



عمیق‌تر از ظاهر زبان

سطح پیشرفته · آموزش جامع

# JavaScript و TypeScript

## پیشرفته و حرفه‌ای

مدرس دوره

مجتبی سجادفر

Mojtaba Sajadfar

از عمق زبان و مدل اجرا تا ناهمزمانی، ماژول‌ها و سیستم نوع‌های TypeScript؛ با مثال، دیاگرام، فلوچارت و تمرین

ES2023+

TypeScript

JavaScript

## درباره من



مجتبی سجادفر  
Mojtaba Sajadfar

Angular

Ionic

NestJS

OpenLayers

TypeScript

immsr.ir



«من مجتبی سجادفر هستم؛ توسعه‌دهنده نرم‌افزار با علاقه به توسعه رابط کاربری، سامانه‌های مبتنی بر نقشه و ساخت اپلیکیشن‌های وب و موبایل. در این مسیر با فناوری‌هایی مانند Angular، Ionic، NestJS، OpenLayers و TypeScript کار می‌کنم و به یادگیری مستمر، حل مسئله و ساخت تجربه‌های دیجیتال کاربردی علاقه‌مندم.»

نقشه‌های تعاملی

WebGIS



توسعه رابط کاربری

Front-End Development



توسعه اپلیکیشن موبایل

Mobile Applications



توسعه وب‌اپلیکیشن

Web Applications



# این دوره برای چه کسی است؟

## روش یادگیری



مفهوم ← دیاگرام ← کد قابل اجرا ← تمرین؛ هر کد را خودتان اجرا کنید.

## پیش نیاز



آشنایی با مبانی HTML، JavaScript و کار با ترمینال و Git.

## مخاطب



برنامه نویسان فرانت اند یا بک اندی که می خواهند JavaScript را عمیق و TypeScript را حرفه ای یاد بگیرند.

## در پایان دوره می توانید

۲ Closure، this و پروتوتایپ را بدون ابهام به کار ببرید

۱ مدل اجرای JavaScript (Call Stack و Event Loop) را توضیح دهید

۴ ماژول ها، عملکرد و امنیت مرورگر را مدیریت کنید

۳ کد ناهمزمان را با Promise و async/await درست طراحی کنید

۶ از Generics، نوع های شرطی و Utility Types استفاده کنید

۵ با TypeScript نوع های دقیق و ایمن بنویسید

# نقشه‌ی چهار بخش دوره

## ۴ TypeScript پیشرفته

- Mapped و Conditional
- Utility Types
- Type Guards
- Declaration Merging
- ابزار و مهاجرت

## ۳ مبانی TypeScript

- نوع‌های روزمره
- Narrowing
- Generics
- کلاس‌ها
- فایل‌های تعریف نوع

## ۲ ناهمزمانی و پلتفرم

- Event Loop
- async/await و Promise
- ماژول‌ها
- DOM، عملکرد و امنیت
- Node.js و تست

## ۱ هسته‌ی JavaScript

- مدل اجرا و Hoisting
- this و Closure
- پروتوتایپ و کلاس
- Generator و Iterator
- Reflect و Proxy

پروژه‌ی نهایی: API Client تایپ‌شده



تمرین و Cheat Sheet در پایان هر بخش



پروژه‌های کوچک بین بخش‌ها



# ابزار و محیط کار

```
ترمینال · bash
$ node -v
v22.x.x
$ mkdir adv-js && cd adv-js
$ npm init -y
$ npm install -D typescript tsx
$ npx tsc --init
$ npx tsx hello.ts
```

## Node.js (نسخه LTS)

اجرای JavaScript بیرون از مرورگر و npm.



## VS Code

با افزونه‌های ESLint و Prettier.



## TypeScript (tsc)

کامپایلر و بررسی نوع‌ها.



## tsx

اجرای مستقیم فایل‌های TS بدون کامپایل دستی.



هسته‌ی زبان

بخش ۱ از ۴

# هسته‌ی JavaScript؛ عمیق‌تر از ظاهر

مدل اجرا، نوع‌ها، تابع‌ها، شیء‌ها، پروتوتایپ و کلاس‌ها

Proxy و Iterator ●

پروتوتایپ و کلاس ●

this و Closure ●

مدل اجرا و نوع‌ها ●

# مدل اجرا: Call Stack و Execution Context

script.js

```
function a() { return b(); }
function b() { return c(); }
function c() { return 42; }
```

```
a(); // a → b → c
```

Call Stack



فریم‌های فراخوانی به‌صورت LIFO.

Heap



حافظه‌ی اشیا و توابع؛ با ارجاع.

Execution Context



متغیرها، this و زنجیره‌ی Scope هر فراخوانی.

Call Stack هنگام اجرای c()

c()

b()

a()

Global



# Temporal Dead Zone و Hoisting

hoisting.js

```
console.log(a);      // undefined (var hoisted)
var a = 1;

console.log(f());    // "ok" (declaration hoisted)
function f() { return "ok"; }

console.log(b);      // ReferenceError (TDZ)
let b = 2;
```

## فاز ساخت و فاز اجرا

اعلان‌ها پیش از اجرای خط‌ها ثبت می‌شوند.



## var

بالا می‌آید و مقدار اولیه undefined دارد.



## اعلان تابع

کامل (با بدنه) بالا می‌آید.



## const (TDZ) و let

تا اعلان، دسترسی خطا می‌دهد.



# تبدیل نوع و برابری: دام‌های رایج

| عبارت               | نتیجه             | علت                             |
|---------------------|-------------------|---------------------------------|
| "5" + 1             | "51"              | با + و رشته: الحاق              |
| "5" - 1             | 4                 | عملگر - عدد می‌خواهد (ToNumber) |
| [] + {}             | "[object Object]" | هر دو به رشته تبدیل می‌شوند     |
| 0 == ""             | true              | هر دو به عدد ۰ تبدیل می‌شوند    |
| null == undefined   | true              | حالت خاص عملگر ==               |
| NaN === NaN         | false             | NaN با خودش برابر نیست          |
| Object.is(NaN, NaN) | true              | الگوریتم SameValue              |
| [] == false         | true              | false → 0 و 0 → "" → []         |

قاعده‌ی عملی: همیشه از === استفاده کنید و تبدیل نوع را صریح انجام دهید.

# مقدار (Primitive) در برابر ارجاع (Reference)

## Reference



- شامل شیء، آرایه، تابع، Map و Set
- در Heap نگهداری می‌شود
- کپی و مقایسه بر اساس ارجاع

```
const x = { n: 1 }; const y = x;
```

```
y.n = 2; // x.n === 2
```

## Primitive



- شامل `string`، `number`، `boolean`، `null`، `undefined`، `bigint` و `symbol`
- تغییرناپذیر (Immutable)
- کپی و مقایسه بر اساس مقدار

```
let a = 1; let b = a; b++; // a === 1
```

# کپی سطحی و عمیق

copy.js

```
const a = { n: 1, d: { k: 1 } };
const s1 = { ...a };           // shallow
const s2 = structuredClone(a); // deep

s1.d.k = 9;
console.log(a.d.k); // 9 (shared)
console.log(s2.d.k); // 1 (independent)
```

## کپی سطحی

... و `Object.assign`؛ تو در تو مشترک می‌ماند.



## کپی عمیق

`structuredClone` برای داده‌ی ساده.



## محدودیت

تابع، DOM و کلاس‌ها کپی نمی‌شوند.



## تर्फند JSON

`Date` و `undefined` را خراب می‌کند.



# Nullish Coalescing و Optional Chaining

operators.js

```
const user = { profile: { name: "Sara" } };
user.profile?.name;           // "Sara"
user.address?.city;           // undefined (no error)
user.getAge?.();               // undefined

const n = 0;
n || 10;                       // 10    (0 is falsy)
n ?? 10;                       // 0
let cfg = {};
cfg.port ??= 3000;             // assign if nullish
```

## ?. زنجیره‌ی امن

برای ویژگی، فراخوانی و اندیس.



## ??

فقط برای `null` و `undefined`.



## || در برابر ??

|| برای هر مقدار falsy پیش‌فرض می‌گذارد.



## ==&& ==|| ==??

انتساب منطقی کوتاه.



# ۱ Closure: تابع و محیط لغوی آن

closure.js

```
function makeCounter() {  
  let count = 0;  
  return function () {  
    count++;  
    return count;  
  };  
}  
const c1 = makeCounter();  
console.log(c1(), c1()); // 1 2  
const c2 = makeCounter();  
console.log(c2()); // 1
```

## Closure = تابع + محیط

تابع، محیط لغوی (Lexical Environment) خود را همراه دارد.



## حالت خصوصی

count فقط از داخل تابع قابل دسترسی است.



## محیط جدید برای هر فراخوانی

c1 و c2 مستقل‌اند.



## زنجیره‌ی Scope

جست‌وجوی متغیر از داخل به بیرون.



## ۲ Closure: الگوها و دام‌ها

closure2.js

```
for (var i = 0; i < 3; i++) setTimeout(() => console.log(i)); // 3 3 3
for (let j = 0; j < 3; j++) setTimeout(() => console.log(j)); // 0 1 2
```

```
const memo = (fn) => {
  const cache = new Map();
  return (x) => {
    if (!cache.has(x)) cache.set(x, fn(x));
    return cache.get(x);
  };
};
const slowSquare = (n) => n * n;
const fast = memo(slowSquare);
```

### var در حلقه

یک متغیر مشترک؛ خروجی 3 3 3.



### memoize

کش نتیجه با Closure و Map.



### ماژول و Factory

حالت خصوصی بدون کلاس.

