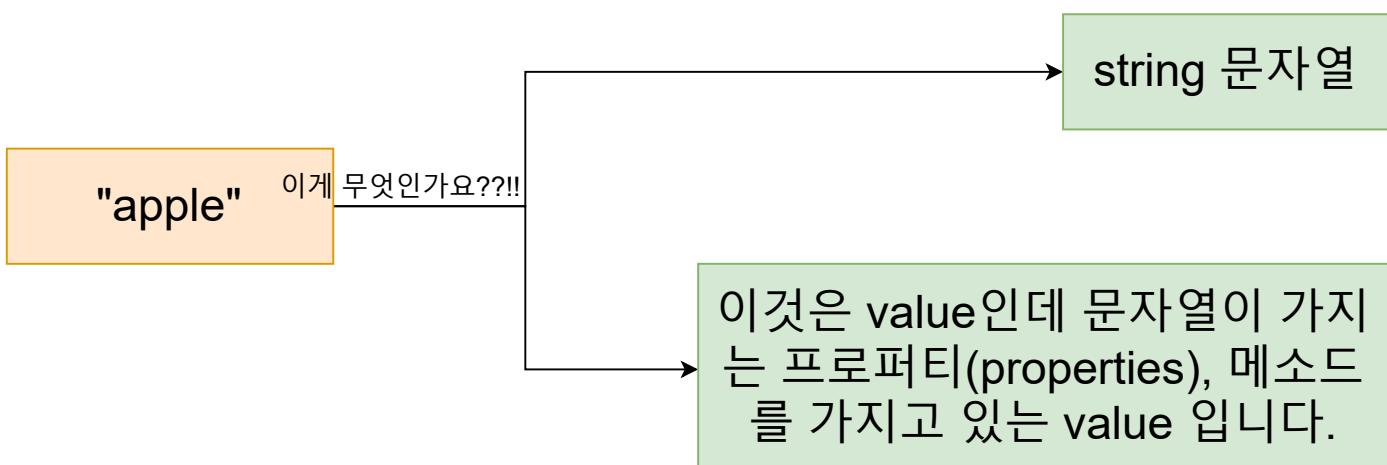


TypeScript Type

In TypeScript,
a type is a convenient way to refer to the different properties
and functions that a value has.

타입이란, 그 value가 가지고 있는 프로퍼티나 함수를 추론할 수 있는
방법이다.



자바스크립트에서 문자열 Properties + Methods



Property

string.length는 문자열의 속성인 문자열의 길이를 제공합니다. 문자열 자체에는 아무 것도 할 줄 알 수 없습니다.

↳	charCodeAt
↳	codePointAt
↳	concat
↳	endsWith
↳	includes

합니다. 문자열 사세에는 아무 것도 아시 않읍니다.

Method

string.toLowerCase()는 문자열을 소문자로 변환합니다. 즉, 문자열에 작업을 수행한 다음 반환합니다.

Types in Typescript

TypeScript는 JavaScript에서 기본으로 제공하는 기본 제공 유형(built-in types)을 상속합니다. TypeScript 유형은 다음과 같이 분류됩니다.

- Primitive Types
- Object Types

Primitive types

Name	Description
string	문자열을 나타냅니다.
number	숫자 값을 나타냅니다.
boolean	true 와 false 값을 가지고 있습니다.
null	하나의 값을 가집니다: null
undefined	하나의 값을 가집니다: undefined. 초기화되지 않은 변수의 기본값입니다.
symbol	고유한 상수 값을 나타냅니다.

