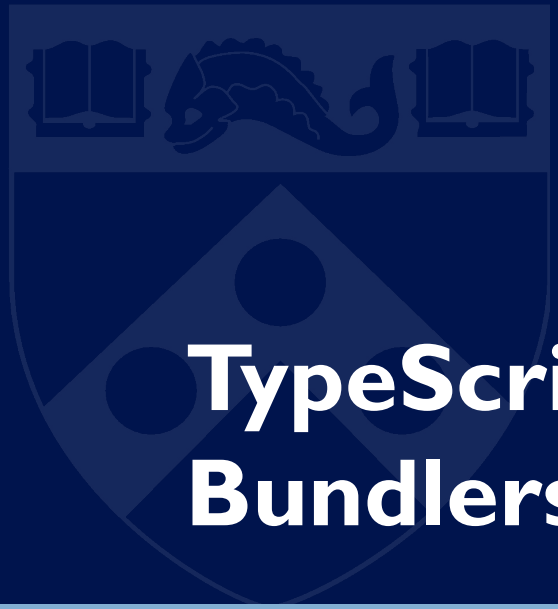


TypeScript, React, and Web Bundlers

CIS 1962 (Fall 2025)
October 27th, 2025

Penn Engineering

Lesson Plan

TypeScript

React & Web Bundlers

PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

PollEv.com/voravichs673

Narrowing

What ways can we use type guards and refine types to be more specific?

Control Flow Analysis

When working with control flow constructs like if/else, ternaries, and loops, type analysis is occurring to check how types affect certain execution paths.

When control flow checks for a certain type before executing code, that check is called a **type guard** or **type assertion**.

Narrowing unknown

The unknown type enforces that you need to narrow or assert that your type is a certain type before you move on.

```
function processData(data: unknown) {
     console.log(data.toLowerCase());
     // Error: 'data' is of type 'unknown'.
}
```

```
function processData(data: unknown) {
     if (typeof data === "string") {
          console.log(data.toLowerCase());
     }
}
```

If Statements: typeof

`typeof`: checks the type of some provided value, returns a string

```
function printLengthOrNumber(x: string | number) {
    if (typeof x === "string") {
        console.log(x.length);
    } else {
        console.log(x.toFixed(2));
    }
}
```

If Statements: "property" in Object

`"property" in Object`: used to narrow types between possible objects by specifying a property specific to one of them

```
type Dog = { bark(): void };
type Cat = { meow(): void };

function speak(animal: Dog | Cat) {
    if ("bark" in animal) {
        animal.bark();
    } else {
        animal.meow();
    }
}
```

If Statements: instanceof

`instanceof`: used to narrow types between existing classes, checks if a value is a certain class

```
class Car { drive() { console.log("Vroom"); } }
class Bike { pedal() { console.log("Pedal pedal"); } }

function move(vehicle: Car | Bike) {
  if (vehicle instanceof Car) {
    vehicle.drive();
  } else {
    vehicle.pedal();
  }
}
```

Custom Type Guards

You can define a **custom type guard** as some function with a return type that specifies the type you want.

```
type Square = { kind: "square", size: number };
type Circle = { kind: "circle", radius: number };

function isSquare(shape: Square | Circle): shape is Square {
     return shape.kind === "square";
}

function area(shape: Square | Circle) {
     if (isSquare(shape)) {
          return shape.size ** 2;
     } else {
          return Math.PI * shape.radius ** 2;
     }
}
```

Assertion Functions

You can use an **assertion function** to throw an error if a type is incorrect instead:

```
function assertIsString(value: unknown): asserts value is string {
  if (typeof value !== "string") {
    throw new Error("Not a string");
  }
}

function shout(x: unknown) {
  assertIsString(x);
  console.log(x.toUpperCase());
}

shout("hey"); // HEY
shout(42); // throws an error
```

Discriminated Unions

When working with discriminated unions, you must make sure all members can be discriminated by some property, and that you **exhaustively handle all cases**. (under "strict" mode)

```ts
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; size: number }
  | { kind: "rectangle"; width: number; height: number };

// Good practice: catch unhandled cases
function assertNever(x: never): never {
  throw new Error("Unexpected object: " + x);
}

function area(shape: Shape): number {
  switch (shape.kind) {
    case "circle": return Math.PI * shape.radius ** 2;
    case "square": return shape.size ** 2;
    case "rectangle": return shape.width * shape.height;
    default: return assertNever(shape);
  }
}
```

