[스파르타코딩클럽] 자바스크립트 문법 뽀개기

매 주차 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

[수업 목표]

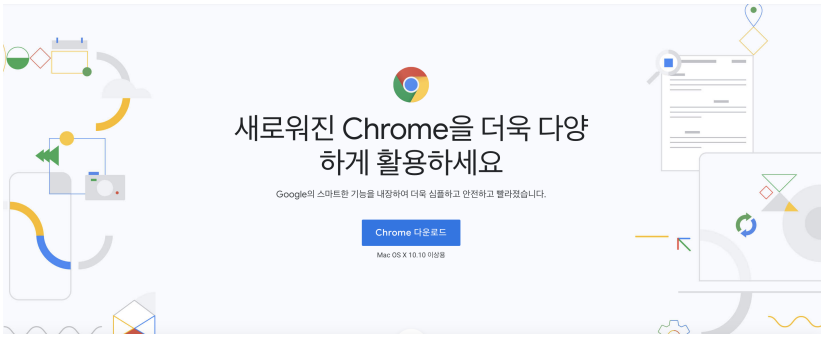
1. 자바스크립트 기초 문법에 익숙해집니다.
2. 퀴즈를 통해 배운 것들을 빠르게 복습합니다.
3. 프로그래밍에 대한 두려움을 없앱니다.

[목차]

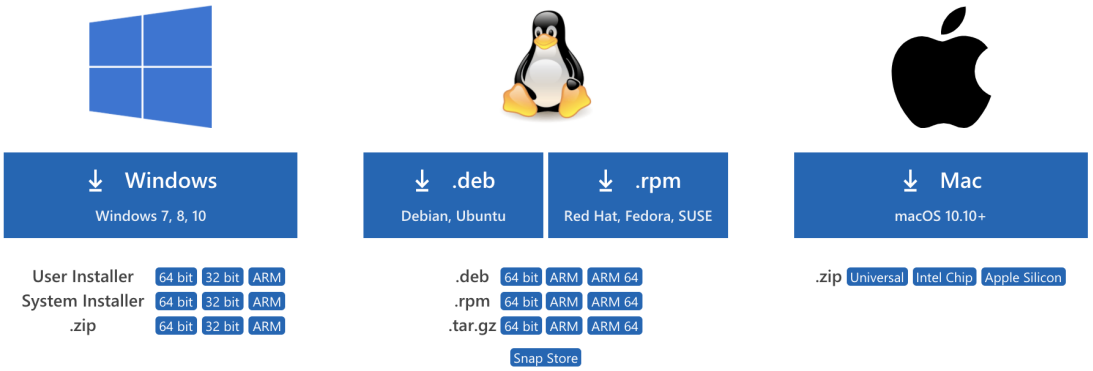
모든 토글을 열고 닫는 단축키
Windows : `Ctrl` + `alt` + `t`
Mac : `⌘` + `⌥` + `t`

1-0. 필수 프로그램 설치

▼ **Chrome : (다운로드 링크)**

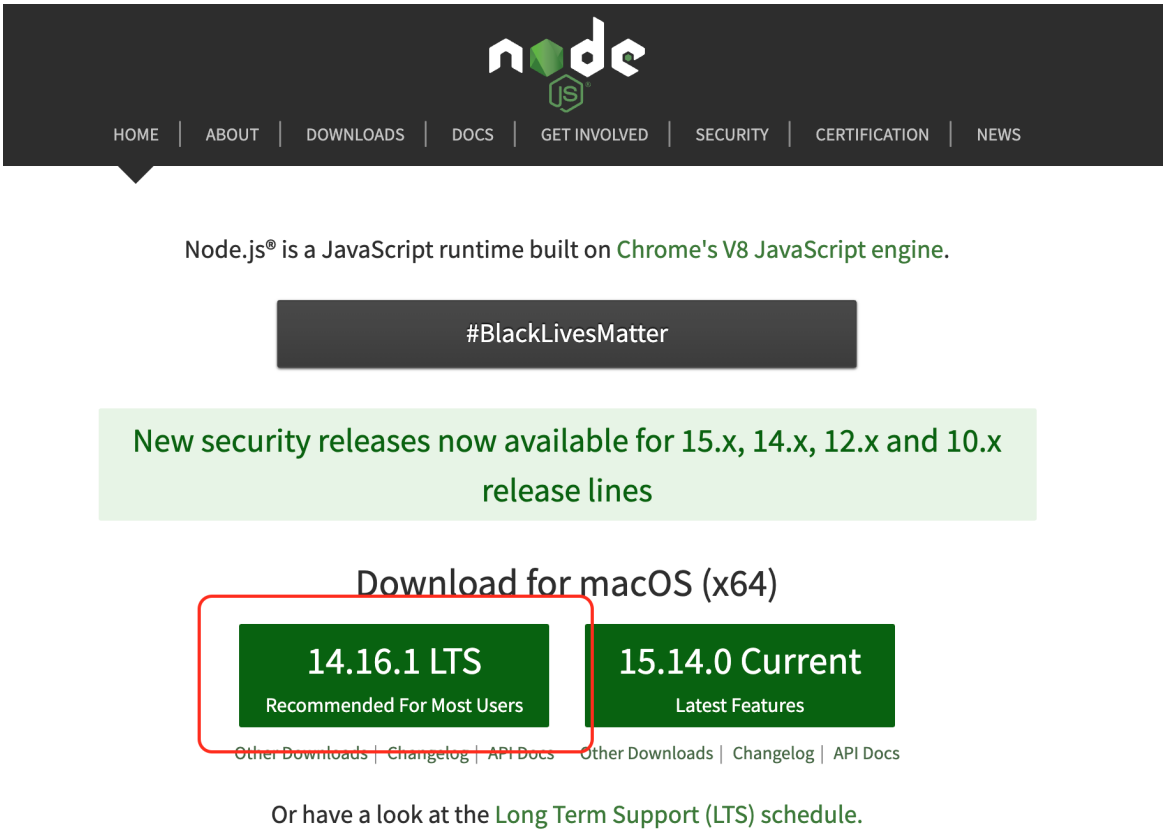


▼ **Visual Studio Code 설치 : (다운로드 링크)**



▼ **Node.js 설치: (다운로드 링크)**

Node.js 사이트에 들어가면 각자의 컴퓨터 운영체제에 맞는 다운로드가 제공됩니다. 빨간색 네모로 표시한 "LTS"라고 적혀 있는 영역을 클릭하여 Node.js 설치 파일을 다운로드 받고 실행하면 쉽게 설치할 수 있습니다. "14.16.1" 과 같은 버젼은 시간이 지남에 따라 계속 올라가는 것이기 때문에 해당 이미지와 달라도 상관 없습니다.



