreateOrderOutput, {
    method: "POST",
    headers: { "content-type": "application/json" },
    body: JSON.stringify(input),
  });
  // Promise<Result<{ orderId; total; placedAt }, ApiError>>
}
```

دقت کنید چه چیزی به دست آوردیم: ورودی در سرور validate می‌شود (به fetch کلاینت اعتمادی نیست — هر کسی می‌تواند POST بزند)، خروجی در کلاینت validate می‌شود (به deploy ناهم‌زمان سرور اعتمادی نیست)، و **هر دو از یک schema** — drift ناممکن.

Server Actions — همان قواعد، صورت جدید

actions.ts

type-checks ✓

```ts
"use server";
import { z } from "zod";

const RenameInput = z.object({ orderId: z.string(), title: z.string().min(1).max(80) });

export async function renameOrder(raw: unknown) {
  const input = RenameInput.parse(raw);       //  ورودی action = ورودی بیرونی!
  await db.orders.rename(input.orderId, input.title);
  return { ok: true as const };
}
declare const db: { orders: { rename(id: string, t: string): Promise<void> } };
```

Server Action از دید نوع‌ها «تابع معمولی» به نظر می‌رسد — همین خطرناکش می‌کند: در واقعیت یک endpoint شبکه است و ورودی‌اش از مرورگرِ قابل دستکاری می‌آید. قاعده: پارامتر را `unknown` بگیرید (یا `FormData`) و اول parse کنید. امضای «تایپ‌شده» بدون parse، همان `as User` فصل ۳۴ است با لباس فریمورک.

مدل ذهنی — هر «use client/server» یک مرز اعتماد است

در Next.js مرزها syntax سبکی دارند — یک directive، یک import — و همین باعث می‌شود فراموش شوند. تمرین ذهنی: هر جا داده از RSC به client component، از فرم به action، یا از fetch به کامپوننت می‌رود، در ذهنتان یک «دیوار JSON» بکشید و بپرسید: این‌طرف دیوار چه کسی تضمین می‌دهد؟ پاسخ درست همیشه یکی است: schema در نقطه ورود (فصل ۳۴)، DTO صریح در نقطه خروج (فصل ۳۶).

در عمل — نوع‌های کمکی Next که باید بشناسید

- پارامترهای مسیر در Next 16 ناهمگام‌اند: `params: Promise<{ id: string }>` — امضای صفحه‌ها را با مستندات نسخه خودتان چک کنید؛ typedRoutes هم لینک‌های `<Link href>` را compile-time می‌کند.
- `NextResponse` / `NextRequest` فقط وقتی لازم‌اند که از امکانات خاص (cookies، geo) استفاده می‌کنید؛ وگرنه Request/Response استاندارد، قابل حمل‌تر است.
- متغیرهای محیطی: schema در boot (فصل ۳۴) + جداسازی سخت `NEXT_PUBLIC_*` — هر چیز بدون این پیشوند، فقط سرور.

پروژه کامل در `examples/next-boundary` (route handler + کلاینت + تست قرارداد). معیار پذیرش: typecheck سبز؛ POST با body خراب → 400 ساخت‌یافته؛ پاسخ دستکاری‌شده → ContractViolation سمت کلاینت. تمرین توسعه: خطاهای 400 سرور را در کلاینت به `kind: "validation"` با نگاشت فیلد→پیام تبدیل کنید تا فرم فصل ۳۹ مستقیم مصرفش کند.

زیر کاپوت — چرا این‌طور است؟

fullstack type safety بر یک مشاهده تلخ بنا شده: **جدی‌ترین مرز اعتماد، بین دو تکه کد خودِ شماست.** سرور و کلاینت را یک تیم می‌نویسد، اما بینشان شبکه است و JSON — و serialization «پاک‌کننده دوم» نوع‌هاست، بی‌رحم‌تر از کامپایلر: تاریخ، string می‌شود؛ undefined حذف می‌شود؛ Map و کلاس، شیء خام. حتی اگر دو طرف یک interface مشترک import کنند، چیزی که واقعاً از سیم رد می‌شود با آن interface فرق دارد — «نوع مشترک» بدون «قرارداد سریال‌سازی»، توهم ایمنی است؛ ناهمگامیِ کلاسیک: سرور فیلدی را rename می‌کند و کلاینت سبزِ کامپایل، در runtime می‌شکند.

راه‌حل‌های این فصل همگی یک ایده‌اند با دو پیاده‌سازی: **قرارداد واحد + استخراج نوع از همان قرارداد.**

مسیر schema (فصل ۳۴ روی سیم): قرارداد هر endpoint یک schema است؛ سرور خروجی را با آن می‌سازد/می‌سنجد، کلاینت ورودی را با آن parse می‌کند، و هر دو نوعشان را با infer از «همان» schema می‌گیرند. تغییر قرارداد = تغییر یک فایل = قرمز شدن هم‌زمان هر دو طرف.

مسیر tRPC ادعای جسورانه‌تری دارد که سازوکارش را باید بدانید: «نوع‌ها بدون codegen از سرور به کلاینت می‌رسند». چطور؟ کلاینت، **نوعِ** router سرور را import می‌کند — `import type { AppRouter }` — که طبق فصل ۲ در خروجی JS هیچ اثری ندارد؛ هیچ کد سروری وارد bundle کلاینت نمی‌شود، فقط گراف نوع در زمان check عبور می‌کند. کامپایلر، خودِ «کانال انتقال قرارداد» شده است. بهایش هم از همین سازوکار معلوم است: کلاینت و سرور باید در یک فضای check مشترک باشند (monorepo) — و مرز با دنیای بیرون (کلاینت‌های ثالث، موبایل) همچنان schema یا OpenAPI می‌خواهد.

قاعده تصمیم: درون یک monorepo هم‌زبان، tRPC/نوع‌مشترک+schema کم‌اصطکاک‌ترین است؛ مرز عمومی یا چندزبانه، قرارداد صریح و مستقل از زبان (OpenAPI + تولید نوع) — چون قرارداد عمومی باید بدون «فضای check مشترک» قابل مصرف باشد.

پرسش‌های مصاحبه این فصل

۱. چرا پارامتر Server Action را unknown می‌گیرید؟

چون action از دید نوع‌ها تابع معمولی است ولی در واقعیت endpoint شبکه است — ورودی از مرورگرِ قابل دستکاری می‌آید و هر کسی می‌تواند مستقیم صدایش بزند. امضای تایپ‌شده بدون parse همان `as User` است با لباس فریمورک. اول schema.parse، بعد منطق.

۲. از RSC به client component چه چیزهایی عبور نمی‌کنند و دفاعتان چیست؟

props باید serializable باشند — Map، class instance، تابع و امثالش عبور نمی‌کنند و نوع ایستا هم این مرز را نمی‌فهمد؛ interface با متد، بی‌خطا کامپایل و در runtime چیز دیگری تحویل می‌شود. دفاع: DTO صریح در مرز (Date → رشته ISO آگاهانه) و schema مشترک برای هر دو سمت.

فرم‌ها، State ناهمگام و Event Emitter تایپ‌شده

پروژه ۷: چرخه کامل فرم + ابزار کتابخانه‌ای با نگاشت رویداد

اهداف یادگیری این فصل

- state فرم را با union صریح (.../idle/submitting) مدل کنی
- لغو عملیات ناهمگام را با AbortController تایپ کنی
- event emitter جنریک با نگاشت رویداد→payload بسازی

این فصل سه قطعه حرفه‌ای را کامل می‌کند که در هر اپ واقعی تکرار می‌شوند: چرخه کامل فرم (**پروژه ۷**)، لغو عملیات ناهمگام با AbortController، و یک event emitter جنریک کتابخانه‌ای (**پروژه ۸ بخش اول**). هر سه، ترکیب مستقیم ابزارهای فصل‌های قبل‌اند — این‌جا دیگر مفهوم جدیدی نیست؛ مهارتِ ترکیب است.

فرم — state صریح به‌جای booleanهای موازی

form-state.ts

type-checks ✓

```
import { z } from "zod";

export const SignupSchema = z.object({
  email: z.email("ایمیل معتبر نیست"),
  password: z.string().min(8, "حداقل ۸ کاراکتر"),
});
export type SignupData = z.infer<typeof SignupSchema>;

// state ماشین حالت، نه کپه — فرم — boolean (فصل ۲۲):
export type FormState =
  | { phase: "editing";    errors: Partial<Record<keyof SignupData, string>> }
  | { phase: "submitting" }
  | { phase: "succeeded";  userId: string }
  | { phase: "failed";     message: string; errors: Partial<Record<keyof SignupData,
string>> };
```

به نوع `errors` دقت کنید: `Partial<Record<keyof SignupData, string>>` — کلیدها به فیلدهای schema قفل‌اند؛ typo در نام فیلد خطای کامپایل است، و فیلد جدید در schema یعنی ادیتور همه‌جا کمکتان می‌کند.

form-logic.ts

type-checks ✓

```ts
import { z } from "zod";
import { SignupSchema, type FormState, type SignupData } from "./form-state.js";
import type { Result } from "./result.js";
import type { ApiError } from "./api-error.js";

export function validate(raw: Record<string, string>):
  | { ok: true; data: SignupData }
  | { ok: false; errors: Partial<Record<keyof SignupData, string>> } {
  const parsed = SignupSchema.safeParse(raw);
  if (parsed.success) return { ok: true, data: parsed.data };
  const errors: Partial<Record<keyof SignupData, string>> = {};
  for (const issue of parsed.error.issues) {
    const key = issue.path[0] as keyof SignupData | undefined;
    if (key && !errors[key]) errors[key] = issue.message;  //  اولین پیام هر فیلد
  }
  return { ok: false, errors };
}

export async function submit(
  state: FormState, raw: Record<string, string>,
  api: (d: SignupData) => Promise<Result<{ userId: string }, ApiError>>
): Promise<FormState> {
  if (state.phase === "submitting") return state;  //  گذار تکراری ممنوع — فصل ۳۶
  const v = validate(raw);
  if (!v.ok) return { phase: "editing", errors: v.errors };
  const result = await api(v.data);
  return result.ok
    ? { phase: "succeeded", userId: result.value.userId }
    : { phase: "failed", message: describeError(result.error), errors: {} };
}

declare function describeError(e: ApiError): string;
```

کل جریان، تایپ‌شده سرتاسر: schema → داده معتبر → state → Result بعدی. UI (React فصل ۳۷ یا هر چیز دیگر) فقط روی `phase` سوییچ می‌کند. هیچ `isLoading && !error && touched` در کار نیست.

لغو — AbortController به‌عنوان قرارداد نوع

جست‌وجوی زنده، کلاسیک‌ترین مشتری لغو است: درخواست قبلی باید بمیرد تا پاسخ دیرآمده، نتیجه تازه را له نکند:

cancellation.ts

type-checks ✓

```ts
export function createSearcher<T>(
  fetcher: (q: string, signal: AbortSignal) => Promise<T>
) {
  let controller: AbortController | null = null;

  return async function search(q: string): Promise<T | "aborted"> {
    controller?.abort();                    // نسل قبلی را بکش
    controller = new AbortController();
    try {
      return await fetcher(q, controller.signal);
    } catch (e) {
      if (e instanceof DOMException && e.name === "AbortError") return "aborted";
      throw e;  // خطای واقعی — بالا برود (فصل ۳۵: تفکیک جنس خطا)
    }
  };
}

