자바스크립트 기본

Hello, world!

- script 태그를 활용하여 자바스크립트 코드 작성 가능

```
<script type="text/javascript">
    alert('Hello, world!');
</script>
```

- type 속성 (type="text/javascript")
  - 문서는 텍스트로 이루어져있으며 내용은 자바스크립트라는 의미의 MIME 타입
  - CSS의 경우 MIME 타입을 text/css로 설정
- type 속성은 필수가 아니므로 생략 가능

외부 스크립트 활용

- 자바스크립트 코드를 분리하고 싶은 경우, 파일에 코드를 작성한 후 **스크립트 태그의 src 속성을 이용**하여 파일 불러오기 가능

```
<script src="/path/to/script.js"></script>
```

- src에 붙은 경로는 사이트의 루트에서부터 파일이 위치한 절대 경로를 나타냄
- URL 전체를 속성으로 사용할 수도 있음

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js">
</script>
```

- 복수개의 스크립트 파일 혹은 여러 스크립트 태그를 삽입 가능

```
<script src="/path/to/script1.js"></script>
<script src="/path/to/script2.js"></script>
<script>
    // ... 내부 스크립트 1 ...
</script>
<script>
    // ... 내부 스크립트 2 ...
</script>
```

- 스크립트가 길어지면 별개의 분리된 파일로 만들어 저장하는 것이 성능상의 이유로 좋음 (캐시를 사용하여 브라우저에 사본 저장 가능)
- 태그 내부에 이벤트 핸들러 관련 속성(onclick, onkeypress 등)을 정의하며 자바스크립트 코드 삽입 가능

- 그러나 html 태그와 자바스크립트 코드가 섞여 유지보수가 어려워지므로 권장되지 않는 방법

```
<button onclick="alert('click')">click me</button>
<input type="text"
    onkeypress="console.log('keypress')"
    onchange="console.log('change')" />
```

코드 구조

- 문(statement)은 어떤 작업을 수행하는 문법 구조(syntax structure)와 **명령어(command)**를 의미
- 주석

```
// 한 줄 주석
/*
    여러 줄 주석
*/
```

변수 상수

- **let 키워드**를 이용하여 변수 생성 가능

```
let message = "Hello";

// 에러 발생
// Uncaught SyntaxError: Identifier 'message' has already been declared
let message = "World";
```

- 가독성을 위해 가급적 한 줄에는 하나의 변수를 작성

변수 명명 규칙

1. 변수명에는 오직 문자와 숫자, 그리고 기호 중 $와 _만 들어갈 수 있습니다.
2. 첫 글자는 숫자가 될 수 없습니다.
3. 대소문자 구별
4. 비라틴계 문자(ex: 한자)도 사용 가능 (권장 X)
5. 예약어 사용 불가
6. 여러 단어를 조합하여 변수명을 만들 땐 **카멜 표기법(camelCase)** 사용 권장

상수

- **const 키워드**를 이용하여 정의
- 처음 값 대입 이후 값 변경 불가
- 상수는 **대문자와 밑줄로 구성된 이름으로 명명**을 권장

```
const MY_CONSTANT = 1234;
```

자료형

- 자바스크립트의 변수는 **자료형에 관계없이 모든 데이터 대입 가능** (자료형에 대한 제약이 적음)
    - **동적 타입 언어**(JS, 파이썬) <=> 정적 타입 언어(JAVA, C)
        - 정적 타입 언어의 경우 변수나 상수를 선언하는 시점에 자료형이 고정됨
- 총 **여덟 가지 자료형**이 존재
- **숫자형**
    - 정수와 실수 모두 포함 (다른 언어(ex: C, Java)처럼 정수형, 실수형 타입을 구분하지 않음)
    - Infinity
        - 0으로 나눌 경우 다른 언어와 같이 예외처리 되지 않고 무한대 상수값을 반환
            - https://stackoverflow.com/questions/8072323/best-way-to-prevent-handle-divide-by-0-in-javascript
    - NaN
        - 숫자로 표현이 불가능한 값인 경우 Not a Number 상수값 반환
- **문자열**
    - 큰따옴표, 작은따옴표, 역따옴표(백틱) 사용 가능
    - 역따옴표 내부에서는 **${...}안에 표현식(expression) 사용** 가능
- 한 글자만 저장하는 글자형(char) 타입 존재하지 않음
- **불린형** (true, false)
- **null**
    - 존재하지 않는(nothing) 값, 비어 있는(empty) 값, 알 수 없는(unknown) 값을 나타내는 데 사용
- **undefined**
    - **"값이 할당되지 않은 상태"**를 나타낼 때 사용
    - 변수는 선언했지만, 값을 할당하지 않았다면 해당 변수에 undefined 값이 할당됨
    - 함수에서 값을 반환하지 않는 경우 undefined 값을 반환함
    - undefined 값을 직접 할당(대입)하는 걸 권장하진 않음, 만약 **값을 모르거나, 값이 없음을 명시해야 한다면 null 값을 할당**
- **객체(object)**
    - 원시(primitive) 자료형과는 다르게 데이터 컬렉션이나 복잡한 개체 표현 가능 (다양한 데이터를 집합으로 표현 가능하다고 이해)
- **객체외에는 모두 원시 자료형**

- **typeof 연산자**로 자료형을 확인 가능
    - typeof x 또는 typeof(x) 형태로 사용

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof true // "boolean"
typeof "foo" // "string"
typeof Math // "object"
typeof null // "object"
typeof alert // "function"
```

- null은 별도의 **고유한 자료형을 가지는 특수 값으로 객체(object)가 아니지만**, 하위 호환성을 유지하기 위해 이런 오류를 수정하지 않고 남겨둔 상황임을 유의
- **함수형은 별도로 존재하지 않음**, 함수는 내부적으로 **객체**로 취급됨 (하지만 typeof 연산자를 사용할 경우 "function" 문자열을 반환)
    - (*) 자바스크립트 언어에서 모든 원시 자료형(number, bigint, string, boolean, undefined, null, symbol)이 아닌 값들은 **모두 객체로 취급**됨

alert, prompt, confirm을 이용한 상호작용

- alert
    - 함수가 실행되면 사용자가 확인(OK) 버튼을 누를 때까지 메시지를 보여주는 모달창이 계속 떠있음
        - 모달이란 단어엔 페이지의 나머지 부분과 상호 작용이 불가능하다는 의미가 내포, 따라서 확인 버튼을 누를 때까지 모달 창 바깥에 있는 버튼을 누른다든가 하는 행동을 할 수 없음
        - prompt, confirm 함수와는 다르게 반환값이 없음 (undefined 반환)
- prompt
    - 함수가 실행되면 텍스트 메시지와 입력 필드(input field), 확인(OK) 및 취소(Cancel) 버튼이 있는 모달 창을 띄워줌

```
// title : 사용자에게 보여줄 문자열
// default : 입력 필드의 초깃값
prompt(title, [default]);

// ex
let age = prompt('나이를 입력해주세요.', 100);
```

- confirm
    - confirm 함수는 매개변수로 받은 question(질문)과 확인 및 취소 버튼이 있는 모달 창을 보여줌

```
let isBoss = confirm("당신이 주인인가요?");
```

        - 사용자가 확인버튼를 누르면 true, 그 외의 경우는 false를 반환

형 변환

- String 함수를 호출해 전달받은 값을 문자열로 변환 가능
    - false는 문자열 "false"로, null은 문자열 "null"로, undefined는 "undefined" 변환
- Number 함수를 호출해 전달받은 값을 숫자로 변환 가능

    - 숫자 이외의 글자가 들어가 있는 문자열을 숫자형으로 변환하려고 하면, 변환이 불가능하므로 결과는 NaN이 됨

```
  Number("임의의 문자열 123");
```

- Boolean 함수를 호출하면 명시적으로 불리언으로의 형 변환을 수행
    - 숫자 0, 빈 문자열(""), null, undefined, NaN과 같이 직관적으로도 "비어있다고" 느껴지는 값(falsy value)들은 false가 되며 그 외의 값(truthy value)은 true로 변환