POLL QUESTION

What should go in the blank in the following code?

```ts
type A = { kind: "a", value: number };
type B = { kind: "b", text: string };

function handle(x: A | B) {
  //       ↓↓↓
  if (__________) {
    console.log(x.text.toLowerCase());
  }
}
```

`x.kind === "b"`

POLL QUESTION

What expression can be used in the blank to narrow APIResponse to the correct type?

```ts
type SuccessStatus = { status: 'ok'; data: { value: number } }
type ErrorStatus = { status: 'error'; errorCode: string }
type ApiResponse = SuccessStatus | ErrorStatus

function processResponse(response: ApiResponse): string {
  //      ↓↓↓
  if (_____________) {
    return `Success: ${response.data.value}`
  } else if (response.status === 'error') {
    return `Error: ${response.errorCode}`
  }
  return 'Unknown response'
}
```

`'data' in response` *OR* `response.status === 'ok'`

TypeScript Classes

How does TypeScript support class syntax?

TypeScript and Classes

TypeScript provides support for JavaScript classes, including some extra syntax enhancements:

- Type Annotations
- Access Modifiers `public`, `protected`, `private` (compatible with ES2022's # private)
- `implements` with TypeScript interfaces
- Abstract classes
- Parameter properties
- Overloading Methods (not specific to classes)

Full Example

```ts
interface CanDrive {
  drive(distance: number): void;
}
```

```ts
class Car extends Vehicle {
  startEngine() {
    this.turnOn();
  }
}
```

```ts
abstract class Vehicle implements CanDrive {
  constructor (
    public readonly brand: string, protected year: number
  ) {}

  private mileage: number = 0;
  #engineOn: boolean = false;

  drive(this: Vehicle, distance: number): void {
    if (!this.#engineOn) {
      console.log("Engine is off!");
      return;
    }
    this.mileage += distance;
    console.log(`${this.brand} drove ${distance} km.`);
  }

  abstract startEngine(): void;

  protected getAge(): number {
    return new Date().getFullYear() - this.year;
  }

  public turnOn(): void {
    this.#engineOn = true;
    console.log(`${this.brand} engine started.`);
  }
}
```

Access Modifiers

`public`: default visibility, can be accessed anywhere

`protected`: unique to TypeScript, only visible to class and subclasses of the class

`readonly`: can prefix any field to prevent assignment outside the constructor

```
abstract class Vehicle implements CanDrive {
  constructor (
    public readonly brand: string, protected year: number
  ) {}
  ...
}
```

Access Modifiers: Private and

`private`: only accessible within the same class.

The private access modifier in TypeScript was implemented before ES2022, but both `private` and # will respect a private access scope. We prefer # because it's JS Native.

```
abstract class Vehicle implements CanDrive {
  ...
  private mileage: number = 0;
  #engineOn: boolean = false;
  ...
}
```

Implementing Interfaces

`implements` will ensure that a class enforces some structure defined by an interface, including fields and methods.

```
interface CanDrive {
     drive(distance: number): void;
}

abstract class Vehicle implements CanDrive {
     ...
     drive(this: Vehicle, distance: number): void {
          if (!this.#engineOn) {
               console.log("Engine is off!");
               Return;
          }
          this.mileage += distance;
          console.log(`${this.brand} drove ${distance} km.`);
     }
     ...
}
```

Abstract Classes

An abstract class cannot be instantiated directly.

These classes exist to extended/subclassed.

In subclasses, abstract methods from the superclass must be implemented.

```
class Car extends Vehicle {
  startEngine() {
    this.turnOn();
  }
}
```

```
abstract class Vehicle implements CanDrive {
  ...
  abstract startEngine(): void;
  ...
  public turnOn(): void {
    this.#engineOn = true;
    console.log(`${this.brand} engine started.`);
  }
}
```

Parameter Properties

TypeScript provides a shorthand for declaring and initializing class properties within the constructor.

```
public readonly brand: string;
protected year: number;

constructor(brand: string, year: number) {
    this.brand = brand;
    this.year = year;
}
```

```
constructor (
    public readonly brand: string,
    protected year: number
) {}
```