// در امضا یعنی «این عملیات لغوپذیر است» — قرارداد، نه کامنت. مصرف‌کننده: signal
const searchUsers = createSearcher((q, signal) =>
  fetch(`/api/users?q=${encodeURIComponent(q)}`, { signal }).then(r => r.json() as Promise<unknown>)
);
void searchUsers;
```

قاعده طراحی: هر API ناهمگامِ «طولانی یا رقابتی» باید `signal: AbortSignal` بپذیرد و آن را به لایه‌های پایین پاس بدهد — `fetch` بومی می‌فهمدش، تایمرها با `AbortSignal.timeout` ترکیب می‌شوند (فصل ۳۵ استفاده کرد). خروجی `"aborted"` به‌جای exception هم تصمیم آگاهانه است: لغو، شکست نیست؛ جریان عادی است.

Event Emitter جنریک — نگاشت رویداد→payload

قلب پروژه ۸: emitter ای که در آن نام رویداد و شکل payload با هم قفل‌اند:

typed-emitter.ts

type-checks ✓

```ts
type Listener<P> = (payload: P) => void;

export class TypedEmitter<EventMap extends Record<string, unknown>> {
  private listeners: { [K in keyof EventMap]?: Set<Listener<EventMap[K]>> } = {};

  on<K extends keyof EventMap>(event: K, fn: Listener<EventMap[K]>): () => void {
    (this.listeners[event] ??= new Set()).add(fn);
    return () => this.listeners[event]?.delete(fn);   // unsubscribe
  }

  emit<K extends keyof EventMap>(event: K, payload: EventMap[K]): void {
    this.listeners[event]?.forEach(fn => fn(payload));
  }
}

//  تعریف قرارداد رویدادهای اپ — literal و متمرکز.
//  نکته ظریف: type نه interface — چون interface ای که از Record<string, unknown>
//  ارث ببرد، index signature رشته‌ای می‌گیرد و هر نام رویدادی را می‌پذیرد!
type AppEvents = {
  "cart:add":    { sku: string; qty: number };
  "cart:clear":  undefined;
  "auth:logout": { reason: "manual" | "expired" };
};

const bus = new TypedEmitter<AppEvents>();

const off = bus.on("cart:add", p => console.log(p.sku, p.qty));  // p استنتاج شد
bus.emit("cart:add", { sku: "k1", qty: 2 });     // ✓
bus.emit("auth:logout", { reason: "expired" });  // ✓
off();
```

emitter-err.ts

× خطای عمدی

```ts
declare const bus: {
  emit<K extends "cart:add">(e: K, p: { sku: string; qty: number }): void;
};
bus.emit("cart:add", { sku: "k1", qty: "two" });   // × payload غلط
```

پیام کامپایلر

```
emitter-err.ts:4:31 - error TS2322: Type 'string' is not assignable
to type 'number'.
```

همه ابزارها این‌جا جمع‌اند: `K extends keyof EventMap` (فصل ۱۶)، indexed access برای payload، mapped type برای storage (فصل ۱۸)، و استنتاج در نقطه مصرف (فصل ۱۵). همین الگو — «نگاشت کلید→شکل» — ستون نوع‌دهی socketها، message busها و IPC در پروژه‌های واقعی است.

مدل ذهنی — سه قطعه، یک قاعده

فرم، لغو، emitter — ظاهراً سه موضوع؛ در عمق یک قاعده: **هر پروتکل زمانی/رویدادی را به شکل داده صریح دربیاور.** فازهای فرم union شدند؛ «لغوپذیری» پارامتر signal شد؛ «چه رویدادی چه داده‌ای دارد» نگاشت نوعی شد. وقتی پروتکل داده شد، کامپایلر ناظرش می‌شود — این فرمول را روی هر الگوی async دیگری هم می‌توانید اجرا کنید.

پروژه‌های `examples/form-flow` و `examples/typed-emitter` نسخه کامل + تست دارند. معیار پذیرش فرم: سناریوی خطای فیلد، خطای شبکه و موفقیت، هر سه فقط با تغییر phase رندر شوند. تمرین توسعه: به emitter متد `once` و `waitFor(event): Promise<EventMap[K]>` اضافه کنید — دومی ترکیب شیرینی از جنریک و Promise است.

زیر کاپوت — چرا این‌طور است؟

طراحی یک API client جنریک، آزمون نهایی چند اصل کتاب است — و دو تصمیم ظریفش را باید با «چرا» بدانید:

چرا نوع پاسخ از پارامتر K (کلید endpoint) استنتاج می‌شود، نه آرگومان صریح؟ چون `client.get("/users")` باید بدون هیچ annotation ای نوع درست بدهد — و این دقیقاً الگوی «گره زدن جایگاه‌ها» فصل ۱۴ است: K از آرگومان literal استنتاج می‌شود (freshness فصل ۵ — literal در جایگاه با انتظار، widen نمی‌شود)، و نوع خروجی `Spec[K]` از همان K مشتق می‌شود. نقشه endpoint ها (`Spec`) تنها منبع حقیقت است: مسیر جدید را یک‌جا اضافه می‌کنید، کل client بدون تغییرِ حتی یک خط، آن را تایپ‌شده می‌شناسد. اگر به‌جای این، `get<User>("/users")` طراحی می‌کردید، نوع و مسیر از هم جدا می‌افتادند — فراخواننده می‌تواند دروغ بگوید و ناهمگامی به‌مرور قطعی است. به‌علاوه، همین Spec جای طبیعی اتصال schema فصل ۳۴ است: پاسخ واقعی شبکه، در مرز parse می‌شود — نوع ایستا از Spec، اثبات runtime از schema، هر دو از یک تعریف.

چرا cancellation با AbortSignal و به‌صورت پارامتر؟ چون لغو، یک «قابلیت» است که فراخواننده باید مالکش باشد: فقط فراخواننده می‌داند چه زمانی نتیجه دیگر به دردش نمی‌خورد (کاربر صفحه را عوض کرد، درخواست جدیدتری جایگزین شد). امضایی که signal اختیاری می‌پذیرد و عبورش می‌دهد، این مالکیت را بدون تحمیل هیچ مدل هم‌زمانی خاصی منتقل می‌کند — همان قراردادی که fetch و اکثر اکوسیستم پذیرفته‌اند؛ client شما با آن ترکیب‌پذیر می‌ماند، نه جزیره.

و خطاها: پاسخ غیر-۲۰۰ «شکست قابل انتظار دامنه» است، پس Result فصل ۳۵، نه throw — فراخواننده باید برای ۴۰۴ و ۴۰۳ «تصمیم» بگیرد و این تصمیم باید در امضا دیده شود. throw را برای نقض قرارداد (پاسخ خارج از schema) نگه دارید: آن دیگر باگ است، نه حالت.

پرسش‌های مصاحبه این فصل

۱. race condition جست‌وجوی زنده را چطور می‌بندید؟

هر درخواست تازه، AbortController قبلی را abort می‌کند؛ fetch با signal لغو می‌شود و پاسخ دیرآمده هرگز state را له نمی‌کند. ظرافت نوعی: لغو، شکست نیست — خروجی `"aborted"` (یا kind مخصوص) جریان عادی است و نباید وارد مسیر خطای واقعی شود.

۲. در event map تایپی، چرا type alias و نه interface؟

interface ای که از `Record<string, unknown>` ارث ببرد index signature رشته‌ای می‌گیرد — یعنی `keyof` آن `string` می‌شود و هر نام رویدادی، حتی typo، پذیرفته می‌شود؛ کل ارزش emitter دود می‌شود. type alias بدون index signature، keyof دقیق نگه می‌دارد و constraint را هم راضی می‌کند.

انتشار پکیج — اعلان عمومی و تست نوع

پروژه ۸: از declaration emit تا جلوگیری از شکستن مصرف‌کننده‌ها

اهداف یادگیری این فصل

- پکیج با d.ts و exports درست منتشر کنی
- تست type-level برای API عمومی بنویسی
- سیاست پایداری API و semver نوع‌ها را تعریف کنی

فصل پایانی، حلقه را می‌بندد: کدی که نوشتید حالا باید **منتشر شود** — با اعلان‌های درست، exports درست و قولی که نسخه‌به‌نسخه نشکند. **پروژه ۸** (بخش دوم): پکیج `typed-emitter` فصل قبل را آماده npm می‌کنیم، با تست‌های type-level که API عمومی را قفل می‌کنند.

آناتومی پکیج مدرن

package.jsonc

```
{
  "name": "@acme/typed-emitter",
  "version": "1.0.0",
  "type": "module",
  "exports": {
    ".": {
      "types": "./dist/index.d.ts",     //  اول types — ۲۸ قاعده فصل
      "default": "./dist/index.js"
    }
  },
  "files": ["dist", "README.md"],
  "scripts": {
    "build": "tsc -p tsconfig.build.json",
    "typecheck": "tsc --noEmit",
    "test": "vitest run",
    "prepublishOnly": "npm run typecheck && npm run test && npm run build"
  },
  "devDependencies": { "typescript": "^7.0.2", "vitest": "^4" }
}
```

tsconfig.build.jsonc

```jsonc
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true,
    "outDir": "dist",
    "rootDir": "src"
  },
  "include": ["src"],
  "exclude": ["src/**/*.test.ts", "src/**/*.test-d.ts"]
}
```

نکته‌هایی که پکیج‌های شکسته را از سالم جدا می‌کنند: `exports` با `types` اول؛ `files` که فقط dist را می‌فرستد (نه سورس و تست)؛ `prepublishOnly` که انتشارِ نچک‌شده را ناممکن می‌کند؛ و اگر CJS هم پشتیبانی می‌کنید، dual publish با `"import"` / `"require"` و دو خروجی — که پیچیدگی واقعی دارد؛ تا مجبور نیستید، ESM-only منتشر کنید (اکوسیستم ۲۰۲۶ کاملاً آماده است).

API عمومی را آگاهانه انتخاب کنید

src/index.ts — type-checks ✓

```ts
// تنها دروازه پکیج — هر چه اینجا نیست، خصوصی است:
export { TypedEmitter } from "./emitter.js";
export type { Listener, EventMapConstraint } from "./emitter.js";
// ابزارهای داخلی عمداً export نمی‌شوند
```

هر export، تعهد نگه‌داری است: حذف یا تغییرش برای مصرف‌کننده‌ها breaking است. دو ابزار کمکی: تگ `/** @internal */` + گزینه `stripInternal` (از d.ts حذف می‌شود)، و ابزارهایی مثل API Extractor یا publint/arethetypeswrong برای ممیزی سطح عمومی و سلامت exports — دومی‌ها را قبل از هر انتشار اجرا کنید؛ نیمی از پکیج‌های npm دقیقاً در همین نقطه خراب‌اند.

تست type-level — قفلِ قرارداد عمومی

رفتار runtime را vitest تست می‌کند؛ اما «API چه نوع‌هایی می‌پذیرد و برمی‌گرداند» هم قرارداد است و رگرسیونش بی‌صدا — مگر تست داشته باشد (فصل ۳۲):

src/index.test-d.ts

type-checks ✓

```ts
import { TypedEmitter } from "./emitter.js";

type Expect<T extends true> = T;
type Equal<A, B> = (<T>() => T extends A ? 1 : 2) extends
  (<T>() => T extends B ? 1 : 2) ? true : false;

type Events = {                          // type، نه interface — فصل ۳۹
  "msg": { text: string };
  "tick": undefined;
};
const bus = new TypedEmitter<Events>();

//  ۱) payload درست استنتاج می‌شود:
bus.on("msg", p => {
  type _T1 = Expect<Equal<typeof p, { text: string }>>;
});

//  ۲) unsubscribe برمی‌گردد:
const off = bus.on("tick", () => {});
type _T2 = Expect<Equal<typeof off, () => void>>;

//  ۳) رویداد ناشناخته رد می‌شود:
// @ts-expect-error — "nope" عضو Events نیست
bus.on("nope", () => {});