기본 연산자와 수학

```
+, -, *, /, %, **
```

- **는 거듭제곱 연산자
- + 연산자를 이용하여 문자열 연결 가능
- 피연산자 중 하나가 문자열이면 다른 하나도 문자열로 변환 (실수 유발 가능)
- + 연산자를 단항 연산자로 사용할 경우 숫자 타입으로 변환 진행

```
// 전혀 권장하지 않지만 동작
+true   // 1
+""     // 0
```

- Number 함수를 호출하여 숫자로 변환하는 것보다 좀 더 짧게 코드를 작성 가능하다는 장점이 있음

```
let apples = "2";
let oranges = "3";

// 이항 덧셈 연산자가 적용되기 전에, 두 피연산자는 숫자형으로 변화합니다.
alert( +apples + +oranges ); // 5
```

- NaN에 대한 숫자 연산은 모두 NaN을 반환 (undefined에 대한 숫자 연산도 NaN을 반환)

```
alert( NaN + 1 )
alert( NaN - 1 )
alert( NaN * 1 )
alert( NaN / 1 )
```

연산자 우선순위

- 후위형, 전위형 증감 연산자의 연산 결과값에 차이가 있음을 주의
- 아래의 경우 증감이 이루어진 후 counter 값이 반환되어 a에 저장됨

```
let counter = 1;
let a = ++counter;

alert(a); // 2
```

- 아래의 경우 증감이 이루어지기 전 counter 값이 반환되어 a에 저장되고 그 이후에 증감 연산이 수행됨

```
let counter = 1;
let a = counter++;

alert(a); // 1
```

- 반환 값을 사용하지 않는 경우라면, 전위형과 후위형엔 차이가 없음

비교 연산자

```
>, >=, <, <=, ==, ===, !=, !==
```

- **엄격한 비교 연산자(strict equality operator)** 사용이 권장됨
  - === 연산자
  - 비교 연산자(==)와 달리 === 연산자는 값의 타입까지 비교함
  - 반대의 케이스는 !== 연산자 사용

자바스크립트 기본

if와 '?'를 사용한 조건 처리

- https://developer.mozilla.org/en-US/docs/Glossary/Falsy
- 다음 값은 모두 false로 평가됨 (false로 평가되므로 falsy value라고도 부름)
  - 숫자 0
  - 빈 문자열 ("")
  - null
  - undefined
  - NaN
    - null과 undefined의 경우 유용하게 사용되는 케이스 있음 (유효한 값이 없을 경우 처리할 로직을 처리하는케이스)
- 반대(그 외 참 값으로 평가되는 값)는 truthy value라고 부름
- 조건부 연산자는 물음표(?)로 표시
  - **피연산자가 세 개**이기 때문에 조건부 연산자를 삼항(ternary) 연산자라고도 칭함
- 평가 대상인 condition이 truthy라면 콜론 기호 기준으로 왼쪽값(value1)이, 그렇지 않으면 오른쪽값(value2)이 반환됨

```
let result = condition ? value1 : value2;
```

- 가독성에 따라 적절히 if 구문이나 삼항 연산자를 선택하여 사용

논리 연산자

- 단락 평가(short circuit evaluation)
  - OR은 왼쪽부터 시작해서 오른쪽으로 평가를 진행하는데, truthy를 만나면 **나머지 값들은 건드리지 않은 채 평가를 멈춤**
  - OR는 하나만 true이면 true를 반환해도 되므로 뒤의 식을 평가할 필요가 없기 때문!
- AND 연산자(&&)도 OR 연산자와 비슷한 순서로 동작
  - 단, 평가된 값이 false인 경우 평가를 멈춤 (OR과 반대)
    - AND는 하나만 false이면 false를 반환해도 되므로 뒤의 식을 평가할 필요가 없기 때문!
- NOT을 두 개 연달아 사용(!!)하면 값을 불린형으로 변환
  - Boolean 함수를 써서 변환하는 것보다 짧게 코드 작성 가능

```
// truthy value => true
alert( !!"non-empty string" ); // true
// falsy value => false
alert( !!null ); // false
```

while과 for 반복문

while 반복문

- 기본 문법

```
while (condition) {
  // 코드
  // '반복문 본문(body)'이라 불림
}
```

- 활용 사례

```
let i = 0;
// 0, 1, 2가 출력
while (i < 3) {
  alert( i );
  i++;
}
```

- do - while 문 사용 가능

```
do {
  // 반복문 본문
} while (condition);
```

- 활용 사례

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

for 반복문

- 기본 문법

```
for (begin; condition; step) {
  // ... 반복문 본문 ...
}
```

- 활용 사례

```
// 0, 1, 2가 출력
for (let i = 0; i < 3; i++) {
  alert(i);
}
```

구성 요소

begin	**i = 0**	**반복문에 진입할 때 단 한 번 실행됩니다. (초기식)**
condition	i < 3	반복마다 해당 조건이 확인됩니다. false이면 반복문을 멈춥니다. (조건식)
body	alert(i)	condition이 truthy일 동안 계속해서 실행됩니다.
step	i++	각 반복의 body가 실행된 이후에 실행됩니다. (증감식)

- 변수를 반복문 안(초기식)에서 선언할 경우 **인라인 변수 선언**이라고 부르며 이렇게 선언한 변수는 반복문 안에서만 접근 가능, 만약 반복문 바깥에서도 변수에 접근하고 싶다면 미리 변수를 선언하면 됨

```
// 반복문 바깥에 변수 정의
let i = 0;

// 기존에 정의된 변수(i)를 사용하여 증감
for (i = 0; i < 3; i++) {
  alert(i);
}

// 3, 반복문 밖에서 선언한 변수이므로 사용할 수 있음
alert(i);
```

- 초기식, 조건식, 증감식 생략 가능

```
// 끊임 없이 본문이 실행 (무한루프)
for (;;) {
    // 본문
}
```

- **break 구문**을 이용하여 반복문 빠져나오기 가능
- **continue 구문**을 이용하여 조건을 만족할 경우 특정 코드를 생략하고 다시 반복을 진행하도록 설정 가능

switch문

- 복수의 if 조건문을 switch문으로 대체 가능

```
switch(x) {
    case 'value1':  // if (x === 'value1')
        ...
        [break]
    case 'value2':  // if (x === 'value2')
        ...
        [break]
    default:
        ...
        [break]
}
```

- 변수 x의 값과 첫 번째 case문의 값 'value1'를 일치 비교(===)한 후, 두 번째 case문의 값 'value2'와 비교하며, 이런 과정을 계속 반복함
- case문에서 변수 x의 값과 일치하는 값을 찾으면 해당 case문의 아래의 코드가 실행되며 break문을 만나면 swtich문 탈출, (만약 break문이 없다면 계속해서 인접한 case문 아래의 코드를 실행)
- **default문**이 있는 경우, 값과 일치하는 case문이 없다면 default문 아래의 코드가 실행

```
let a = 2 + 2;

switch (a) {
    case 3:
        alert( '비교하려는 값보다 작습니다.' );
        break;
    case 4:
        alert( '비교하려는 값과 일치합니다.' );
        break;
    case 5:
        alert( '비교하려는 값보다 큽니다.' );
        break;
    default:
        alert( "어떤 값인지 파악이 되지 않습니다." );
}
```

- 아래 코드의 경우 break문이 없으므로 case 4이후의 모든 case 내부 코드가 실행됨

```
let a = 2 + 2;

switch (a) {
    case 3:
        alert( '비교하려는 값보다 작습니다.' );
    // 이 시점에 a와 값이 일치하는데, break문이 없으므로 case 4, 5, default 아래 코드
모두 실행
    case 4:
        alert( '비교하려는 값과 일치합니다.' );
    case 5:
        alert( '비교하려는 값보다 큽니다.' );
    default:
        alert( "어떤 값인지 파악이 되지 않습니다." );
}
```

- switch 문 오른쪽 괄호와 case 오른쪽에 **표현식 사용 가능**

```
let a = "1";
let b = 0;

// switch 문 오른쪽 괄호에 표현식 사용 가능
switch (+a) {
    // case 오른쪽에 표현식 사용 가능
    case b + 1:
        alert("표현식 +a는 1, 표현식 b+1는 1이므로 이 코드가 실행됩니다.");
        break;
    default:
        alert("이 코드는 실행되지 않습니다.");
}
```

