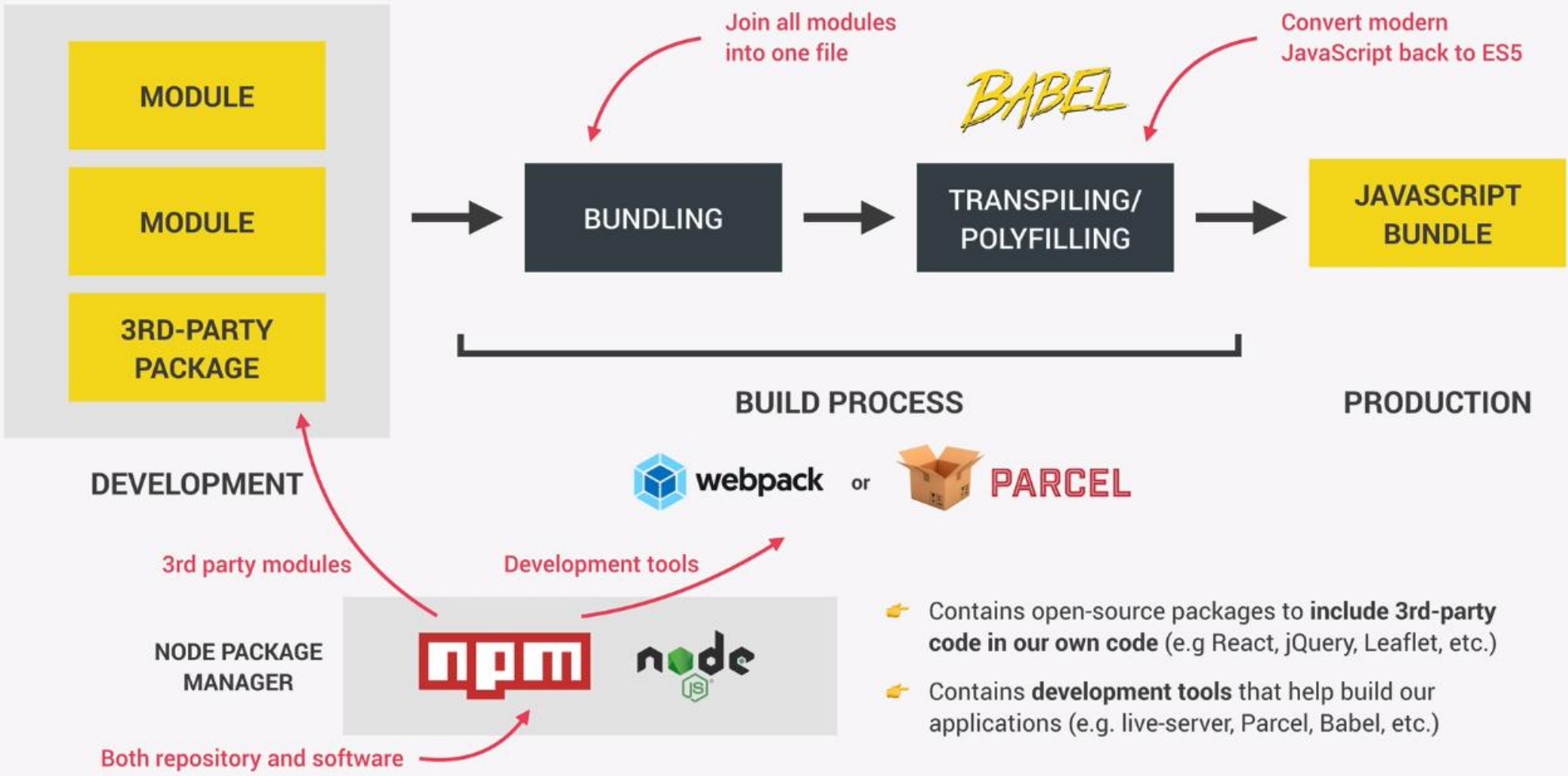
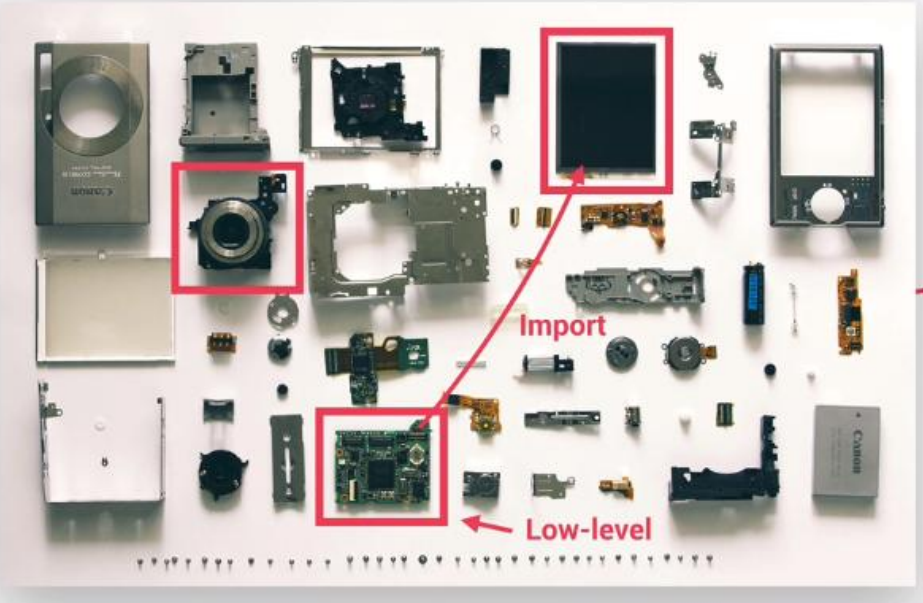
MODERN JAVASCRIPT DEVELOPMENT