//  ۴) payload غلط رد می‌شود:
// @ts-expect-error — text باید string باشد
bus.emit("msg", { text: 42 });

export {};
```

این فایل در `tsc --noEmit` بررسی می‌شود — اگر refactor داخلی، یکی از این قراردادها را عوض کند، CI قرمز می‌شود **قبل از** مصرف‌کننده‌ها. تست ۳ و ۴ (تست‌های منفی با `@ts-expect-error`) را کم نگیرید: «چه چیزی نباید کامپایل شود» نیمی از قرارداد یک API تایپ‌شده است.

semver برای نوع‌ها — چه چیزی breaking است؟

تغییر	semver
گسترش ورودی پذیرفتنی (پارامتر union عضو جدید می‌گیرد)	minor — کد قدیمی کامپایل می‌ماند
محدودتر کردن ورودی، یا گسترش **خروجی** (union خروجی عضو جدید)	**major** — switchهای exhaustive مصرف‌کننده می‌شکنند!
فیلد required جدید در options ورودی	major

تغییر	semver
فیلد جدید در شیء خروجی	معمولاً minor؛ اما با excess check و spread مصرف‌کننده، گاهی دردسر
ارتقای حداقل نسخه TypeScript موردنیاز	major (یا حداقل اعلام صریح)

قاعده ذهنی: **ورودی‌ها contravariant و خروجی‌ها covariant رفتار می‌کنند** (فصل ۲۴!) — گشادتر کردن ورودی امن است، گشادتر کردن خروجی شکننده. برای پکیج‌های جدی، snapshot گرفتن از d.ts تولیدی (`git diff dist/index.d.ts`) در CI یا API Extractor report) آخرین تور ایمنی است — همان‌طور که فصل ۳۳ برای گذار کامپایلر توصیه کرد.

> **مدل ذهنی — پکیج یعنی قرارداد منجمد**
>
> داخل اپلیکیشن، refactor ارزان است: کامپایلر همه مصرف‌کننده‌ها را نشانتان می‌دهد. پشت مرز npm، مصرف‌کننده‌ها **نامرئی‌اند** — تنها نماینده‌شان d.ts منتشرشده و تست‌های type-level شماست. هر چه این نماینده‌ها کامل‌تر، انتشار بی‌ترس‌تر. این همان آخرین پله «مدل را بفهم، رفتار را پیش‌بینی کن، نوع را طراحی کن، پیاده‌سازی را راستی‌آزمایی کن» است — این بار برای کسانی که هرگز نمی‌بینیدشان.

پروژه کامل در `examples/typed-emitter` : سورس، تست runtime، تست نوع، build و دستور publint. معیار پذیرش: هر سه مرحله prepublishOnly سبز؛ `npx publint` بدون خطا؛ و دمو که پکیج build شده را از dist مصرف می‌کند. تمرین توسعه نهایی کتاب: به v1.1 برسانیدش — متد `once` اضافه کنید، تست‌های نوع بنویسید و استدلال کنید چرا minor است نه major.

زیر کاپوت — چرا این‌طور است؟

انتشار یک کتابخانه تایپ‌شده یعنی پذیرفتن این‌که **فایل‌های d.ts شما، «باینریِ» API نوعی‌تان هستند** — همان‌قدر محصول نهایی که فایل‌های js. از این زاویه، سه تصمیم فصل منطق روشنی دارند:

چرا نوع‌ها عضو قرارداد semver اند؟ چون تغییر نوع، build مصرف‌کننده را می‌شکند — دقیقاً همان کاری که تغییر رفتار runtime با برنامه‌اش می‌کند. و جهت شکستن را variance فصل ۲۴ تعیین می‌کند، نه شهود: سخت‌تر کردن «ورودی» یک تابع (پارامتر باریک‌تر) breaking است؛ غنی‌تر کردن «خروجی» معمولاً امن. اضافه‌کردن عضو required به interface ای که مصرف‌کننده «می‌سازد» breaking است (implementation های موجود ناقص می‌شوند)؛ به interface ای که فقط «می‌خواند» امن. یعنی پاسخِ «این تغییر نوعی major است؟» از جدول تولیدکننده/مصرف‌کننده درمی‌آید — یکی از عملی‌ترین کاربردهای variance در کل کتاب. ابزارهایی مثل تست‌های نوع فصل ۳۲ روی نمونه‌های مصرف واقعی، همین قرارداد را در CI نگهبانی می‌کنند.

چرا خروجی را هم ESM می‌دهیم هم CJS، با نگاشت دقیق exports؟ چون مصرف‌کننده‌های شما در هر دو جهان زندگی می‌کنند و فیلد exports (فصل ۲۸) تنها مرز رسمی بسته است — و هر شاخه‌اش (import/require) باید «types مخصوص خودش» را اعلام کند؛ اشتباه رایجِ یک d.ts واحد برای هر دو شاخه، در حالت‌هایی به resolution نادرست می‌انجامد. قاعده ساده: هر entry point، جفتِ کاملِ کد + اعلان خودش.

چرا any در API عمومی ممنوع و unknown مجاز؟ ویروس فصل ۸ این‌جا برون‌مرزی می‌شود: any در امضای عمومی شما، به کد «مشتری» نشت می‌کند و checker او را خاموش می‌کند — آسیبی که از مرز بسته رد شده. unknown همان «نمی‌دانم» را صادقانه می‌گوید و مشتری را به narrowing موظف می‌کند.

و ترفند پایانی برای کیفیت: بسته‌تان را پیش از انتشار، از دید مصرف‌کننده check کنید — پروژه کوچکی که بسته pack شده را نصب و با strict کامل مصرفش می‌کند. آنچه شما در سورس می‌بینید، آن چیزی نیست که مشتری از d.ts می‌گیرد؛ تنها آزمون واقعی، همان مسیر مصرف است.

پرسش‌های مصاحبه این فصل

۱. گشاد کردن union «خروجی» یک پکیج چرا major است ولی گشاد کردن «ورودی» minor؟

منطق variance روی semver: ورودی گشادتر یعنی هر کد قدیمی هنوز قابل قبول است — سازگار. خروجی گشادتر یعنی switch های exhaustive مصرف‌کننده‌ها دیگر کامل نیستند و کامپایلشان می‌شکند. خروجی پکیج شما، «ورودی» کد مشتری است — از چشم او نگاه کنید.

۲. چک‌لیست پیش از publish شما؟

`prepublishOnly` زنجیره typecheck + تست runtime + تست‌های type-level (قفل API عمومی با Equal و ts-expect-error)؛ بعد build + publint و arethetypeswrong برای سلامت exports/types؛ و diff روی d.ts تولیدی نسبت به نسخه قبل — چون پشت مرز npm، مصرف‌کننده‌ها نامرئی‌اند و d.ts تنها نماینده‌شان است.

A

۵۰ پرسش مصاحبه با پاسخ تشریحی

از Junior تا Senior — پاسخ‌هایی که مدل ذهنی را نشان می‌دهند، نه حفظیات را

علاوه بر پرسش‌های پایان هر فصل (دو پرسش تخصصی با پاسخ)، این پیوست ۵۰ پرسش عرضی را سطح‌بندی می‌کند. پاسخ خوب در مصاحبه، تعریف حفظی نیست — نشان دادن مدل ذهنی است: چرا، کی، و با چه هزینه‌ای. پاسخ‌های زیر با همین منطق نوشته شده‌اند و هر یک به فصل مرجع ارجاع می‌دهد. سطح‌بندی: پرسش ۱–۱۸ پایه، ۱۹–۳۶ میانی، ۳۷–۵۰ ارشد.

سطح پایه (۱–۱۸)

۱. TypeScript دقیقاً چه چیزی به JavaScript اضافه می‌کند و چه چیزی اضافه نمی‌کند؟

یک لایه تحلیل ایستا و syntax نوع‌ها — که هنگام کامپایل کاملاً پاک می‌شود (type erasure). هیچ رفتار زمان اجرا اضافه نمی‌کند (استثناهای تاریخی: enum و namespace و parameter properties که کد تولید می‌کنند). پس «بررسی نوع در runtime» از TypeScript برنمی‌آید؛ آن کار schema/validation است. (فصل ۱)

۲. تفاوت interface و type را با معیار «کی کدام» توضیح دهید.

هر دو شکل شیء را بیان می‌کنند؛ تفاوت واقعی: type می‌تواند union/tuple/conditional/mapped باشد و interface می‌تواند merge شود. قاعده عملی: قراردادهای شیئی عمومی ⇒ interface؛ هر نوع ترکیبی/مشتق ⇒ type؛ در تیم یکدست. (فصل ۱۳)

۳. چرا `const x = "a"` نوع `"a"` دارد ولی `let x = "a"` نوع `string` ؟

Widening: برای جایگاه قابل‌تغییر، کامپایلر literal را به نوع عمومی پهن می‌کند چون مقدار می‌تواند عوض شود؛ const تغییر نمی‌کند پس literal می‌ماند. propertyهای شیء حتی در const پهن می‌شوند — راه‌حل‌ها: satisfies، as const، annotation. (فصل ۵)

۴. structural typing یعنی چه و یک پیامد عملی آن؟

سازگاری بر اساس شکل است نه نام: هر مقداری که اعضای لازم را داشته باشد، قابل انتساب است. پیامد: دو alias هم‌شکل (مثل UserId و OrderId روی string) قابل تعویض‌اند — اگر تفکیک واقعی می‌خواهید، branded type لازم است. (فصل ۶ و ۲۳)

۵. excess property check چیست و چرا فقط گاهی فعال است؟

بررسی سخت‌گیرانه‌تر برای object literal تازه که مستقیماً به جایگاه تایپ‌دار داده می‌شود — چون عضو اضافه در literal تازه تقریباً همیشه typo است. از مسیر متغیر واسط عبور می‌کند، پس «جلوگیری از فیلد اضافه» نیست؛ سرویس typo یابی است. (فصل ۶)

۶. تفاوت any و unknown؟ کی unknown انتخاب درست است؟

هر دو هر مقداری را می‌پذیرند؛ اما any بررسی را خاموش می‌کند و ویروسی منتشر می‌شود، در حالی که unknown قبل از هر استفاده اثبات نوع (narrowing) می‌خواهد. هر داده‌ای با منشأ نامعلوم — ورودی JSON، catch، پیام postMessage — unknown است. (فصل ۸)

۷. never کجاها ظاهر می‌شود؟

نوع «مجموعه خالی»: خروجی تابعی که همیشه throw/loop می‌کند، باقیمانده union بعد از پوشش همه اعضا (پایه exhaustive check)، intersection ناسازگار مثل string & number، و توزیع conditional روی union خالی. دیدن never غیرمنتظره = سرنخ دیباگ. (فصل ۸ و ۹)

۸. strictNullChecks چه چیزی را عوض می‌کند؟

بدون آن null/undefined به هر نوعی قابل انتساب‌اند و کامپایلر درباره غیبت مقدار ساکت است. با آن، T و T | null دو نوع متفاوت‌اند و دسترسی بدون بررسی خطاست — بزرگ‌ترین کلاس باگ JS زیر پوشش نوع می‌رود. (فصل ۷)

۹. تفاوت ?? با || ؟

|| برای هر falsy (صفر، رشته خالی، false) سمت راست را برمی‌گرداند؛ ?? فقط برای null/undefined. برای مقدار پیش‌فرض تقریباً همیشه ?? درست است — باگ کلاسیک: port || 3000 که پورت 0 را می‌بلعد. (فصل ۷)

۱۰. narrowing چیست؟ سه سازوکار نام ببرید.

باریک شدن نوع بر اساس بررسی‌های مسیر اجرا (control-flow analysis): typeof برای primitiveها، instanceof و in برای اشیاء، مقایسه با literal برای discriminated unionها؛ به‌علاوه truthiness و Array.isArray. کامپایلر if/return/throw را می‌خواند و نوع را در هر نقطه به‌روز می‌کند. (فصل ۱۰)

۱۱. چرا روی string | number نمی‌توان toUpperCase صدا زد؟

union یعنی «یکی از این‌ها، نمی‌دانم کدام» — فقط اعضای مشترک مجازند. بعد از narrowing (مثلاً typeof)، عملیات اختصاصی آزاد می‌شود. این صداقت نوع است، نه سخت‌گیری. (فصل ۹)

۱۲. tuple با آرایه چه فرقی دارد و کجا به‌کارش می‌برید؟

tuple طول ثابت و نقش موقعیتی دارد: [string, number] یعنی «اولی این، دومی آن». جایی که destructuring معنادار است (خروجی hookها، جفت کلید/مقدار). عناصر هم‌جنس بی‌نقش ⇒ آرایه. (فصل ۱۲)

۱۳. as چه می‌کند و چرا خطرناک است؟

type assertion: به کامپایلر می‌گوید «این را این نوع فرض کن» — بدون هیچ بررسی runtime. خطا را ساکت می‌کند نه حل؛ اگر ادعا غلط باشد، باگ تایپ‌امن به نظر می‌رسد. موارد قابل دفاع: دروازه‌های brand، بعضی مرزهای DOM/Object.keys — همیشه با کامنت چرا. (فصل ۱، ۶، ۲۳)

۱۴. پیام «Object is possibly 'undefined'» را چطور درست رفع می‌کنید؟