1-1. 시작하기에 앞서

▼ 프로그래밍이란?

- 자바스크립트 문법 뽀개기에 오신 여러분들을 환영합니다! 프로그래밍의 세계는 정말 방대합니다. 처음 접하시는 분들은 어디서부터 무엇을 시작해야할지 알기 어렵고, 처음부터 너무 깊은 내용을 들여다보려 하면 금방 지치고 흥미를 잃기 쉽습니다. 그래서 기초부터 차근차근 하나씩 하나씩 알아가면서 프로그래밍이라는 세계에 익숙해지는게 중요합니다. 이번 자바스크립트 문법 뽀개기를 통해 수강생분들이 프로그래밍 세계에 첫발을 잘 내딛으실 수 있도록 제가 여러분의 길잡이가 되겠습니다.

👉 프로그래밍, 코딩이란 무엇일까요?
우리는 이미 컴퓨터나 스마트폰으로 거의 모든 일을 하고 있습니다. 회사에서는 엑셀이라는 프로그램을 사용해 업무를 처리하고, 지금 하고 있으신 것처럼 노션이라는 웹사이트에 접속해 강의교재를 보며 공부도 하고, 쿠팡이라는 스마트폰앱을 사용해 손쉽게 쇼핑도 합니다.
방금 얘기한 엑셀, 노션, 쿠팡의 공통점은 무엇일까요? 이것들이 다 프로그래밍을 통해 만들어진 프로그램이라는 점입니다. 프로그래머들이 "어떤 도구"를 이용해 컴퓨터가 작동시킬 수 있는 프로그램을 만들었고, 우리는 마우스 클릭이나 키보드 입력과 같은 익숙한 방식으로 사용하는 것이죠.
우리가 이러한 프로그램을 사용하는 과정을 다시 한번 들여다볼게요. 마우스를 클릭하거나 키보드로 타이핑을 한다는 것은 데이터를 "입력"한다는 것입니다. 쇼핑몰에서 우리가 상품을 고르고 배송지와 결제카드번호 등을 입력하는 것을 떠올려보면 되겠죠. 그리고 최종 구매를 하겠다는 버튼을 누르면 그 정보들은 어딘가에 저장이 되어서 "처리"가 될 겁니다. 최종 결제 금액은 얼마인지, 언제 어떤 물건을 구매했는지 기록이 남을거고 배송이 진행되면서 배송 현황도 계속해서 업데이트가 됩니다. 그리고 이러한 모든 기록과 실시간 업데이트 상황이 화면에 "출력"되어서 우리가 쉽게 확인을 할 수 있습니다.
지금 얘기한 데이터의 입력 ⇒ 처리 ⇒ 출력이 바로 프로그램이 하는 일이고, 이렇게 프로그램이 정해진 방식에 따라 일할 수 있도록 작성해놓는게 프로그래밍이라고 할 수 있습니다.

▼ 자바스크립트란?

- 앞서 프로그래밍이란 무엇인가에 대해서 얘기하면서 프로그래머들이 "어떤 도구"를 이용해 코딩을 한다고 했습니다. 이 때 이용하는 도구가 바로 프로그래밍 언어입니다. 인간이 쓰는 언어와 마찬가지로 프로그래밍 언어도 정해진 문법이 있습니다. 이 문법에 따라 여러분들이 코딩을 하면 컴퓨터는 이를 해석해 우리가 원하는대로 데이터를 입력받아 처리하고 출력해줄 수 있습니다.
- 프로그래밍 언어도 인간이 쓰는 언어처럼 매우 다양하고 계속 발전합니다. 언어 고유의 특성과 장단점이 있지만 좀 더 많이 쓰이는 언어들이 존재하구요. 자바스크립트는 이러한 언어들 중에서 몇 손가락 안에 들만큼 많이 쓰이는 언어이고, 많은 프로그래머들로부터 사랑받고 있는 언어입니다.

👉 우리가 설치한 Node.js 란 무엇인가요?
"Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser"
Node.js를 정의하는 문장입니다. 아직은 한번에 딱 와닿지 않으시죠?
조금 더 쉬운 말로 얘기하자면 Node.js 는 우리가 자바스크립트 언어의 문법에 맞게 코딩을 했을 때 컴퓨터가 이를 해석해서 처리할 수 있도록 해주는 일종의 전용번역기라고 생각하시면 됩니다. 이 번역기가 이전에는 웹브라우저에만 있었지만, 우리 컴퓨터에 Node.js만 설치해주면 웹브라우저가 아닌 곳에서도 자바스크립트 코드를 실행하고 그 결과물을 바로 확인할 수 있어요.

1-2. Hello World

▼ Hello World

> 🔥 우리가 보통 코딩을 처음 시작하게 되면 하는 일이 "Hello World"라는 문구를 출력해보는 일입니다. 자바스크립트 프로그래밍 세계에 첫발을 내딛는 여러분을 격하게 환영합니다!

- 여러분 각자의 컴퓨터에서 원하는 폴더 위치에 "javascript" 라는 이름의 폴더를 만듭니다.

  VSCode를 실행하고 메뉴의 File - Open에서 이 폴더를 열어줍니다.

  메뉴의 File - New File을 클릭해 파일을 만들고 아래와 같이 타이핑을 한 이후에 hello.js 라고 저장해줍니다. 저장은 메뉴의 File - Save를 누르면 됩니다. "." 뒤에 js는 이 파일이 자바스크립트 파일이라는 것을 알려주는 확장자입니다. 이렇게 확장자를 명시해주면 코딩을 할 때 VSCode와 같은 에디터 툴이 지원해주는 여러 도움을 받을수도 수 있어요.

```
console.log('Hello World')
// Hello World 라는 문자열을 출력하는 명령어입니다.
// 앞으로도 계속 console.log 라는 명령어를 활용해 우리가 프로그래밍한 결과물들을 출력해볼거예요!
```

> 👉 console.log('Hello World') 밑에 있는 // 는 무엇인가요?
> 주석이라고 부르는 것입니다. 우리가 익숙히 알고 있는 주석과 같은 의미인데요, 컴퓨터가 해석해서 실행하는 부분이 아니기 때문에 코딩을 하면서 특별히 설명을 달고 싶은 것들이 있을 때 사용합니다.

- VSCode 상단에 위치한 메뉴바에서 Terminal - New Terminal을 클릭합니다. VSCode에서 화면이 상하로 분할되는게 보이시나요?
  아래쪽 화면에 커서를 위치하고 "node hello.js"라고 타이핑한 후 엔터를 누르면 "Hello World" 출력 완료!