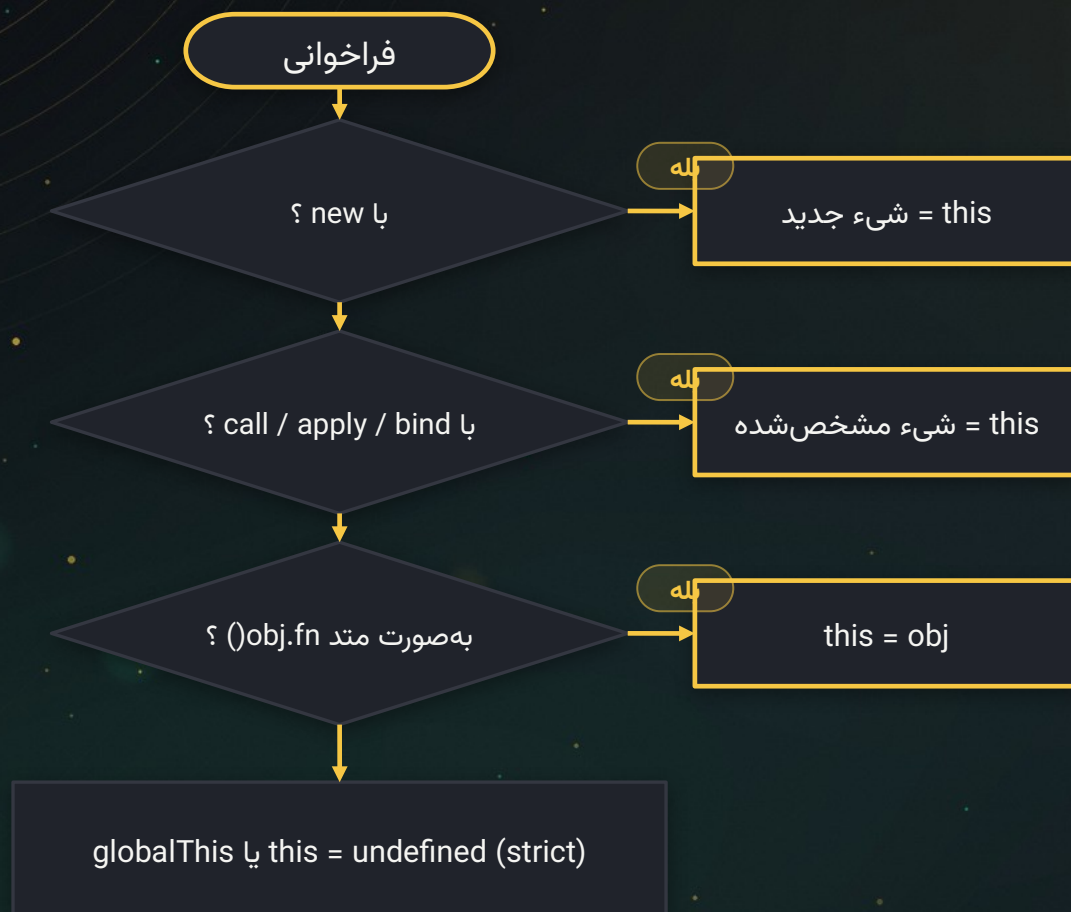


### نشت حافظه

Closure مرجع متغیرها را نگه می‌دارد.



# this: چه کسی صدا زده؟



this.js

```

const o = {
  n: "A",
  f() { return this.n; },
  g: () => this, // lexical this
};
o.f(); // "A"
const h = o.f;
h(); // undefined (this lost)
  
```

تابع arrow



this ندارد؛ آن را از محیط لغوی بیرونی می‌گیرد، پس برای callbackها مناسب است ولی برای متدهای شیء نه.

# bind و call، apply

bind.js

```
function greet(g, p) { return `${g}, ${this.name}${p}`; }  
const u = { name: "Sara" };
```

```
greet.call(u, "Hi", "!");           // "Hi, Sara!"  
greet.apply(u, ["Hello", "?"]);    // "Hello, Sara?"
```

```
const bound = greet.bind(u, "Hey");  
bound("...");                      // "Hey, Sara..."
```

## call

فراخوانی فوری؛ آرگومان‌ها جدا.



## apply

فراخوانی فوری؛ آرگومان‌ها در آرایه.



## bind

تابع جدید با **this** ثابت (و آرگومان جزئی).

## bind و arrow

روی تابع arrow اثری ندارد.



# توابع پیشرفته: Rest، Currying و HOF

functions.js

```
const sum = (...nums) => nums.reduce((a, b) => a + b, 0);  
Math.max(...[1, 2, 3]); // 3  
const hi = (name = "guest") => `Hi ${name}`;  
  
const curry = (f) => (a) => (b) => f(a, b);  
curry((a, b) => a + b)(2)(3); // 5  
  
const compose = (...fs) => (x) =>  
  fs.reduceRight((v, f) => f(v), x);  
const inc = (x) => x + 1, dbl = (x) => x * 2;  
compose(dbl, inc)(3); // 8
```

## Spread و Rest

جمع و پخش آرگومان‌ها.



## Currying

تبدیل  $f(a,b)$  به  $f(a)(b)$ .

## HOF

تابعی که تابع می‌گیرد یا برمی‌گرداند.



## Compose

ترکیب توابع کوچک.



# بازگشت (Recursion) و Call Stack

recursion.js

```
function fact(n) {
  if (n <= 1) return 1;      // base case
  return n * fact(n - 1);    // recursive case
}
console.log(fact(4)); // 24
```

حالت پایه



سرریز Stack



Memoization



جلوگیری از محاسبه‌ی تکراری (مثل fib).

فریم‌ها هنگام  $\text{fact}(4)$  $\text{fact}(1) \rightarrow 1$  $\text{fact}(2) \rightarrow 2 \cdot 1$  $\text{fact}(3) \rightarrow 3 \cdot 2$  $\text{fact}(4) \rightarrow 4 \cdot 6$ 

بالاترین فریم ابتدا برمی‌گردد و مقدار را پایین می‌دهد

# ویژگی‌های ویژگی‌ها: Property Descriptors

descriptors.js

```
const o = {};  
Object.defineProperty(o, "id", {  
  value: 7,  
  writable: false,  
  enumerable: false,  
  configurable: false,  
});  
o.id = 9; // ignored (TypeError in strict)  
Object.keys(o); // []  
Object.getOwnPropertyDescriptor(o, "id");  
Object.freeze(o);  
Object.isFrozen(o); // true
```

## writable

آیا مقدار قابل تغییر است؟



## enumerable

آیا در `for...in` و `Object.keys` دیده می‌شود؟

## configurable

آیا حذف و تغییر فلگ‌ها ممکن است؟



## freeze / seal

قفل سطحی شیء.



# Object و Setter و Getter

objects.js

```
const user = {
  first: "Sara", last: "Ahmadi",
  get full() { return `${this.first} ${this.last}`; },
  set full(v) { [this.first, this.last] = v.split(" "); },
};
user.full = "Ali Rezaei";
```

```
const key = "role";
const o = { [key]: "admin" };           // computed key
Object.entries({ a: 1 });               // [["a", 1]]
Object.fromEntries([["a", 1]]);         // { a: 1 }
Object.hasOwn(o, "role");               // true
```

## get / set

ویژگی محاسبه‌شده با رفتار تابع.



## کلید محاسباتی

[expr] به‌عنوان نام ویژگی.



## fromEntries و entries

تبدیل شیء و جفت‌ها.



## hasOwn

جایگزین امن `.hasOwnProperty`

# زنجیره‌ی پروتوتایپ (Prototype Chain)

proto.js

```
const animal = { eats: true };
const rabbit = Object.create(animal);
rabbit.jumps = true;

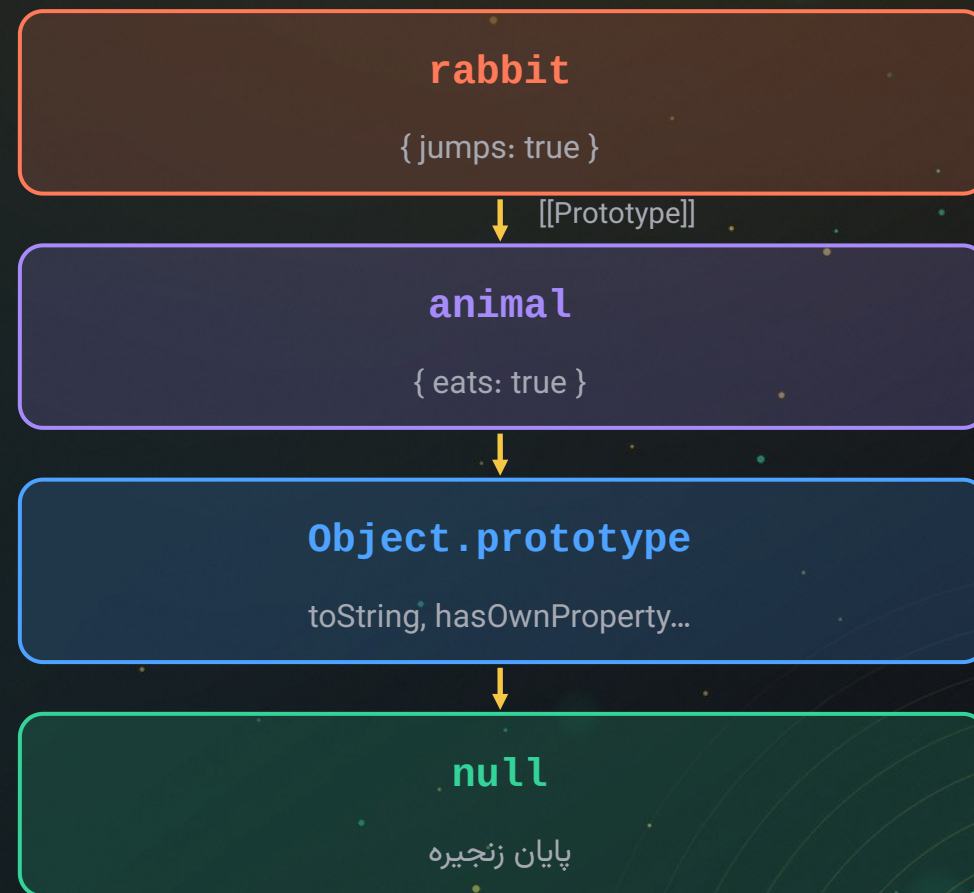
rabbit.eats; // true
Object.getPrototypeOf(rabbit) === animal; // true
"eats" in rabbit; // true
Object.hasOwn(rabbit, "eats"); // false
```

جست‌وجو

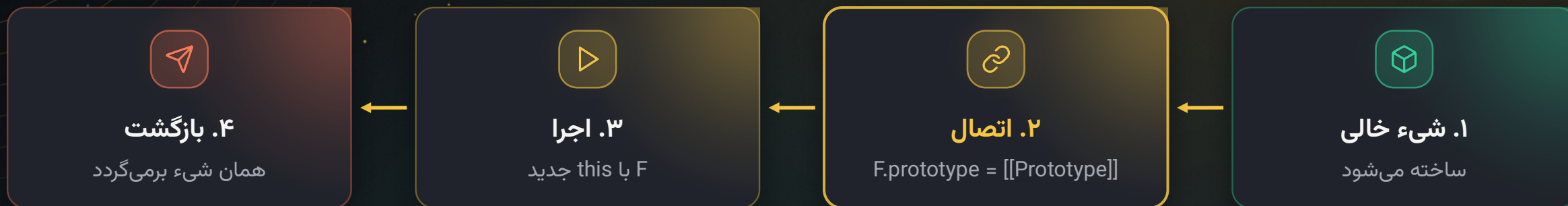
اگر ویژگی پیدا نشد، به پروتوتایپ می‌رود.