به ترتیب ترجیح: narrowing واقعی (if/early return)، ?. وقتی ادامه‌ندادن منطقی است، ?? برای پیش‌فرض، throw صریح با پیام وقتی نبودنش باگ است. ! آخرین گزینه و فقط با اثبات قابل کامنت. (فصل ۷)

۱۵. نقش tsconfig چیست و سه گزینه مهم نام ببرید.

قرارداد سه‌طرفه کد/کامپایلر/محیط اجرا. حیاتی‌ها: strict (چتر بررسی‌ها)، module/moduleResolution (باید آینه runtime واقعی باشد)، target/lib (syntax خروجی و APIهای شناخته‌شده). (فصل ۳ و ۲۸)

۱۶. چرا esbuild/Vite خطای نوع نمی‌گیرند؟

آن‌ها فقط transpile می‌کنند — نوع‌ها را حذف می‌کنند بدون فهمیدن. بررسی نوع کار جداگانه tsc --noEmit (یا سرویس native نسخه ۷) است که باید در CI باشد. «dev server سبز» یعنی هیچ‌چیز درباره نوع‌ها. (فصل ۲)

۱۷. optional property (`a?: T`) با `a: T | undefined` چه فرقی دارد؟

اولی می‌تواند اصلاً غایب باشد؛ دومی باید حاضر باشد ولو با مقدار undefined. تفاوت در in و Object.keys و spread واقعی است؛ گزینه exactOptionalPropertyTypes این تمایز را دقیق enforce می‌کند. (فصل ۲۷)

۱۸. enum یا union of literals؟

پیش‌فرض مدرن: union of literals — erasure کامل، سازگار با JSON و ابزارهای تک‌فایلی. اگر مقادیر runtime هم لازم است: شیء as const + استخراج نوع با keyof typeof. enum باقی‌مانده تاریخی است با چند دام (const enum، رفتار عددی). (فصل ۵ و ۱۶)

سطح میانی (۱۹–۳۶)

۱۹. جنریک چه مشکلی را حل می‌کند که any و کپی‌کاری نمی‌توانند؟

حفظ «رابطه» بین نوع‌ها: خروجی از جنس عنصر ورودی، کلید↔مقدار، callback↔داده. any رابطه را نابود می‌کند، کپی‌کاری nمی‌برابرش می‌کند. قاعده: اگر رابطه‌ای برای حفظ کردن نیست، جنریک لازم نیست. (فصل ۱۴)

۲۰. `T extends HasId` با پارامترِ ساده `HasId` چه فرقی دارد؟

با constraint، نوع «کامل» آرگومان حفظ می‌شود و خروجی جزئیات را نگه می‌دارد؛ با پارامتر ساده، همه‌چیز به HasId فرو می‌ریزد و فیلدهای اضافه از دید نوع گم می‌شوند. constraint یعنی «حداقل این»، نه «دقیقاً این». (فصل ۱۴)

۲۱. keyof و indexed access را با یک مثال عملی توضیح دهید.

`getProp<T, K extends keyof T>(obj: T, key: K): T[K]` — کلید باید عضو واقعی باشد (typo = خطای کامپایل) و خروجی دقیقاً نوع همان کلید. پایه الگوی «منبع حقیقت واحد»: نوع از داده مشتق می‌شود نه موازی با آن. (فصل ۱۶)

۲۲. conditional type توزیعی یعنی چه و چطور جلویش را می‌گیرید؟

وقتی پارامتر نوع برهنه در شرط باشد، union عضوبه‌عضو ارزیابی و نتایج union می‌شوند: ToArray<A|B> = A[]|B[]. برای خاموش کردن: دو طرف را tuple کنید — extends [T] [U]. حالت مرزی: never توزیع می‌شود به هیچ، پس نتیجه never. (فصل ۱۷)

۲۳. infer چه می‌کند؟ ReturnType را بنویسید.

در شاخه شرط، متغیر نوع از ساختار استخراج می‌کند — سطح pattern matching نوع: `type ReturnType<T extends (...a: any) => any> = T extends (...a: any) => infer R ? R : never`.
همین الگو برای عنصر آرایه، مقدار Promise و آرگومان‌ها. (فصل ۱۷)

۲۴. mapped type چیست؟ Partial و Omit را پیاده کنید.

حلقه روی کلیدها: `type Partial<T> = { [K in keyof T]?: T[K] }`؛ `type Omit<T, K> = Pick<T, Exclude<keyof T, K>>`. با as می‌توان کلید را بازنویسی/فیلتر کرد (نگاشت به never = حذف). (فصل ۱۸)

۲۵. satisfies چه مشکلی را حل می‌کند که annotation نمی‌تواند؟

annotation بررسی می‌کند اما نوع را به قرارداد پهن می‌کند و جزئیات literal گم می‌شود؛ satisfies همان بررسی را انجام می‌دهد ولی نوع استنتاجی دقیق را نگه می‌دارد. ایده‌آل برای پیکربندی‌ها و نگاشت‌های exhaustive. (فصل ۲۱)

۲۶. discriminated union چیست و چرا از boolean فlag ها بهتر است؟

union اشیایی با یک فیلد literal مشترک (tag)؛ کامپایلر با بررسی tag، کل عضو را انتخاب می‌کند. برتری: حالت‌های بی‌معنا (loading همراه error) اصلاً قابل ساخت نیستند و exhaustive check در refactor همه نقاط جامانده را قرمز می‌کند. (فصل ۲۲)

۲۷. الگوی exhaustive check با never را بنویسید.

در default سوئیچ: `const x: never = value; throw new Error(...)` یا تابع assertNever. اگر عضوی پوشش داده نشود، انتساب به never خطای کامپایل می‌دهد — switch شما «آینده‌بین» می‌شود. (فصل ۸ و ۲۲)

۲۸. type predicate چیست و خطرش کجاست؟

امضای `x is T` که narrowing را به توابع شما می‌آورد. خطر: کامپایلر بدنه را با ادعا مقایسه نمی‌کند — predicate دروغگو باگ تایپ‌امن‌نما می‌سازد. برای داده بیرونی، schema بهتر از predicate دست‌نویس است. (فصل ۲۳ و ۳۴)

۲۹. branded type چیست و چه وقت می‌ارزد؟

شبیه‌سازی نوع اسمی با intersection یک مهر فقط-نوعی: `string & { [brand]: "UserId" }`. ساخت فقط از دروازه‌ای که validate می‌کند — نتیجه: هر UserId در برنامه، گواهی اعتبار است و Id های هم‌شکل قاطی نمی‌شوند. برای شناسه‌ها و واحدهای حساس می‌ارزد. (فصل ۲۳)

۳۰. «پارامتر تابع contravariant است» یعنی چه؟

تابع جانشین می‌تواند پارامتر «عمومی‌تر» بپذیرد، نه خاص‌تر: void<=(Animal) جانشین void<=(Dog) مجاز است، برعکس نه — چون جانشین باید از پس همه ورودی‌های قرارداد بربیاید. خروجی برعکس: covariant. (فصل ۲۴)

۳۱. چرا آرایه قابل‌نوشتن covariant ناسالم است و دفاع شما چیست؟

Dog[] به Animal[] قابل انتساب است؛ سپس push گربه به «آرایه سگ‌ها» ممکن می‌شود — crash در runtime. تسامح عمدی زبان برای کاربردپذیری است. دفاع: پارامترهای فقط-خواندنی را readonly T[] بگیرید که covariance اش سالم است. (فصل ۲۴)

۳۲. declaration merging و module augmentation کجا به‌کار می‌آیند؟

interfaceهای هم‌نام merge می‌شوند؛ با declare module می‌توان نوع کتابخانه دیگران را گسترش داد: req.user در Express، تایپ process.env. قاعده: هر declare یک ادعاست — جفتش کنید با اثبات runtime (schema روی env). (فصل ۲۵)

۳۳. overload کی ابزار درست است و قواعد حیاتی‌اش؟

وقتی رابطه ورودی→خروجی چند حالت «بی‌قاعده» دارد. قواعد: خاص‌ترین امضا بالاتر (اولین match برنده است)؛ امضای پیاده‌سازی از بیرون نامرئی است و با امضاهای عمومی فقط سازگاری حداقلی دارد — پس تست لازم. برای رابطه قاعده‌مند، جنریک + نگاشت بهتر است. (فصل ۲۵)

۳۴. نوع پاسخ fetch چرا دروغ بالقوه است و راه درست چیست؟

res.json() نوع any دارد و as User فقط ادعاست؛ سرور می‌تواند هر چیزی بفرستد. راه درست: schema validation در مرز (Zod) و استنتاج نوع از z.infer — schema. نوع ایستا برای درون، schema برای مرز. (فصل ۳۴)

۳۵. الگوی Result در برابر exception — چه زمانی کدام؟

شکست قابل انتظار (شبکه، نبود رکورد، validation) جزو قرارداد است و باید در نوع دیده شود ⇒ Result با discriminated union خطاها. باگ و نقض invariant ⇒ exception که بالا برود و گزارش شود. امضای Result مصرف‌کننده را مجبور به تصمیم می‌کند. (فصل ۳۵)

۳۶. در React، props تایپ‌شده یک دکمه سفارشی را چطور طراحی می‌کنید؟

extends از "ComponentPropsWithoutRef<"button"> برای propsهای بومی + فیلدهای خودی با literal union برای variant؛ children: ReactNode؛ rest spread به button. در React 19، ref هم prop معمولی است. (فصل ۳۷)

سطح ارشد (۳۷–۵۰)

۳۷. «state ناممکن را غیرقابل‌ساخت کن» را با مثال معماری توضیح دهید.

به‌جای شیء صاف با optionalهای موازی، هر حالت دامنه یک عضو union با فیلدهای تضمین‌شده‌اش: سفارش shipped حتماً trackingCode و txId دارد. گذارها هم تابع‌های حالت‌مشخص (Extract روی union) — گذار نامعتبر کامپایل نمی‌شود و «نگهبان قواعد» از review به کامپایلر منتقل می‌شود. (فصل ۳۶)

۳۸. هزینه‌های نوع‌های type-level پیچیده را فهرست کنید.

سه هزینه: کارایی کامپایل/ادیتور (instantiationهای انفجاری)، خوانایی پیام خطا برای مصرف‌کننده (چهل خط mapped/conditional بازشده)، و bus factor نگه‌داری. پیچیدگی را جایی خرج کنید که ضریب استفاده بالاست (هسته کتابخانه)، و پشت امضای عمومی ساده پنهانش کنید. (فصل ۲۶)

۳۹. تست type-level چیست و حداقل ابزارش؟

قفل کردن قرارداد نوع‌ها علیه رگرسیون: ترفند Equal دوجنریکه + Expect، و @ts-expect-error برای تست‌های منفی («این نباید کامپایل شود»). در tsc --noEmit اجرا می‌شود؛ برای API عمومی پکیج واجب است. ابزار بالغ‌تر: tsd، expect-type. (فصل ۳۲ و ۴۰)

۴۰. کامپایل کند شده — روش سیستماتیک شما؟

اول اندازه‌گیری: tsc --extendedDiagnostics (شمار Types/Instantiations)، بعد --analyze-trace + generateTrace برای یافتن فایل/نوع مقصر. الگوهای مقصر: union انفجاری، conditional بازگشتی، خروجی‌های عمومی بدون annotation، کتابخانه نوع‌سنگین. راه‌حل ساختاری: project references؛ و از ۲۰۲۶، کامپایلر ۱۰× native. (فصل ۳۲)

۴۱. برنامه مهاجرت یک codebase بزرگ JS به TS؟

فاز صفر: allowJs+checkJs+noEmit و typecheck در CI — ارزش بدون ریسک. سپس ماژول‌به‌ماژول از برگ‌های پرمصرف (utils/models) به بالا؛ نردبان سخت‌گیری گزینه‌به‌گزینه (noImplicitAny → strictNullChecks → strict) و قفل هر پله. سنجه سلامت: ترند any، نه درصد فایل ts. دام اصلی: مهاجرت any-محور. (فصل ۳۰)

۴۲. project references چه می‌دهد و مرز پروژه را کجا می‌برید؟

گراف build رسمی: بازبررسی فقط پروژه‌های اثرپذیرفته، مصرف upstream از d.ts (ارزان‌تر)، و enforcement معماری — import خلاف جهت ممکن نیست. مرز را جایی ببرید که مالکیت/نرخ تغییر عوض می‌شود: دامنه پایدار جدا از اپ پرتغییر. (فصل ۳۱)

۴۳. module: nodenext با bundler چه تفاوت رفتاری دارد؟

nodenext آینه دقیق Node است: اجبار پسوند در import نسبی ESM، احترام کامل به exports، تفکیک cjs/esm از type/پسوند. bundler همان resolution مدرن منهای اجبار پسوند — چون bundler حلش می‌کند. انتخاب بر اساس «اجراکننده نهایی». پیکربندی سخاوتمندتر از runtime واقعی = شکست در deploy. (فصل ۲۸)

۴۴. فیلد exports در package.json چه تغییری در دنیای نوع‌ها داد؟