> 🔥 우리가 터미널에서 node hello.js 라고 명령함으로서 컴퓨터는 Node.js를 활용해 hello.js 파일을 해석하고 그 결과물을 우리에게 출력해준 것입니다.
> 다시 한번 강조하지만 프로그래밍은
> 데이터 입력 ⇒ 입력받은 데이터 처리 ⇒ 처리된 최종 데이터를 출력하는 과정을 위한 작업입니다.
> 방금 우리가 해본 것도 "Hello World"라는 문자 데이터를 입력하고 이를 컴퓨터가 해석해서 다시 우리에게 출력해준 것이었습니다. 자바스크립트 문법 뽀개기를 통해 더 다양한 데이터를 입력하고 처리해서 출력해보는 과정을 반복해보도록 할거예요!

1-3. 변수

▼ 변수 선언과 데이터 할당

- 자바스크립트에서 변수를 선언할 때는 let 이라는 키워드를 사용해요!

  let 변수이름 = 값

  우리는 이걸 보통 "변수 <변수이름> 를 선언했고 해당 변수에 <값>을 할당했다" 고 얘기합니다.

👉 변수란 무엇일까요?
보통 프로그래밍 세계, 그리고 자바스크립트에서는 값(데이터)을 저장해놓고 사용하기 위해 변수라는 것을 사용합니다. 변수는 저장해놓은 값을 가리키는 일종의 '이름표'라고 생각할 수 있어요.
이렇게 변수를 사용하면 해당 데이터가 의미하는 바를 변수이름을 통해 정확히 표현할 수 있고, 계속해서 재사용도 할 수 있는데요, 이것들이 정확히 무엇을 의미하는지는 차근차근 감을 잡아보도록 할게요.

```
let name = 'Sinok Kim' // name이라는 변수에 Sinrok Kim 이라는 값을 할당
console.log(name) // 변수 name이 가리키고 있는 값 Sinrok Kim 을 출력

name = 'William' // 위에서 선언했던 name이라는 변수에 "William"이라는 값을 재할당
console.log(name) // 변수 name이 가리키고 있는 값 "William"을 출력
```

- 자바스크립트에서는 변수를 선언하는 const 라는 키워드도 있어요!

  const 변수이름 = 값

  const는 let과 달리 변수에 값을 재할당할 필요가 없을 때 써요. 해당 변수가 고정된 값을 계속 갖고 있을 때 쓰면 좋겠죠?

```
const name = "Sinok Kim" // name이라는 변수에 "Sinrok Kim"이라는 값을 할당
console.log(name) // 변수 name이 가리키고 있는 값 "Sinrok Kim"을 출력

name = "William" // 위에서 선언했던 name이라는 변수에 "William"이라는 값을 다시 재할당하려는 것이지만 실패. 에러 발생!
```

🔥 변수를 선언할 때 var 라는 키워드를 혹시 보신 적이 있나요?
let, const가 있기 전에는 실제로 var 를 써서 변수를 선언해야만 했습니다.
하지만 최신 자바스크립트에서는 let과 const를 쓰는 것이 맞습니다. var는 여러 단점들이 있기 때문에 더 이상 사용하지 않아야 해요.

1-4. 데이터 타입

▼ 데이터 타입

- 앞서 값(데이터)를 변수에 할당하고 출력하는 것을 해봤는데요, 자바스크립트에서는 여러 종류의 데이터 타입들이 존재해요. 이번 챕터에서는 기본 또는 원시형 (primitive) 타입이라고 불리는 number, string, boolean, null, undefined에 대해서 배워볼게요!

👉 왜 이런 데이터 타입들이 존재할까요?
프로그래밍이란 것은 수많은 데이터를 입력 ⇒ 처리 ⇒ 출력하는 과정을 컴퓨터가 알아들을 수 있는 언어로 서술하는 것이라고 할 수 있어요.
그런데 세상에는 정말 무수히 많은 데이터들이 존재하겠죠? 이러한 데이터들을 컴퓨터가 보다 빠르고 효율적으로 처리하기 위해 자바스크립트라는 프로그래밍 언어가 구분해놓은 것이라고 생각하면 좋을거 같아요.

- 숫자 (Number)

  말 그대로 숫자 데이터!

  나이, 거리, 무게, 가격.. 정말 무수히 많은 종류의 데이터를 숫자로 표현할 수 있어요!

```
console.log(10) // 10을 출력

const myAge = 37
const yourAge = 25

console.log(myAge) // 37을 출력
console.log(yourAge) // 25를 출력
```

- 문자열 (String)

  말 그대로 문자열 데이터. 이중 따옴표("")나 작은따옴표('')로 데이터를 감싸야 합니다.

  이름, 브랜드명, 제품명.. 역시 많은 종류의 데이터를 문자열로 나타낼 수 있겠죠!

```


const firstName = 'Sinrok'
const lastName = 'Kim'

console.log(firstName) // Sinrok을 출력
console.log(lastName) // Kim을 출력
```

- Boolean

  자바스크립트에서 참과 거짓을 나타내는 true / false를 표현하는 데이터

  나중에 배우게 될 비교연산자에서 많이 활용하게 돼요!

```
const isMan = true
const isWoman = false

console.log(isMan)
console.log(isWoman)
```

- null, undefined

  null은 텅텅 비어 있는 값을 의미합니다.

  undefined은 변수를 선언만 하고 값이 할당되어 있지 않은 것입니다.

```
let name1 = null
console.log(name) // null을 출력

let name2
console.log(name2) // undefined를 출력
```

- 자바스크립트에는 기본 타입 이외에 객체형 타입이라는 데이터도 존재해요. 이건 2주차 강의에서 배워보도록 할거예요.

▼ ✍️Q. 내가 원하는 변수들을 자유롭게 선언해보고 데이터를 할당한 뒤 출력해보아요.

```
const brandName = 'Nike'
const modelName = 'Air Jordan'
console.log(brandName)
console.log(modelName)

const shoesPrice = 200000
const capPrice = 180000
console.log(shoesPrice)
console.log(capPrice)
```

1-5. 연산자(1)

▼ 문자열 붙이기

- + 를 사용하여 문자열을 이어 붙일 수 있어요.