여러 개의 case문 묶기

- switch/case문에서 break문이 없는 경우엔 **조건에 상관없이 다음 case문이 실행**되므로 주의!

```
let a = 3;

switch (a) {
    case 4:
        alert('계산이 맞습니다!');
        break;
    // (*) 두 case문을 묶음 (즉, a가 3이거나 5이면 case 5 아래 코드 실행)
    // 즉, case 3과 case 5에서 실행하려는 코드가 같은 경우
    case 3:
    case 5:
        alert('계산이 틀립니다!');
        alert("수학 수업을 다시 들어보는걸 권유 드립니다.");
        break;
    default:
        alert('계산 결과가 이상하네요.');
}
```

자바스크립트 기본

함수

- 유사한 동작을 하는 코드가 여러 곳에 있을 경우 묶어 주기
- 함수는 프로그램을 구성하는 **주요 구성 요소(building block)**로 함수를 이용하면 중복 없이 **유사한 동작을 하는 코드를 여러 번 호출 가능**
- 언어 자체에서 제공하는 빌트인(built-in) 함수(parseInt, eval 등등)와 브라우저에서 제공하는 빌트인 함수(alert, prompt, confirm 등등)가 존재함
- 언어에서 제공하는 빌트인 함수외에도 직접 **사용자 함수를 정의**하여 호출 가능

함수 선언

```
function 함수이름(파라미터) {
  // ...함수 본문...
}
```

- 함수 정의 및 호출

```
function showMessage() {
  alert( '안녕하세요!' );
}

// 함수 호출
showMessage();
```

- 함수 내에서 선언한 변수인 **지역 변수(local variable)는 함수 안에서만 접근 가능**

```
function showMessage() {
  let message = "안녕하세요!"; // 지역 변수

  // 함수 내부에서만 message 접근 가능
  alert( message );
}
```

- 함수 내부에서 외부 영역에 정의된 외부 변수(outer variable)에 접근 가능 (값 수정도 가능)

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);

  // 외부 변수값 수정
  userName = "Bob";
}
```

- 매개변수(parameter)를 이용하면 임의의 데이터를 함수 안에 전달 가능
    - 함수에 전달된 매개변수는 복사된 후 함수의 지역변수가 됨

```
// 두 개의 파라미터 존재(from, text)
function showMessage(from, text) {
  alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!');
```

- 매개변수에 값을 전달하지 않으면 그 값은 undefined가 됨
    - 다른 언어와 달리 함수에 정의된 **파라미터 값을 모두 전달하지 않아도 정상동작**함을 유의!

```
// 이 경우 text에는 undefined가 할당됨
showMessage("Ann");
```

- 기본값 할당 가능
    - 매개변수 오른쪽에 대입 기호(=)을 붙이고 undefined 대신 설정하고자 하는 기본값을 써주기

```
// 만약 text 값이 전달되지 않을 경우 "no text given" 값을 기본값으로 사용
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}
```

매개변수 기본값 설정할 수 있는 또 다른 방법

- **ES6 이전 문법으로 코드 작성** 시 활용할 수 있는 방법
- 매개변수를 undefined와 비교

```
function showMessage(text) {
    if (text === undefined) {
        text = '빈 문자열';
    }

    alert(text);
}
```

- OR 연산자 사용

```
function showMessage(text) {
    text = text || '빈 문자열';
    // ...
}
```

- null 병합 연산자(nullish coalescing operator) 사용

```
function showCount(count) {
    alert(count ?? "unknown");
}

// 0은 falsy 값이지만 값은 엄연히 존재하므로 0 출력
showCount(0);
// null => unknown 출력
showCount(null);
// undefined => unknown 출력
showCount();
```

반환 값

- **return 지시자를 이용**하여 함수 반환값 설정 가능

```
function sum(a, b) {
    return a + b;
}
```

- return문이 없거나 return 지시자만 있는 함수는 **undefined를 반환**함을 유의! (함수는 모두 값을 반환한다고 봐도 무방)

```
function doNothing() { /* empty */ }

// true
alert( doNothing() === undefined );
```

- return문을 만나면 함수는 즉시 종료됨

함수 표현식

- **함수 선언문(Function Declaration)** 방식으로 함수 만들기

```
function sayHi() {
  alert( "Hello" );
}
```

- **함수 표현식(Function Expression)** 을 사용해서 함수 만들기
    - 함수를 생성하고 **변수에 값을 할당하는 것처럼 함수가 변수에 할당**됨
- 자바스크립트에서 함수는 값이므로 **함수를 값처럼 취급**할 수 있음

```
// 함수 표현식을 사용해서 함수 만들기
// 함수를 만들고 그 함수를 변수 sayHi에 할당
let sayHi = function() {
    alert( "Hello" );
};

// 함수도 값처럼 취급되므로 대입하여 함수 복사 가능 (sayHi 함수에 괄호가 없음(=함수 호출 아님)을 유의!)
let func = sayHi;

// "Hello" 경고문 출력
func();
```

- 함수는 값이므로 **함수의 인자값으로 함수 전달 가능**

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}

function showOk() {
    alert( "동의하셨습니다." );
}

function showCancel() {
    alert( "취소 버튼을 누르셨습니다." );
}

// 사용법: 함수 showOk와 showCancel가 ask 함수의 인수로 전달됨
ask("동의하십니까?", showOk, showCancel);
```

- 함수 ask의 인수, showOk와 showCancel은 **콜백 함수(callback function) 또는 콜백**이라고 불리움
    - 왜냐하면 함수를 직접 호출하는 것이 아니고 **파라미터로 전달한 함수가 나중에 호출(called back)**되기 때문

```
// 다음과 같이 사용 가능
ask(
    "동의하십니까?",
    function() { alert("동의하셨습니다."); },
    function() { alert("취소 버튼을 누르셨습니다."); }
);
```

- 위와 같이 이름 없이 선언한 함수는 **익명 함수(anonymous function)**라고 불리움

함수 표현식 vs 함수 선언문

- **차이1** : 문법
    - 함수를 선언하는 생김새가 다름

```
// 함수 선언문
function sum(a, b) {
    return a + b;
}

// 함수 표현식
let sum = function(a, b) {
    return a + b;
};
```

- **차이2** : 함수 생성 시점
    - 함수 표현식은 실제 **실행 흐름이 해당 함수에 도달했을 때 함수를 생성**
    - 함수 선언문은 **함수 선언문이 정의되기 전에도 호출** 가능

```
// 이 시점에 함수 호출 가능
sayHi("John");

// 함수 호출 구문 뒤에 함수 정의해도 사용 가능
function sayHi(name) {
    alert( `Hello, ${name}` );
}
```

함수 표현식은 함수 표현식 구문을 해석한 시점 이후부터 호출 가능

```
// 호출 불가 (Uncaught ReferenceError: sayHi is not defined)
sayHi("John");

// 함수 표현식을 이용한 값 할당 이후 호출 가능
let sayHi = function(name) {
    alert( `Hello, ${name}` );
}

// 이 시점에서는 호출 가능
sayHi("John");
```

- 특별히 함수 표현식을 써야 할 이유가 있지 않다면 **함수 선언문 방식으로 함수를 정의하는 것이 권장**됨

화살표 함수 기본 (*)

- 함수 표현식보다 **단순하고 간결한 문법으로 함수를 선언**하는 방식 => **화살표 함수(arrow function)** 사용
- 화살표 함수 선언 문법
    - 화살표 오른쪽의 **표현식이 평가되고 반환**됨
        - 구문(statement)과 표현식(expression)의 차이 생각해보기

```
let func = (arg1, arg2, ...argN) => expression
```

- 활용 사례

```
let sum = (a, b) => a + b;
```

- **전달할 파라미터가 하나**밖에 없다면 파라미터를 감싸는 괄호를 생략 가능

```
// 전달할 파라미터가 한 개이므로 괄호 생략 가능
let double = n => n * 2;

// 물론 써줘도 무방함
let double = (n) => n * 2;
```

- 전달할 파라미터가 하나도 없을 경우 괄호만 쓰기 (이때는 괄호 생략 불가)

```
let sayHi = () => alert("안녕하세요!");
```