نوشتن

همیشه روی خود شیء انجام می‌شود.



# تابع سازنده و new



new.js

```
function Person(name) { this.name = name; }
Person.prototype.hello = function () { return "Hi " + this.name; };

const p = new Person("Sara");
p.hello(); // "Hi Sara"
p instanceof Person; // true
Object.getPrototypeOf(p) === Person.prototype; // true
```

F.prototype

پروتوتایپ مشترک همه‌ی نمونه‌ها.



instanceof

زنجیره‌ی پروتوتایپ را بررسی می‌کند.



# کلاس‌ها: Private، Static و Accessor

class.js

```
class Account {  
  #balance = 0; // private  
  static count = 0;  
  constructor(owner) { this.owner = owner; Account.count++; }  
  deposit(x) { this.#balance += x; return this; }  
  get balance() { return this.#balance; }  
  static compare(a, b) { return a.balance - b.balance; }  
}  
const a = new Account("Sara").deposit(50);  
a.balance; // 50  
// a.#balance // SyntaxError
```

## private#

فیلد و متد واقعاً خصوصی.



## static

عضو خود کلاس، نه نمونه.



## شکل خوانا

روی پروتوتایپ‌ها ساخته شده است.



## get / set

ویژگی محاسبه‌شده در کلاس.



# وراثت: extends و super و Mixin

inherit.js

```
class Animal {
  constructor(name) { this.name = name; }
  speak() { return `${this.name} makes a sound`; }
}
class Dog extends Animal {
  speak() { return super.speak() + " (woof)"; }
}
const d = new Dog("Rex");
d instanceof Animal; // true

const Swimmer = (Base) => class extends Base {
  swim() { return `${this.name} swims`; }
};
class Duck extends Swimmer(Animal) {}
```

## extends

کلاس فرزند و پروتوتایپ والد.



## super

فراخوانی سازنده و متد والد.



## پیش از this

در سازنده، اول `super()`.



## Mixin

ترکیب رفتار بدون وراثت چندگانه.



# Symbol و پروتکل Iterator

iterator.js

```
const id = Symbol("id");
const o = { [id]: 1, name: "x" };
Object.keys(o);           // ["name"] (symbols hidden)

const range = {
  from: 1, to: 3,
  [Symbol.iterator]() {
    let c = this.from, t = this.to;
    return {
      next: () => c <= t
        ? { value: c++, done: false }
        : { value: undefined, done: true },
    };
  },
};
console.log([...range]);    // [1, 2, 3]
```

## Symbol

کلید یکتا و پنهان.



## Iterable

متد [Symbol.iterator]



## Iterator

next() با { value, done }



## spread و for...of

روی هر iterable کار می‌کنند.



# Generator: تابع‌هایی که متوقف می‌شوند

generator.js

```
function* idGen() {  
  let id = 1;  
  while (true) yield id++;  
}  
  
const g = idGen();  
g.next().value;    // 1  
g.next().value;    // 2  
  
function* talk() {  
  const name = yield "name?";  
  yield `hi ${name}`;  
}  
  
const t = talk();  
t.next();  
t.next("Sara").value;    // "hi Sara"
```

**yield**

اجرا را متوقف و مقدار می‌دهد.

**تنبل (Lazy)**

مقدار فقط هنگام نیاز تولید می‌شود.

**دوطرفه**`next(v)` مقدار به داخل می‌فرستد.**\*yield**

واگذاری به generator دیگر.



# WeakSet و Map، Set، WeakMap

| ساختار  | کلید           | پیمایش                 | کاربرد                       |
|---------|----------------|------------------------|------------------------------|
| Object  | رشته یا Symbol | ترتیب درج با قواعد عدد | رکورد ساده با کلید رشته‌ای   |
| Map     | هر نوع         | ترتیب درج              | کلید-مقدار با کلید دلخواه    |
| Set     | – (مقدار یکتا) | ترتیب درج              | حذف تکراری و عضویت سریع      |
| WeakMap | فقط شیء        | قابل پیمایش نیست       | داده‌ی خصوصی هر شیء بدون نشت |
| WeakSet | فقط شیء        | قابل پیمایش نیست       | برچسب‌گذاری اشیا             |

قاعده: کلید غیررشته‌ای یا نیاز به ترتیب و اندازه ← Map؛ داده‌ی وابسته به عمر شیء ← WeakMap.

# آرایه و رشته: reduce، flatMap و Intl

arrays.js

```
const orders = [
  { u: "a", t: 10 }, { u: "b", t: 5 }, { u: "a", t: 7 },
];
const byUser = orders.reduce((acc, o) => {
  (acc[o.u] ??= []).push(o.t);
  return acc;
}, {}); // { a: [10, 7], b: [5] }

[[1, 2], [3]].flatMap((x) => x); // [1, 2, 3]
[10, 9, 1].sort((a, b) => a - b); // [1, 9, 10]
[1, 2, 3].at(-1); // 3
new Intl.NumberFormat("fa-IR").format(1234567);
```

## reduce

ساخت هر ساختار از آرایه.



## flatMap

map و flat در یک گام.



## sort

بدون comparator، عددها را رشته‌ای مرتب می‌کند.



## Intl

قالب‌بندی و مقایسه‌ی محلی.



# عبارت‌های باقاعده (Regular Expressions)

regex.js

```
const re = /(?<year>\d{4})-(?<month>\d{2})/u;
const m = "Date: 2025-03".match(re);
m.groups.year; // "2025"

"a1b22".replace(/\d+/g, (n) => `<${n}>`); // "a<1>b<22>"
/(?<=\$)\d+/.exec("cost $45")[0]; // "45"
/<.+?>/g; // lazy quantifier
/\p{Script=Arabic}+/u.test("سلام"); // true
```

فلگ‌ها

.g i m u s y



گروه‌های نام‌دار

.m.groups و (...<name>?)



Lookbehind و Lookahead

شرط بدون مصرف کاراکتر.



Backtracking

الگوهای تو در توی حریص می‌توانند کند شوند.



# اعداد، Date، BigInt، Intl و Intl

numbers.js

```
0.1 + 0.2; // 0.30000000000000004
(0.1 + 0.2).toFixed(2); // "0.30"
Number.isSafeInteger(2 ** 53); // false

const big = 2n ** 64n; // BigInt
typeof big; // "bigint"

new Date("2025-03-01T10:00:00Z").toISOString();
new Intl.DateTimeFormat("fa-IR").format(new Date(0));
```

## double

دقت محدود اعشار؛ مقایسه با **EPSILON**.

## BigInt

عدد صحیح بدون سقف دقت، با پسوند **n**.

## Date

از ISO و UTC برای ذخیره استفاده کنید.



## Intl

قالب‌بندی تاریخ و عدد فارسی.



# JSON و Destructuring، Spread پیشرفته

data.js

```
const { a, b: { c = 5 } = {}, ...rest } = { a: 1, x: 2, y: 3 };  
const [first, , third = 0, ...others] = [1, 2];  
const merged = { ...defaults, ...options };
```

```
JSON.stringify({ d: new Date(0), u: undefined }, null, 2);  
JSON.parse('{"n":1}', (k, v) => (k === "n" ? v + 1 : v));
```

```
class P { toJSON() { return { kind: "P" }; } }  
JSON.stringify(new P()); // '{"kind":"P"}'
```

## تخریب پیشرفته

پیش فرض، تغییر نام و rest.



## Spread

ادغام و کپی سطحی.



## reviver و replacer

کنترل سریالایز و بازیابی.



## Map و Date

در JSON درست ذخیره نمی‌شوند.



# مدیریت خطا: try، catch، finally و cause

errors.js

```
class AppError extends Error {  
  constructor(msg, opts) { super(msg, opts); this.name = "AppError"; }  
}  
try {  
  try { JSON.parse("{bad}"); }  
  catch (e) { throw new AppError("config invalid", { cause: e }); }  
} catch (err) {  
  console.log(err.name, err.cause.name); // AppError SyntaxError  
  if (!(err instanceof AppError)) throw err; // rethrow  
} finally {  
  console.log("cleanup");  
}
```

## نوع‌های خطا

...TypeError، RangeError، SyntaxError



## cause

نگه‌داشتن علت اصلی خطا.



## finally

همیشه اجرا می‌شود؛ برای پاک‌سازی.



## دوباره پرتاب

خطای ناشناخته را نبلیید.



# Reflect و Proxy

proxy.js

```
const target = { age: 20 };
const guarded = new Proxy(target, {
  get(t, k, r) {
    return k in t ? Reflect.get(t, k, r) : `no ${String(k)}`;
  },
  set(t, k, v) {
    if (k === "age" && (!Number.isInteger(v) || v < 0))
      throw new TypeError("bad age");
    return Reflect.set(t, k, v);
  },
});
guarded.age = 21;    // ok
guarded.size;       // "no size"
```

## Trap

...get, set, has, deleteProperty, apply



## Reflect

اجرای رفتار پیش فرض هر trap.



## کاربرد

اعتبارسنجی، لاگ و واکنش پذیری.



## هزینه

رهگیری کندتر از دسترسی مستقیم است.



# کیفیت کد و آزمون خودکار

sum.test.js

```
import { describe, it, expect } from "vitest";
import { sum } from "../sum.js";

describe("sum", () => {
  it("adds numbers", () => {
    expect(sum(2, 3)).toBe(5);
  });
  it("returns 0 for no input", () => {
    expect(sum()).toBe(0);
  });
});
```

## "use strict"

خطاهای خاموش را آشکار می‌کند.



## Prettier و ESLint

بررسی خودکار سبک و باگ.



## آزمون واحد

کوچک، سریع و مستقل.



## CI

اجرای خودکار آزمون در هر تغییر.



## وقت تمرین

## تمرین‌های هسته‌ی JavaScript

آسان

تابع memoize بنویسید که نتیجه‌ی فراخوانی با آرگومان یکسان را کش کند. راهنما: Map و Closure

۱

آسان

شمارنده‌ای با Closure بسازید که increment، decrement و reset داشته باشد. راهنما: حالت خصوصی

۲

متوسط

deepClone ساده برای شیء و آرایه بنویسید (بدون structuredClone). راهنما: بازگشت و typeof

۳

متوسط

کلاس Stack با فیلد items# و متدهای push، pop و peek بسازید. راهنما: فیلد خصوصی

۴

متوسط

iterable ای بنویسید که با گام دلخواه عدد تولید کند (range یا step). راهنما: Symbol.iterator یا generator

۵

چالشی

یک Proxy بسازید که دسترسی به کلید ناموجود را لاگ کند و مقدار پیش‌فرض بدهد. راهنما: trap get

۶

پس از حل، رفتار را با آزمون‌های Vitest بسنجید.