  추가로 문자열과 숫자를 이어붙이면 숫자가 문자로 인식된다는 사실!

```
console.log('My' + ' car') // My car를 출력
console.log('1' + 2) // 12를 출력
```

- 템플릿 리터럴 (Template literals)

  백틱(``) 을 사용하여 문자열 데이터를 표현할 수 있어요. 이중 따옴표나 작은 따옴표로 문자열을 표현할 때보다 간결하게 문자열 붙이기가 가능합니다.

```
const shoesPrice = 200000
console.log(`이 신발의 가격은 ${shoesPrice}원입니다`)
// console.log('이 신발의 가격은 ' + shoesPrice + '원입니다') 와 동일
// + 를 활용한 문자열 붙이기보다 간결하게 표현할 수 있다는 게 보이시나요?
```

▼ 산술연산자 (Numeric operators)

- 숫자 데이터에 대한 여러 연산들이 가능해요

  우리가 일상생활에서 많이 쓰는 사칙연산(+, -, *, /) 뿐만 아니라 // (나머지 연산), ** (거듭제곱) 도 있습니다.

```
console.log(2 + 1) // 3을 출력
console.log(2 - 1) // 1을 출력
console.log(4 / 2) // 2를 출력
console.log(2 * 3) // 6을 출력
console.log(10 % 3) // 나머지(remainder) 연산자. 1을 출력
console.log(10 ** 2) // exponentiation. 10의 2승인 100을 출력
```

▼ 증감연산자 (Increment and Decrement operators)

- 자기 자신의 값을 증가시키거나 감소시키는 연산자(++, —)라고 생각하시면 좋을거 같아요.
  이 증감연산자를 변수앞에 놓는냐, 변수뒤에 놓느냐에 따라 차이가 있는데요, 코드를 통해 확인해보도록 할게요.

```
let count = 1
const preIncrement = ++count
```

```
// 증감연산자를 앞에 놓게 되면 아래 주석으로 처리한 두 줄의 코드와 같은 내용입니다.
// 먼저 자기 자신에게 1을 더해서 재할당 한 후, 이를 preIncrement 에 할당했다는 의미입니다.
// count = count + 1
// const preIncrement = count
console.log(`count: ${count}, preIncrement: ${preIncrement}`) // count: 2, preIncrement: 2
```

```
let count = 1
const postIncrement = count++
// 증감연산자를 뒤에 놓게 되면 아래 주석으로 처리한 두 줄의 코드와 같은 내용입니다.
// postIncrement에 자기 자신의 값을 먼저 할당하고, 이후에 1을 더해서 재할당합니다.
// const postIncrement = count
// count = count + 1
console.log(`count: ${count}, postIncrement: ${postIncrement}`) // count: 1, postIncrement: 2
```

🔥 혹시 count 변수를 const 가 아닌 let 구문으로 선언한 이유를 눈치채셨나요?
우리가 증감연산자를 활용해 count의 값을 계속 증가시키고 다시 count에 할당하고 있기 때문에 const를 사용하면 에러가 발생해요!

▼ 대입연산자 (Assignment operators)

- 앞서 어떤 값을 어떤 변수에 할당한다는 표현을 많이 했는데요, 그게 바로 대입연산자를 사용한다는 의미였어요.

  = 뿐만 아니라 +=, -= 같은 것들을 통해서 연산과 대입을 한번에 할 수도 있어요.

```
const shirtsPrice = 100000
const pantsPrice = 80000
let totalPrice = 0

totalPrice += shirtsPrice // totalPrice = totalPrice + shirtsPrice 와 동일
console.log(totalPrice)
totalPrice += pantsPrice // totalPrice = totalPrice + pantsPrice 와 동일
console.log(totalPrice)

totalPrice -= shirtsPrice // totalPrice = totalPrice - shirtsPrice 와 동일
console.log(totalPrice)
```

1-6. 연산자(2)

▼ 비교연산자 (Comparison operators)

- 말 그대로 숫자값을 비교하는 연산자예요! 그리고 이러한 비교연산자를 통해서 얻는 값이 바로 boolean!

  뒤에서 배우게 될 조건문과 같이 많이 활용할겁니다.

```
console.log(1 < 2) // 1이 2보다 작은가? true
console.log(2 <= 2) // 2가 2보다 작거나 같은가? true
console.log(1 > 2) // 1이 2보다 큰가? false
console.log(1 >= 2) // 1이 2보다 크거나 같은가? false
```

▼ 논리연산자 (Logical operators)

- || (or), && (and), ! (not) 과 같은 연산자를 말해요. 이것 역시 조건문과 찰떡궁합입니다.

|| 는 연산 대상 중 하나만 true 여도 true 리턴

&& 는 연산 대상이 모두 true 여야만 true 리턴

! 는 true를 false로, flase를 true로 바꿔서 리턴

```
let isOnSale = true
let isDiscountItem = true

console.log(isOnSale && isDiscountItem) // true && true 이므로 true
console.log(isOnSale || isDiscountItem) // true || true 이므로 true

isOnSale = false
console.log(isOnSale && isDiscountItem) // false && true 이므로 false
console.log(isOnSale || isDiscountItem) // false || true 이므로 true

isDiscountItem = false
console.log(isOnSale && isDiscountItem) // false && false 이므로 false
console.log(isOnSale || isDiscountItem) // false || false 이므로 false

console.log(!isOnSale) // !false 이므로 true
```

▼ 일치연산자 (Equality operators)

- 두 값이 일치하는지를 비교해요!

```
console.log(1 === 1) // true
console.log(1 === 2) // false
console.log('Javascript' === 'Javascript') // true
console.log('Javascript' === 'javascript') // 대소문자나 띄워쓰기도 다 정확히 일치해야 해요. 따라서 false
```

🔥 일치연산자는 == 도 있지 않나요?
맞습니다. 자바스크립트에는 두 가지의 일치연산자가 있어요. 우리가 배운 === 는 엄밀한 (strict) 일치연산자여서 비교하는 두 값의 데이터타입과 값 자체가 정확히 일치해야만 true를 리턴합니다. 반면 == 는 비교하는 두 값의 데이터타입이 일치하지 않을 때 해당 값의 데이터타입을 자동으로 변환해주는 자바스크립트만의 특성이 있어요. 이 특성이 자칫 개발자의 실수를 유발할 가능성이 크기 때문에 실무에서는 거의 쓰지 않고 있습니다. 우리도 이번 과정에서는 정확한 프로그래밍을 위해 === 만 쓸거예요!