Object types

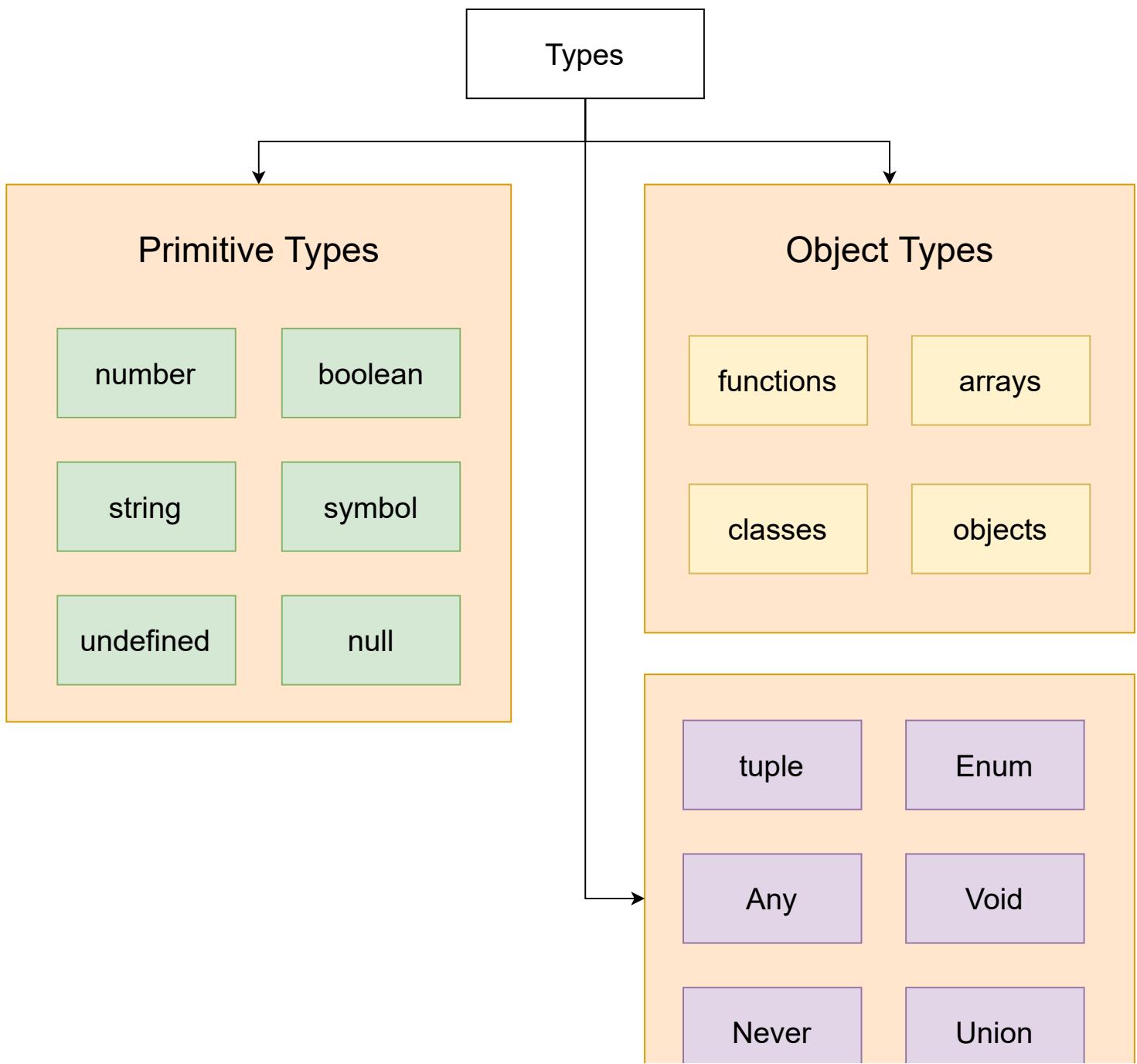
Name	Description
function	함수를 나타냅니다.
array	배열을 나타냅니다.
classes	클래스를 나타냅니다.
object	객체를 나타냅니다.

```
// functions
const getNumber = (i: number): void => {
  console.log(i)
