1장. 들어가며

1.1 웹 개발의 역사

1. 자바스크립트의 탄생

- 1990년대 가장 많이 사용되는 웹 브라우저들

마이크로소프트(Microsoft)	인터넷 익스플로러(Internet Explorer)
넷스케이프 커뮤니케이션즈	넷스케이프 내비게이터

- 1995년 넷스케이프의 **브랜든 아이크** → 웹의 다양한 콘텐츠 표현하기 위해 **자바스크립트** 만듦
  - 웹 대응 언어로서 완벽한 해결책으로 제시하기보다,
    빠르게 시장 반응을 확인할 수 있는 프로토타입 언어로 출시할 것을 목표

➡️ 당시에는 사용자에게 **유의미한 편의성**을 제공하지 못함.

2. 자바스크립트 표준, ECMAScript의 탄생

- 경쟁 관계이던 넷스케이프, 마이크로소프트는 자신들의 브라우저에서 새로운 기능을 늘림
  → **각자의 브라우저에서만 동작**
- 특히 익스플로러, 내비게이터의 DOM 구조는 완전히 다름
  → **크로스 브라우징(Cross Browsing) 이슈 발생**
- 초기 JS는 브라우저 생태계를 고려해서 작성된 것이 X, 단지 새로운 기능이 추가되는 형태
  ➡️ 결국 **브라우저는 자바스크립트의 변화를 따라가지 못함**

✅ 이러한 **문제**를 해결하기 위해 **폴리필(polyfill)**, **트랜스파일(transpile)** 개념 등장

> 💡 **폴리필(polyfill)**
> : 브라우저가 지원하지 않는 코드를 브라우저에서 사용할 수 있도록 변환한 코드 조각 & 플러그인
> ex) core.js, polyfill.io
>
> **트랜스파일(transpile)** : 최신 버전 코드 → 예전 버전 코드로 변환하는 과정 ex) 바벨(Babel)
>
> = 둘 다 최신 기능을 구버전의 실행 환경에서 동작할 수 있게 바꿔주는 역할

- 따라서, jQuery(브라우저 호환성 고민하지 않고 한 번에 개발) 같은 라이브러리 유행

하지만, 언제까지나 라이브러리에 기대어 크로스 브라우징 이슈 해결할 수는 없고,
모든 브라우저에서 동일하게 동작하는 **표준화된 자바스크립트의 필요성**이 제기됨.

- 넷스케이프는 Ecma 인터내셔널(국제 표준화 기구)에 자바스크립트의 기술 규격을 제출했고,
  Ecma 인터내셔널은 **ECMAScript**라는 이름으로 자바스크립트 표준화를 공식화 함.

3. 웹사이트에서 웹 어플리케이션으로의 전환

웹사이트

- 수집된 데이터 및 정보를 특정 페이지에 표시하기 위한 정적인 웹
- **단방향**으로 정보를 제공하기 때문에 사용자와 상호 작용 X
- HTML에 링크가 연결된 웹 페이지 모음으로 콘텐츠가 동적으로 업데이트 X

웹 애플리케이션

- 사용자와 상호작용하는 **쌍방향** 소통의 웹
- 본격적으로 시대의 서막을 연 서비스 : Google Maps

이러한 웹의 성장은 개발자가 구현해야 하는 결과물 규모가 이전에 비해 커졌다는 것을 의미
개발 규모가 커진 만큼, 개발 생태계도 이런 흐름에 맞춘 **변화**가 필요

4. 개발 생태계의 발전

1. 하나의 웹 페이지를 통으로 개발 X
   컴포넌트 단위로 개발하는 방식이 생겨남.
2. 비동기 요청 사용해서 **페이지의 일부 데이터 로드 가능**

- 웹 서비스는 점차 페이지에서 애플리케이션의 특성 갖게 됨.
  - 서비스 규모 ↑, 데이터양 ↑
  - 표현해야 하는 화면 다양 & 복잡
  - 디바이스 다양화
  - 사용자는 저마다의 디바이스에 최적화된 UX/UI를 기대

→ **컴포넌트 베이스 개발Component Based Development(CBD) 방법론** 등장

: 재사용할 수 있는 컴포넌트를 개발 또는 조합해서 하나의 애플리케이션을 만드는 개발 방법론

- 컴포넌트는 다른 컴포넌트와의 의존성을 최소화하거나 없애야 함.

  ➡️ 개발자는 컴포넌트 간의 의존성을 파악해야 함.

5. 개발자 협업의 필요성 증가

- 결과물 커짐 → 서비스 개발 후 유지보수에 있어서 **협업의 중요성 ↑**

  = 거대한 프로덕트 개발할 때 효과적인 유지보수를 위한 **협업 보완책의 필요성** 제기

1.2 자바스크립트의 한계

1. 동적 타입 언어

- 자바스크립트는 동적 타입 언어

  = 변수에 타입을 명시적으로 지정 X, 코드가 실행되는 런타임에 변숫값이 할당될 때 해당 값의 타입에 따라 변수 타입이 결정된다는 것

2. 동적 타이핑 시스템의 한계

```
// 이 함수는 숫자 a, b의 합을 반환
const sumNumber = (a, b) => {
    return a + b;
};

sumNumber(1, 2); // 3
```

- 실행하면 어떠한 에러 발생 X, 정상적으로 동작

```
// 이 함수는 숫자 a, b의 합을 반환
const sumNumber = (a, b) => {
    return a + b;
};

sumNumber(100); // NaN
sumNumber("a", "b"); // ab
```

- 이또한 어떠한 에러 발생 X, 정상적으로 동작

그러나, 주석과 함수 이름 때문에 **개발자 의도와는 다르게 동작**

⇒ 기계 입장에서는 정상적이지만 **사람 입장에서는 정상적이지 않은 코드**

3. 한계 극복을 위한 해결 방안

JSDoc

장점

- @ts- check를 추가하면 타입 및 에러 확인 가능
- 타입 힌트를 제공하는 HTML 문서 생성 가능

단점

- 어디까지나 주석의 성격이라 강제성 부여해서 사용하기는 어려움

propTypes

장점

- 리액트에서 prop에 유효한 값이 전달됐는지 확인 가능

단점

- 전체 애플리케이션의 타입 검사에는 사용 불가능
- 리액트라는 특정 라이브러리에서만 사용 가능

다트

장점

- JS를 대체하기 위해 제시한 새로운 언어 → 근본적인 해결책으로 여겨짐

단점

- JS가 이미 자리매김한 상황에 새로운 언어의 등장으로 개발의 파편화 야기 가능

결론

자바스크립트 스스로가 인터페이스를 기술할 수 있는 언어로 발전해야 한다.

4. 타입스크립트의 등장

마이크로소프트는 자바스크립트의 슈퍼셋(Superset) 언어, **타입스크립트(TypeScript)** 공개

안전성 보장

- 타입스크립트는 정적 타이핑을 제공
  - 컴파일 단계에서 타입 검사 → JS에서 빈번한 타입 에러를 줄일 수 있음
  - 런타임 에러를 사전에 방지 → 안전성↑

개발 생산성 향상

- VSCode 등의 IDE에서 타입 자동 완성 기능을 제공 → 개발 생산성 크게 향상

협업에 유리

- 복잡한 애플리케이션 개발 & 협업에 유리
- 복잡한 애플리케이션일수록 협업하는 개발자 수↑
  - 자동 완성 기능이나 기술된 인터페이스 코드로 쉽게 파악 가능

자바스크립트에 점진적으로 적용 가능

- 자바스크립트의 슈퍼셋이기 때문에 일괄 전환이 아닌 **점진적 도입 가능**