- == 과 === 의 차이 확인해보기

```
console.log(1 === "1") // false를 출력
console.log(1 == "1" // true를 출력
```

▼ ✍️ Q. 상품 가격을 나타내는 2개의 변수를 선언하고 각각의 변수에 원하는 가격값을 할당해보아요.
두 상품을 더한 가격 역시 총가격을 나타내는 변수에 할당해봅니다. 마지막으로 총가격의 20% 할인된 가격을 구해서 '총 몇 원에 물건을 구입합니다.' 라는 문자열을 출력합니다.

```
const shoesPrice = 200000
const capPrice = 100000
const totalPrice = shoesPrice + capPrice

consoel.log(`총 ${totalPrice * 0.8}원에 물건을 구입합니다.`)
```

1-7. 조건문(1)

▼ if

- if 구문을 활용해 조건을 만족했을 때만 코드를 실행하도록 할 수 있어요. 이 조건의 결과값이 바로 앞서 우리가 배운 Boolean입니다. Boolean을 리턴하는 연산자는 비교연산자, 논리연산자, 일치연산자가 있었다는 것 기억나시나요?

  if (조건) { 조건을 만족할 때 실행할 코드 }

```
const shoesPrice = 40000
if (shoesPrice < 50000) { // 신발 가격이 50000원보다 작으므로 해당 코드가 실행됨
  console.log('신발을 사겠습니다.')
}

const capPrice = 50000
if (capPrice < 50000) {
  console.log('모자를 사지 않겠습니다.') // 모자 가격이 50000원보다 작지 않으므로 해당 코드가 실행되지 않음
}
```

if 구문의 body(중괄호 안쪽) 코드를 작성할 때 들여쓰기를 했다는게 보이시나요? 자바스크립트에서는 들여쓰기를 할 때 보통 탭 (Tab) 키를 사용합니다.
사실 들여쓰기는 반드시 하지 않아도 상관없습니다. 하지만 들여쓰기를 통해 해당 코드가 if 구문안에 속하는 코드라는 것을 보다 명시적으로 보여줄 수가 있습니다.
비록 코드를 최종적으로 해석하는 것은 컴퓨터지만 이 코드를 읽고 작성하는 주체는 우리와 같은 사람입니다. 그래서 코드를 작성할 때는 가독성을 최대한 높일 수 있도록 하는게 좋습니다!

1-8. 조건문(2)

▼ else, else if

- if 구문의 조건을 만족하지 않았을 때 실행하고 싶은 코드를 else 구문과 함께 작성합니다.

```
const shoesPrice = 50000
if (shoesPrice < 40000) {
  console.log('신발을 사겠습니다.')
} else {
  console.log('너무 비싸요. 신발을 사지 않겠습니다.') // 신발 가격이 40000원보다 작지 않으므로 해당 코드가 실행됨
}
```

- else if 구문을 활용하면 보다 더 많은 조건을 판단하고 코드를 실행할 수 있습니다.

```
const shoesPrice = 50000
if (shoesPrice < 40000) {
  console.log('신발을 사겠습니다.')
} else if (shoesPrice <= 50000) {
  console.log('고민을 해볼게요...')  // 신발 가격이 50000원보다 작거나 같으므로 않으므로 해당 코드가 실행됨
} else {
  console.log('너무 비싸요. 신발을 사지 않겠습니다.')
}
```

▼ ✍️ Q. 거리를 의미하는 변수를 선언하고 원하는 숫자값 (단위는 km라고 가정) 을 할당합니다. 2km 미만이면 "걸아가자"를, 2km 이상이고 5km 미만이면 "택시를 타자"를, 그 외에는 "기차를 타자"를 출력해봅니다.

```
const distance = 2
if (distance < 2) {
  console.log("걸어가자")
} else if (distance >= 2 && distance < 5) { // 논리연산자를 && 를 이렇게 활용할 수 있어요!
  console.log("택시를 타자")
} else {
  console.log("기차를 타자")
}
```

2-1. 반복문(1)

▼ while

- 반복문을 활용해서 특정 코드를 반복해서 실행할 수 있어요. 이 때 조건을 설정해서 우리가 원하는 만큼만 반복할 수 있도록 합니다!

  while (조건) { 조건을 만족할 때 실행할 코드 }

  ```
  let temperature = 20
  while (temperature < 25) {
    console.log(`${temperature}도 정도면 적당한 온도입니다.`)
    temperature++ // 증감연산자를 활용해서 온도를 변화시킵니다.
  }
  ```

  여기서 주의할 것은 반복문의 조건에 포함된 변수의 값을 계속 변화시켜줘서 언젠가는 반복문이 끝날 수 있도록 해줘야 한다는 겁니다. 위의 코드에서도 온도를 1도씩 계속 올려서 반복문의 조건이 언젠가는 false를 리턴하고 바디의 코드가 실행되지 않습니다. 반복문의 조건이 계속해서 true 를 리턴한다면 무한루프에 빠져서 프로그램이 끝나지 않습니다!

  🔥 실수로 무한루프에 빠져서 프로그램의 실행이 끝나지 않는다면 ctrl + c 를 눌러서 중단시킵니다.

2-2. 반복문(2)

▼ for

- while 과 같은 반복문입니다. 좀 더 명시적으로 반복문의 조건을 표현할 수 있습니다.

  for (begin; condition; step) { 조건을 만족할 때 실행할 코드 }

  ```
  for (let temperature = 20; temperature < 25; temperature++) {
    console.log(`${temperature}도 정도면 적당한 온도입니다.`)
  }
  ```

  for문이 실행되는 순서를 한번 정리해볼까요?

  1. temperature라는 변수를 선언하고 값을 할당합니다. (begin)

2. temperature가 25보다 작은지 연산합니다. 결과값이 true라면 계속 실행. false라면 for 문 종료 (condition)
3. 중괄호 안의 코드가 실행됩니다.
4. temperature에 1을 더해서 재할당하고 2번 과정부터 다시 반복합니다. (step)

2-3. 반복문과 조건문 활용

▼ 반복문과 조건문의 만남

- 반복문과 조건문은 코딩하는데 있어 핵심 중의 핵심입니다. 조금 과장을 보태서 우리가 짜는 프로그램은 반복문과 조건문의 무수한 집합이라고도 할 수 있습니다. 입력된 데이터에 대해 조건문을 활용해 수많은 조건으로 분기하고 반복문을 활용해 반복해서 처리하는 게 프로그램이 주로 하는 일이기 때문이죠.
- 반복문과 조건문을 같이 활용하여 1 ~ 10까지의 숫자중 3으로 나누었을 때 나머지가 0인 숫자를 구해서 출력해봐요.

```
for (let number = 1; number <= 10; number++) {
  if (number % 3 === 0) {
    console.log(`${number}는 3으로 나눠서 떨어지는 숫자입니다.`)
  }
}
```

반복문 안에 넣은 조건문이 number의 값을 검사하는 코드가 이해되시나요? 이 때 코드의 가독성을 높이기 위해 들여쓰기 규칙을 지킨 것도 다시 한번 짚고 넘어가요!