}

// array
const arr: string[] = ['a', 'b', 'c']

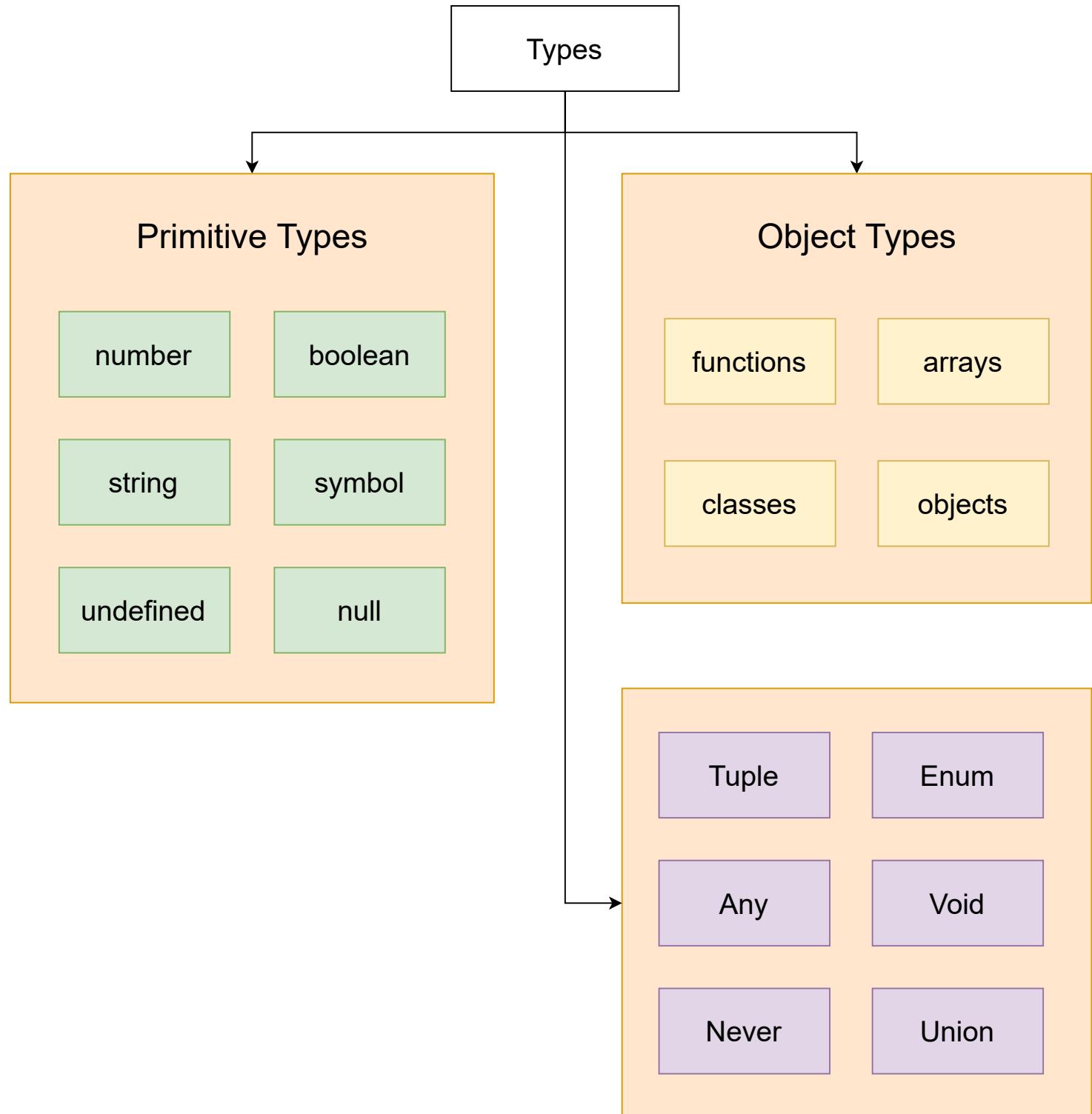
// class
class Music { }
let music: Music = new Music()

// object
let point: { x: number; y: number } = { x: 20, y: 10 }
```