## خلاصه‌ی سریع

## خلاصه‌ی هسته‌ی JavaScript (Cheat Sheet)

TDZ و Hoisting

`var let const`

فریم‌های فراخوانی / حافظه‌ی اشیا

`Call Stack / Heap`

دسترسی امن و پیش‌فرض

`?. ?? ??=`

برابری بدون coercion

`=== / Object.is``new > call/bind > obj.fn() > global``this`

تابع + محیط لغوی

`Closure`

ترکیب و HOF

`...rest compose`تعیین صریح `this``call apply bind`

زنجیره‌ی پروتوتایپ

`Object.create`

فلگ‌های ویژگی

`defineProperty`

پروتوکل iterable

`Symbol.iterator`وراثت و `private#``class extends super`

رهگیری عملیات شیء

`Proxy + Reflect`

Generator تنبل

`function* yield`

ناهمزمانی و پلتفرم

بخش ۲ از ۴

# ناهمزمانی، ماژول و پلتفرم

Node.js، Event Loop، Promise، ماژول‌ها، عملکرد، امنیت و

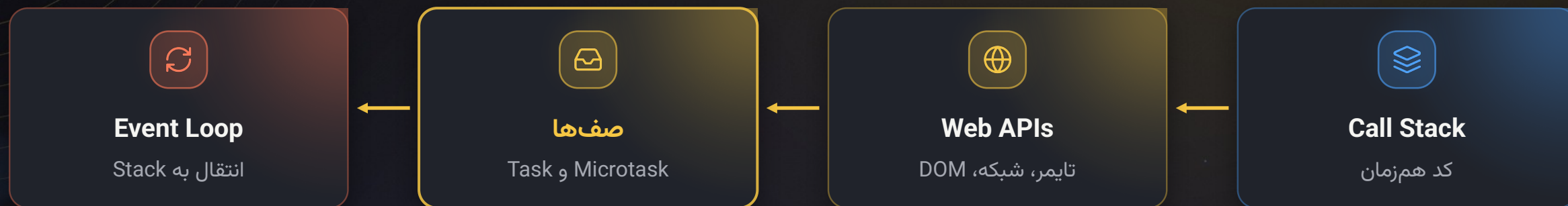
عملکرد و امنیت ●

ماژول‌ها ●

async و Promise ●

Event Loop ●

# Event Loop: قلب ناهمزمانی



## قاعده‌ی اجرا

Stack خالی ← همه‌ی Microtask ها ← یک Task ← رندر.

## Microtask

`Promise.then`, `queueMicrotask`, `MutationObserver`

## Task (Macrotask)

`setTimeout`, رویدادهای DOM و I/O

# ترتیب اجرا: Microtask پیش از Task

● ● ● order.js

```
console.log("1 sync");
setTimeout(() => console.log("4 timeout"), 0);
Promise.resolve().then(() => console.log("3 promise"));
queueMicrotask(() => console.log("3b microtask"));
console.log("2 sync");
```

● ● ● خروجی · Console

نمایشی

```
1 sync
2 sync
3 promise
3b microtask
4 timeout
```

## ۱. همزمان

ابتدا همه‌ی خط‌های عادی.



## ۲. Microtask

همه‌ی Promise‌ها قبل از Task بعدی.



## ۳. Task

setTimeout(..., 0) بعد از Microtask‌ها.



# promisify و Callback، Callback Hell

callbacks.js

```
getUser(1, (err, user) => {  
  if (err) return handle(err);  
  getPosts(user.id, (err, posts) => {  
    if (err) return handle(err);  
    getComments(posts[0].id, (err, c) => {  
      /* pyramid of doom */  
    });  
  });  
});
```

```
import { promisify } from "node:util";  
const getUserP = promisify(getUser);
```

## Callback Hell

تو در توی عمیق؛ خوانایی کم.



## error-first

قرارداد: آرگومان اول خطا.



## promisify

تبدیل callback به Promise.



# Promise: حالت‌ها و زنجیره



promise.js

```

fetchUser(1)
  .then((u) => fetchPosts(u.id))
  .then((posts) => posts.length)
  .catch((err) => { console.error(err); return 0; })
  .finally(() => console.log("done"));
  
```

## then زنجیره می‌سازد

هر **then** یک Promise جدید برمی‌گرداند.



## catch

خطا تا نزدیک‌ترین **catch** پایین می‌رود.



## finally

همیشه اجرا می‌شود.



# ترکیب Promise ها: race، allSettled، all و any

| متد                | منتظر چه می ماند | نتیجه                              | اگر یکی رد شود              |
|--------------------|------------------|------------------------------------|-----------------------------|
| Promise.all        | همه موفق شوند    | آرایه ی نتیجه ها                   | بلافاصله رد می شود          |
| Promise.allSettled | همه تمام شوند    | آرایه ی { status, value   reason } | هرگز رد نمی شود             |
| Promise.race       | اولین تمام شده   | نتیجه ی همان اولی                  | اگر اولی رد باشد، رد می شود |
| Promise.any        | اولین موفق       | نتیجه ی اولین موفق                 | همه رد شوند: AggregateError |

برای درخواست های مستقل از Promise.all یا allSettled استفاده کنید تا زمان کل کمتر شود.

# async/await: ترتیبی در برابر موازی

async.js

```
// sequential: time = a + b
const u = await getUser();
const p = await getPosts();

// parallel: time = max(a, b)
const [u2, p2] = await Promise.all([getUser(), getPosts()]);

try {
  const data = await load();
} catch (err) {
  console.error(err);
}
```

## ترتیبی

زمان کل برابر جمع زمان‌ها.



## موازی

زمان کل برابر بلندترین درخواست.



## try/catch

برای خطای `await` ضروری است.

## await در حلقه

هر دور منتظر می‌ماند؛ در صورت امکان موازی کنید.



# پیمایش ناهمزمان: for await و async generator

pages.js

```
async function* pages(url) {
  let next = url;
  while (next) {
    const res = await fetch(next);
    const data = await res.json();
    yield data.items;
    next = data.next;
  }
}

for await (const items of pages("/api/users?page=1")) {
  console.log(items.length);
}
```

## \*async function

generator که می‌تواند await کند.



## for await...of

پیمایش دنباله‌ی ناهمزمان.



## جریان (Stream)

پردازش داده‌ی بزرگ تکه‌تکه.



# زمان بندی: `requestAnimationFrame` و `debounce`، `throttle`

● ● ● timing.js

```
const debounce = (fn, ms) => {  
  let id;  
  return (...a) => { clearTimeout(id); id = setTimeout(() => fn(...a), ms); };  
};  
const throttle = (fn, ms) => {  
  let last = 0;  
  return (...a) => {  
    const now = Date.now();  
    if (now - last >= ms) { last = now; fn(...a); }  
  };  
};  
input.addEventListener("input", debounce(search, 300));  
requestAnimationFrame(draw);
```

## debounce

بعد از توقف رویداد اجرا می شود.



## throttle

حداکثر هر `ms` یک بار.



## requestAnimationFrame

همگام با رندر؛ مخصوص انیمیشن.



# Fetch پیشرفته: POST، لغو و timeout

http.js

```
async function postJSON(url, body, { timeout = 5000 } = {}) {  
  const ctrl = new AbortController();  
  const id = setTimeout(() => ctrl.abort(), timeout);  
  try {  
    const res = await fetch(url, {  
      method: "POST",  
      headers: { "Content-Type": "application/json" },  
      body: JSON.stringify(body),  
      signal: ctrl.signal,  
    });  
    if (!res.ok) throw new Error(`HTTP ${res.status}`);  
    return await res.json();  
  } finally {  
    clearTimeout(id);  
  }  
}
```

**res.ok**

fetch برای 404 رد نمی‌شود.



**AbortController**

لغو و timeout درخواست.



**Content-Type**

برای ارسال JSON لازم است.



# export و ES Modules: import

modules.js

```
// math.js
export const PI = 3.14;
export function add(a, b) { return a + b; }
export default class Calculator {}

// app.js
import Calculator, { PI, add as sum } from "./math.js";
import * as math from "./math.js";

const heavy = await import("./heavy.js"); // dynamic
```

## Default و Named

چند export نام‌دار و یک پیش‌فرض.



## Live Binding

import پیوند زنده است، نه کپی.



## import () پویا

بارگذاری هنگام نیاز؛ code splitting.



## Top-level await

فقط در ماژول‌ها.



# ES Modules در برابر CommonJS

| ویژگی             | CommonJS                              | ES Modules                                          |
|-------------------|---------------------------------------|-----------------------------------------------------|
| نحو               | <code>require / module.exports</code> | <code>import / export</code>                        |
| بارگذاری          | همزمان                                | ناهمزمان؛ تحلیل ایستا                               |
| Tree-shaking      | دشوار                                 | ممکن (حذف کد استفاده نشده)                          |
| Top-level await   | ندارد                                 | دارد                                                |
| در مرورگر         | نیاز به bundler                       | مستقیم با <code>"type"="module"</code>              |
| فعال سازی در Node | پیش فرض قدیمی                         | <code>"type": "module"</code> یا <code>mjs</code> . |

پروژه‌ی جدید را با ESM شروع کنید؛ bundlerهایی مثل Vite کد را برای انتشار بسته‌بندی و بهینه می‌کنند.

# رویدادها: Capturing، Bubbling و Delegation

events.js

```
list.addEventListener("click", (e) => {
  const li = e.target.closest("li");
  if (!li || !list.contains(li)) return;
  console.log("clicked", li.dataset.id);
});

btn.addEventListener("click", handler, { once: true });

btn.dispatchEvent(new CustomEvent("saved", {
  detail: { id: 1 }, bubbles: true,
}));
```

## Capturing

از ریشه به هدف (با `{ capture: true }`).

## Bubbling

از هدف به ریشه؛ پیش فرض.



## Delegation

یک شنونده روی والد برای فرزندان پویا.



## CustomEvent

رویداد سفارشی با `detail`.

# عملکرد DOM و Observer ها

observer.js

```
const io = new IntersectionObserver((entries) => {  
  for (const e of entries) {  
    if (e.isIntersecting) {  
      e.target.classList.add("visible");  
      io.unobserve(e.target);  
    }  
  }  
}, { threshold: 0.2 });  
  
document.querySelectorAll(".lazy").forEach((el) => io.observe(el));
```

## Repaint و Reflow

خواندن/نوشتن متناوب Layout را پرهزینه می‌کند.