▼ ✍️Q. 1부터 20까지의 숫자중 홀수인 경우는 '숫자 ...은 홀수입니다.'를 짝수인 경우는 '숫자 ...은 짝수입니다' 를 출력하는 프로그램을 작성해봅니다.

```
for (let number = 1; number <= 20; number++) {
  if (number % 2 === 0) {
    console.log(`숫자 ${number}는 짝수입니다.`)
  } else {
    console.log(`숫자 ${number}는 홀수입니다.`)
  }
}
```

2-4. 함수(1)

▼ 함수란?

- 함수는 특정 작업을 수행하는 코드의 집합이라고 볼 수 있습니다.
  잠시 코드를 통해서 이 함수라는게 왜 필요한 것인지 생각해보도록 할게요.

```
const priceA = 1000
const priceB = 2000
// 두 상품가격의 합과 평균을 구해서 출력해야 하는 코드 작성
const sum1 = priceA + priceB
console.log(`두 상품의 합계는 ${sum1}입니다.`)
const avg1 = sum1 / 2
console.log(`두 상품의 평균은 ${avg1}입니다.`)

const priceC = 3000
const priceD = 4000
```

```
// 이번에도 두 상품가격의 평균을 구해서 출력해야 한다면? 위와 동일한 코드를 또 작성...
const sum2 = priceC + priceD
console.log(`두 상품의 합계는 ${sum2}입니다.`)
const avg2 = sum2 / 2
console.log(`두 상품의 평균은 ${avg2}입니다.`)

// 한두번은 괜찮은데 이렇게 수십, 수백번씩 평균을 구하고 출력해야 한다면?
```

- 이렇게 반복되는 특정 작업을 수행해야 한다면 그 코드 자체를 어딘가에 만들어서 저장해놓고 사용할 수 있지 않을까요? 마치 변수에 데이터를 할당해놓고 계속 사용하는 것처럼요. 바로 이 때 등장하는게 함수입니다!

▼ 함수의 선언과 호출

- 함수의 선언
  변수를 선언하고 해당 변수에 값을 할당했던 것처럼, 함수도 선언을 하고 해당 함수가 실행할 코드의 집합을 만들어서 저장해줍니다.

  function 함수명(매개변수들...) {

  이 함수에서 실행할 코드들

  return 반환값

  }

  여기서 함수명은 함수가 하는 일들을 대표할 수 있는 이름이면 좋겠죠?

  매개변수(parameter) 는 해당 함수의 바디에 있는 코드에서 사용할 수 있는 일종의 변수이고, 함수 호출시 전달하게 됩니다. 함수를 실행하기 위해서 필요한 일종의 input 이라고 생각하면 될거 같아요.
  중괄호 안에는 이 함수가 해야할 일들을 코드로 쭉 작성하고, 반환해야 할 값을 명시해줍니다.
  예시를 통해서 같이 볼게요!

```
// 함수의 선언
function calculateAvg(price1, price2) {
  const sum = price1 + price2 // 매개변수인 price1, price2을 변수처럼 활용!
  console.log(`두 상품의 합계는 ${sum}입니다.`)
  const avg = sum / 2
  return avg // 평균가격을 리턴!
}
```

- 함수의 호출

  함수를 선언만 하고 끝내면 안되겠죠? 실제 이 함수를 사용하기 위해서는 호출을 해야 합니다.

  const 변수명 = 선언한 함수명(매개변수들...)

```
const priceA = 1000
const priceB = 2000
// 함수의 호출
const avg1 = calculateAvg(priceA, priceB)
console.log(`두 상품의 평균은 ${avg1}입니다.`)

const priceC = 3000
const priceD = 4000
// 함수의 호출
const avg2 = calculateAvg(priceC, priceD)
console.log(`두 상품의 평균은 ${avg2}입니다.`)
```

2-5. 함수(2)

▼ 함수 호출시 코드의 흐름

- 함수 호출시 어떻게 코드가 전개되는건지 헷갈리시지는 않나요? 다시 한번 코드의 흐름을 복기해볼게요!
  1. 함수 calculateAvg를 호출하면서 변수 priceA와 priceB를 매개변수로 전달
  2. 함수 calculateAvg의 바디 코드가 실행됨. 이 때 1번에서 전달한 매개변수의 값이 함수를 선언할 때 썼던 매개변수명인 price1, price2에 할당되었다고 보면 됨
  3. 함수의 바디 코드가 최종적으로 변수 avg를 리턴하고 있고, 이것이 변수 avg1에 할당됨

▼ ✍️Q. 세 개의 물건가격을 매개변수로 전달받아 평균값을 리턴하는 함수를 정의하고, 그 함수를 호출해서 평균값을 출력해보세요.

```
function calculateAvg(price1, price2, price3) {
  const avg = (price1 + price2 + price3) / 3
  return avg
}

const priceA = 1000
const priceB = 2000
const priceC = 3000
const avg = calculateAvg(priceA, priceB, priceC)
console.log(`평균가격: ${avg}`)
```

2-6. 클래스와 객체(1)

▼ 객체(Object) 타입

- 변수 시간에 배웠던 데이터의 기본 타입들을 아직 잊지 않으셨죠? 데이터의 기본 타입을 활용해서 필요한 데이터들을 적절히 표현하고 연산도 할 수 있었습니다. 그런데 이런 필요성이 생깁니다. 관련있는 데이터들을 묶어서 한번에 잘 나타낼수 있는 데이터 타입은 없을까? 노트북 전문 쇼핑몰을 생각해볼게요. 여기서 파는 모든 노트북들은 이름, 가격, 제조사와 같은 데이터들을 가지고 있을 겁니다. 그런데 이러한 것들을 기본 타입만으로 표현한다면 그 데이터들을 묶어서 표현한다는 게 쉽지 않습니다.

  ```
  // 노트북1을 것을 표현하기 위한 데이터들
  // 변수명을 명시적으로 하는 것 이외에는 이 데이터들의 관계를 표현해줄 수 있는 방법이 없음
  const notebook1Name = 'Macbook'
  const notebook1Price = 2000000
  const notebook1Company = 'Apple'

  // 이름, 가격, 제조사와 같은 정보를 다 담을 수 있는 좀 더 큰 범위의 데이터 타입이 있으면 좋지 않을까?
  ```

  이 때 사용하는 데이터 타입이 바로 객체 타입 입니다. 객체를 좀 어렵게 얘기하면 '물리적으로 존재하거나 추상적으로 생각할 수 있는 것중에서 자신의 속성을 갖고 있고 다른 것과 식별 가능한 것' 을 의미합니다. 위의 예시 코드를 통해 다시 생각해보면 노트북1은 이름, 가격, 제조사라는 자신만의 속성을 갖고 있고 다른 노트북들과 식별 가능한 것이니 객체로 나타낼 수 있는 것이죠!