TypeScript 추가 제공 타입



source: <https://www.tutorialsteacher.com/typescript/>,
<https://typescript-kr.github.io/pages/basic-types.html>

Any

```
let something: any = "Hello World!";
something = 23;
something = true;
```

애플리케이션을 만들 때, 잘 알지 못하는 타입을 표현해야 할 수가 있습니다. 이 값들은 사용자로부터 받은 데이터나 서드 파티 라이브러리 같은 동적인 컨텐츠에서 올 수도 있습니다. 이 경우 타입 검사를 하지 않고, 그 값들이 컴파일 시간에 검사를 통과하길 원합니다. 이를 위해, any 타입을 사용할 수 있습니다.

하지만 이 타입을 최대한 쓰지 않는게 좋습니다. 그래서 noImplicitAny라는 옵션을 주면 any를 셋을 때 오류가 나오게 할 수 있습니다.

```
let arr: any[] = ["John", 212, true];
arr.push("Smith");
console.log(arr); //Output: [ 'John', 212, true, 'Smith' ]
```

Union

```
let code: (string | number);
code = 123; // OK
code = "ABC"; // OK
code = false; // Compiler Error

let empId: string | number;
empId = 111; // OK
empId = "E111"; // OK
empId = true; // Compiler Error
```

TypeScript를 사용하면 변수 또는 함수 매개변수에 대해 둘 이상의 데이터 유형을 사용할 수 있습니다. 이것을 유니온 타입이라고 합니다.

Tuple

TypeScript에서는 배열 타입을 보다 특수한 형태로 사용할 수 있는 [tuple](#) 타입을 지원합니다. tuple에 명시적으로 지정된 형식에 따라 아이템 순서를 설정해야 되고, 추가되는 아이템 또한 tuple에 명시된 타입만 사용 가능합니다.

```
var employee: [number, string] = [1, "Steve"];
var person: [number, string, boolean] = [1, "Steve", true];

var user: [number, string, boolean, number, string];// declare tuple variable
user = [1, "Steve", true, 20, "Admin"];// initialize tuple variable
```

배열 Tuple

```
var employee: [number, string] = [1, "Steve"];
```

```
var employee: [number, string][];  
employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]];
```

Tuple에 요소 추가

```
var employee: [number, string] = [1, "Steve"];  
employee.push(2, "Bill");  
console.log(employee); //Output: [1, 'Steve', 2, 'Bill']
```

에러가 나는 경우

```
employee.push(true)
```

튜플은 'number | string'은 숫자와 문자열 값만 저장할 수 있습니다.

<https://www.typescriptlang.org/play?ts=4.5.4>

Enum

enum은 enumerated type(열거형)을 의미합니다.
Enum은 값들의 집합을 명명하고 이를 사용하도록 만듭니다.
여기서는 PrintMedia라 불리는 집합을 기억하기 어려운 숫자 대신 친숙한 이름으로 사용하기 위해 enum을 활용할 수 있습니다. 열거된 각 PrintMedia는 별도의 값이 설정되지 않은 경우 기본적으로 0부터 시작합니다.

```
enum PrintMedia {  
    Newspaper, //0  
    Newsletter, //1  
    Magazine, //2  
    Book //3  
}
```

아래 코드에서 mediaType 변수에 할당된 값은 3입니다. 설정된 PrintMedia 열거형 데이터의 Book의 값이 숫자 3이기 때문입니다.

```
let mediaType: number = PrintMedia.Book // 3
```

enum에 설정된 아이템에 값을 할당할 수도 있습니다. 값이 할당되지 않은 아이템은 이전 아이템의 값에 +1된 값이 설정됩니다.

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter = 50,
    Magazine = 55,
    Book // 55 + 1
}
```

아래 코드에서 mediaType 변수에 할당된 값은 56입니다. 설정된 PrintMedia 열거형 데이터의 Book의 값이 숫자 56이기 때문입니다.

```
let mediaType: number = PrintMedia.Book // 56
```

enum 타입의 편리한 기능으로 숫자 값을 통해 enum 값의 멤버 이름을 도출할 수 있습니다.

```
let type: string = PrintMedia[55] // 'Magazine'
```

또한 어떠한 언어 코드를 정의하는 코드를 작성할 때 언어의 집합을 만들 때도 enum을 사용 할 수 있습니다.