## دسته‌ای کردن

DocumentFragment و تغییر یک‌جا.



## IntersectionObserver

lazy loading و انیمیشن اسکرول.

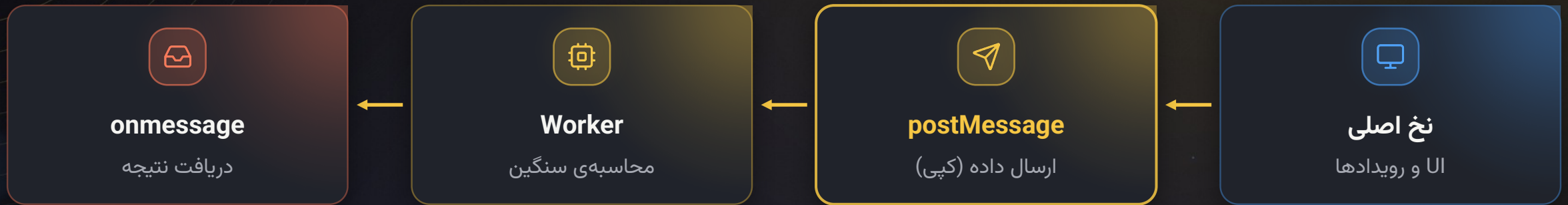


## Resize/Mutation

تغییر اندازه و تغییر DOM.



# Web Worker: کار سنگین خارج از نخ اصلی



worker.js

```
// main.js
const w = new Worker(new URL("./worker.js", import.meta.url), { type: "module" });
w.postMessage({ n: 40 });
w.onmessage = (e) => console.log("fib =", e.data);

// worker.js
self.onmessage = (e) => self.postMessage(fib(e.data.n));
```

## بدون DOM

Worker به DOM دسترسی ندارد.

## کپی داده

پیام‌ها کپی می‌شوند (یا Transferable).

# ذخیره‌سازی در مرورگر: کدام را کی؟

| کاربرد                     | نوع داده  | همزمانی  | ظرفیت تقریبی    | روش            |
|----------------------------|-----------|----------|-----------------|----------------|
| نشست و ارتباط با سرور      | رشته      | همزمان   | حدود ۴ کیلوبایت | Cookie         |
| ترجیحات کاربر              | رشته      | همزمان   | حدود ۵ مگابایت  | localStorage   |
| داده‌ی موقت هر تب          | رشته      | همزمان   | حدود ۵ مگابایت  | sessionStorage |
| داده‌ی ساخت‌یافته و آفلاین | شیء، Blob | ناهمزمان | بسیار بزرگ      | IndexedDB      |
| منابع و حالت آفلاین        | Response  | ناهمزمان | بسته به مرورگر  | Cache API      |

داده‌ی حساس را در localStorage نگذارید؛ با XSS قابل خواندن است.

# مدیریت حافظه و Garbage Collection

leaks.js

```
const cache = new Map();
function remember(obj) {
  cache.set(obj.id, obj);           // leak: never removed
}

const meta = new WeakMap();
function tag(el) {
  meta.set(el, { clicks: 0 });      // freed with el
}

const timer = setInterval(tick, 1000);
// clearInterval(timer) when finished
```

## Reachability

هر چیز قابل رسیدن از ریشه زنده است.



## نشت‌های رایج

تایمر، شنونده، کش بی‌حد و DOM جداشده.



## WeakRef و WeakMap

ارجاع ضعیف، بدون جلوگیری از آزادسازی.

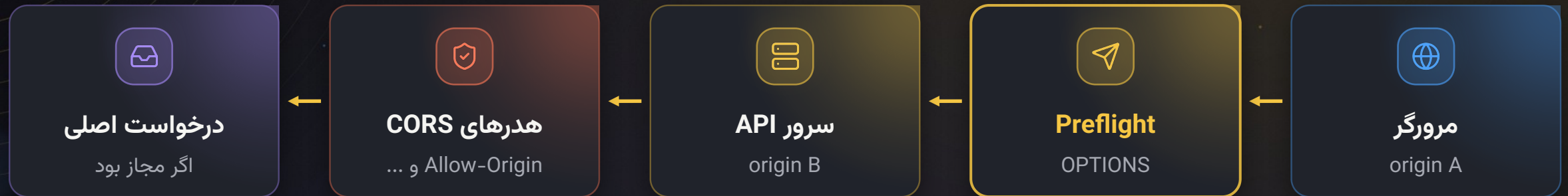


## ابزار

Heap Snapshot در تب Memory.



# امنیت مرورگر: XSS، CORS، و CSRF



headers

```
Access-Control-Allow-Origin: https://app.example.com
Content-Security-Policy: default-src 'self'; script-src 'self'
Set-Cookie: sid=...; HttpOnly; Secure; SameSite=Lax
```

XSS

textContent، پاکسازی و CSP.

CSRF

SameSite و توکن.

# Node.js: اجرای JavaScript روی سرور

read.mjs

```
import { readFile } from "node:fs/promises";
import { join } from "node:path";

const file = join(process.cwd(), "data.json");
const data = JSON.parse(await readFile(file, "utf8"));

console.log(process.argv.slice(2), data.length);
```

## رانتایم

همان موتور V8 بدون مرورگر.



## ماژول‌های داخلی

.node:fs, node:path, node:http



## package.json

وابستگی‌ها و اسکریپت‌های npm.



## process

آرگومان‌ها و متغیرهای محیطی.



# آزمون نویسی و چرخه‌ی TDD



user.test.js

```
vi.mock("./api.js", () => ({
  getUser: vi.fn().mockResolvedValue({ id: 1 }),
}));

it("loads user", async () => {
  expect(await loadUser(1)).toEqual({ id: 1 });
});
```

**Mock**  
جایگزین وابستگی بیرونی.



**AAA**  
.Arrange, Act, Assert



دست به کد شو

# پروژه‌ی کوچک: جست‌وجوی زنده‌ی ایمن

search.js

```
const cache = new Map();
let ctrl;

async function search(q) {
  if (cache.has(q)) return render(cache.get(q));
  ctrl?.abort();
  ctrl = new AbortController();
  try {
    const url = `/api/search?q=${encodeURIComponent(q)}`;
    const res = await fetch(url, { signal: ctrl.signal });
    const data = await res.json();
    cache.set(q, data);
    render(data);
  } catch (e) {
    if (e.name !== "AbortError") showError(e);
  }
}

input.addEventListener("input", debounce((e) => search(e.target.value.trim()), 300));
```

## debounce

کم کردن تعداد درخواست.



## AbortController

لغو پاسخ قدیمی.



## کش با Map

جلوگیری از درخواست تکراری.



## خطای Abort

آن را خطای واقعی ندان.



وقت تمرین

# تمرین‌های ناهمزمانی و پلتفرم

آسان

۱ خروجی یک قطعه کد با Promise، setTimeout و کد همزمان را پیش‌بینی و بعد اجرا کنید. راهنما: Microtask پیش از Task

آسان

۲ تابع sleep(ms) بنویسید که Promise برگرداند و با await استفاده کنید. راهنما: new Promise و setTimeout

متوسط

۳ تابع retry(fn, n) بنویسید که تا n بار تلاش دوباره کند. راهنما: حلقه و try/catch

متوسط

۴ سه درخواست مستقل را موازی اجرا و زمان کل را با ترتیبی مقایسه کنید. راهنما: Promise.all و console.time

متوسط

۵ برنامه را به ماژول‌های ESM تقسیم کنید و یک ماژول را به صورت پویا بارگذاری کنید. راهنما: import()

چالشی

۶ صف وظایف با محدودیت همزمانی (حداکثر ۳ کار همزمان) پیاده‌سازی کنید. راهنما: Promise.race و صف

برای هر تمرین، آزمون Vitest بنویسید.

## خلاصه‌ی سریع

## خلاصه‌ی ناهمزمانی و پلتفرم (Cheat Sheet)

pending / fulfilled / rejected

Promise

Stack ← Microtask ← Task

Event Loop

ترتیبی یا موازی

async / await

ترکیب Promise‌ها

all allSettled

کنترل نرخ اجرا

debounce throttle

پیمایش ناهمزمان

for await...of

ماژول ESM

import / export

لغو درخواست

AbortController

lazy و انیمیشن اسکرول

IntersectionObserver

رویداد روی والد

closest + delegation

داده‌ی بزرگ مرورگر

IndexedDB

کار سنگین خارج از UI

Worker + postMessage

آزمون واحد

Vitest + mock

امنیت مرورگر

CORS CSP SameSite

# مبانی TypeScript: نوع‌ها برای کد مطمئن

نوع‌های روزمره، Narrowing، Generics، کلاس‌ها و فایل‌های تعریف نوع

● کلاس و ماژول

● Generics

● Narrowing

● نوع‌های روزمره

# TypeScript چیست و چرا؟



```

index.ts

function area(w: number, h: number): number {
  return w * h;
}
area(2, "3");
// compile error: string is not assignable to number
  
```

- Type Erasure**  نوع‌ها فقط در زمان کامپایل‌اند.
- Superset**  هر JavaScript معتبر، TypeScript معتبر است.
- تجربه‌ی بهتر**  خطای زود هنگام و تکمیل خودکار.