دروازه رسمی پکیج: فقط مسیرهای اعلام‌شده قابل import اند (کپسوله‌سازی واقعی) و شرط types-first در هر entry، لازمه دیده شدن اعلان‌ها توسط resolverهای مدرن. خطای TS2307 روی پکیج سالم، معمولاً resolution قدیمی مصرف‌کننده است. ابزار ممیزی: arethetypeswrong، publint. (فصل ۲۸ و ۴۰)

۴۵. برای نوع‌های یک پکیج npm، چه چیزی breaking change است؟

با منطق variance: محدودتر کردن ورودی یا گشادتر کردن خروجی (عضو جدید در union خروجی، switchهای مصرف‌کننده می‌شکند) = major؛ گشادتر کردن ورودی = minor؛ فیلد required جدید در ورودی = major. ارتقای حداقل نسخه TS هم اعلام می‌خواهد. تور ایمنی: diff روی d.ts منتشرشده و تست‌های type-level. (فصل ۴۰)

۴۶. وضعیت TypeScript 6/7 را برای تصمیم تیمی خلاصه کنید.

6.0 (مارس ۲۰۲۶) آخرین نسخه JS-based و پل deprecationهاست؛ 7.0 (ژوئن ۲۰۲۶) پورت native با Go، حدود ۱۰× سریع‌تر، رفتار type-checking هم‌ارز 6.0، اما API کامپایلر (Strada) پشتیبانی نمی‌شود. مسیر: اول 6.0 تمیز (بدون ignoreDeprecations دائمی)، بعد 7.0 با تست کل pipeline و diff اعلان‌ها. «زبان» ثابت ماند؛ ابزارها جداگانه باید سازگار باشند. (فصل ۳۳)

۴۷. امنیت: نوع‌ها چه تضمینی نمی‌دهند؟

هیچ تضمین runtime: نوع string روی ورودی کاربر نه XSS را می‌گیرد نه SQLi؛ assertion و augmentation فقط ادعا هستند؛ داده هر مرز (Server Action، postMessage، storage، env، fetch) باید validate شود. نوع‌ها قرارداد «درون» برنامه‌اند؛ امنیت در مرزها و لایه‌های کلاسیک خودش. (فصل ۱، ۳۴، ۳۸)

۴۸. کی annotation نوع بازگشتی اجباری می‌دانید و چرا؟

API عمومی و توابع صادراتی: قرارداد قفل می‌شود، خطا به محل علت می‌رود (نه محل استفاده)، کامپایل سریع‌تر cache پذیرتر، و برای declaration emit تک‌فایلی (isolated declarations) اصلاً لازم. توابع محلی کوچک: استنتاج کافی. (فصل ۴، ۱۱، ۲۹)

۴۹. یک خطای چهل‌خطی جنریک — قدم‌های دیباگ شما؟

عمیق‌ترین سطر تورفته را بخوان (علت واقعی)؛ دو نوع دعوا را مشخص کن و بپرس کدام طرف غلط است (شاید annotation خودم)؛ hover روی پارامترهای جنریک برای دیدن استنتاج واقعی (never/union غیرمنتظره = سرنخ)؛ عبارت را به متغیرهای میانی annotate شده برش بزن تا خطا روی قطعه مقصر بنشیند؛ در نهایت حداقل بازتولید. (فصل ۱۲ و ۳۲)

۵۰. اگر فقط سه قاعده برای یک تیم TypeScript می‌گذاشتید؟

۱) strict همیشه + هر سرکوب (@ts-expect-error ،!، as) فقط با کامنتِ «چرا». ۲) هر داده بیرونی از schema عبور کند و نوعش از schema مشتق شود — درون برنامه هیچ validation تکراری و هیچ state (۳ .any و خطاها با discriminated union مدل شوند و همه switchها exhaustive — تا refactor، کار کامپایلر باشد نه شانس. این سه، ۸۰٪ ارزش کل این کتاب‌اند. (فصل‌های ۲۲، ۲۷، ۳۴، ۳۵)

B

واژه‌نامه فارسی–انگلیسی

اصطلاح‌شناسی یکدست کتاب برای جست‌وجو در مستندات انگلیسی

معادل‌های فارسی این واژه‌نامه همان‌هایی‌اند که در سراسر کتاب به‌کار رفته‌اند. برای بسیاری از اصطلاحات، رویه حرفه‌ای در فارسی استفاده از خود واژه انگلیسی است — در این موارد، ستون «معادل» توضیح مفهومی است نه جایگزین رایج. ارجاع فصل، محل تعریف اصلی است.

مفاهیم بنیادی

اصطلاح	معادل / توضیح	فصل
static type checking	بررسی ایستای نوع — تحلیل پیش از اجرا	۱
type erasure	پاک‌شدن نوع‌ها در خروجی کامپایل	۱
type annotation	حاشیه‌نویسی نوع (اعلان صریح)	۴
type inference	استنتاج نوع	۴
contextual typing	نوع‌دهی از زمینه (جهت معکوس استنتاج)	۴
structural typing	نوع‌دهی ساختاری — سازگاری بر پایه شکل	۶
nominal typing	نوع‌دهی اسمی — سازگاری بر پایه نام	۶، ۲۳
assignability	قابلیت انتساب	۶
excess property check	بررسی ویژگی اضافه در literal تازه	۶
widening / narrowing	پهن‌شدن / باریک‌شدن نوع	۵، ۱۰
control-flow analysis	تحلیل جریان کنترل	۱۰
literal type	نوع literal (مقدار-به‌مثابه-نوع)	۵
union / intersection	اجتماع / اشتراک نوع‌ها	۹
discriminated union	union تفکیک‌شده با فیلد tag	۲۲
exhaustiveness check	بررسی جامعیت شاخه‌ها (با never)	۸، ۲۲
top / bottom type	نوع فراگیر (unknown) / نوع تهی (never)	۸
type assertion	ادعای نوع (as) — بدون بررسی اجرا	۶
non-null assertion	ادعای غیرتهی‌بودن (!)	۷
optional chaining / nullish coalescing	زنجیره اختیاری (?.) / ادغام تهی (??)	۷

نوع‌های پیشرفته

اصطلاح	معادل / توضیح	فصل
generic / type parameter	جنریک / پارامتر نوع	۱۴
generic constraint	قید جنریک (extends)	۱۴
indexed access type	نوع دسترسی اندیسی T[K]	۱۶
keyof operator	عملگر استخراج کلیدها	۱۶
conditional type	نوع شرطی	۱۷
distributive conditional	شرطیِ توزیعی روی union	۱۷
infer	استخراج نوع از ساختار در شاخه شرط	۱۷
mapped type	نوع نگاشتی (حلقه روی کلیدها)	۱۸
key remapping	بازنویسی کلید با as در mapped type	۱۸
template literal type	نوع رشته قالبی سطح نوع	۱۹
utility types	نوع‌های ابزاری استاندارد (Partial، Pick، …)	۲۰
const assertion	تثبیت literal با as const	۵، ۲۱
satisfies	بررسی قرارداد بدون پهن‌کردن نوع	۲۱
type predicate	گزاره نوع (x is T)	۲۳
assertion function	تابع اثباتی (asserts x is T)	۲۳
branded / opaque type	نوع مهردار — تفکیک اسمی روی ساختار	۲۳
variance (co/contra/in)	وردایی: هم‌سو / پادسو / ناوردا	۲۴
declaration merging	ادغام اعلان‌های هم‌نام	۲۵
module augmentation	گسترش نوع‌های یک ماژول دیگر	۲۵
overload	چندامضایی تابع	۲۵
recursive type	نوع بازگشتی	۲۶

کامپایلر و ابزار

اصطلاح	معادل / توضیح	فصل
transpile	ترجمه syntax بدون بررسی نوع	۲
type-check (tsc --noEmit)	بررسی نوع بدون تولید خروجی	۲

اصطلاح	معادل / توضیح	فصل
strict mode family	خانواده گزینه‌های سخت‌گیرانه	۲۷
module resolution	سازوکار یافتن ماژول‌ها	۲۸
declaration file (.d.ts)	پرونده اعلان — نوع بدون پیاده‌سازی	۲۹
ambient declaration	اعلان محیطی (declare)	۲۹
isolated declarations	تولید اعلان تک‌فایلی با امضاهای صریح	۲۹
project references	ارجاع بین پروژه‌ها — گراف build	۳۱
incremental build	ساخت افزایشی (.tsbuildinfo)	۳۱
type-level test	تست سطح نوع (Equal/Expect)	۳۲
language service	سرویس زبان (هوش ادیتور)	۲، ۳۳

معماری و اجرا

اصطلاح	معادل / توضیح	فصل
runtime validation	اعتبارسنجی زمان اجرا	۳۴
schema / parse, don't validate	طرح‌واره / «تجزیه کن، فقط وارسی نکن»	۳۴
single source of truth	منبع حقیقت واحد (نوع مشتق از schema)	۱۶، ۳۴
trust boundary	مرز اعتماد	۳۴، ۳۸
Result type	نوع نتیجه — خطای قابل انتظار در امضا	۳۵
invariant	قاعده همیشه-برقرار دامنه	۳۶
state machine	ماشین حالت	۳۶
make illegal states unrepresentable	«حالت ناممکن را غیرقابل‌ساخت کن»	۲۲، ۳۶
DTO (data transfer object)	شیء انتقال داده — مدل مخصوص مرز	۳۶، ۳۸
serialization boundary	مرز serialize (سرور/کلاینت)	۳۸
cancellation (AbortSignal)	لغو عملیات ناهمگام	۳۹
semantic versioning (semver)	نسخه‌بندی معنایی	۴۰
breaking change	تغییر شکننده قرارداد	۴۰

C

مرجع تصمیم گزینه‌های کامپایلر

هر گزینه: چه می‌گیرد، چه هزینه‌ای دارد، چه زمانی روشن شود

این پیوست مرجع تصمیم است، نه فهرست کامل گزینه‌ها: برای هر گزینه مهم — مقدار توصیه‌شده، چرا، و کی استثنا قائل شوید. مبنا: TypeScript 7.0 (رفتار گزینه‌ها با 6.0 هم‌ارز است — فصل ۳۳). نقطه شروع پیشنهادی کتاب، همان tsconfig فصل ۳ است.

پایه — برای همه پروژه‌ها

گزینه	توصیه	چرا / استثنا
`strict`	`true` ، همیشه	چتر همه بررسی‌های جدی؛ پروژه جدید بدون آن شروع نکنید. استثنا: فقط میانه مهاجرت (فصل ۳۰) و موقت.
`noUncheckedIndexedAccess`	`true`	`arr[i]` و `rec[key]` را صادقانه `T | undefined` می‌کند؛ خارج از strict است و باید جدا روشن شود. استثنا: کدِ پرحلقه‌ی index محور قدیمی — ولی هدف نهایی روشن بودنش است.
`noImplicitOverride`	`true`	جلوی override تصادفی/یتیم در کلاس‌ها. تقریباً بدون هزینه.
`exactOptionalPropertyTypes`	پروژه جدید `true`	تمایز «غایب» از «undefined صریح» (فصل ۲۷). روی codebase موجود نویز مهاجرتی دارد.
`verbatimModuleSyntax`	`true`	import نوع‌ها باید `import type` باشد؛ لازمه ابزارهای تک‌فایلی (esbuild/swc) و حذف import ارواح.
`skipLibCheck`	`true`	بررسی d.ts کل node_modules گران و پر از دعواهای بین‌کتابخانه‌ای است. هزینه پنهان: خطای واقعی اعلان‌ها هم رد می‌شود — جبران با تست نوع خودتان (فصل ۳۲).
`forceConsistentCasingInFileNames`	`true` (پیش‌فرض نسخه‌های جدید)	باگ «روی مک کامپایل شد، روی لینوکس نه».

ماژول و خروجی — بر اساس «اجراکننده نهایی» (فصل ۲۸)

سناریو	گزینه‌ها
کتابخانه/سرویس Node	`module: "nodenext"` + `target: "es2023"` + پسوند در importهای نسبی
اپ با bundler (Vite/Next/esbuild)	`module: "esnext"` ؛ `moduleResolution: "bundler"` ؛ target بر عهده bundler، tsc فقط typecheck

سناریو	گزینه‌ها
پکیج npm منتشرشدنی	مثل Node + `declaration: true` + `declarationMap: true` + `outDir` / `rootDir`؛ tsconfig.build جدا که تست‌ها را exclude می‌کند (فصل ۴۰)
فقط بررسی، بدون خروجی	`noEmit: true` — الگوی استاندارد CI کنار bundler

قاعده طلایی: tsconfig باید **آینه runtime واقعی** باشد. سخاوتمندتر از واقعیت (bundler-style) برای کدی که Node خام اجرایش می‌کند) = سبز محلی، شکست در deploy.

lib و jsx

گزینه	توصیه
`lib`	Node خالص: `["es2023"]` ؛ مرورگر: `["es2023", "dom", "dom.iterable"]` . هر API ای که lib اعلامش نکند «وجود ندارد» — خطای TS2550 روی متد جدید یعنی lib عقب است، نه TypeScript خراب.
`jsx`	`"react-jsx"` برای React 17+ (بدون import React)؛ `"preserve"` وقتی فریمورک خودش JSX را تبدیل می‌کند (Next).
`types`	پیش‌فرض (همه @types) معمولاً درست است؛ محدودکردنش فقط وقتی تایپ‌های global ناخواسته (مثلاً jest در کد production) نشت می‌کنند.

کارایی و مقیاس (فصل ۳۱–۳۲)

گزینه	توصیه
`incremental`	`true` + `tsBuildInfoFile` مشخص؛ cache در CI کلیددار با نسخه TS.