- 화살표 함수를 이용해서 더 간결한 코드 작성 가능
  - **함수 본문이 한 줄이거나, 표현식으로 대체 가능한 간단한 함수**는 화살표 함수를 사용해서 만드는 것이 편리

```
let age = prompt("나이를 알려주세요.", 18);

let welcome = (age < 18) ?
    () => alert('안녕') :
    () => alert("안녕하세요!");
```

- 평가해야 할 표현식이나 구문이 여러 개인 복잡한 함수를 화살표 함수로 정의할 경우 **중괄호 블록과 명시적인 return 문을 이용**

```
// 중괄호는 본문 여러 줄로 구성되어 있음을 알려줌
let sum = (a, b) => {
  let result = a + b;
  // 중괄호를 사용했다면, return문을 이용하여 결괏값을 반환해야 함
  return result;
};
```

객체:기본

ps. "프로퍼티"라는 용어와 "속성"이라는 용어가 같은 의미를 가리킴을 유의 (자주 혼용해서 사용)

객체

- 객체형은 원시형과 달리 **다양한 데이터**를 함께 담을 수 있음
- 중괄호 안에는 **키(key): 값(value) 쌍으로 구성된 프로퍼티(property)**를 여러 개 넣을 수 있는데, 키엔 **문자형**, 값엔 **모든 자료형**이 허용됨
- 객체를 생성하는 두 가지 방법

```
// '객체 생성자' 문법
let user = new Object();

// '객체 리터럴' 문법 (중괄호 이용)
// 객체를 선언할 땐 주로 객체 리터럴 방법을 사용
let user = {};
```

- 중괄호 {...} 안에는 **키:값 쌍**으로 구성된 프로퍼티(=속성)가 들어감
    - **콜론(:)**을 기준으로 **왼쪽엔 키가, 오른쪽엔 값이 위치**

```
// '객체 리터럴' 문법으로 객체 생성
let user = {
    // 키: "name", 값: "John"
    name: "John",
    // 키: "age", 값: 30
    age: 30
};
```

- **점 표기법(dot notation)**을 이용하여 프로퍼티 값에 접근 가능

```
// 점 왼쪽에 객체의 이름이, 오른쪽에는 속성 이름이 위치
alert( user.name ); // John
alert( user.age ); // 30
```

- 객체에 **동적으로 속성을 추가 가능**
    - 클래스에 미리 정의해놓은 속성(ex: 자바의 클래스 필드)만 사용 가능한 타 언어와 다른 점

```
user.isAdmin = true;
```

- **delete 연산자**를 사용하여 프로퍼티 삭제 가능
    - delete 연산자 사용시 **프로퍼티 삭제가 성공했으면 true, 실패했으면 false를 반환**

```
// age 프로퍼티 삭제
delete user.age;
console.log(user.age); // undefined

// 반환값이 필요한 경우
let result = delete user.age;
console.log(result); // true

// Math 객체의 PI 상수는 삭제 불가
console.log(delete Math.PI); // false
console.log(Math.PI); // 3.14159...
```

- 스페이스나 하이픈(-)과 같은 기호를 조합해서 프로퍼티 이름을 만들어야 할 경우, 프로퍼티 이름을 **따옴표로 감싸야 함** (비권장)

```
let user = {
    name: "John",
    age: 30,
    // 스페이스나 하이픈이 포함된 프로퍼티 이름을 사용해야 할 경우 따옴표로 묶어주기
    "likes birds": true
};
```

- **상수로 선언된 객체도 객체의 내용은 수정 가능함**을 유의
    - 상수 => **새로운 값의 대입(값의 변경)이 불가능**하다는 의미

```
const user = {
    name: "John"
};

// 상수에는 새로운 값의 "대입"이 불가능함! (Uncaught TypeError: Assignment to
constant variable.)
user = {}

// 객체를 통해 접근하여 객체의 속성값을 수정하는 것은 가능함
user.name = "Pete";
```

- **대괄호**를 이용하여 프로퍼티 접근, 수정(추가), 삭제 가능
    - 스페이스나 하이픈(-)과 같은 기호를 조합해서 프로퍼티 이름을 만든 경우, **대괄호와 문자열을 통해서만 접근 가능** => 점 표기법으로는 접근 불가

```
let user = {};

// 수정 (추가)
user["likes birds"] = true;
// 점 표기법으로는 접근 불가능!
// user.likes birds = true;
// user.likes-birds = true;

// 접근
alert(user["likes birds"]);

// 삭제
delete user["likes birds"];
```

- 만약에 **코드 실행 중에 변하는 값(ex: 변수)을 이용하여 프로퍼티에 접근**하고 싶은 경우 대괄호를 이용하여 접근

```
let user = {
    name: "John",
    age: 30
};

let key = prompt("사용자의 어떤 정보를 얻고 싶으신가요?", "name");

// 변수를 이용하여 프로퍼티 접근 (변수값이 평가되고 이후 평가된 값을 키로 이용하여 속성값 접근)
alert( user[key] );
// 밑의 방법은 "key라는 이름의 속성"에 접근하는 것임을 유의
// alert( user.key );
```

값 단축 구문 사용

- **값 단축 구문(property value shorthand)**을 이용한 객체 생성 간소화
    - 프로퍼티 값을 변수에서 받아온 값으로 설정할 때 유용하게 사용 가능

```
let name = "John";
let age = 20;

// 값 단축 구문없이 객체 정의
let user = {
    name: name,
    age: age
}

console.log( user ); // { name: "John", age: 20 }

// "변수와 속성 이름이 같은 경우" 다음과 같이 축약하여 객체 정의 가능
user = {
    // name: name과 같은 의미
    name,
    // age: age와 같은 의미
    age
}

console.log( user ); // { name: "John", age: 20 }

function makeUser(name, age) {
    return { name, age };
}

console.log( makeUser("John", 20) ); // { name: "John", age: 20 }
```

- 다음과 같이 일반 프로퍼티와 단축 프로퍼티를 함께 사용하는 것도 가능

```
let name = "John";

let user = {
    // name에만 값 단축 구문 사용
    name,
    age: 30
};
```

프로퍼티 이름의 제약사항

- 변수 이름엔 for, let, return 같은 예약어를 사용할 수 없지만, 객체 프로퍼티엔 이런 제약이 적용되지 않음

```
// 예약어를 키로 사용해도 무방함
let obj = {
    for: 1,
    let: 2,
    return: 3
};

alert( obj.for + obj.let + obj.return ); // 6
```

- 객체의 키로 문자열 자료형(+심볼형)만 사용 가능하므로, **문자열이나 심볼형에 속하지 않은 값은 문자열로 자동 형 변환됨**

```
let obj = {
    // 숫자형이므로 문자열로 변환 (0 => "0")
    // 즉, "0": "test"와 동일
    0: "test"
};

alert( obj["0"] ); // "test"
// 여기서 전달된 숫자 0은 문자열 "0"으로 변환됨을 유의
alert( obj[0] ); // "test"
```

in 연산자를 사용한 속성 존재 여부 확인

- 자바스크립트 객체의 중요한 특징 => **존재하지 않는 프로퍼티에 접근하려 해도 에러가 발생하지 않고 undefined를 반환**함

```
let user = {};

// noSuchProperty 프로퍼티가 존재하지 않으므로 true
alert( user.noSuchProperty === undefined );
```

- **in 연산자**를 사용하여 특정 속성이 객체에 존재하는지 여부를 파악 가능

```
let user = {
    name: "John"
};

// name 속성이 존재하므로 true 출력
alert( "name" in user );
// noSuchProperty 속성이 존재하지 않기 때문에 false 출력
alert( "noSuchProperty" in user );
```

- 존재하지 않는 속성값과 undefined를 비교하여 속성값 존재 여부를 파악하는 것도 가능하지만, **속성값이 undefined인 경우 문제 발생 가능**

```
let user = {
    name: undefined
};

// true
alert( "name" in user );
// name 속성은 존재하지만 undefined이므로 true 출력
alert( user.name === undefined );
```

for .. in 반복문

- for .. in 반복문을 사용하면 **객체의 모든 키를 순회** 가능
    - 단, 배열과 같은 자료구조에 **저장된 요소들을 순회**하기 위해서 for .. in 반복문을 사용하면 안됨, 그런 상황에는 **for .. of 반복문**을 사용해야 함