```
export enum LanguageCode {
    korean = 'ko',
    english = 'en',
    japanese = 'ja',
    chinese = 'zh',
    spanish = 'es',
}

const code: LanguageCode = LanguageCode.english
```

이렇게 enum을 이용해서 언어 집합을 만들어주면 어떠한 코드가 어떠한 나라의 언어 코드가 무엇인지 알지 못해도 쉽게 코드를 작성해 줄 수 있고 코드를 읽는 사람 입장에서도 가독성이 높아지게 됩니다.

```
let LanguageCode = {
    korean : 'ko',
    english : 'en',
    japanese : 'ja',
    chinese : 'zh',
    spanish : 'es',
}
```

}

이렇게 보면 enum과 JS의 object를 사용하는 것과 별 차이가 없어 보입니다. 사실 enum은 그 자체로 객체이기도 합니다.
그래서 Object.keys(LanguageCode) 를 하면 실제 키 값이 배열에 담겨 나옵니다. => ['korean', 'english']
Object.values(LanguageCode) 를 하면 value 값이 ...
=> ['ko', 'en']

enum과 객체의 차이점

object 는 코드내에서 새로운 속성을 자유롭게 추가할 수 있지만, enum 은 선언할 때 이후에 변경할 수 없습니다.

object 의 속성값은 JS가 허용하는 모든 타입이 올 수 있지만, enum 의 속성값으로는 문자열 혹은 숫자만 허용됩니다.

Void

Java와 같은 언어와 유사하게 데이터가 없는 경우 void가 사용됩니다. 예를 들어 함수가 값을 반환하지 않으면 반환 유형으로 void를 지정할 수 있습니다.

타입이 없는 상태이며, any 와 반대의 의미를 가집니다.
void 소문자로 사용해주셔야하며, 주로 함수의 리턴이 없을 때 사용해주시면 됩니다.

```
function sayHi(): void {
    console.log('Hi!')
}

let speech: void = sayHi();
console.log(speech); //Output: undefined
```

Never

TypeScript는 절대 발생하지 않을 값을 나타내는 새 Type never를 도입했습니다.

Never 유형은 어떤 일이 절대 일어나지 않을 것이라고 확신할 때 사용됩니다. 일반적으로 함수의 리턴 타입으로 사용됩니다. 함수의 리턴 타입으로 never가 사용될 경우, 항상 오류를 리턴하거나 리턴 값을 절대로 내보내지 않음을 의미합니다.

```
function throwError(errorMsg: string): never {
    throw new Error(errorMsg);
}

function keepProcessing(): never {
    while (true) {
        console.log('I always does something and never ends.')
    }
}
```

```
}
```

Void 와 Never의 차이

Void 유형은 값으로 undefind이나 null 값을 가질 수 있으며 Never는 어떠한 값도 가질 수 없습니다.

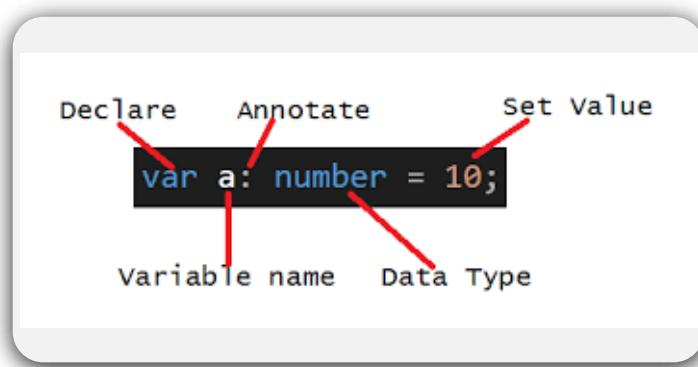
```
let something: void = null;  
let nothing: never = null; // Error: Type 'null' is not assignable to type 'never'
```

TypeScript에서 값을 Return하지 않는 함수는 실제로 undefined를 반환합니다.