Overloading

TypeScript introduces overloading within functions and class methods. This involves multiple compatible function signatures and an implementation signature.

```
class Greeter {
  greet(person: string): string; // overload
  greet(age: number): string; // overload
  greet(personOrAge: any): string { // implementation
    if (typeof personOrAge === "string") {
      return `Hello, ${personOrAge}`;
    } else {
      return `You are ${personOrAge} years old.`;
    }
  }
}
```

POLL QUESTION

TypeScript identifies an error in this code. What line is it on and why is it an error?

```
1  interface Loggable {
2    log(): void
3  }
4
5  abstract class BaseLogger implements Loggable {
6    abstract log(): void
7
8    printTimestamp() {
9      console.log(Date.now())
10   }
11 }
12
13 class CustomLogger extends BaseLogger {
14   log() {
15     console.log('Custom log')
16   }
17
18   printTimestamp() {
19     console.log('Timestamp:', Date.now())
20   }
21 }
22
23 const logger: Loggable = new CustomLogger()
24 logger.log()
25 logger.printTimestamp()
```

Line 25: Property 'printTimestamp' does not exist on type 'Loggable'.
Logger is of Type Loggable, and thus does not have visibility on the methods of BaseLogger or CustomLogger, despite the object itself being a CustomLogger.

POLL QUESTION

What error will TypeScript produce for the line involving combine("hi", 2)?

```
function combine(a: string, b: string): string;
function combine(a: number, b: number): number;
function combine(a: any, b: any): any {
  return a + b;
}

const result = combine("hi", 2);
```

Error: No overload matches this call

Utility Types

How do we organize many types within a project and derive new types from old ones?

Too Many Types!

Often, you'll want type definitions for items that have very similar properties. These may be types or interfaces that share fields with other types like so:

```
interface RedPanda {
    id: number;
    name: string;
    age: number;
    zoo: string;
}
```

```
interface RedPandaNoId {
    name: string;
    age: number;
    zoo: string;
}
```

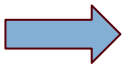
```
interface
RedPandaDetails {
    id: number;
    name: string;
    age: number;
    zoo: string;
    heightCm: number;
    weightKg: number;
}
```

This often leads to too many separate types!

Single Source of Truth

Instead, we can use features of TypeScript like extends, inference, and type transformations through Utility Types to keep a **single source of truth**, and require you to only update one source if changes need to be made.

```ts
interface RedPanda {
    id: number;
    name: string;
    age: number;
    zoo: string;
}
```



```ts
interface RedPandaDetails extends RedPanda {
    heightCm: number;
    weightKg: number;
}
```

Utility Type: Partial<T>

`Partial<T>` turns all properties in type T optional.

For instance, you can use this to easily update only parts of a certain object.

```
function update(redPanda: RedPanda, patch:
Partial<RedPanda>) {
     return {...redPanda, ...patch}
}

const klein = {1, “Klein”, 3}

const updated = update(klein, {
     age: 13,
     zoo: “Philadelphia Zoo”
})
```

```
interface RedPanda {
     id: number;
     name: string;
     age: number;
     zoo?: string;
}
```

```
// Partial<RedPanda>
// looks like:
{
     id?: number;
     name?: string;
     age?: number;
     zoo?: string;
}
```

Utility Type: Required<T>

`Required<T>` turns all properties in type T required.

For instance, you can use this to make sure all components of something with optional fields are present.

```
function completeRedPanda(redPanda:
Required<RedPanda>) {
     console.log(`${redPanda.name} is a
${redPanda.age} old red panda from
${redPanda.zoo}.`)
}

completeRedPanda({id: 1, name: "Klein", age: 13,
zoo: "Philadelphia Zoo"})
completeRedPanda({id: 2, name: "Liu"})
// problem: missing age and zoo from
Required<RedPanda>
```

```
interface RedPanda {
     id: number;
     name: string;
     age?: number;
     zoo?: string;
}
```

```
// Required<RedPanda>
// looks like:
{
     id: number;
     name: string;
     age: number;
     zoo: string;
}
```

Utility Type: Record<Keys, Type>

`Record<Keys, Type>` can be used to map the properties of one type to another type. The `Keys` are used as properties to access this new type.

```ts
type Name = "klein" | "wallace" | "darcy";

interface RedPandaInfo {
    id: number;
    age: number;
}

const redPandas: Record<Name, RedPandaInfo> = {
    klein: { id: 1, age: 13 },
    wallace: { id: 2, age: 8 },
    darcy: { id: 3, age: 5 },
};

console.log(redPandas.klein)
```