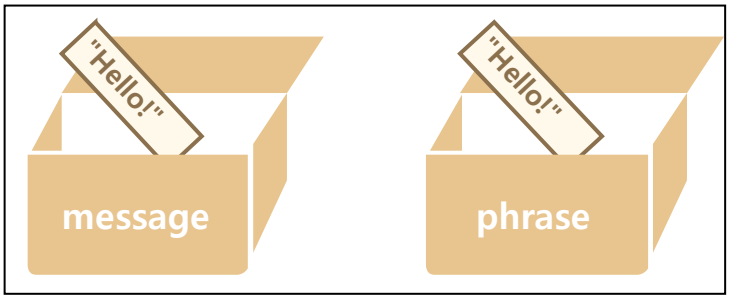
```
let user = {
    name: "John",
    age: 30,
    isAdmin: true
};

// name : John
// age : 30
// isAdmin : true
for (let key in user) {
    // key 변수에는 "name", "age", "isAdmin"이 순차적으로 대입됨
    alert(key + " : " + user[key]);
}
```

참조에 의한 객체 복사

- 객체와 원시 타입의 근본적인 차이 중 하나는 **객체는 참조에 의해(by reference) 저장되고 복사된다는 점**
- 원시 값을 대입할 경우 말 그대로 **값의 복사(deep copy)**가 일어남

```
let message = "Hello!";
// 원시값 대입으로 인하여 값의 복사가 발생
let phrase = message;
```

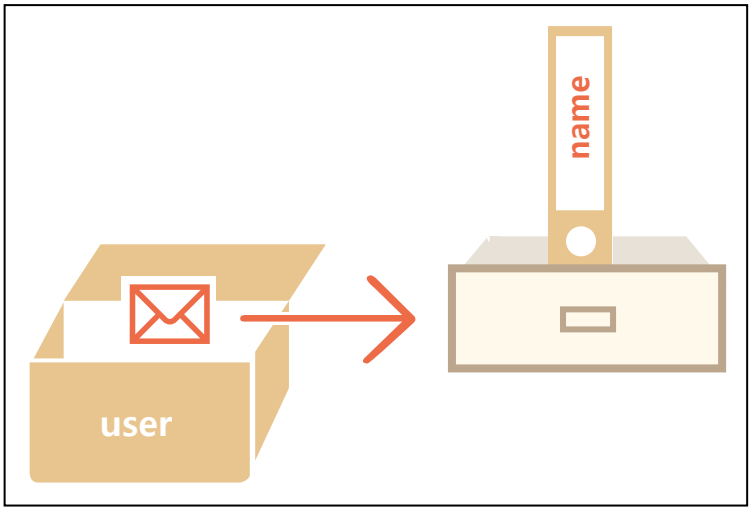
- 위 코드는 다음과 같이 **서로 다른 변수에 각기 독립된 값("Hello")을 저장**



- 객체를 변수에 대입할 경우 **객체에 대한 참조(=메모리 주소, 화살표)가 저장**됨 (C로 치자면 포인터와 비슷한 개념으로 이해)

```
let user = {
    name: "John"
};
```

다음과 같이 값 자체가 아닌, 해당 **값(객체)을 가리키고 있는 화살표를 저장**한다고 이해하기




```
let user = { name: "John" };

// 참조값을 대입(복사)함, 따라서 admin과 user 변수는 "같은 객체"를 가리키게 됨
let admin = user;

// admin 참조에 의해서 값이 변경됨
admin.name = 'Pete';

// (둘이 가리키는 객체가 결국 "같은 참조를 가진 같은 객체"이므로) user 참조 값을 이용해서도 변경사항을 확인 가능
alert(user.name); // Pete 출력
```

- 함수의 인수로 전달받는 과정에서도 참조값의 복사가 진행됨

```
let user = { name: "John" };

function changeName(user) {
    // 원본 객체에 영향을 끼침
    user.name = "Pete";
}

alert(user.name); // "John"
changeName(user);
alert(user.name); // "Pete"
```

- **객체끼리 값 비교시 ==, === 연산자는 동일한 결과를 출력**
    - 어차피 비교 대상이 둘 다 객체이므로 타입까지 검사하는 === 연산자와 차이가 없어짐
    - 비교 후 **피연산자인 두 객체가 동일한 참조를 가진 객체인 경우에 참을 반환**함

```
let a = {};
let b = a; // 참조에 의한 복사

alert( a == b ); // true, 두 변수는 같은 객체를 참조합니다.
alert( a === b ); // true, == 연산자와 결과 같음
```

내용이 같아도 참조가 다르면 false를 반환함을 유의

```
let a = {};
let b = {}; // 독립된 두 객체
alert( a == b ); // false, 같은 객체를 참조하지 않음

let user1 = { name: "John" };
let user2 = { name: "John" };
alert( user1 == user2 ); // false
// name 속성값은 객체가 아니므로 값을 그대로 비교하게되어 true가 반환됨
alert( user1.name == user2.name ); // true
```

객체:기본

메서드와 this

- 자바스크립트에선 객체의 **프로퍼티에 함수를 할당**해 객체에게 행동할 수 있는 능력을 부여
  - **객체 지향 프로그래밍 => 정보 + 행동**
  - 자바스크립트에서는 **정보(값)와 행동(함수) 모두 속성에 정의**

메서드 만들기

- **객체 속성에 할당된 함수를 메서드(method)**라고 부름

```
let user = {
    name: "John",
    age: 30
};

// sayHi 속성에 함수 할당 (=> 메소드 정의)
user.sayHi = function() {
    alert("안녕하세요!");
};

// 메소드 호출
user.sayHi();
```

- 다음과 같이 **기존에 정의된 함수를 객체의 메소드로 등록**하는 것도 가능
  - 다른 언어와 같이 **함수와 메소드가 서로 분리된 개념이 아님**

```
let user = {
    // ...
};

// 함수 선언
function sayHi() {
    alert("안녕하세요!");
};

// 선언된 함수를 메서드로 등록
user.sayHi = sayHi;
```

- 다음과 같이 객체 내부에 속성을 정의하는 동시에 곧바로 메소드 정의 가능

```
let user = {
    // 속성 정의하며 곧바로 메소드 정의
    sayHi: function() {
        alert("Hello");
    }
};
```

- 위와 똑같은 역할을 하는 **단축 구문을 사용 가능** (콜론, function 키워드 안 쓰고 메서드 정의)

```
let user = {
    // 단축 구문 이용하여 메소드 정의
    sayHi() {
        alert("Hello");
    }
};
```

메서드와 this

- (100%는 아니지만) 보통 메서드에는 **현재 객체의 속성값에 접근하거나 변경하는 코드**가 포함됨
    - 따라서 메서드 내부에서 객체를 참조할 방법이 필요함
    - **this 키워드를 사용하면 객체에 접근** 가능

```
let user = {
    name: "John",
    age: 30,
    sayHi() {
        // 이 예제에서 this는 "현재 객체"를 나타냄
        alert(this.name);
    }
};

// John 출력
user.sayHi();
```

자유로운 this (*)

- 다른 객체지향 언어(ex: Java, C++)와는 다르게 **this 값은 메소드가 호출되는 맥락에 따라 바뀔 수 있음**
    - 즉, **this 값은 코드를 실행하며 함수가 호출되는 시점에 유연하게 결정**될 수 있음
    - 기본적으로는 **메소드 호출 시점의 점(.) 앞 객체를 참조**함

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
    alert( this.name );
}

user.f = sayHi;
admin.f = sayHi;
```

```
// this 값은 기본적으로 "메소드 호출 시점에 점(.) 앞 객체"를 참조
user.f(); // "John" 출력 (메소드 호출 시점에 점 앞의 객체는 user 따라서, this는 user)
admin.f(); // "Admin" 출력 (메소드 호출 시점에 점 앞의 객체는 admin 따라서, this는 admin)

// 대괄호를 통해서도 메소드 접근 가능
admin['f']();
```

- 만약 **특별한 맥락없이(=점 앞에 아무 객체도 없는 상태) 메소드를 호출할 경우, 엄격 모드 적용 여부에 따라 다음과 같이 다른 결과가 발생**함

엄격 모드인 경우