`composite`	فقط برای project references؛ declaration را هم اجباری می‌کند.
`isolatedModules`	`true` — تضمین سازگاری با transpile تک‌فایلی (زیرمجموعه الزامات verbatimModuleSyntax).
`isolatedDeclarations`	برای پکیج‌های با build موازی؛ هزینه: annotation صریح همه صادرات (که خودش فضیلت است — فصل ۲۹).

گزینه‌هایی که باید درباره‌شان «نه» بگویید

- `suppressImplicitAnyIndexErrors` ، `keyofStringsOnly` و خویشاوندان: deprecated و در 7.0 حذف شده‌اند (فصل ۳۳).
- `experimentalDecorators` برای کد جدید: دکوراتور استاندارد ES را بدون این پرچم بنویسید؛ فقط فریمورک‌های وابسته (NestJS/Angular قدیمی) آن را موجه می‌کنند.
- `baseUrl` برای alias: به‌جایش `paths` نسبی + تنظیمات معادل در bundler؛ و یادتان باشد paths فقط به tsc آدرس می‌دهد، runtime را شما باید هم‌آهنگ کنید.

- خاموش کردن نقطه‌ای strict (`strictNullChecks: false` زیر strict): چراغ عقب‌گرد؛ اگر لازم شد، مشکل معماری دارید نه پیکربندی.

در عمل — سه پروفایل آماده

در ریشه مخزنِ همراه کتاب، سه tsconfig آماده مطابق همین جدول‌ها هست: `examples/` (اپ bundler محور)، `examples/api-client` (کتابخانه‌وار) و `examples/typed-emitter` (پکیج قابل انتشار با build جدا). کپی‌برداری از پروفایل نزدیک‌تر، سریع‌تر از سرهم‌بندی گزینه‌به‌گزینه است — بعد فقط «اجراکننده نهایی» را بازبینی کنید.

D

راهنمای عیب‌یابی سیستم نوع

از علامت خطا تا علت ریشه‌ای — جدول مراجعه سریع

فصل ۱۲ الگوریتم چهارمرحله‌ای خواندن خطا را داد: ۱) کد خطا را جدی بگیر، ۲) عمیق‌ترین سطر تورفته را بخوان، ۳) دو نوعِ درگیر را مشخص کن، ۴) بپرس کدام طرف غلط است. این پیوست جدول مرجع آن الگوریتم است: پرتکرارترین کدهای خطا، معنای واقعی (نه ترجمه تحت‌اللفظی)، و مسیر رفع درست.

جدول کدهای پرتکرار

کد	معنای واقعی	رفع درست (و رفع غلط رایج)
TS2322	«این مقدار به آن جایگاه قابل انتساب نیست» — عمومی‌ترین خطا	دو نوع را مقایسه کن؛ شاید annotation خودت غلط است. رفع غلط: as. (فصل ۶)
TS2345	همان 2322 برای آرگومان تابع	نوع واقعی آرگومان را hover کن؛ در جنریک‌ها اول استنتاج T را ببین. (فصل ۱۴)
TS2339	«این عضو روی این نوع وجود ندارد»	اگر union است: narrowing لازم است نه as. اگر DOM/API جدید: شاید lib عقب است. (فصل ۹، ۱۰)
TS2531/2532/18047/18048	«احتمالاً null/undefined»	نردبان فصل ۷: throw → ?? → ?. → narrowing صریح؛ ! فقط با اثبات.
TS7006	پارامتر، implicit any شد	annotation بده؛ در callback، اگر زمینه نیست یعنی جایگاه تایپ‌دار نیست — نوع تابع را تعریف کن. (فصل ۴، ۱۱)
TS7016	ماژول JS بدون declaration	نردبان فصل ۲۹: @declaration → types/x محلی حداقلی → آخرین چاره declare module با any.
TS7053	index با رشته عمومی روی شیء بدون index signature	اگر کلیدها محدودند: keyof + narrowing (typedKeys فصل ۱۶)؛ اگر واقعاً باز است: Record مناسب.
TS2307	ماژول پیدا نشد	اول typo/نصب؛ بعد moduleResolution — پکیج exports دار با resolution قدیمی دیده نمی‌شود. (فصل ۲۸)
TS2305/2614	ماژول هست، عضو صادرشده نیست	named در برابر default import را چک کن؛ در پکیج دوگانه cjs/esm، شاید نسخه اشتباه resolve شده. (فصل ۲۸)
TS2352	«این as حتی به‌عنوان ادعا هم بعید است»	کامپایلر دارد از assertion بدتر نجاتت می‌دهد؛ مسیر unknown فقط وقتی واقعاً مطمئنی. (فصل ۶)
TS2367	مقایسه‌ای که هیچ‌وقت true نمی‌شود	معمولاً باگ واقعی: مقایسه literal های ناهم‌پوشان یا فراموشی await.

کد	معنای واقعی	رفع درست (و رفع غلط رایج)
TS2589	instantiation بی‌نهایت/خیلی عمیق	نوع بازگشتی بدون شرط توقف یا ورودی عمیق؛ شمارنده عمق یا ساده‌سازی مدل. (فصل ۲۶)
TS2769	هیچ overload ای match نشد	از بالا به پایین امضاها را با آرگومان‌هایت مقایسه کن؛ خطای واقعی معمولاً در «نزدیک‌ترین» overload گزارش می‌شود. (فصل ۲۵)
TS2742	«نوع استنتاجی قابل نام‌گذاری نیست بدون ارجاع به …»	خروجی صادراتی را صریح annotate کن — همان قاعده فصل ۱۱/۲۹؛ معمولاً در monorepo با نسخه‌های تکراری پکیج.
TS4111	دسترسی به env فقط با index	process.env.X با noPropertyAccessFromIndexSignature؛ بهتر: env تایپ‌شده مرکزی (فصل ۳۴).
TS1484	این import باید import type باشد	verbatimModuleSyntax فعال است و درست می‌گوید؛ type را جدا import کن. (فصل ۲۸)
TS2554	تعداد آرگومان غلط	اگر بعد از refactor فوران کرد: خبر خوب است — کامپایلر دارد همه call site ها را برایت پیدا می‌کند.

نشانه‌شناسی — وقتی خطا «عجیب» است

- **نوع `never` غیرمنتظره** در hover: دو narrowing متوالی ناسازگار، exhaustive check ای که زودتر کامل شده، یا آرایه خالی بدون annotation (فصل ۴). never را «بن‌بست منطقی» بخوان و عقب‌گرد کن ببین کجا مجموعه خالی شد.
- **union غول‌آسا در پیام**: معمولاً template literal یا mapped روی union بزرگ منفجر شده — پیام را نخوان، منبع union را پیدا کن (فصل ۲۶).
- **خطا در فایلی که دست نزده‌ای**: نوع صادراتی تغییر کرده و مصرف‌کننده دوردست شکسته. `tsc --noEmit` محلی قبل از push، این سورپرایز را از CI به ادیتور می‌آورد.
- **در ادیتور سبز، در CI قرمز (یا برعکس)**: دو نسخه TypeScript. ادیتور را به نسخه workspace سوییچ کن؛ نسخه را در package.json قفل کن (فصل ۳۳).
- **خطا فقط بعد از پاک‌کردن node_modules**: cache یا skipLibCheck تفاوت را پنهان می‌کرد؛ lockfile و نسخه‌های @types را بررسی کن.

پروتکل شکار باگ نوعی (جمع‌بندی فصل ۱۲ + ۳۲)

۱) کمینه‌سازی: عبارت پیچیده را به متغیرهای میانی با annotation صریح برش بزن — خطا روی قطعه مقصر می‌نشیند. ۲) حقیقت‌سنجی: hover روی هر میانی؛ اولین جایی که نوع با انتظارت فرق دارد، صحنه جرم است. ۳) جهت‌سنجی: بپرس «نوع غلط است یا انتظار من؟» — نیمی از خطاها annotation خوش‌بینانه خود شماست. ۴) اگر هنوز گیر است: بازتولید حداقلی در یک فایل تمیز؛ اگر آن‌جا درست کار کرد، تفاوت در tsconfig یا اعلان‌های محیطی پروژه است.

E

چک‌لیست مهاجرت از JavaScript

مرحله‌به‌مرحله، قابل برگشت، با معیار پایان هر مرحله

نسخه عملیاتی فصل ۳۰ به شکل چک‌لیست قابل چاپ. هر فاز را کامل کنید، قفل کنید (CI)، بعد فاز بعد. سنجه پیشرفت: **ترند تعداد any و خطاهای سرکوب‌شده**، نه درصد فایل‌های ts.

فاز ۰ — ارزش بدون تغییر کد (نیم‌روز تا دو روز)

- [ ] tsconfig با `allowJs: true` ، `checkJs: true` ، `noEmit: true` ، strict خاموش — فقط دید.
- [ ] `tsc --noEmit` در CI، فعلاً به‌صورت گزارش (non-blocking)؛ شمار خطاها را ثبت کنید — خط مبنا.
- [ ] نصب @types برای وابستگی‌های اصلی (react، express، node، …).
- [ ] خطاهای «مفت» را درو کنید: typo ها، آرگومان جاافتاده، null های آشکار — همین فاز معمولاً چند باگ واقعی production می‌گیرد.
- [ ] JSDoc روی ۵–۱۰ ماژول پرمصرف (`@param` / `@returns` / `@type`) — تایپ بدون تغییر پسوند.

فاز ۱ — زیرساخت و قواعد تیم (قبل از اولین rename)

- [ ] نسخه TypeScript قفل‌شده در package.json (فصل ۳۳)؛ ادیتور همه روی نسخه workspace.
- [ ] قاعده مکتوب: فایل جدید فقط .ts/.tsx؛ فایل لمس‌شده در PR، کاندید مهاجرت.
- [ ] قاعده سرکوب: فقط `@ts-expect-error` با کامنت علت + شناسه issue؛ `@ts-ignore` ممنوع.
- [ ] ابزار سنجش: `type-coverage` یا شمارش ساده any در CI؛ ترند در داشبورد تیم.
- [ ] ترتیب مهاجرت از گراف وابستگی: برگ‌های پرمصرف اول (constants، models، utils)، بعد سرویس‌ها، آخر ورودی‌های اپ.

فاز ۲ — مهاجرت ماژول‌به‌ماژول (هسته کار)

برای هر ماژول، همین ترتیب:

- [ ] rename به .ts؛ خطاها را **در همین ماژول** صفر کنید — نه با any، با نوع واقعی یا TODO تایپ‌دار (`unknown` + issue).
- [ ] قراردادهای صادراتی را صریح annotate کنید (فصل ۱۱) — مصرف‌کننده‌های JS هنوز از آن‌ها سود می‌برند.
- [ ] داده بیرونی این ماژول (fetch/env/storage) دارد؟ همین حالا schema (فصل ۳۴) — مهاجرت بهترین بهانه برای بستن مرزهاست.
- [ ] تست‌های موجود سبز؛ commit کوچک، نه big-bang.
- [ ] هر ۱۰–۱۵ ماژول: بازبینی ترند any و زمان typecheck.

فاز ۳ — نردبان سخت‌گیری (گزینه‌به‌گزینه، قفل پس از هر پله)

- [ ] `noImplicitAny: true` — دردناک‌ترین و پرسودترین پله.
- [ ] `strictNullChecks: true` — با نردبان فصل ۷، نه با پاشیدن `!`.
- [ ] `strict: true` کامل.
- [ ] `noUncheckedIndexedAccess: true` + `noImplicitOverride: true`.
- [ ] اگر مخزن بزرگ است: per-directory tsconfig — دایرکتوری‌های مهاجرت‌کرده strict، بقیه هنوز نه (فصل ۳۰).

فاز ۴ — تثبیت

- [ ] `tsc --noEmit` در CI اکنون **blocking**.
- [ ] حذف allowJs وقتی آخرین js رفت؛ حذف همه سرکوب‌های TODO دار (issue ها بسته شوند).
- [ ] فایل‌های d.ts موقتی که برای JS داخلی نوشتید، با مهاجرت خود ماژول حذف شوند (فصل ۲۹).
- [ ] قراردادهای مرزی تیم (فصل ۲۷): جدول گزینه‌های نهایی + قواعد as/any در CONTRIBUTING.
- [ ] جشن اندازه‌گیری‌شده: خط مبنای فاز ۰ در برابر امروز — تعداد باگ نوعی مرده را به مدیریت نشان دهید.

سه دام که مهاجرت‌ها را می‌کشند

۱) **مهاجرت any-محور**: rename همه فایل‌ها + any برای ساکت کردن — عددِ «۱۰۰٪ TypeScript» می‌دهد و هیچ ایمنی؛ بدتر از JS، چون توهم پوشش می‌سازد. ۲) **مهاجرت جدا و طولانی branch**: merge جهنمی؛ مهاجرت باید روی main و قطره‌قطره باشد. ۳) **refactor هم‌زمان**: «حالا که دستم اینجاست بازنویسی‌اش کنم» — دو ریسک در یک commit؛ اول مهاجرت هم‌رفتار، بعد refactor با پشتیبانی کامپایلر.

F

الگوهای طراحی نوع قابل‌استفاده مجدد

دوازده الگوی محک‌خورده با کد کامل و موارد استفاده

دوازده الگوی نوعی که در کتاب ساخته شدند، این‌جا به شکل «کارت مرجع» جمع‌اند: مسئله ← الگو ← کد حداقلی. همه snippet ها خودبسنده‌اند و با strict کامپایل می‌شوند؛ نسخه با توضیح کامل در فصل ارجاعی.

۱. منبع حقیقت واحد — نوع از داده (فصل ۱۶، ۲۱)

p01-source-of-truth.ts — type-checks ✓