Utility Type: Pick<Type, Keys>

`Pick<Type, Keys>` creates a new type by picking the `Keys`, either a string literal or union of string literals, from `Type`.

```
interface RedPanda {
     id: number;
     name: string;
     age: number;
     zoo: string;
}

type SimpleRedPanda = Pick<RedPanda, "name" | "id">;

const simple: SimpleRedPanda = { id: 1, name: "klein"};

console.log(simple)
```

Utility Type: Omit<Type, Keys>

`Omit<Type, Keys>` creates a new type by picking the ALL properties from `Type` and removing the designated `Keys`.

```
interface RedPanda {
     id: number;
     name: string;
     age: number;
     zoo: string;
}

type SimpleRedPanda = Omit<RedPanda, “zoo”>;

const simple: SimpleRedPanda = { id: 1, name: "klein", age: 13};

console.log(simple)
```

POLL QUESTION

Fill in the blank:

What goes in to the "Keys" part of this Record?

```
const animalWeights: Record<________, number> = {
  elephant: 5000,
  cat: 4.5,
  rabbit: 2,
};
```

```
'elephant' | 'cat' | 'rabbit'
```

Generics

How do we define reusable components that work for a variety of types?

Generics

Generics allows you to create reusable logic that works with many types instead of a single one.

Using generics allows you to capture a type in the function header to be used as an argument or return type. For instance:

```
function identity<Type>(arg: Type): Type {
     return arg;
}

const out = identity<string>("mystring")
console.log(out)
```

Calling Generic Functions

When you call generic functions, you can choose to explicitly pass the type argument <>, or omit it to allow type argument inference to kick in.

```
async function fetchFromApi<T>(url: string): Promise<T |
undefined> {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error("Network response was not ok");
    }
    return await response.json();
  } catch (error) {
    console.error("Error fetching data:", error);
    return undefined;
  }
}

const comments = await fetchFromApi(
  "https://jsonplaceholder.typicode.com/todos/1",
);

console.log(typeof comments) // object
```

Generic Constraints

A generic constraint will constrain the generic type to conform to a certain object shape, through an object or interface.

This can be achieved through the `extends` keyword after the generic type.

```
function logLength<T extends { length: number }>(item: T): void {
    console.log(item.length);
}

logLength([1, 2, 3]); // arrays have length
logLength("hello"); // strings have length
logLength(42); // Error: numbers do not have a length property
```

Generic Types and Classes

Generics may be used in type and class definitions as well.

```
interface Entity {
  id: number;
}

type ItemCollection<T extends Entity> = T[];

function addItemToCollection<T extends Entity>(
      collection: ItemCollection<T>, item: T
): ItemCollection<T> {
  return [...collection, item];
}

class RedPanda<Details> implements Entity {
  constructor(
    public id: number,
    public name: string,
    public details: Details
  ) {}
}
```

```
interface RedPandaDetails {
  age: number;
  zoo: string;
}

const panda = new RedPanda<RedPandaDetails>(1,
"Klein", {
  age: 13,
  zoo: "Philadelphia Zoo"
});


let pandas:
ItemCollection<RedPanda<RedPandaDetails>> = [];
pandas = addItemToCollection(pandas, panda);
```

POLL QUESTION

What line produces a TypeScript error?

```
1   type WithId = { id: number };
2
3   function mergeWithId<T extends WithId, U>(obj: T, extra: U): T & U {
4     return { ...obj, ...extra };
5   }
6
7   const a = { id: 1, name: "Cat" };
8   const b = { breed: "Siamese" };
9
10  const c = mergeWithId(a, b);
11  const d = mergeWithId(b, a);
```

```
11: Argument of type '{ breed: string; }' is not assignable to parameter of type 'WithId'.
    Property 'id' is missing in type '{ breed: string; }' but required in type 'WithId'.
```

5-Minute Break!

Additional TypeScript Syntax

enums, Conditional Types, infer, dynamic and mapped types, Template Union Types

enums

Unions with string literals can give us "enum-like" code.