```
"use strict" // 엄격 모드를 적용하기 위해서 소스 코드 첫 줄에 "use strict" 문자열 사용

function sayHi() {
    alert(this);
}

// undefined (메소드를 호출하는 주체가 없다고 해석)
sayHi();
```

엄격 모드가 아닌 경우

```
function sayHi() {
    console.log(this);
}

// 어쨌든 this 값 바인딩이 필요하고 따로 주체가 없으므로 글로벌 객체로 설정 (브라우저에서는 window 객체)
sayHi();
```

- 자바스크립트에서 **this는 런타임에 결정**됨
    - 메서드가 **어디에 정의되었는지에 상관없이 this는 점 앞의 객체가 무엇인가에 따라 자유롭게 결정**됨
    - 다른 언어(ex: Java)에서는 this가 메서드가 정의된 객체를 가리키게 되는데 이러한 this를 **"bound this"**라고 함 (추후에 배울 bind 메서드를 사용하면 자바스크립트에서도 this 값을 고정 가능)

```
"use strict"

let obj = {
    x: 100,
    logThis() {
        console.log(this);
    }
};
obj.logThis(); // { x: 100 }

let another = { y: 200 };
another.logThis = obj.logThis;
another.logThis(); // { y: 200 }

let logThis = another.logThis;
```

```
logThis(); // Window 객체 (엄격 모드가 적용된 경우 undefined)

let o = {
    inner: {
        z: 300
    }
};

o.inner.logThis = obj.logThis;
// 여기서 점 앞의 객체는 o가 아닌 inner임을 유의
o.inner.logThis() // { z: 300 }
```

new 연산자와 생성자 함수

- **new 연산자와 생성자 함수를 사용**하면 유사한 객체 여러 개를 쉽게 생성 가능

생성자 함수 (*)

- 생성자 함수도 그냥 함수와 별반 다르지 않음, 그러나 보통 다음의 관례를 따름 (실무에서는 반드시 따라서 지켜야 함)

```
1. 함수 이름의 "첫 글자는 대문자"로 시작
2. 함수를 호출하며 "new 연산자"를 붙여서 호출
```

생성자 함수 정의 및 활용 코드

```
// 함수의 첫 글자는 대문자
function User(name) {
    // (new 연산자와 함께 함수 호출 시) 내부적으로 다음과 같은 코드가 동작
    // this = {}

    // 새로운 속성을 this에 추가 (생성자 역할 수행)
    this.name = name;
    this.isAdmin = false;

    // (new 연산자와 함께 함수 호출 시) 내부적으로 다음과 같은 코드가 동작
    // return this
}

// new 연산자와 함께 함수 호출
let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

- `new 생성자함수(인자값)` 호출 시 다음 순서로 동작

```
1. 빈 객체를 만들어 this 값에 할당
2. 함수 본문을 실행하며 this 객체에 새로운 속성을 추가하여 객체 구성
3. this 값을 반환
```

- **생성자 함수는 재사용할 수 있는, 객체 생성 및 구성 코드**가 담긴 함수
- 생성자 함수에서는 **자동으로 this 값이 반환되므로 return 문이 불필요**
- 전달할 인수값이 없는 생성자 함수는 괄호를 생략해서 호출 가능

```
// 생성자 함수로 전달할 인수가 없으므로 괄호 없이 호출 가능
let user = new User;
// 아래 코드는 위 코드와 똑같이 동작
let user = new User();
```

- 생성자에서 메서드 추가 가능

```
function User(name) {
    this.name = name;

    // 생성자에서 메서드 추가
    this.sayHi = function() {
        alert( "My name is: " + this.name );
    };
}

let john = new User("John");

// My name is: John
john.sayHi();

let sam = new User("Sam");

alert( john.sayHi === sam.sayHi ); // false, 서로 다른 참조를 가진 함수 객체
```

- **class 키워드**를 이용해서 생성자 함수를 이용한 객체 생성을 대체할 수 있음 (단, class 키워드는 ES6에서 추가된 새로운 문법)

자료구조와 자료형

숫자형

- 일반적인 숫자(number 타입)는 **배정밀도 부동소수점 숫자(double precision floating point number)로 알려진 64비트 형식의 IEEE-754**에 저장
  - 자바스크립트는 **정수, 실수 타입의 구분이 없음** (모두 number 타입)
- 값이 매우 큰 정수는 bigInt 타입의 숫자로 저장

과학적 표기법을 이용한 큰, 작은 수 표현

```
// 10억, 1과 9개의 0
let billion = 1e9; // 1 * 10^9

// 73억 (7,300,000,000)
alert( 7.3e9 ); // 7.3 * 10^9

// 0.000001
let ms = 1e-6; // 1 * 10^-6
```

어림수 구하기

- 내림, 올림, 반올림 적용하기

Math.floor

소수점 첫째 자리에서 내림 (3.1 => 3)

Math.ceil

소수점 첫째 자리에서 올림 (3.1 => 4)

Math.round

소수점 첫째 자리에서 반올림 (3.1 => 3, 3.6 => 4)

부정확한 계산

- 숫자가 너무 커서 **64비트 저장 범위를 초과한 경우** 값을 Infinity로 처리함을 유의

```
// Infinity
alert( 1e500 );
```

- 소수점 값을 연산하는 과정에서 **정밀도 손실(loss of precision) 현상 발생 가능**

```
// false
alert( 0.1 + 0.2 == 0.3 );
// 0.30000000000000004 (미세한 오차 발생)
console.log( 0.1 + 0.2 );
```

- 정밀도 손실 현상 해결하기 위해서 toFixed 메서드 사용 가능

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(1) ); // "0.3" 출력
alert( +sum.toFixed(1) ); // 0.3 출력
```

- 두 소수점 값을 비교하는 방법
    - https://stackoverflow.com/questions/3343623/javascript-comparing-two-float-values

해결법 1) toFixed 메서드 사용

```
let c = 1.2;
let r = c * 3;
// false
console.log(r == 3.6);
// true (단, toFixed의 결과는 문자열이므로 문자열 비교가 된다는 점을 주의!)
console.log(r.toFixed(1) == (3.6).toFixed(1));
```

해결법 2) 입실론 값을 사용

```
let abs = Math.abs(r - 3.6);
console.log(abs);
console.log(abs < 0.0000001);
// 임의의 아주 작은 값(입실론 값)을 대입
const DELTA = 0.0000001;
// 두 값을 뺀 차가 앞서 정의한 입실론 값보다 적으면 값이 같다고 취급하기
if(Math.abs(r - 3.6) < DELTA) {
    console.log("Math.abs(r - 3.6) < DELTA == true");
}
```

isNaN과 isFinite

- 특수한 숫자값
    - Infinity와 -Infinity
        - 그 어떤 숫자보다 큰 혹은 작은 특수 숫자 값
    - NaN
        - 숫자 연산 중 발생한 에러를 나타내는 값
- 다음과 같이 **비교 연산자를 이용하여 NaN 값 직접 비교 불가**

```
// ==, === 연산자 모두 비교 불가!
alert( NaN == NaN ); // false
alert( NaN === NaN ); // false

// a는 NaN
let a = "asdf" / 100;
if(a === NaN) {
    alert("a === NaN");
} else {
    // "else"가 출력됨을 유의!
    alert("else");
}
```

- NaN 값을 구분하기 위해서 isNaN 함수를 사용

```
// true
isNaN(NaN);
// 전달한 인수를 먼저 숫자로 변환한 후에 NaN 여부를 확인하므로 다음 결과값도 true
isNaN("str");
```

- Infinity의 경우 비교 연산자를 이용해 값 비교가 가능

```
// true
Infinity == Infinity
// true
Infinity === Infinity
```

- NaN/Infinity/-Infinity가 아닌 **일반 숫자인지 여부를 반환**하는 isFinite 함수가 존재

```
// isFinite 내부에서 먼저 숫자 변환을 시도하므로, 그리고 변화한 결과값이 숫자 15이므로
true를 반환
alert( isFinite("15") );
// false 반환, 변환 결과가 NaN이기 때문
alert( isFinite("str") );
// false 반환, 반환 결과가 Infinity이기 때문
alert( isFinite(Infinity) );
```

- isFinite 함수는 **문자열의 내용이 일반 숫자인지 검증**하기 위해서 주로 사용