```ts
const ROLES = ["admin", "editor", "viewer"] as const;
type Role = (typeof ROLES)[number];          // "admin" | "editor" | "viewer"

const PERMS = { admin: 7, editor: 3, viewer: 1 } as const satisfies Record<Role, number>;
type Perm = (typeof PERMS)[Role];
```

۲. ماشین حالت با discriminated union (فصل ۲۲، ۳۶)

p02-state.ts — type-checks ✓

```ts
type State =
  | { status: "idle" }
  | { status: "loading" }
  | { status: "done"; data: string }
  | { status: "failed"; error: string };

function assertNever(x: never): never { throw new Error(String(x)); }
function label(s: State): string {
  switch (s.status) {
    case "idle": return "آماده";
    case "loading": return "…";
    case "done": return s.data;
    case "failed": return s.error;
    default: return assertNever(s);
  }
}
```

۳. Result برای خطای قابل انتظار (فصل ۳۵)

p03-result.ts — type-checks ✓

```ts
type Result<T, E> = { ok: true; value: T } | { ok: false; error: E };
const ok = <T,>(value: T): Result<T, never> => ({ ok: true, value });
const err = <E,>(error: E): Result<never, E> => ({ ok: false, error });

function parsePort(s: string): Result<number, string> {
  const n = Number(s);
  return Number.isInteger(n) && n > 0 && n < 65536 ? ok(n) : err(`پورت نامعتبر: ${s}`);
}
```

۴. Branded type — تفکیک اسمی (فصل ۲۳)

p04-brand.ts — type-checks ✓

```ts
declare const brand: unique symbol;
type Brand<T, B extends string> = T & { readonly [brand]: B };

type UserId = Brand<string, "UserId">;
function asUserId(raw: string): UserId {
  if (!/^u_[a-z0-9]{8,}$/.test(raw)) throw new Error("شناسه نامعتبر");
  return raw as UserId;        // مجاز: داخل دروازه تنها as
}
```

۵. دروازه parse, don't validate — schema (فصل ۳۴)

p05-schema.ts — type-checks ✓

```ts
import { z } from "zod";

const User = z.object({ id: z.string(), email: z.email() });
type User = z.infer<typeof User>;            // نوع، مشتق — نه موازی

function fromJson(raw: unknown): User { return User.parse(raw); }
```

۶. Builder/factory با استنتاج در نقطه مصرف (فصل ۱۵)

p06-factory.ts — type-checks ✓

```ts
function defineConfig<T extends Record<string, unknown>>(cfg: T): Readonly<T> {
  return Object.freeze(cfg);
}
const config = defineConfig({ port: 3000, host: "0.0.0.0" });
// config.port: number — بدون annotation جنریک نامرئی، مصرف‌کننده
```

۷. نگاشت رویداد→payload (فصل ۳۹)

p07-eventmap.ts — type-checks ✓

```ts
interface Events { save: { id: string }; reset: undefined }

function on<K extends keyof Events>(event: K, fn: (p: Events[K]) => void): void {
  void event; void fn;
}
on("save", p => p.id);        // p استنتاج شد؛ خطای کامپایل "svae"
```

۸. RequireAtLeastOne / گزینه‌های وابسته (فصل ۲۰)

p08-atleastone.ts — type-checks ✓

```ts
type RequireAtLeastOne<T, K extends keyof T = keyof T> =
  K extends keyof T ? Omit<T, K> & Required<Pick<T, K>> : never;

interface Notify { email?: string; sms?: string }
declare function send(target: RequireAtLeastOne<Notify>): void;
send({ email: "a@b.c" });     //  ✓ — ولی {} رد می‌شود
```

۹. PickByType — فیلتر عضوها بر اساس نوع مقدار (فصل ۱۸)

p09-pickbytype.ts — type-checks ✓

```ts
type PickByType<T, V> = { [K in keyof T as T[K] extends V ? K : never]: T[K] };

interface Row { id: number; name: string; active: boolean; score: number }
type Sortable = keyof PickByType<Row, string | number>;  // "id" | "name" | "score"
```

۱۰. تبدیل سطح‌نوع رشته‌ها — کلیدهای مسیر (فصل ۱۹)

p10-pathparams.ts — type-checks ✓

```ts
type PathParams<S extends string> =
  S extends `${string}:${infer P}/${infer Rest}` ? P | PathParams<`/${Rest}`> :
  S extends `${string}:${infer P}` ? P : never;

type P = PathParams<"/users/:userId/posts/:postId">;  // "userId" | "postId"
```

۱۱. DeepReadonly بازگشتی (فصل ۲۶)

p11-deepreadonly.ts — type-checks ✓

```ts
type DeepReadonly<T> = T extends (...a: never[]) => unknown ? T
  : T extends readonly unknown[] ? { readonly [K in keyof T]: DeepReadonly<T[K]> }
  : T extends object ? { readonly [K in keyof T]: DeepReadonly<T[K]> }
  : T;

type Frozen = DeepReadonly<{ a: { b: number[] } }>;
```

۱۲. قفل قرارداد با تست نوع (فصل ۳۲، ۴۰)

p12-typetest.ts — type-checks ✓