▼ 클래스(Class) 선언

- 객체를 만들때 마치 설계도처럼 사용하는 것이 바로 클래스입니다. 그래서 흔히들 클래스는 템플릿이고 객체는 이를 구체화한 것이라고도 하죠. 우리가 함수를 정의하고 해당 함수를 필요할 때 계속 사용할 수 있었다는 것 기억하시나요? 마찬가지로 클래스를 미리 정의해놓으면 필요할 때마다 그 클래스를 사용해서 동일한 모양을 가진 객체를 만들 수 있습니다.

```
class Notebook {
  constructor(name, price, company) {
    this.name = name
    this.price = price
    this.company = company
  }
}
```

해당 코드를 한번 찬찬히 살펴볼게요.

1. class 키워드와 클래스명

   class는 클래스를 선언하는 문구이고 그 뒤에 바로 클래스 명이 나옵니다. 클래스명도 마치 변수명처럼 내가 표현하고자 하는 데이터를 잘 나타낼 수 있는 이름이 좋겠죠? 위의 예에서 Notebook 대신 Person 같은 이름을 쓴다면 다른 사람들이 코드를 봤을 때 이상하다고 생각할 겁니다.

2. 생성자 (constructor)

   중괄호 안에는 생성자라는 것을 적어줍니다. 혹시 생성자가 함수와 많이 비슷하다는 것 눈치채셨나요? 이 생성자는 말 그대로 나중에 객체가 '생성'이 될 때 자바스크립트 내부에서 호출이 되는 함수라고 생각해주시면 됩니다. 생성자를 좀 더 살펴보면 3개의 매개변수를 정의했고 각각의 이름은 name, price, company 네요.

3. this와 속성(property)

   생성자의 바디를 보면 this 라는 키워드가 등장하네요. 이 this는 클래스를 사용해 만들어질 객체 자기 자신을 의미하고 this 뒤에 붙는 name, price, company는 객체의 속성입니다.
   생성자의 바디에서는 함수 호출시 전달할 매개변수 name, price, compay를 객체의 속성 name, price, company에 각각 할당하고 있는 것입니다.

▼ 객체 만들기

- 객체를 만들어요!

  const 변수명 = new 클래스명(생성자에서 정의한 매개변수들...)

```
const notebook1 = new Notebook('MacBook', 2000000, 'Apple')
```

클래스를 활용해 객체를 만들 때는 new 라는 키워드를 먼저 써주고 클래스명을 마치 함수처럼 호출하면서 매개변수값을 전달해주면 됩니다. 그러면 해당 클래스의 생성자가 호출되면서 객체가 생성되고 객체의 속성들에 매개변수값들이 할당되겠죠.

만들어진 객체는 변수에 할당해줍니다. 기본 타입의 데이터들을 변수에 할당하면 변수를 사용해 해당 데이터에 접근할 수 있었습니다. 객체도 마찬가지입니다. 객체도 변수에 할당하고 나면 해당 변수를 활용해 객체에 접근할 수 있습니다.

객체의 속성 하나하나에 접근해 데이터를 갖고와야 할 때도 있겠죠? 이 때는
this.속성명 을 사용합니다.

```
console.log(notebook) // Notebook { name: 'Macbook', price: 2000000, company: 'Apple' }
console.log(notebook.name) // MacBook
consoel.log(notebook.price) // 2000000
console.log(notebokk.company) // Apple
```

2-7. 클래스와 객체(2)

▼ 메소드 (method)

- 클래스에는 데이터(값)를 나타내는 속성 뿐만 아니라 함수와 같이 특정 코드를 실행할 수 있는 메소드도 정의할 수 있습니다. 객체를 생성한 후, 만들어진 객체의 메소드를 호출하면 됩니다.

```
// 클래스 선언
class Product {
  constructor(name, price) {
    this.name = name
    this.price = price
  }

  printInfo() {
    console.log(`상품명: ${this.name}, 가격: ${this.price}원`)
  }
}

// 객체 생성 및 메소드 호출
const notebook = new Product('Apple Macbook', 2000000)
notebook.printInfo() // 상품명: Apple Macbook, 가격: 2000000원
```

▼ 객체 리터럴(Object Literal)

- 자바스크립트에서는 객체 리터럴을 활용해서 바로 객체를 만들 수도 있습니다. 객체 리터럴은 클래스와 같은 템플릿 없이 빠르게 객체를 만들 수 있는 방법이라고 생각해주시면 될 거 같아요. 2개 이상의 속성과 메소드가 있을 때는 쉼표로 구별해주고 가독성을 위해서 줄바꿈도 해주는 게 좋습니다.

  const 변수명 = {

  속성명: 데이터,

  메소드명: function () { 메소드 호출시 실행할 코드들 }

  }

```
const computer = {
  name: 'Apple Macbook',
  price: 20000,
  printInfo: function () {
    console.log(`상품명: ${this.name}, 가격: ${this.price}원`)
  }
}

computer.printInfo()
```

  name, price라는 속성과 printInfo 라는 메소드를 가지고 있는 객체를 만들어서 computer라는 변수에 할당하는 코드입니다. 해당 객체의 printInfo 메소드를 바로 호출까지 해보았습니다.

  결과적으로는 클래스를 활용해 객체를 만든 것과 동일합니다. 그렇다면 왜 굳이 복잡하게 클래스를 정의하는 걸까요? 바로 재사용성 때문입니다. 한번 클래스를 만들어두면 같은 속성과 메소드를 갖고 있는 객체를 훨씬 더 간결한 코드로 만들 수 있습니다.

▼ ✍️Q. 여러분만의 의류 쇼핑몰을 만들려고 합니다. 옷의 종류는 많지만 기본적으로 색깔, 사이즈, 가격의 속성을 갖고 있네요. 그리고 이 옷들의 세 속성을 바로 확인할 수 있게 출력해주는 메소드가 필요할 거 같습니다. 클래스와 객체를 활용해 작성해보아요.

```
class Cloth {
  constructor(color, size, price) {
    this.color = color
```

```
    this.size = size
    this.price = price
  }

  printInfo() {
    console.log(`색깔: ${this.color}, 사이즈: ${this.size}, 가격: ${this.price}`)
  }
}

const shirts = new Cloth('white', 'M', '50000')
const coat = new Cloth('navy', 'L', '200000'

shrits.printInfo()
coat.printInfo()
```

2-8. 배열(1)

▼ 배열(Array) 이란?