```
function sayHi(): void {  
    console.log('Hi!')  
}  
  
let speech: void = sayHi();  
console.log(speech); // undefined
```

위의 예에서 볼 수 있듯이 sayHi 함수는 반환 유형이 void인 경우에도 내부적으로 undefined를 반환하기 때문에 speech는 undefined가 됩니다. Never 유형을 사용하는 경우 void는 Never에 할당할 수 없기 때문에 Speech:never는 컴파일 시간 오류를 발생시킵니다.

type annotation, type inference



type annotation

개발자가 타입을 타입스크립트에게
직접 말해주는 것

```
const rate: number = 5 // number 타입 지정
```

type inference

타입스크립트가 알아서 타입을 추론하
는 것

```
const rate = 5 // **변수 선언과 동시에 초기화할 경우** 타입을 알아서 추론한다
```

타입을 추론하지 못해서
타입 annotation을 꼭 해줘야하는 경우

any 타입을 리턴하는 경우

coordinates에 hover해보면 `const coordinates: any` 라고 뜨는 것을 볼 수 있습니다. `JSON.parse`는 `json`을 파싱해줍니다. 인풋으로 들어가는 `json`을 확인하면 대충 어떤 타입이 리턴될지 개발자는 예상할 수 있지만, 타입스크립트는 여기까지 지원하지 않습니다. 리턴 타입이 일정하지 않으므로 `any`를 리턴한다고 추론해버립니다. 그러므로 이 경우에는 타입 애노테이션을 해주어야 합니다.

```
const json = '{"x": 4, "y": 7}'
```

```
const json = '{"x": 1, "y": 2}'  
const coordinates = JSON.parse(json)  
console.log(coordinates)
```

변수 선언을 먼저하고 나중에 초기화하는 경우

변수 선언과 동시에 초기화하면 타입을 추론하지만, 선언을 먼저하고 나중에 값을 초기화할 때에는 추론하지 못합니다.

```
let greeting  
greeting = 'hello' // let greeting: any
```

변수에 대입될 값이 일정치 못하는 경우

여러 타입이 지정되어야 할 때에는 | (or statement)로 여러 타입을 애노테이션 해줍니다.

```
let num = [-7, -2, 10]  
let numAboveZero: boolean | number = false  
  
for (let i = 0; i < num.length; i++) {  
  if (num[i] > 0) {  
    numAboveZero = num[i]  
  }  
}
```

type assertion

type assertion이란 ?

TypeScript에서는 시스템이 추론 및 분석한 타입 내용을 우리가 원하는 대로 얼마든지 바꿀 수 있습니다. 이때 "타입 표명(type assertion)"이라 불리는 메커니즘이 사용됩니다. TypeScript의 타입 표명은 프로그래머가 컴파일러에게 내가 너보다 타입에 더 잘 알고 있고, 나의 주장에 대해 의심하지 말라고 하는 것과 같습니다.

type assertion을 사용하면 값의 type을 설정하고 컴파일러에 이를 유추하지 않도록 지시할 수 있습니다. 이것은 프로그래머로서 TypeScript가 자체적으로 추론할 수 있는 것보다 변수 유형에 대해 더 잘 이해하고 있을 때입니다.

```
var foo = {};
foo.bar = 123; // 오류: 속성 'bar'가 '{}'에 존재하지 않음
foo.bas = 'hello'; // 오류: 속성 'bar'가 '{}'에 존재하지 않음
```

컴파일러는 foo type이 속성이 없는 {}라고 가정하기 때문에 위의 예에서는 컴파일러 오류가 발생합니다. 그러나 아래와 같이 type assertion을 사용하면 이러한 상황을 피할 수 있습니다.

```
interface Foo {
  bar: number;
  bas: string;
}
var foo = {} as Foo;
foo.bar = 123;
foo.bas = 'hello';
```

as Foo , <Foo>

타입 표명은 위에 두가지 방식으로 표현할 수 있습니다. 하지만 리액트를 사용할 때는 <Foo> 키워드는 JSX의 문법과 겹치기 때문에 as Foo를 공통적으로 사용하는게 추천됩니다.