```
// prompt의 반환값은 문자열
let num = +prompt("숫자를 입력하세요.", '');

// 숫자가 아닌 값을 입력하거나 Infinity, -Infinity를 입력하면 false가 출력
alert( isFinite(num) );
```

- 빈 문자열이나 공백만 있는 문자열은 isFinite를 포함한 모든 숫자 관련 내장 함수에서 0으로 취급

```
// 빈 문자열이므로 0으로 취급하여 true 반환
isFinite("");
// 공백문자(스페이스, 개행, 탭 문자)만 포함하므로 0으로 취급하여 true 반환
isFinite(" \n\t");
```

parseInt와 parseFloat

- + 단항 연산자나 Number를 이용하여 숫자로 변환 시에는 적용되는 규칙이 엄격함
  - **숫자 혹은 숫자 표현과 관련되지 않은 다른 내용이 존재하면 변환 실패**
    - 단, 과학적 표기법을 이용한 "1e5"와 같은 문자열은 잘 변환됨 (숫자 표현과 연관된 내용)
- parseInt, parseFloat 변환 함수는 **숫자가 포함된 영역까지는 변환을 시도**한다는 차이 존재
  - parseInt => 정수 변환 함수
  - parseFloat => 실수 변환 함수

```
// "100"까지 읽고 p부터 실패하므로 문자열 "100"을 재료로 해서 값 변환 시도
alert( parseInt('100px') );
// a부터 실패하므로 123 반환
alert( parseInt("123abc456") );
// "12.5"까지 읽고 e부터 실패하므로 문자열 "12.5"를 재료로 해서 값 변환 시도
alert( parseFloat('12.5em') );
// 과학적 표기법은 문제 없이 변환 가능
alert( parseFloat('12.5e2') ); // 1250

// parseInt 함수를 사용하여 변환했으므로 정수 부분만 반환
alert( parseInt('12.3') );
// 두 번째 점은 무시되므로 12.3 반환
alert( parseFloat('12.3.4') );

// 시작하자마자 a를 만나 변환을 실패하므로 NaN 반환
alert( parseInt('a123') );
```

- parseInt의 두 번째 매개 변수(radix, 진법)는 선택적으로 사용 가능
  - 값을 전달하지 않을 경우 10진법을 이용하여 변환

```
// 16진법 표기법 숫자를 변환 (두 번째 매개 변수로 16전달)
alert( parseInt('0xff', 16) );
// 0x 없어도 잘 동작
alert( parseInt('ff', 16) );
```

기타 수학 함수

Math.random

- 0부터 1을 제외한 범위의 난수 반환

```
alert( Math.random() ); // 0 ~ 0.99999999999999999 사이(1은 제외)의 무작위 수
```

- 1부터 특정 정수를 포함한 범위의 수 구하기

```
// 1 ~ 10(포함)까지 범위 사이의 수 반환
let r = Math.floor(Math.random() * 10) + 1;
console.log(r);
```

- 특정 범위 사이의 수 구하기

```
let max = 100;
let min = 1;
// 1 ~ 100(포함)까지 범위 사이의 수 반환
let r = Math.floor(Math.random() * (max - min + 1)) + min;
```

- 함수 만들기

```
function randInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
};
```

Math.max / Math.min

- **가변 인수**를 전달받아 최대, 최소값을 반환
    - **배열을 전달받지 않는다는 점에 유의**

```
// 가변 인수를 전달받으므로 값 개수는 자유롭게 전달 가능
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1

// 배열 전달 X, NaN 출력
alert( Math.max([3, 5, -10, 0, 1]) );
```

Math.pow(n, power)

- n을 power번 거듭제곱한 결과값을 반환

```
// 2의 10제곱 = 1024
alert( Math.pow(2, 10) );
```

문자열

- 자바스크립트엔 글자 하나만 저장할 수 있는 별도의 자료형(ex: char)이 없음
    - 즉, **길이에 상관없이 모두 문자열로 저장**
- 자바스크립트에서 문자열은 페이지의 인코딩 방식과 상관없이 내부적으로 UTF-16 방식으로 저장
- 문자열은 작은따옴표(')나 큰따옴표("), 백틱(`)으로 감쌀 수 있음

```
let single = '작은따옴표';
let double = "큰따옴표";
let backticks = `백틱`;
```

- **표현식을 ${ ... }로 감싸고 이를 백틱으로 감싼 문자열 중간에 넣어주면** 해당 표현식의 결괏값을 문자열 내부에 삽입 가능 => **템플릿 리터럴(template literal)**

```
function sum(a, b) {
    return a + b;
}

// 1 + 2 = 3.
alert(`1 + 2 = ${sum(1, 2)}.`);
```

- 백틱을 사용하면 문자열을 **여러 줄에 걸쳐 작성**할 수 있음
    - 따로 개행문자(\n)를 삽입하지 않아도 알아서 엔터를 개행 문자로 인식

```
let guestList = `손님:
 * John
 * Pete
 * Mary
`;

alert(guestList);
```

-----(여기까지 시험범위)-----

특수 기호

- 모든 특수 문자(**이스케이프 문자(escape character)**라고도 불리움)는 **역슬래시(\)(backslash character)**로 시작함

특수 문자	설명
`\n`	줄 바꿈
`\r`	캐리지 리턴(carriage return). Windows에선 캐리지 리턴과 줄 바꿈 특수 문자를 조합(`\r\n`)해 줄을 바꿉니다. 캐리지 리턴을 단독으론 사용하는 경우는 없습니다.
`\'`, `\"`	따옴표
`\\`	역슬래시
`\t`	탭
`\b`, `\f`, `\v`	각각 백스페이스(Backspace), 폼 피드(Form Feed), 세로 탭(Vertical Tab)을 나타냅니다. 호환성 유지를 위해 남아있는 기호로 요즘엔 사용하지 않습니다.
`\xXX`	16진수 유니코드 `XX`로 표현한 유니코드 글자입니다(예시: 알파벳 `'z'`는 `'\x7A'`와 동일함).
`\uXXXX`	UTF-16 인코딩 규칙을 사용하는 16진수 코드 `XXXX`로 표현한 유니코드 기호입니다. `XXXX`는 반드시 네 개의 16진수로 구성되어야 합니다(예시: `\u00A9`는 저작권 기호 `©`의 유니코드임).
`\u{X…XXXXXX}` (한 개에서 여섯 개 사이의 16진수 글자)	UTF-32로 표현한 유니코드 기호입니다. 몇몇 특수한 글자는 두 개의 유니코드 기호를 사용해 인코딩되므로 4바이트를 차지합니다. 이 방법을 사용하면 긴 코드를 삽입할 수 있습니다.

- **length 속성**을 통해 문자열 길이 읽어오기 (속성이고, 메서드가 아님을 유의!)

```
// 3 출력
// \n은 '특수 문자' 한 개로 취급
alert( `My\n`.length );
```

- 문자열 내 특정 위치에 있는 글자에 접근하려면 대괄호를 이용하거나, charAt 메서드를 호출 (위치는 0부터 시작)

```
let str = "Hello";

// 첫 번째 글자 H 출력
alert( str[0] );
// charAt 메소드 사용
alert( str.charAt(0) );

// 마지막 글자 o 출력
alert( str[str.length - 1] );
```

- 대괄호와 charAt 차이 => 해당 위치에 글자를 찾을 수 없는 경우 반환값
    - 보통 charAt 보다는 대괄호를 많이 사용함

```
let str = "Hello";

// 대괄호 => 위치에 글자가 없을 경우 undefined 반환
alert( str[1000] );
// charAt => 위치에 글자가 없을 경우 빈 문자열('') 반환
alert( str.charAt(1000) );
```

- **for .. of 반복문**을 사용하여 문자열 순회 가능
    - for .. in 반복문이 객체의 key 값을 순회하기 위해서 사용된다면, **for .. of 반복문은 배열을 순회하기 위해서 사용**됨 (정확히 말하면 iterable 객체 (ex: 배열, 문자열 등))