However, TypeScript also has its own unique `enum` keyword, in order to define named constants.

```
enum Compass {
    North = 1,
    East,
    South,
    West
}
```

Numeric Enums have auto-incrementing behavior.

```
enum Compass {
    Up = "NORTH",
    Right = "EAST",
    Down = "SOUTH",
    Left = "WEST"
}
```

String Enums allow more serializable and readable values

Conditional Types

Using Generics and ternaries, we can build Conditional Types, using the following syntax:

```
type NewType = SomeType extends OtherType ? TypeIfTrue :
TypeIfFalse;
```

This allows us to filter out certain types in a type union, like so:

```
type IsHouseCat<Cat> = Cat extends {sound: “meow”} ? Cat: never;

type Cats = Lion | Cheetah | Tiger | Tabby
type HouseCat = IsHouseCat<Cats> // Cheetah | Tabby
```

infer keyword

The infer keyword can be used with conditional types to extract a type from another type within the conditional structure.

We can use it to do things like:

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

Extract a return type from a function T

```
type ElementType<T> = T extends (infer U)[] ? U : never;
```

Extract an array type from array T

Index Signatures

Index signatures allow you to define a shape for an object that can have any number of properties, so long as it fits the types of the key and value.

```
type Dict = {
  [key: string]: string;
};

const dictionary: Dict = {
     "apple": "lorem ipsum",
     "alliance": "dolor est",
     ...
};
```

Mapped Types

A **mapped type** uses similar syntax to index signatures to structure a new type based on some input type.

```
type Feline = {
     species: string,
     classification: “housecat”| “bigcat”;
};

type OptionalFeline = {
     [K in keyof Feline]?: Feline[K];
}
```

```
// OptionalFeline now looks like:
type OptionalFeline = {
     species?: string,
     classification?: “housecat”| “bigcat”;
};
```

Template Union Types

A template literal string can combine together different types for all possible combinations.

```
type Vertical = "north" | "south"
type Horizontal = "east" | "west"

type Diagonal = `${Vertical}${Horizontal}`
```

We can use a conditional type to view the result in an editor:

```
type Debug<T> = T extends any ? (x: T) => void : never;
type _debug = Debug<Diagonal>;
// shows Diagonal type has “northeast”, “northwest”,
“southeast”, and “southwest”
```

TypeScript Cheat Sheet!

Overwhelmed with syntax? Use the cheat sheet!

https://www.typescriptlang.org/cheatsheets/

The TypeScript documentation provides a cheatsheet for various syntax, including a lot of what we covered today.

Introduction to React and Web Bundlers

JavaScript Frameworks

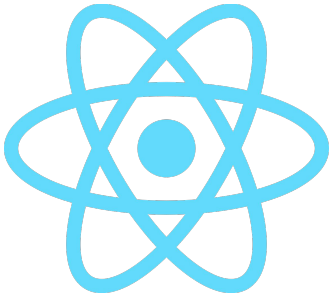
How does a JavaScript framework help developers build modern web applications quickly?

JavaScript Frameworks

JavaScript frameworks help build web applications efficiently by providing structure and reusable code.

They provide benefits like:

- Structure and conventions
- Built-in utilities for common tasks
- Faster development



The Rise of Frameworks

Vanilla JS uses DOM manipulation and works with scripts and ids. This often leads to problems with code maintenance and handling UI states, as things like automatically updating UI requires complex code.

Libraries like jQuery can improve the experience of working with DOM, but this still leads to complex code bases. This gets more complex as Single-Page Applications (SPAs) got more popular.

Using Frameworks

Each JavaScript framework is structured in a different way around JavaScript's features.

They each will provide certain **conventions** and **tools** to organize and enhance JavaScript, while allowing the use of JavaScript logic, including all of its syntax.

What is React?

React: Your View Layer

React is a declarative, component-based JavaScript library for building user interfaces.

It is not an all-in-one framework, as it only focuses on the view layer of the app, not the models or backend components.

It does contain a rich ecosystem of tailor-made libraries for it, to make up for things it lacks (routers, styling, forms, global state management)

How React Works

React has:

- **Component-Based UI**: UI is made up of reusable pieces
- **Declarative UI**: You describe what the UI looks like while React handles the updates
- **Virtual DOM**: React creates a virtual copy of the DOM, and only needs to update what is changed when changes occur.
- **Unidirectional Data Flow**: Data flows from parent to child within the DOM

Example: React Component

React components are written as JS functions that return JSX, or JavaScript XML, that is HTML-like code for React elements.

```
const Navbar = () => {
     const navLinks = [
          ...
     ];

     return (
          <nav>
               {navLinks.map((link) => (
                    <li key={link.href}>
                         ...
                    </li>
               ))}
          </nav>
     )
}

export default Navbar;
```

Example: Rendering Components

Once you have your component, you can insert it into other components. The starting point for rendering is made with the React API function `createRoot()`, which creates a top-level node that is now managed by React for rendering.