```
import ts from 'ts'
import path from 'path'
import matter from 'gray-matter'

const postsDirectory = path.join(process.cwd(), 'posts')
console.log('process.cwd()', process.cwd());
// /Users/johnahn/Downloads/next-typescript
console.log('postsDirectory', postsDirectory);
// /Users/johnahn/Downloads/next-typescript/posts

export function getSortedPostsData() {
    // Get file names under /posts
    const fileNames = fs.readdirSync(postsDirectory)
    console.log('fileNames', fileNames);
    // fileNames [ 'pre-rendering.md', 'ssg-ssr.md' ]
    const allPostsData = fileNames.map(fileName => {
        // Remove ".md" from file name to get id
        const id = fileName.replace /\.md$/, ''

        // Read markdown file as string
        const fullPath = path.join(postsDirectory, fileName)
        const fileContents = fs.readFileSync(fullPath, 'utf8')

        // Use gray-matter to parse the post metadata section
        const matterResult = matter(fileContents)

        // Combine the data with the id
        return {
            id,
            ...(matterResult.data as { date: string; title: string })
        }
    })
    // Sort posts by date
    return allPostsData.sort((a, b) => {
        if (a.date < b.date) {
            return 1
        } else {
            return -1
        }
    })
}
```

getStaticProps를 이용한 포스트 리스트 나열

빌드 타임에 포스트 자료 가져오기

```
export const getStaticProps: GetStaticProps = async () => {
  const allPostsData = getSortedPostsData()
  return {
    props: {
      allPostsData
    }
  }
}
```

```
import { GetStaticProps } from 'next'
import { getSortedPostsData } from '../lib/posts'
```

props으로 포스트 데이터 가져오기

```
export default function Home({ allPostsData }: {
  allPostsData: {
    date: string
    title: string
    id: string
  }[]
}) {
```

리스트 나열하기

```
<ul className={homeStyles.list}>
  {allPostsData.map(({ id, date, title }) => (
    <li className={homeStyles.listItem} key={id}>
      <a>{title}</a>
      <br />
      <small className={homeStyles.lightText}>
        {date}
      </small>
    </li>
  )))
</ul>
```

포스트 자세히 보기 페이지로 이동(file system 기반의 라우팅)

파일기반 네비게이션 가능

리액트에서는 route를 위해서 react-router라는 라이브러리를 사용하지만

Next.js에는 페이지 개념을 기반으로 구축된 파일 시스템 기반 라우터가 있습니다.

파일이 페이지 디렉토리에 추가되면 자동으로 경로로 사용할 수 있습니다.

페이지 디렉토리 내의 파일은 가장 일반적인 패턴을 정의하는 데 사용할 수 있습니다.

파일 생성 예시

pages/index.js → /

pages/blog/index.js → /blog

pages/blog/first-post.js → /blog/first-post

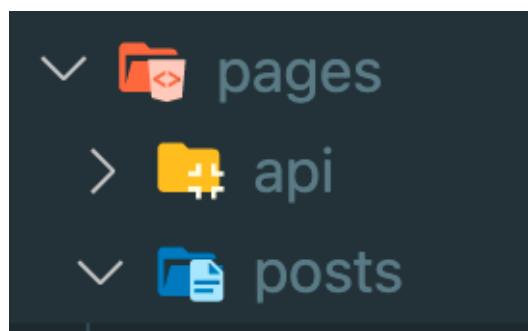
pages/dashboard/settings/username.js → /dashboard/settings/username

pages/blog/[slug].js → /blog/:slug (/blog/hello-world)

pages/[username]/settings.js → /:username/settings (/foo/settings)

pages/post/[...all].js → /post/* (/post/2020/id/title)

포스트 파일 생성



```
export default function Post() {
  return (
    <div>
      ...
    </div>
  )
}
```

Link 함수를 이용한 페
이지 이동

```
<li className={homeStyles.listItem} key={id}>
  <Link href={`/posts/${id}`}>
    <a>{title}</a>
  </Link>
  <br />
  <small className={homeStyles.lightText}>
    {date}
  </small>
</li>
```

포스트 데이터를 가져와서 보여주기(remark)

getStaticPaths

- 동적 라우팅이 필요할 때 getStaticPaths로 경로 리스트를 정의하고, HTML에 build 시간에 렌더 됩니다.
- Nextjs는 pre-render에서 정적으로 getStaticPaths에서 호출하는 경로들을 가져옵니다.

Post 데이터를 가져와야 하는 경로 목록을 가져오기