# نصب و تنظیم: tsconfig.json

tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "outDir": "dist",
    "skipLibCheck": true
  },
  "include": ["src"]
}
```

## strict

گروه بررسی‌های سخت‌گیرانه؛ همیشه روشن.



## module و target

نسخه‌ی خروجی و نظام ماژول.



## include و outDir

پوشه‌ی خروجی و منبع.



## اجرا

`npx tsc` یا `npx tsc --noEmit`



# نوع‌های روزمره (Everyday Types)

index.ts

```
let name: string = "Sara";
let age: number = 30;
let ok: boolean = true;
let tags: string[] = ["a", "b"];
let ids: Array<number> = [1, 2];
let anything: any = 1; // opt out of checking
let n: null = null;
function log(msg: string): void {
  console.log(msg);
}
```

**string number boolean**

نوع‌های ابتدایی با حرف کوچک.

**آرایه**

T یا &lt;Array&gt;T.

**any**

بررسی را خاموش می‌کند؛ پرهیز.

**null و void**

بدون مقدار برگشتی و مقدار خالی.



# const و Inference، Assertion

index.ts

```
let n = 5; // inferred: number
const s = "a"; // literal type "a"

const el = document.getElementById("app") as HTMLDivElement;
const user = maybeUser!; // non-null assertion

const cfg = { mode: "dark" } as const;
// { readonly mode: "dark" }
```

## Inference

نوع از مقدار اولیه استنتاج می‌شود.



## as

تحمیل نوع؛ مسئولیت با شماست.



## (non-null) !

فقط وقتی مطمئنید.



## as const

readonly و literal



# void و any، unknown، never

| نوع     | معنی                 | کاربرد                       | نکته                           |
|---------|----------------------|------------------------------|--------------------------------|
| any     | خاموش کردن بررسی نوع | مهاجرت موقت از JS            | خطرناک؛ سرایت می‌کند           |
| unknown | مقدار ناشناخته‌ی امن | ورودی خارجی (JSON، API)      | پیش از استفاده باید narrow شود |
| never   | هرگز مقداری ندارد    | تابع پرتاب‌کننده، بررسی کامل | زیرنوع همه‌ی نوع‌ها            |
| void    | بدون مقدار برگشتی    | تابع بدون return             | با undefined یکی نیست          |

قاعده: به جای any از unknown استفاده کنید و با narrowing ایمن کنید.

# نوع دهی توابع

index.ts

```
function greet(name: string, title?: string): string {
  return title ? `${title} ${name}` : name;
}

const add = (a: number, b = 1): number => a + b;
const sum = (...n: number[]): number =>
  n.reduce((a, b) => a + b, 0);

type Handler = (e: Event) => void;

function pad(v: string): string;
function pad(v: number): string;
function pad(v: string | number): string {
  return String(v).padStart(3, "0");
}
```

## اختیاری و پیش فرض

`b = 1` و `title?: string`



## Rest

`.[]n: number...`



## نوع تابع

`.void <= (e: Event)`



## Overload

چند امضا برای یک تابع.



# نوع‌های شیء: type و interface

index.ts

```
interface User {  
  readonly id: number;  
  name: string;  
  email?: string;  
  [extra: string]: unknown;  
}  
interface Admin extends User {  
  role: "admin";  
}  
type Point = { x: number; y: number };  
type Named = { name: string } & { id: number };
```

## readonly

غیرقابل تغییر پس از ساخت.



## ویژگی اختیاری

email?: string



## Index Signature

کلیدهای دلخواه با نوع مشخص.



## & و extends

وراثت و اشتراک نوع‌ها.



# Literal Types و Union

index.ts

```

type ID = string | number;
type Mode = "light" | "dark";

function setMode(m: Mode) {}
setMode("dark");           // ok
setMode("blue");           // compile error

function show(id: ID) {
  return typeof id === "string"
    ? id.toUpperCase()
    : id.toFixed(0);
}

```

## Union

یکی از چند نوع:  $A \mid B$ .

## Literal

مقدار مشخص به عنوان نوع.



## جایگزین enum

اتحاد literalها ساده و بدون کد اضافه.

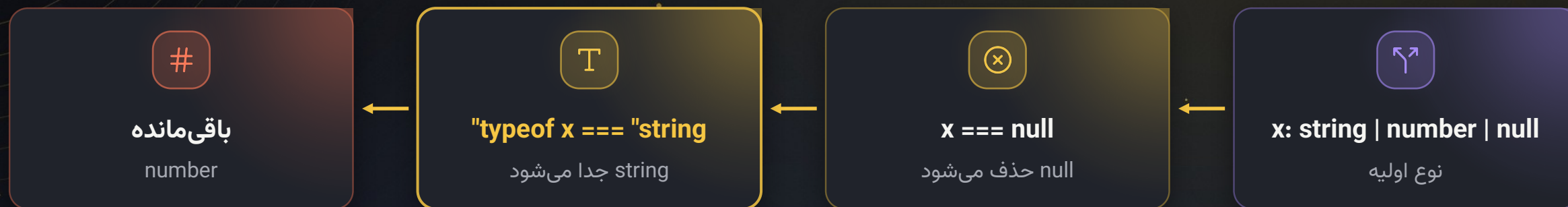


## نیاز به narrowing

پیش از استفاده از ویژگی‌های مخصوص.



# Narrowing: تنگ کردن نوع



● ● ● narrow.ts

```

function f(x: string | number | null) {
  if (x === null) return;           // equality
  if (typeof x === "string") return x.length; // typeof
  return x.toFixed(1);             // number
}
if ("swim" in animal) animal.swim(); // in
if (e instanceof HttpError) e.status; // instanceof
  
```

## ابزارها

`typeof` `instanceof` `in` `.truthiness` برای و

## جریان کد

نوع در هر شاخه فرق می‌کند.

# Discriminated Union و بررسی کامل با never

index.ts

```
type Shape =  
  | { kind: "circle"; r: number }  
  | { kind: "rect"; w: number; h: number };  
  
function area(s: Shape): number {  
  switch (s.kind) {  
    case "circle": return Math.PI * s.r ** 2;  
    case "rect": return s.w * s.h;  
    default: {  
      const _exhaustive: never = s;  
      return _exhaustive;  
    }  
  }  
}
```

## ویژگی تمایزدهنده

literal `kind` مشترک.



## switch و narrowing

در هر `case` نوع دقیق است.



## بررسی کامل

`never` حالت فراموش شده را آشکار می‌کند.



# Enum و Tuple، readonly

index.ts

```
const nums: readonly number[] = [1, 2];
type Pair = [string, number];
const p: Pair = ["age", 30];
type Named = [name: string, age?: number];

enum Role { Admin, User }           // generates code
type RoleU = "admin" | "user";       // preferred

const Colors = { red: "#f00", blue: "#00f" } as const;
type Color = keyof typeof Colors;    // "red" | "blue"
```

## Tuple

طول و نوع هر خانه مشخص.



## readonly آرایه

جلوگیری از تغییر.



## enum

کد اضافه تولید می‌کند.



## as const + keyof typeof

جایگزین سبک و ایمن.



# کلاس‌ها در TypeScript

index.ts

```
interface Drawable { draw(): void }

abstract class Shape {
  constructor(public readonly name: string, protected id = 0) {}
  abstract area(): number;
}

class Circle extends Shape implements Drawable {
  constructor(private r: number) { super("circle"); }
  override area() { return Math.PI * this.r ** 2; }
  draw(): void {}
}
```

## public/private/protected

فقط در زمان کامپایل؛ برای خصوصی واقعی #.



## Parameter Properties

تعریف و انتساب ویژگی در سازنده.



## implements و abstract

قرارداد و کلاس پایه.



## override

تضمین بازنویسی واقعی.



# ۱ Generics: تابع و محدودیت

index.ts

```
function first<T>(arr: T[]): T | undefined {
  return arr[0];
}
first([1, 2]); // number | undefined

function pick<T, K extends keyof T>(o: T, k: K): T[K] {
  return o[k];
}
pick({ a: 1, b: "x" }, "b"); // string

function len<T extends { length: number }>(x: T) {
  return x.length;
}
```

## پارامتر نوع

<T>؛ نوع همراه ورودی حفظ می‌شود.



## محدودیت extends

T extends { length: number }



## keyof

K extends keyof T



## استنتاج

اغلب لازم نیست T را بنویسید.



# interface: ۲ Generics، کلاس و پیش فرض

index.ts

```
interface Box<T> { value: T }

class Stack<T> {
  private items: T[] = [];
  push(x: T) { this.items.push(x); }
  pop(): T | undefined { return this.items.pop(); }
}

const s = new Stack<number>();

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

generic interface و کلاس

&lt;Box&gt;T و &lt;Stack&gt;T



نوع پیش فرض

E = Error



الگوی Result

خطا بخشی از نوع می شود.



# type یا interface؟

| ویژگی                        | interface                   | type                          |
|------------------------------|-----------------------------|-------------------------------|
| ترکیب و وراثت                | <b>extends</b>              | & (اشتراک)                    |
| Declaration Merging          | بله؛ چند اعلان ادغام می‌شود | خیر؛ تعریف تکراری خطاست       |
| Tuple و Union                | خیر                         | بله                           |
| نوع‌های Conditional و Mapped | خیر                         | بله                           |
| پیام خطا و عملکرد            | معمولاً خواناتر             | برای ترکیب‌های عمیق خوب       |
| پیشنهاد                      | قرارداد اشیا و کلاس‌ها      | ترکیب‌ها و نوع‌های محاسبه‌شده |

یک قرارداد تیمی بگذارید و در کل پروژه ثابت بمانید.

# Excess Property Check و Structural Typing

index.ts

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

const p3 = { x: 1, y: 2, z: 3 };
const a: Point = p3;           // ok: same shape
const b: Point = { x: 1, y: 2, z: 3 };
// compile error: excess property 'z'

class A { x = 1 }
class B { x = 2 }
const c: A = new B();          // ok: same shape
```

## بر اساس شکل

نام نوع مهم نیست؛ شکل مهم است.



## Excess Property

فقط برای object literal مستقیم.



## سازگاری کلاس‌ها

کلاس‌های هم‌شکل سازگارند.



# ماژول‌ها و فایل‌های تعریف نوع (d.ts.)

index.ts

```
import type { User } from "../types";
import { api } from "../api.js";
export type { User };
```

```
// globals.d.ts
declare const __APP_VERSION__: string;
declare module "*.svg" {
  const url: string;
  export default url;
}
```