```ts
type Expect<T extends true> = T;
type Equal<A, B> = (<T>() => T extends A ? 1 : 2) extends
  (<T>() => T extends B ? 1 : 2) ? true : false;

type _T1 = Expect<Equal<ReturnType<() => string>, string>>;
// @ts-expect-error — رگرسیون عمدی باید قرمز بماند
type _T2 = Expect<Equal<string, number>>;
export {};
```

در عمل — ترتیب انتخاب الگو

وقتی مسئله نوعی جلویتان است، به این ترتیب بپرسید: آیا داده موجود می‌تواند منبع نوع شود؟ (الگوی ۱) آیا حالت‌ها union تفکیک‌شده‌اند؟ (۲) آیا شکست قابل انتظار است؟ (۳) آیا مرز بیرونی است؟ (۵) آیا تفکیک اسمی لازم است؟ (۴) — و فقط اگر هیچ‌کدام، سراغ mapped/conditional سفارشی (۸–۱۱) بروید؛ و هر چه ساختید، با الگوی ۱۲ قفل کنید.

G

نقشه راه نهایی

از توسعه‌دهنده JavaScript تا مهندس TypeScript در سطح Production

این پیوست، کتاب را به برنامه عمل تبدیل می‌کند: سه مسیر خواندن/تمرین بر اساس نقطه شروع شما، و پس از آن نقشه نگه‌داشتن دانش در اکوسیستمی که هر سال عوض می‌شود.

مسیر ۱ — از JavaScript می‌آیید (۴–۶ هفته، روزی ~۱ ساعت)

هفته	تمرکز	خروجی قابل لمس
۱	فصل ۱–۷: مدل ذهنی، ابزار، استنتاج، null safety	یک utility کوچک JS خودتان را به TS ببرید؛ strict از روز اول
۲	فصل ۸–۱۳: union/narrowing/خواندن خطا	ماژول «پیام‌رسانی» با union تفکیک‌شده + exhaustive switch
۳	فصل ۱۴–۱۶ + ۲۰–۲۲: جنریک‌های کاربردی، utility types، state	پروژه ۳ (ماشین حالت سفارش) را از صفر بازسازی کنید
۴	فصل ۲۷–۲۸ + ۳۴–۳۵: پیکربندی، مرزها، خطا	پروژه ۱ و ۲: boundary با Zod + کلاینت Result
۵–۶	فصل ۳۷–۳۹ بسته به استک + پیوست ها	فیچر React/Next تایپ‌شده سرتاسر؛ پیوست A را شفاهی جواب بدهید

فصل‌های ۱۷–۱۹ و ۲۶ (نوع‌های سطح بالا) را در این مسیر رد شوید و بعداً برگردید — بدون آن‌ها هم مهندس TypeScript مولدی هستید.

مسیر ۲ — TypeScript کار کرده‌اید، عمق می‌خواهید (۲–۳ هفته)

فصل ۴ (استنتاج دوجهته)، ۶ (ساختاری/انتساب)، ۱۰ (control flow) را حتی اگر «بلدید» بخوانید — بیشترِ شهودهای غلط همین‌جاها لانه دارند. سپس: ۱۷–۱۹ و ۲۶ (type-level کامل + مرز پیچیدگی)، ۲۳–۲۵ (predicate/brand/variance)، ۳۲ (augmentation). تست نوع و کارایی — احتمالاً غایب اصلی دانش شما)، و پروژه‌های ۴–۸. آزمون نهایی: پرسش‌های ۳۷–۵۰ پیوست A بدون نگاه به پاسخ.

مسیر ۳ — تیم‌لید/معمار (گلچین معماری)

فصل ۲۲ و ۳۶ (مدل‌سازی دامنه) → ۲۷ (قرارداد سخت‌گیری تیم) → ۳۰ (مهاجرت) → ۳۱ (مرزبندی monorepo) → ۳۳ (برنامه گذار 6/7) → ۴۰ (انتشار پکیج). خروجی‌های مکتوب برای تیمتان: جدول گزینه‌های tsconfig (پیوست C را شخصی‌سازی کنید)، قاعده سه‌گانه پرسش ۵۰ پیوست A، و چک‌لیست مهاجرت (پیوست E) اگر JS قدیمی دارید.

به‌روز ماندن — چرخه سالانه

- **منابع درجه‌یک** (پیوست H): وبلاگ رسمی TypeScript برای هر نسخه minor؛ بخش breaking changes و release notes را همیشه، بقیه را گزینشی.
- **قاعده نسخه**: در پروژه‌ها یک minor عقب‌تر از آخرین نسخه بمانید مگر قابلیت مشخصی لازمتان باشد؛ ارتقا را با پروتکل فصل ۳۳ (branch + diff اعلان‌ها + تست کامل pipeline) انجام دهید.
- **تمرین عضله type-level**: چالش‌های type-challenges (مخزن معروف GitHub) — هفته‌ای یکی کافی است؛ بیشترش، تمرین پیچیدگی بی‌مشتری است (هشدار فصل ۲۶).
- **خواندن کد دیگران**: اعلان‌های d.ts کتابخانه‌هایی که استفاده می‌کنید را گاهی باز کنید (zod، React types) — بهترین کلاس الگوهای API تایپ‌شده واقعی.

مدل ذهنی — پایان کتاب، شروع چرخه

چهار حرکتی که در تک‌تک فصل‌ها تکرار شد، دیگر باید رفلکس شده باشد: **مدل را بفهم** (داده واقعاً چه شکل‌هایی دارد؟)، **رفتار را پیش‌بینی کن** (کامپایلر این‌جا چه استنتاج می‌کند؟)، **نوع را طراحی کن** (کدام حالت‌ها نباید قابل بیان باشند؟)، **راستی‌آزمایی کن** (schema در مرز، تست نوع در قرارداد). ابزارها نسخه عوض می‌کنند — کامپایلر Go شد و باز هم خواهد شد — اما این چهار حرکت همان چیزی است که با شما می‌ماند. کتاب تمام شد؛ کامپایلر منتظر است.

H

منابع و مآخذ

مستندات رسمی و منابع بررسی‌شده در تدوین این کتاب — تاریخ پژوهش: مهر ۱۴۰۵ (اکتبر ۲۰۲۶)

تمام ادعاهای نسخه‌ای و تاریخی کتاب به منابع این پیوست متکی‌اند. تاریخ راستی‌آزمایی همه موارد: **۱۶ اکتبر ۲۰۲۶** (۲۴ مهر ۱۴۰۴). URLها در متن چاپی کامل آمده‌اند تا بدون لینک فعال هم قابل پیگیری باشند.

مبنای نسخه‌ها (راستی‌آزمایی‌شده از registry رسمی npm)

ابزار	نسخه مبنای کتاب	منبع
TypeScript (پایدار)	7.0.2 — انتشار ۸ ژوئیه ۲۰۲۶	registry.npmjs.org/typescript، تگ latest
TypeScript (آخرین JS-based)	6.0.3	همان منبع
Node.js LTS	x «Krypton» (24.22.0).24	nodejs.org
React	19.3.1	registry.npmjs.org/react
Next.js	16.4.2	registry.npmjs.org/next
Zod	4.6.7	registry.npmjs.org/zod

کدهای کتاب با TypeScript 7.0.2 (و برای سازگاری، 6.0.3) زیر strict کامل بررسی شده‌اند — جزئیات در گزارش اعتبارسنجی همراه کتاب (/reports).

منابع فصل ۳۳ — گذار TypeScript 6/7

۱. **"Announcing TypeScript 7.0"** — وبلاگ رسمی تیم TypeScript (devblogs.microsoft.com/typescript)، ۱۸ ژوئن ۲۰۲۶. مبنای ادعاهای: پورت native با Go (کد «Corsa»)، شتاب حدوداً ۱۰×، هم‌روندی چندنخی، حذف گزینه‌های deprecated شده در 6.0، عدم پشتیبانی API جاوااسکریپتی کامپایلر قدیم (Strada).

۲. **"Announcing TypeScript 6.0"** — همان وبلاگ، ۶ مارس ۲۰۲۶. مبنای ادعاهای: آخرین نسخه JS-based و نقش «پل» 5.9→7.0، سازگاری API با 5.9، دروازه `"ignoreDeprecations": "6.0"`، پرچم `--stableTypeOrdering` برای هم‌ارزسازی خروجی با 7.0، نبودِ 6.1 (فقط patch).

۳. **مستندات و release notes رسمی** — typescriptlang.org/docs و بخش release notes هر نسخه؛ مرجع جداول گزینه‌های پیوست C و رفتارهای زبانی فصل‌های ۴–۲۶.

تفکیکی که کتاب رعایت کرده و از همین منابع می‌آید: 6→7 تغییر **پیاده‌سازی کامپایلر** است، نه زبان؛ رفتار type-checking دو نسخه هم‌ارز طراحی شده است. ادعاهای کارایی، اعداد اعلامی سازنده‌اند نه اندازه‌گیری مستقل ما — در متن نیز با همین قید آمده‌اند.

مراجع موضوعی

- **Zod v4** — zod.dev: مرجع API فصل‌های ۳۴ و ۳۸ (treeifyError، safeParse، z.infer، اعتبارسنج‌های سطح بالا مانند z.email).
- **React 19** — react.dev: ref به‌عنوان prop معمولی، الگوهای فصل ۳۷؛ تایپ‌ها از پکیج‌های @types/react و @types/react-dom.
- **Next.js 16 (App Router)** — nextjs.org/docs: مرز params، Server Actions، Server/Client Component ناهمگام — فصل ۳۸.
- **Node.js** — nodejs.org/docs: type stripping (اجرای مستقیم .ts) و محدودیت‌هایش، AbortSignal.timeout — فصل‌های ۲، ۳۵، ۳۹.
- **"Parse, don't validate"** — مقاله Alexis King (lexi-lambda.github.io، ۲۰۱۹): خاستگاه شعار معماری فصل ۳۴.
- **type-challenges** (github.com/type-challenges/type-challenges) و **type-fest** (github.com/sindresorhus/type-fest): تمرین و کتابخانه نوع‌های ابزاری — فصل‌های ۲۰، ۲۶، پیوست G.
- **expect-type / tsd / publint / arethetypeswrong** — ابزارهای فصل ۳۲ و ۴۰، هر یک در مخزن رسمی خود در npm/GitHub.

فونت‌ها و مجوزها

- **Vazirmatn** (متن فارسی) — طراحی صابر راستی‌کردار؛ مجوز SIL Open Font License 1.1 — استفاده و embed آزاد.
- **JetBrains Mono** (کد) — JetBrains؛ مجوز SIL Open Font License 1.1.

هر دو فونت با مجوزشان در بسته منابع کتاب همراه‌اند و در PDF embed شده‌اند.

درباره این کتاب

متن، مثال‌ها و پروژه‌ها برای این کتاب نوشته شده‌اند و کدها در مخزن همراه (/examples) قابل اجرا و بررسی‌اند. ساختار ویراستاری کتاب از سنت کتاب‌های مرجع فنی فارسی الهام گرفته است. خطاها و پیشنهادها را به نویسنده برسانید — نسخه‌های بعدی، مثل changelog، semver خواهند داشت.

مرجع جامع TypeScript

بیشتر توسعه‌دهندگان syntax تایپ‌اسکریپت را بلدند؛ اما وقتی کامپایلر یک انتساب ساده را رد می‌کند، وقتی جنریک استنتاج نمی‌شود، یا وقتی خطای چهل‌خطی ظاهر می‌شود، به حدس و خطا می‌افتند. این کتاب فاصله بین «بلد بودن syntax» و «به‌کار بردن TypeScript به‌عنوان ابزار مهندسی» را پر می‌کند.

- مدل ذهنی سیستم نوع: استنتاج، سازگاری ساختاری، narrowing و تحلیل جریان کنترل
- طراحی نوع پیشرفته: جنریک‌ها، conditional و mapped typeها، satisfies و branded types
- کامپایلر و ابزار: tsconfig آگاهانه، ماژول‌ها، مهاجرت تدریجی و گذار TypeScript 6/7
- مهندسی Production: اعتبارسنجی runtime با Zod، مدل خطای صریح، React و Next.js
- ۸ پروژه کامل، بیش از ۱۳۰ پرسش مصاحبه با پاسخ تشریحی، واژه‌نامه و مرجع تصمیم‌گیری

وعده آموزشی کتاب در یک جمله: **مدل را بفهم، رفتار را پیش‌بینی کن، نوع را طراحی کن، پیاده‌سازی را راستی‌آزمایی کن.**

محمدرضا رضائیان · مرجع جامع TypeScript · نسخه ۱٫۰٫۰ · ۲۰۲۶
مبتنی بر TypeScript 7.0 / 6.0 · مثال‌ها با کامپایلر رسمی بررسی شده‌اند