AN OVERVIEW OF MODULES



WHY MODULES?

- **Compose software:** Modules are small building blocks that we put together to build complex applications;
- **Isolate components:** Modules can be developed in isolation without thinking about the entire codebase;
- **Abstract code:** Implement low-level code in modules and import these abstractions into other modules;
- **Organized code:** Modules naturally lead to a more organized codebase;
- **Reuse code:** Modules allow us to easily reuse the same code, even across multiple projects.

IMPORT (DEPENDENCY)

MODULE

```
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

Module code

EXPORT (PUBLIC API)

NATIVE JAVASCRIPT (ES6) MODULES

ES6 MODULES

Modules stored in files, **exactly one module per file.**

```
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

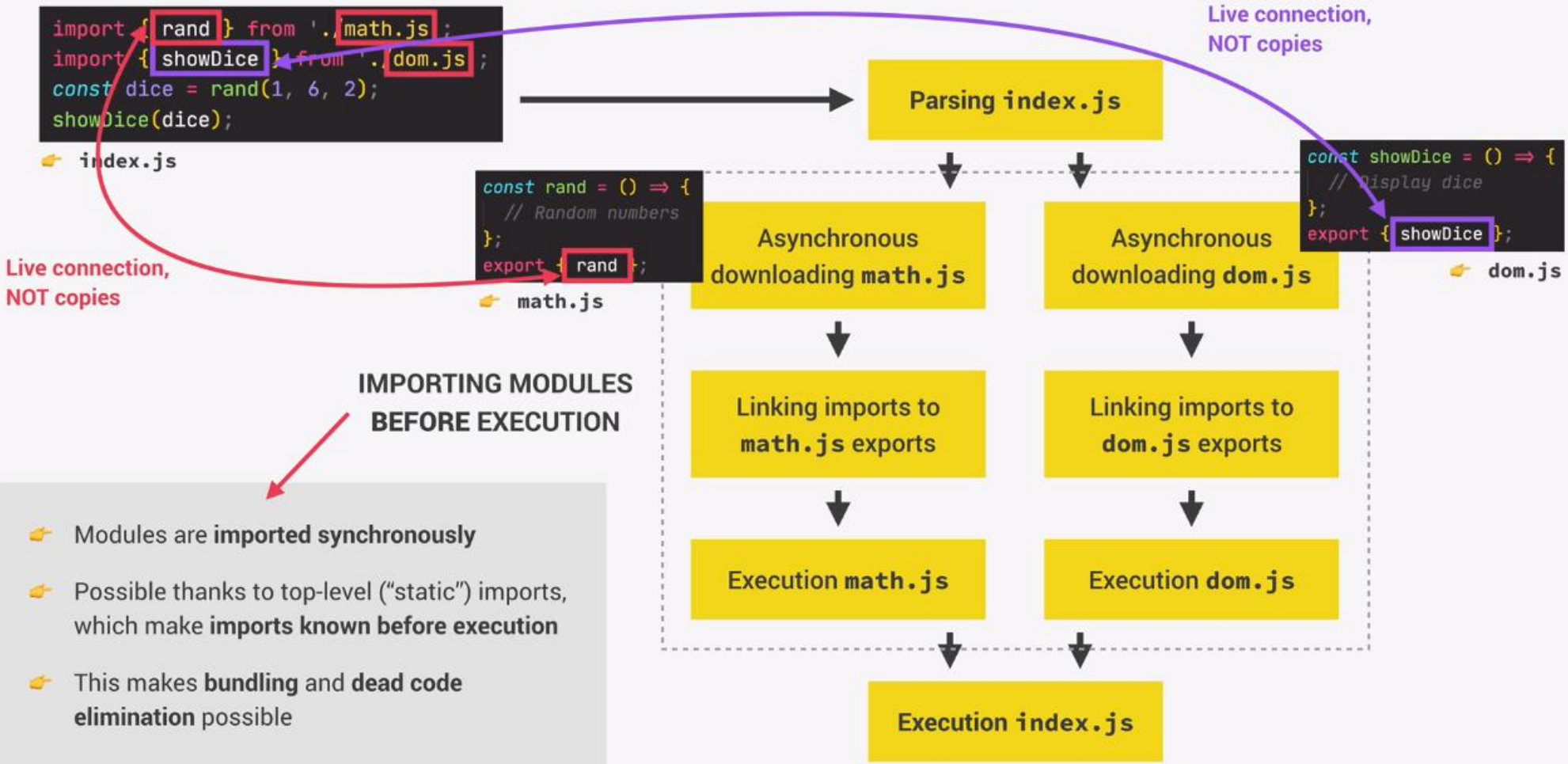
`import` and `export` syntax

	ES6 MODULE	SCRIPT
👉 Top-level variables	Scoped to module	Global
👉 Default mode	Strict mode	"Sloppy" mode
👉 Top-level `this`	`undefined`	`window`
👉 Imports and exports	✅ YES	🚫 NO
👉 HTML linking	`<script type="module">`	`<script>`
👉 File downloading	Asynchronous	Synchronous

☝ Need to happen at top-level

Imports are hoisted!

HOW ES6 MODULES ARE IMPORTED



REVIEW: MODERN AND CLEAN CODE

READABLE CODE

- Write code so that **others** can understand it
- Write code so that **you** can understand it in 1 year
- Avoid too "clever" and overcomplicated solutions
- Use descriptive variable names: **what they contain**
- Use descriptive function names: **what they do**

GENERAL

- Use DRY principle (refactor your code)
- Don't pollute global namespace, encapsulate instead
- Don't use `var`
- Use strong type checks (=== and `!==`)

FUNCTIONS

- Generally, functions should do **only one thing**
- Don't use more than 3 function parameters
- Use default parameters whenever possible
- Generally, return same data type as received
- Use arrow functions when they make code more readable

OOP

- Use ES6 classes
- Encapsulate data and **don't mutate** it from outside the class
- Implement method chaining
- Do **not** use arrow functions as methods (in regular objects)

REVIEW: MODERN AND CLEAN CODE

AVOID NESTED CODE

- Use early `return` (guard clauses)
- Use ternary (conditional) or logical operators instead of `if`
- Use multiple `if` instead of `if/else-if`
- Avoid `for` loops, use array methods instead
- Avoid callback-based asynchronous APIs

ASYNCHRONOUS CODE

- Consume promises with async/await for best readability
- Whenever possible, run promises in **parallel** (`Promise.all`)
- Handle errors and promise rejections