```
// npm i -D @types/node
```

## import type

در خروجی حذف می‌شود.



## d.ts.

فقط اعلان نوع، بدون کد.



## \*/types@

نوع کتابخانه‌های بدون نوع.



## declare

اعلان مقدار یا ماژول سراسری.



# خواندن پیام‌های خطای TypeScript

| کد      | پیام (خلاصه)                            | معنی و راه حل                                                    |
|---------|-----------------------------------------|------------------------------------------------------------------|
| TS2322  | 'Type 'X' is not assignable to type 'Y' | نوع مقدار با نوع موردانتظار نمی‌خواند؛ مقدار یا نوع را درست کنید |
| TS2339  | Property does not exist on type         | ویژگی روی نوع نیست؛ narrowing بدهید یا نوع را کامل کنید          |
| TS2345  | Argument of type ... is not assignable  | آرگومان با پارامتر سازگار نیست؛ نوع آرگومان را بررسی کنید        |
| TS18048 | 'x' is possibly 'undefined'             | بررسی <b>undefined</b> بنویسید یا از ?? و ?. استفاده کنید        |
| TS7006  | Parameter implicitly has an 'any' type  | برای پارامتر نوع بنویسید                                         |

خط آخر پیام خطا معمولاً علت اصلی است؛ از آنجا شروع کنید.

## وقت تمرین

## تمرین‌های مبانی TypeScript

آسان

۱ برای یک تابع formatUser نوع ورودی (interface) و نوع خروجی را بنویسید. راهنما: interface و ?:

آسان

۲ نوع Status را با literal union بسازید و تابعی بنویسید که برای هر مقدار پیام برگرداند. راهنما: switch و never

متوسط

۳ تابع generic به نام groupBy بنویسید که آرایه را بر اساس یک کلید گروه‌بندی کند. راهنما: K extends keyof T

متوسط

۴ نوع &lt;Result&lt;T, E&gt; را بنویسید و تابع parseJSON با آن ایمن کنید. راهنما: union و narrowing

متوسط

۵ کلاس abstract Shape با دو کلاس فرزند بنویسید و آرایه‌ای از Shape‌ها را پردازش کنید. راهنما: abstract و override

چالشی

۶ یک فایل JavaScript کوچک را به TypeScript مهاجرت دهید و تمامی خطاهای strict را رفع کنید. راهنما: tsc --noEmit ابتدا

همه‌ی تمرین‌ها باید با گزینه‌ی strict بدون خطا کامپایل شوند.

## خلاصه‌ی سریع

## خلاصه‌ی مبانی TypeScript (Cheat Sheet)

اختیاری و فقط خواندنی

`x?: T / readonly`

annotation نوع

`x: string`

Literal Union

`"a" | "b"`

Union

`A | B`

ابزارهای Narrowing

`typeof in instanceof`

ناشناخته‌ی امن / غیرممکن

`unknown / never`

شکل اشیا و ترکیب‌ها

`interface / type`

Discriminated Union

`kind: "circle"`

کلیدهای نوع

`keyof T`

Generic با محدودیت

`<T extends X>`

قرارداد کلاس

`class implements`

readonly و literal

`as const`

بررسی سخت‌گیرانه

`strict: true`

نوع بدون کد

`import type`

# TypeScript پیشرفته: برنامه‌نویسی در سطح نوع

Utility Types، Template Literal، Mapped، Conditional و الگوهای حرفه‌ای

ابزار و مهاجرت

Type Guards

Utility Types

Mapped و Conditional

# Indexed Access و keyof، typeof

types.ts

```
type User = { id: number; name: string; tags: string[] };
type Keys = keyof User;           // "id" | "name" | "tags"
type Name = User["name"];         // string
type Tag = User["tags"][number]; // string

const cfg = { port: 3000, host: "x" };
type Cfg = typeof cfg;             // { port: number; host: string }
type K = keyof typeof cfg;         // "port" | "host"
```

keyof

اتحاد کلیدهای نوع.



typeof (نوع)

نوع یک مقدار موجود.



T[K]

نوع ویژگی؛ [number] برای عنصر آرایه.



# infer و Conditional Types

types.ts

```
type IsString<T> = T extends string ? true : false;
type A = IsString<"x">; // true

type Unwrap<T> = T extends Promise<infer U> ? U : T;
type B = Unwrap<Promise<number>>; // number

type ElementOf<T> = T extends (infer E)[] ? E : never;
type C = ElementOf<string[]>; // string
```

**T extends U ? X : Y**

انتخاب نوع بر اساس شرط.



**infer**

شکار بخشی از نوع در شرط.



**پایه Utility ها**

ReturnType، Awaited و ...



# Distributive Conditional و حذف با never

types.ts

```
type TArray<T> = T extends unknown ? T[] : never;
type D = TArray<string | number>; // string[] | number[]

type TArray2<T> = [T] extends [unknown] ? T[] : never;
type E = TArray2<string | number>; // (string | number)[]

type NonNull<T> = T extends null | undefined ? never : T;
type F = NonNull<string | null>; // string

type OnlyStrings<T> = T extends string ? T : never;
type G = OnlyStrings<"a" | 1 | "b">; // "a" | "b"
```

## Distributive

روی هر عضو union جدا اعمال می‌شود.



## extends [U] [T]

خاموش کردن توزیع.



## فیلتر با never

never در union ناپدید می‌شود.



# Mapped Types: تبدیل نوع‌ها

types.ts

```
type Optional<T> = { [K in keyof T]?: T[K] };
type Mutable<T> = { -readonly [K in keyof T]: T[K] };
type Required2<T> = { [K in keyof T]-?: T[K] };

type Getters<T> = {
  [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K];
};
type G = Getters<{ name: string; age: number }>;
// { getName: () => string; getAge: () => number }
```

## [K in keyof T]

پیمایش کلیدها و ساخت نوع تازه.



## + - readonly ?

افزودن یا حذف مدیفایر.



## as (Key Remapping)

بازنویسی نام کلیدها.



# Template Literal Types

types.ts

```
type Ev = "click" | "focus";
type Handler = `on${Capitalize<Ev>}`; // "onClick" | "onFocus"

type Route = `/users/${number}`;
const r: Route = "/users/42";
type Px = `${number}px`;

type Split<S extends string> =
  S extends `${infer A}.${infer B}` ? [A, ...Split<B>] : [S];
type P = Split<"a.b.c">; // ["a", "b", "c"]
```

## ترکیب رشته‌ها

`on${Capitalize<Ev>}`

## infer در رشته

تجزیه‌ی رشته در سطح نوع.



## کاربرد

مسیرها، رویدادها و کلیدهای قراردادی.



# ۱ Utility Types: تبدیل شکل اشیا

| ابزار                           | کار                        | مثال                                         |
|---------------------------------|----------------------------|----------------------------------------------|
| <code>Partial&lt;T&gt;</code>   | همه‌ی ویژگی‌ها اختیاری     | <code>Partial&lt;User&gt;</code>             |
| <code>Required&lt;T&gt;</code>  | همه‌ی ویژگی‌ها اجباری      | <code>Required&lt;Config&gt;</code>          |
| <code>Readonly&lt;T&gt;</code>  | همه‌ی ویژگی‌ها فقط‌خواندنی | <code>Readonly&lt;State&gt;</code>           |
| <code>Pick&lt;T, K&gt;</code>   | فقط کلیدهای انتخاب‌شده     | <code>Pick&lt;User, "id"   "name"&gt;</code> |
| <code>Omit&lt;T, K&gt;</code>   | همه به‌جز کلیدهای حذف‌شده  | <code>Omit&lt;User, "password"&gt;</code>    |
| <code>Record&lt;K, V&gt;</code> | نقشه‌ی کلید به مقدار       | <code>Record&lt;string, number&gt;</code>    |

اغلب لازم نیست نوع جدید بنویسید؛ از ترکیب این ابزارها بسازید.

## ۲ Utility Types: اتحادها و توابع

| ابزار                             | کار                   | مثال و نتیجه                                               |
|-----------------------------------|-----------------------|------------------------------------------------------------|
| <code>Exclude&lt;U, X&gt;</code>  | حذف اعضا از union     | <code>"Exclude&lt;"a"   "b", "a"&gt; ← "b"</code>          |
| <code>Extract&lt;U, X&gt;</code>  | نگه داشتن اعضای مشترک | <code>Extract&lt;string   1, string&gt; ← string</code>    |
| <code>NonNullable&lt;T&gt;</code> | حذف null و undefined  | <code>NonNullable&lt;string   null&gt; ← string</code>     |
| <code>ReturnType&lt;F&gt;</code>  | نوع خروجی تابع        | <code>ReturnType&lt;typeof fn&gt;</code>                   |
| <code>Parameters&lt;F&gt;</code>  | تاپل پارامترهای تابع  | <code>Parameters&lt;typeof fn&gt;</code>                   |
| <code>Awaited&lt;T&gt;</code>     | نوع نهایی Promise     | <code>Awaited&lt;Promise&lt;number&gt;&gt; ← number</code> |

# Assertion Function و Type Guard

types.ts

```
function isString(x: unknown): x is string {  
    return typeof x === "string";  
}  
  
function assertDefined<T>(x: T | undefined): asserts x is T {  
    if (x === undefined) throw new Error("undefined");  
}  
  
const isUser = (v: unknown): v is User =>  
    typeof v === "object" && v !== null && "id" in v;  
  
const data: unknown = JSON.parse(raw);  
if (isUser(data)) console.log(data.id);
```

**x is T**

تنگ کردن نوع در شاخه‌ی true.

**asserts x is T**

پرتاب خطا؛ پس از آن نوع دقیق است.

**مسئولیت با شماست**

کامپایلر درستی بدنه‌ی guard را بررسی نمی‌کند.



# الگوهای پیشرفته‌ی Brand و Generic: Factory

types.ts

```
type Brand<T, B> = T & { readonly __brand: B };
type UserId = Brand<number, "UserId">;
type OrderId = Brand<number, "OrderId">;

const uid = 5 as UserId;
function getUser(id: UserId) {}
getUser(uid);           // ok
// getUser(5);          // compile error

function create<T>(C: new (...a: any[]) => T): T {
  return new C();
}
```

## Branded Type

جلوگیری از اشتباه گرفتن مقادیرهای هم‌ساختار.



## Factory Generic

کار با نوع سازنده.



## فقط در زمان کامپایل

نشانه‌ی brand در اجرا وجود ندارد.



# نوع‌دهی ایمن به API و Async

types.ts

```
type ApiResult<T> =  
  | { ok: true; data: T }  
  | { ok: false; status: number };  
  
async function api<T>(url: string): Promise<ApiResult<T>> {  
  const res = await fetch(url);  
  if (!res.ok) return { ok: false, status: res.status };  
  return { ok: true, data: (await res.json()) as T };  
}  
  
const r = await api<User[]>("/api/users");  
if (r.ok) console.log(r.data.length);  
else console.error(r.status);
```

## ApiResponse

حالت موفق و ناموفق در نوع.



## ok با narrowing

پس از بررسی ok نوع دقیق است.



## as T

اعتبارسنجی نمی‌کند؛ از schema استفاده کنید.



# Module Augmentation و Declaration Merging

types.ts

```
interface Window { appVersion: string } // merges with global

declare global {
  interface Array<T> { last(): T | undefined }
}
Array.prototype.last = function () {
  return this[this.length - 1];
};

declare module "express" {
  interface Request { user?: { id: number } }
}
```

## ادغام interface

اعلان‌های هم‌نام ترکیب می‌شوند.



## declare global

گسترش نوع سراسری.



## declare module

گسترش نوع کتابخانه.



## فقط نوع

پیاده‌سازی را خودتان بنویسید.



# using و satisfies, const Generic

types.ts