- 자바스크립트에서 데이터를 표현하기 위한 방법으로 기본타입과 객체를 배웠습니다. 이걸로도 우리는 분명 많은 것들을 할 수 있죠. 하지만 같은 형식의 많은 데이터를 순서대로 저장하고자 할 때는 데이터의 수만큼 변수들을 선언해줄 수밖에 없었습니다. 이 때 쓰는 것이 바로 배열입니다.

  같은 타입의 데이터들을 하나의 변수에 할당하여 관리하기 위해 사용하는 데이터 타입이라고 기억해두시면 좋을거 같아요!

▼ 배열의 선언

- 숫자 1, 2, 3, 4, 5로 이루어진 배열을 선언하는 방법은 아래처럼 두 가지가 있습니다.

```
// 1번째 방법
const arr1 = new Array(1, 2, 3, 4, 5)

// 2번째 방법
const arr2 = [1, 2, 3, 4, 5]
```

  1번째 방법은 앞서 우리가 클래스를 활용해 객체를 만든 것과 똑같지 않나요? 맞습니다. Array라는 클래스를 활용해서 객체를 만들었다고 생각해주시면 될 거 같아요. 우리가 직접 Array라는 클래스를 선언한 적은 없지만 자바스크립트 내부적으로 이미 갖고 있는 것이구요. 그래서 우리는 바로 사용을 하기만 하면 됩니다.

  2번째 방법은 배열을 바로 만드는 방법입니다. 대괄호 안에 우리가 배열로 저장할 데이터를 쭉 나열해주면 됩니다. 정말 간단하죠? 그래서 배열을 만들때는 1번째보다는 2번째 방법을 많이 사용합니다.

▼ 배열 안의 데이터

- 배열에 있는 데이터 각각을 우리는 요소 (element) 라고 부릅니다. 당연히 이 요소들에 쉽게 접근해서 바로 쓸수도 있어야 하지 않을까요? 객체도 속성명을 통해 해당 데이터에 쉽게 접근할 수 있었습니다. 배열에서는 인덱스(index) 가 객체의 속성명과 같은 역할을 해줍니다.

  인덱스는 배열 안의 데이터들이 자리잡은 순서라고 생각해주시면 될 거 같아요. 특이한 점은 이 인덱스가 0부터 시작한다는 사실! 코드로 바로 확인해볼게요.

```
const rainbowColors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']

console.log(rainbowColors[0]) // 0번 인덱스를 통해서 데이터에 접근. red
console.log(rainbowColors[1]) // 1번 인덱스를 통해서 데이터에 접근. orage
console.log(rainbowColors[2]) // 2번 인덱스를 통해서 데이터에 접근. yellow
console.log(rainbowColors[3]) // 3번 인덱스를 통해서 데이터에 접근. green
console.log(rainbowColors[4]) // 4번 인덱스를 통해서 데이터에 접근. blue
console.log(rainbowColors[5]) // 5번 인덱스를 통해서 데이터에 접근. indigo
console.log(rainbowColors[6]) // 6번 인덱스를 통해서 데이터에 접근. violet
```

▼ 배열의 길이

- 배열은 같은 형식의 데이터를 순서대로 저장하는 것이라고 했습니다. 그렇다면 이 배열이 얼마나 많은 데이터를 갖고 있는지, 그 길이를 알 필요도 있지 않을까요? 위의 예시에서 rainbowColors는 요소의 갯수가 7개니까 그나마 눈으로 셀 수 있었지만 배열 안의 요소가 이보다 훨씬 더 많다면 정말 만만치 않은 일이 될 겁니다. 이 때 사용하는 것이 바로 length 라는 속성입니다!

```
const rainbowColors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']

console.log(rainbowColors.length) // 7을 출력

console.log(rainbowColors[rainbowColors.length - 1]) // length 를 응용하여 배열의 마지막 요소도 쉽게 찾아서 출력 가능!
```

2-9. 배열(2)

▼ 요소 추가와 삭제

- 배열을 선언하고 난 이후에 새로운 요소를 더하거나 빼야 할 필요도 있지 않을까요?

  이 때 사용하는 것이 push 와 pop 이라는 메소드입니다.

```
const rainbowColors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']

rainbowColors.push('ultraviolet') // 배열의 마지막에 ultarviolet 추가
console.log(rainbowColors) // ultraviolet이 추가된 rainbowColors 출력

rainbowColors.pop() // 배열의 마지막 요소 ultraviolet을 제거
console.log(rainbowColors) // ultraviolet이 제거된 rainbowColors 출력
```

▼ 배열과 반복문

- 배열의 요소들을 하나씩 꺼내서 출력해야 하는 코드를 작성해야한다고 했을 때 좀 더 간결한 방법이 없을까요? 바로 반복문을 활용하면 됩니다.

```
const rainbowColors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']

for (let i = 0; i < rainbowColors.length; i++) {
  console.log(rainbowColors[i])
}
```

  이 반복문을 좀 더 자세히 분석해볼까요?

  1. 배열의 인덱스는 0부터 시작이니 변수 i의 시작값도 0으로!
  2. i가 배열의 길이보다 작을 때에만 반복문 안의 코드 실행
  3. 모든 요소를 빠짐없이 다 출력해야 하므로 i는 1씩 증가

- 배열과 함께 좀 더 자주 쓰이는 간단한 형식의 for 문도 있어요!

```
const rainbowColors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']

for (const color of rainbowColors) {
  console.log(color)
}
```

배열에서 요소들을 차례대로 하나씩 찾아 color라는 변수에 할당합니다. 자동으로 배열의 끝까지 반복문이 실행되기 때문에 첫번째 for 문보다 쓰기 편합니다.

▼ ✍️Q. 열 개의 상품 가격 데이터를 갖고 있는 배열을 만듭니다. 반복문을 활용해 상품들의 가격 합계와 평균을 구해보아요.

```
const priceList = [1000, 2000, 5000, 7000, 10000, 9000, 3000, 15000, 20000, 17000]
let sum = 0

for (const price of priceList) {
  sum += price
}

const avg = sum / priceList.length
console.log(`합계: ${sum}, 평균: ${avg}`)
```

반복문을 활용해 배열 안의 요소들에 접근하고 그 값들을 sum이라는 변수에 계속 누적시키고 있는 코드가 이해되시나요?
평균을 구할 때는 앞서 구한 sum과 배열의 length를 활용하면 됩니다!