```
export const getStaticPaths: GetStaticPaths = async () => {
  const paths = getAllPostIds()
  console.log('paths', paths)
  // [ { params: { id: 'pre-rendering' } }, { params: { id: 'ssg-ssr' } } ]
  return {
    paths,
    fallback: false
  }
}
```

fallback

- false라면 getStaticPaths로 리턴되지 않는 것은 모두 404 페이지가 됩니다.
- true라면 getStaticPaths로 리턴되지 않는 것은 404로 뜨지 않고, fallback 페이지가 뜨게 됩니다.

```
export function getAllPostIds() {
  const fileNames = fs.readdirSync(postsDirectory)
  return fileNames.map(fileName => {
    return {
      params: {
        id: fileName.replace(/\.\md$/, '')
      }
    }
  })
}
```

```
    }
}
```

전달받은 아이디를 이용해서 해당 포스트의 데이터 가져오기

```
export const getStaticProps: GetStaticProps = async ({ params }) => {
  console.log('params', params);
  // { id: 'ssg-ssr' }
  const postData = await getpostData(params.id as string)
  return {
    props: {
      postData
    }
  }
}
```

```
export async function getpostData(id: string) {
  const fullPath = path.join(postsDirectory, `${id}.md`)
  const fileContents = fs.readFileSync(fullPath, 'utf8')

  // Use gray-matter to parse the post metadata section
  const matterResult = matter(fileContents)

  // Use remark to convert markdown into HTML string
  const processedContent = await remark()
    .use(html)
    .process(matterResult.content)
  const contentHtml = processedContent.toString()

  // Combine the data with the id and contentHtml
  return {
    id,
    contentHtml,
    ...(matterResult.data as { date: string; title: string })
  }
}
```

HTML

npm install remark remark-html --save

```
processedContent VFile {
  data: {},
  messages: [],
  history: [],
  cwd: '/Users/johnahn/Downloads/next-learn-master/basics/typescript-fina
  value: `<p>We recommend using <strong>Static Generation</strong> (with
use your page can be built once and served by CDN, which makes it much fa
e on every request.</p>\n` +
  `<p>You can use Static Generation for many types of pages, including:
  <ul>\n` +
  `<li>Marketing pages</li>\n` +
  `<li>Blog posts</li>\n` +
  `<li>E-commerce product listings</li>\n` +
  `<li>Help and documentation</li>\n` +
  `</ul>\n` +
  `<p>You should ask yourself: "Can I pre-render this page <strong>ahead
answer is yes, then you should choose Static Generation.</p>\n` +
  `<p>On the other hand, Static Generation is <strong>not</strong> a go
ahead of a user's request. Maybe your page shows frequently updated data,
request.</p>\n` +
  `<p>In that case, you can use <strong>Server-Side Rendering</strong>
page will always be up-to-date. Or you can skip pre-rendering and use c
p>\n`
}

contentHtml <p>We recommend using <strong>Static Generation</strong> (wi
cause your page can be built once and served by CDN, which makes it much
age on every request.</p>
<p>You can use Static Generation for many types of pages, including:</p>
<ul>
  <li>Marketing pages</li>
  <li>Blog posts</li>
  <li>E-commerce product listings</li>
  <li>Help and documentation</li>
```

가져온 데이터 화면에서 보여주기

```
export default function Post({
  postData
}: {
  postData: {
    title: string
    date: string
    contentHtml: string
  }
}) {
```

```
<div>
  <Head>
    <title>{postData.title}</title>
```

```
</Head>
<article>
  <h1 className={homeStyles.headingXl}>{postData.title}</h1>
  <div className={homeStyles.lightText}>
    {postData.date}
  </div>
  <div dangerouslySetInnerHTML={{ __html: postData.contentHtml }} />
</article>
</div>
```

애플리케이션 스타일링

```
.container {  
    max-width: 36rem;  
    padding: 0 1rem;  
    margin: 3rem auto 6rem;  
}
```