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import Navbar from '../Components/Navbar'

createRoot(document.getElementById('root')).render(
     <NavBar/>
)
```

React Ecosystem

Due to the popularity of React, there are many libraries made for it to provide extra features to round it out.

We will cover many React libraries, including:

- Routing libraries like **React Router**
- **Redux** for state management normal React states

Additionally we'll be covering **Next.js**, an entire framework built on React that provides its own routing, server-side rendering, and more.

Web Bundlers

How do web bundlers help
create, configure, and manage a React project?

What is a Web Bundler?

Modern JS projects, especially those that include frameworks, have a lot of JS modules and scripts, CSS, and image files.

Large projects can easily become an unoptimized mess with browser compatibility issues and module incompatibilities!

Web Bundlers solve this issue by optimizing and packaging everything for proper deployment, including module support in browsers.

Web Bundlers

There are many bundlers out there that have different performance features, ease of setup, and capabilities.

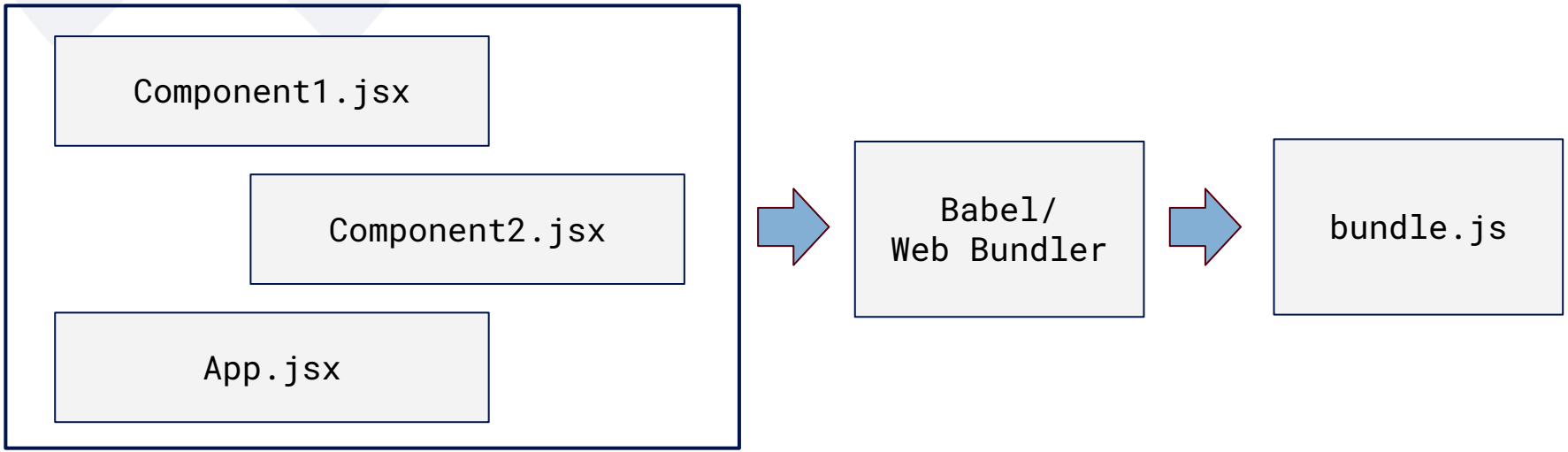
Webpack

Parcel

Vite (build tool + dev server that uses Rollup)

Bundlers and React

Browsers don't natively understand JSX, so bundlers will provide transpilation tools such as Babel to turn JSX into JS, among other optimizations.



Deprecated: create react app

You may find many resources that suggest create react app (CRA) as a way to create a React project. This method uses WebPack and Babel in the background.

However, as of earlier this year (2025), this method is now deprecated, with a warning if you do still use it!

Modern Build Tools: Vite

We'll suggest you use **Vite** (`npm create vite@latest`) to create your React projects.

It includes:

- **esbuild**, a code transformer for quickly transpiling JSX to JS
- **Rollup** as a bundling tool
- A very fast dev server for fast updates and startup

Activity: Create a React Project

We'll be using Vite to make a React project and explore the different components.