```
const routes = {  
  home: "/",  
  user: "/u/:id",  
} satisfies Record<string, string>;  
routes.home; // type "/" (literal kept)  
  
function tuple<const T extends readonly unknown[]>(x: T) {  
  return x;  
}  
  
const t = tuple([1, "a"]); // readonly [1, "a"]  
  
{ using res = openFile(); } // disposed at block end
```

## satisfies

بررسی نوع بدون از دست دادن literal.



## const Generic

استنتاج دقیق‌تر.



## using

آزادسازی خودکار منبع؛ وابسته به runtime.



# Decorator ها (استاندارد 5 TypeScript)

types.ts

```
function logged<This, Args extends unknown[], R>(
  fn: (this: This, ...a: Args) => R,
  ctx: ClassMethodDecoratorContext,
) {
  return function (this: This, ...a: Args): R {
    console.log("call", String(ctx.name));
    return fn.call(this, ...a);
  };
}

class Api {
  @logged
  get(id: number) { return id; }
}
```

## دکوراتور متد

تابعی که متد را می‌پیچد.



## ctx

اطلاعات عضو دکوراته شده.



## کاربرد

Angular، NestJS و ORM ها.



# Variance و سازگاری نوع تابع

types.ts

```
type Animal = { name: string };
type Dog = { name: string; bark(): void };

let fa: (a: Animal) => void = (d: Dog) => {};
// compile error (strictFunctionTypes)
let fd: (d: Dog) => void = (a: Animal) => {}; // ok

const dogs: Dog[] = [];
const animals: Animal[] = dogs; // ok (covariant, unsound)
```

## Contravariance

ورودی کلی‌تر امن است.



## strictFunctionTypes

اعمال قاعده‌ی بالا روی توابع.



## آرایه Covariant

ناایمن ولی پذیرفته شده.



# گزینه‌های strict و مهاجرت تدریجی از JavaScript

| گزینه                                     | اثر                                                        | چرا؟                                    |
|-------------------------------------------|------------------------------------------------------------|-----------------------------------------|
| <code>noImplicitAny</code>                | نوع any ضمنی خطاست                                         | جلوگیری از بی‌نوعی                      |
| <code>strictNullChecks</code>             | <code>null</code> و <code>undefined</code> جدا از بقیه‌اند | حذف رایج‌ترین باگ                       |
| <code>strictFunctionTypes</code>          | بررسی سخت‌گیرانه‌ی پارامتر توابع                           | سازگاری امن                             |
| <code>strictPropertyInitialization</code> | ویژگی کلاس باید مقداردهی شود                               | جلوگیری از <code>undefined</code> پنهان |
| <code>noUncheckedIndexedAccess</code>     | <code>arr[i]</code> ممکن است <code>undefined</code> باشد   | ایمنی اندیس                             |
| <code>allowJs + checkJs</code>            | بررسی فایل‌های JS با JSDoc                                 | مهاجرت تدریجی                           |

مسیر مهاجرت: `checkJs` ← `allowJs` ← تبدیل فایل‌ها به `strict` ← `ts` کامل.

# اکوسیستم ابزارها



## بررسی نوع ≠ ساخت

ابزارهای سریع فقط نوع را حذف می‌کنند؛ بررسی را **tsc** انجام می‌دهد.

CI

**tsc --noEmit** و ESLint را در هر تغییر اجرا کنید.

# نوع ها در زمان اجرا نیستند: اعتبارسنجی و آزمون نوع

types.ts

```
import { z } from "zod";
const User = z.object({ id: z.number(), email: z.string().email() });
type User = z.infer<typeof User>;
const parsed = User.parse(await res.json()); // runtime + typed

type Equal<A, B> =
  (<T>() => T extends A ? 1 : 2) extends (<T>() => T extends B ? 1 : 2)
    ? true : false;
type Expect<T extends true> = T;
type t1 = Expect<Equal<ReturnType<typeof fn>, User>>;
```

## Type Erasure

نوع ها در اجرا وجود ندارند.



## Schema

اعتبارسنجی اجرایی و نوع از یک منبع.



## Type Test

آزمون نوع با `Expect` و `Equal`.



دست به کد شو

## پروژه‌ی کوچک: Event Emitter با نوع ایمن

types.ts

```

type Events = { login: { id: number }; logout: void };

class Emitter<E extends Record<string, unknown>> {
  private h: { [K in keyof E]?: ((p: E[K]) => void)[] } = {};
  on<K extends keyof E>(k: K, fn: (p: E[K]) => void) {
    (this.h[k] ??= []).push(fn);
  }
  emit<K extends keyof E>(k: K, p: E[K]) {
    this.h[k]?.forEach((f) => f(p));
  }
}

const bus = new Emitter<Events>();
bus.on("login", (p) => console.log(p.id));
bus.emit("login", { id: 1 });

```

**K extends keyof E**

نام رویداد فقط از نقشه.

**E[K]**

نوع payload هر رویداد.

**ایمنی**

emit اشتباه کامپایل نمی‌شود.



دست به کد شو

# پروژه‌ی کوچک: Store ساده با Reducer تایپ‌شده

types.ts

```

type Action = { type: "inc" } | { type: "add"; n: number };
type State = { count: number };

const reducer = (s: State, a: Action): State => {
  switch (a.type) {
    case "inc": return { count: s.count + 1 };
    case "add": return { count: s.count + a.n };
  }
};

function createStore(init: State) {
  let state = init;
  const subs = new Set<() => void>();
  return {
    get: () => state,
    dispatch(a: Action) { state = reducer(state, a); subs.forEach((f) => f()); },
    subscribe(f: () => void) { subs.add(f); return () => subs.delete(f); },
  };
};

```

100

## Action = union

حالت‌های مجاز مشخص و بسته.



## reducer کامل

فراموشی یک case خطای کامپایل می‌دهد.



## subscribe

اطلاع‌رسانی پس از هر تغییر.



## وقت تمرین

## تمرین‌های TypeScript پیشرفته

متوسط

نوع `<DeepReadonly<T` بنویسید که همه‌ی سطح‌ها را فقط خواندنی کند. راهنما: `Mapped` و بازگشت

متوسط

نوع `<PickByValue<T, V` بنویسید که فقط کلیدهایی با مقدار نوع `V` را نگه دارد. راهنما: `as` + `Mapped` + شرط

متوسط

نوع `Awaited` ساده خودتان را با `infer` پیاده کنید. راهنما: `<T extends Promise<infer U`

چالشی

نوع `<CamelCase<S` را برای تبدیل `snake_case` به `camelCase` بنویسید. راهنما: `infer` و `Template Literal`

چالشی

یک `fetch-wrapper` تایپ‌شده بنویسید که نوع پاسخ را از یک نقشه‌ی مسیر به نوع بخواند. راهنما: `Record` و `Generic`

چالشی

برای نوع‌های بالا `type test` با `Expect` و `Equal` بنویسید. راهنما: نوع `Equal`برای هر نوع، حالت‌های مرزی (`union`، `never`، `null`) را هم بسنجید.

## خلاصه‌ی سریع

## خلاصه‌ی TypeScript پیشرفته (Cheat Sheet)

`typeof x`

نوع یک مقدار

`infer U`

شکار نوع در شرط

`as `get${...}``

Key Remapping

`Partial Pick Omit`

تبدیل شکل اشیا

`x is T / asserts`

Type Guard

`satisfies`

بررسی بدون گسترش

`strict + noEmit`

سخت‌گیری و CI

`keyof T / T[K]`

کلیدها و نوع ویژگی

`T extends U ? X : Y`

نوع شرطی

`{ [K in keyof T]: ... }`

Mapped Type

``on${Capitalize<E>}``

Template Literal

`Exclude ReturnType Awaited`

اتحاد و تابع

`Brand<T, B>`

نوع نشان‌دار

`declare global`

ادغام و گسترش

# بعد از این دوره: کجا برویم؟

## الگوهای طراحی و معماری



Clean Architecture، SOLID و الگوهای رایج.

## بک‌اند با Node



NestJS یا Express با TypeScript؛ API و پایگاه داده.

## فریم‌ورک‌های فرانت‌اند



React، Angular یا Vue با TypeScript؛ ساخت برنامه‌های بزرگ.

## ابزار تیمی



CI/CD، Monorepo و Code Review.

## آزمون پیشرفته



آزمون یکپارچه، E2E و Contract Test.

## عملکرد پیشرفته



پروفایل، Web Vitals و بهینه‌سازی بسته.

# برنامه‌ی پیشنهادی دوازده جلسه‌ای

۱ مدل اجرا، Hoisting، نوع‌ها و coercion

تمرین: تمرین‌های نوع و برابری

۲ Closure، this و توابع پیشرفته

تمرین: memoize و شمارنده

۳ شیء، پروتوتایپ و کلاس

تمرین: Stack با private#

۴ Regex و Iterator، Generator، Map/Set

تمرین: range و iterable

۵ خطا، Proxy و کیفیت کد

تمرین: آزمون با Vitest

۶ Event Loop، Promise و async/await

تمرین: sleep و retry

۷ ماژول‌ها، DOM، fetch و عملکرد

تمرین: جست‌وجوی زنده

۸ امنیت، Node.js و آزمون

تمرین: ابزار CLI کوچک

۹ مبانی ۱ TypeScript: نوع‌ها و Narrowing

تمرین: تابع‌ها و union

۱۰ مبانی ۲ TypeScript: Generics، کلاس و ماژول

تمرین: Result و groupBy

۱۱ TypeScript پیشرفته: Conditional و Mapped

تمرین: DeepReadonly

۱۲ Utility، Type Guard و پروژه‌ی نهایی

تمرین: API Client تایپ‌شده

دست به کد شو

# پروژه‌ی پایانی: API Client تایپ‌شده با رابط کاربری

## منطق ناهمزمان



- fetch با timeout و AbortController
- کش و retry
- debounce برای جست‌وجو

## نوع‌ها



- نقشه‌ی مسیرها به نوع پاسخ
- Result و Type Guard
- بدون any و با strict

## تحویل



- مخزن با Git و README
- اسکریپت‌های npm
- نمونه‌ی اجرا

## آزمون



- آزمون واحد با Vitest و mock
- آزمون نوع با Expect/Equal
- tsc --noEmit در CI

جمع‌بندی دوره



مجتبی سجادفر

Mojtaba Sajadfar

# زبان را عمیق بفهمید؛ نوع‌ها را هوشمندانه بنویسید.

آزمون

پلتفرم و امنیت

ناهمزمانی

هسته‌ی JavaScript

پروژه‌ی واقعی

Utility Types

Generics

نوع‌های TypeScript

نمونه‌کارها و ادامه‌ی مسیر را در پرتفولیو ببینید

[immsr.ir](https://immsr.ir)