IMPERATIVE VS. DECLARATIVE CODE

Two fundamentally different ways of writing code (paradigms)

IMPERATIVE

- Programmer explains **"HOW to do things"**
- We explain the computer *every single step* it has to follow to achieve a result
- **Example:** Step-by-step recipe of a cake

```
const arr = [2, 4, 6, 8];
const doubled = [];
for (let i = 0; i < arr.length; i++)
  doubled[i] = arr[i] * 2;
```

DECLARATIVE

- Programmer tells **"WHAT do do"**
- We simply *describe* the way the computer should achieve the result
- The **HOW** (step-by-step instructions) gets abstracted away
- **Example:** Description of a cake

```
const arr = [2, 4, 6, 8];
const doubled = arr.map(n ⇒ n * 2);
```

FUNCTIONAL PROGRAMMING PRINCIPLES

FUNCTIONAL PROGRAMMING

- **Declarative** programming paradigm
- Based on the idea of writing software by combining many **pure functions**, avoiding **side effects** and **mutating** data
- **Side effect:** Modification (mutation) of any data **outside** of the function (mutating external variables, logging to console, writing to DOM, etc.)
- **Pure function:** Function without side effects. Does not depend on external variables. **Given the same inputs, always returns the same outputs**.
- **Immutability:** State (data) is **never** modified! Instead, state is **copied** and the copy is mutated and returned.
- **Examples:** 

FUNCTIONAL PROGRAMMING TECHNIQUES

- Try to avoid data mutations
- Use built-in methods that don't produce side effects
- Do data transformations with methods such as `.map()`, `.filter()` and `.reduce()`
- Try to avoid side effects in functions: this is of course not always possible!

DECLARATIVE SYNTAX

- Use array and object destructuring
- Use the spread operator (`...`)
- Use the ternary (conditional) operator
- Use template literals

COMPONENTS OF ANY ARCHITECTURE

BUSINESS LOGIC	STATE	HTTP LIBRARY	APPLICATION LOGIC (ROUTER)	PRESENTATION LOGIC (UI LAYER)
Code that **solves the actual business problem**;	Essentially **stores all the data** about the application	Responsible for making and receiving AJAX requests	Code that is only concerned about the **implementation of application itself**;	Code that is concerned about the **visible part** of the application
Directly related to what business does and what it needs;	Should be the "single source of truth"	Optional but almost always necessary in real-world apps	Handles navigation and UI events	Essentially displays application state
Example: sending messages, storing transactions, calculating taxes, ...	UI should be kept in sync with the state			
	State libraries exist