```
for(let char of "Hello") {
    alert(char); // H,e,l,l,o (char는 순차적으로 H, e, l, l, o가 됩니다.)
}

let lst = [1, 2, 3];
// 1, 2, 3 출력
for(let item of lst) {
    console.log(item);
}
```

문자열의 불변성

- 문자열은 불변값이므로 **문자열의 값을 직접 수정할 수 없음**

```
let str = 'Hi';

// Error: Cannot assign to read only property '0' of string 'Hi'
str[0] = 'A';

// 변경되지 않은 채로, "Hi" 출력
alert( str[0] );
```

- 문자열 내용을 수정하기 위해서는 **완전히 새로운 문자열을 하나 만든 다음, 이 문자열을 할당**해야 함

```
let str = 'Hi';

// 문자열 전체를 교체함
str = 'h' + str[1];
// 혹은 바로 값을 대입
// str = 'hi';

// "hi" 출력
alert( str );
```

부분 문자열 찾기

str.indexOf(substr, pos) 메서드

문자열 str의 pos에서부터 시작해, 부분 문자열 substr이 어디에 위치하는지를 찾아주는 메소드

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, str은 'Widget'으로 시작함
alert( str.indexOf('widget') ); // -1, indexOf는 대·소문자를 따지므로 원하는 문자열을 찾지 못함

alert( str.indexOf("id") ); // 1, "id"는 첫 번째 위치에서 발견됨 (Widget에서 id)
```

- str.indexOf(substr, pos)의 두 번째 매개변수 pos는 선택적으로 사용할 수 있는데, 이를 명시하면 검색이 해당 위치부터 시작

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

모든 "as"의 위치 찾기 코드

```
let str = 'As sly as a fox, as strong as an ox';
let target = 'as';
let pos = 0;

while (true) {
    let foundPos = str.indexOf(target, pos);
    if (foundPos == -1) break;

    alert( `위치: ${foundPos}` );
    // 다음 위치를 기준으로 검색을 이어갑니다.
    pos = foundPos + 1;
}
```

짧게 줄인 코드

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( `위치: ${pos}` );
}
```

부분 문자열 여부를 검사하려면 아래와 같이 -1과 비교

```
let str = "Widget with id";

// 0번 위치에 있는 경우 0이 false로 평가되므로 코드 실행이 안됨!
// if (str.indexOf("Widget")) {
// -1과 비교 필요
if (str.indexOf("Widget") != -1) {
    alert("찾았다!"); // 의도한 대로 동작합니다.
}
```

includes, startsWith, endsWith

includes : 부분 문자열의 위치 정보는 필요하지 않고 포함 여부만 알고 싶을 때 적합한 메서드

```
alert( "Widget with id".includes("Widget") ); // true
alert( "Hello".includes("Bye") ); // false
```

startsWith, endsWith : 메서드 이름 그대로 문자열 str이 특정 문자열로 시작하는지(start with) 여부와 특정 문자열로 끝나는지(end with) 여부를 확인

```
alert( "Widget".startsWith("Wid") ); // true, "Widget"은 "Wid"로 시작합니다.
alert( "Widget".endsWith("get") ); // true, "Widget"은 "get"으로 끝납니다.
```

부분 문자열 추출하기

str.slice(start [, end])

```
let str = "stringify";
alert( str.slice(0, 5) ); // 'strin', 0번째부터 5번째 위치까지(5번째 위치의 글자는 포함하지 않음)
alert( str.slice(0, 1) ); // 's', 0번째부터 1번째 위치까지(1번째 위치의 자는 포함하지 않음)
```

- 두 번째 인수가 생략된 경우엔, 명시한 위치부터 문자열 끝까지를 반환

```
let str = "stringify";
// ringify, 2번째부터 끝까지
alert( str.slice(2) );
```

- start와 end는 음수가 될 수도 있습니다. 음수를 넘기면 문자열 끝에서부터 카운팅을 시작

```
let str = "stringify";
// 끝에서 4번째부터 시작해 끝에서 1번째 위치까지
alert( str.slice(-4, -1) ); // gif
```

str.substring(start [, end])

substring은 slice와 아주 유사하지만 start가 end보다 커도 괜찮다는 데 차이

```
let str = "stringify";

// 동일한 부분 문자열을 반환합니다.
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// slice를 사용하면 결과가 다릅니다.
alert( str.slice(2, 6) ); // "ring" (같음)
alert( str.slice(6, 2) ); // "" (빈 문자열)
```

- substring은 음수 인수를 허용하지 않습니다. 음수는 0으로 처리

str.substr(start [, length])

start에서부터 시작해 length 개의 글자를 반환 (끝 위치 대신에 **길이를 기준으로 문자열을 추출**한다는 점에서 substring과 slice와 차이)

```
let str = "stringify";
alert( str.substr(2, 4) ); // ring, 두 번째부터 글자 네 개
```

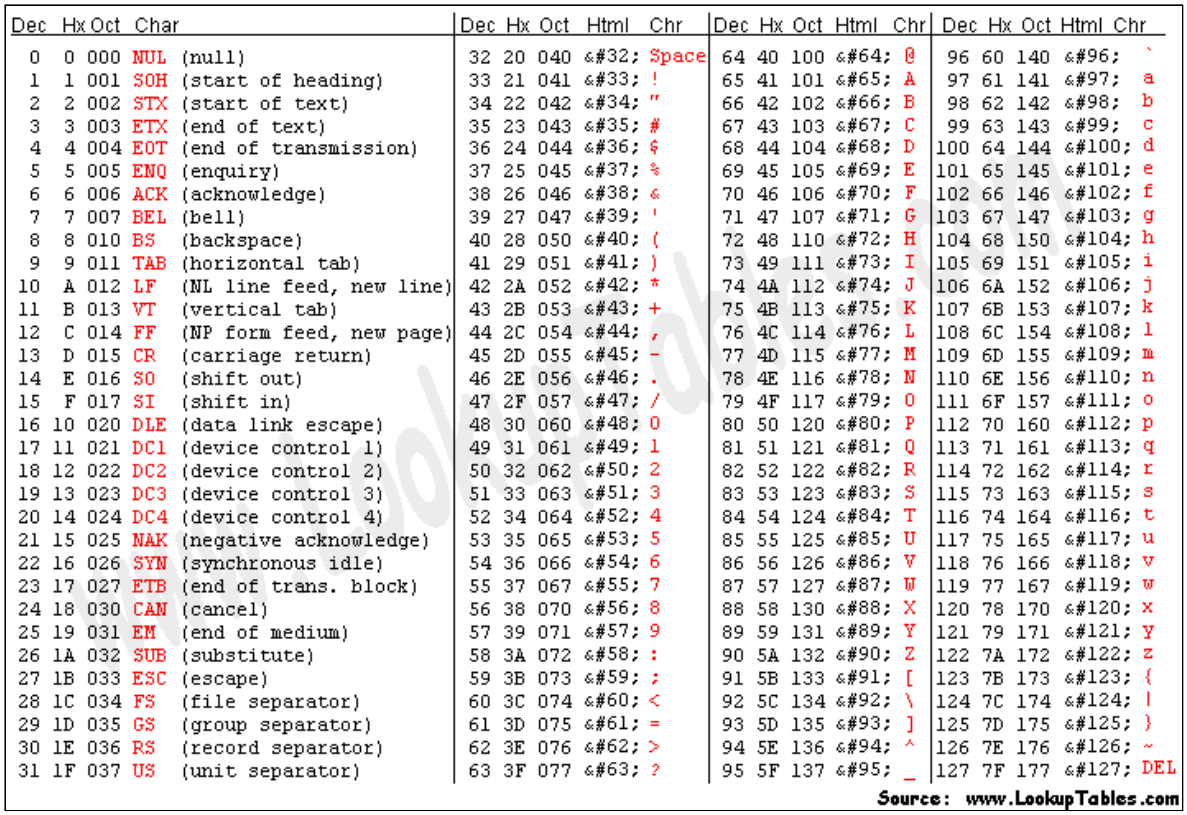
- 첫 번째 인수가 음수면 뒤에서부터 개수

```
let str = "stringify";
alert( str.substr(-4, 2) ); // gi, 끝에서 네 번째 위치부터 글자 두 개
```

메서드	추출할 부분 문자열	음수 허용 여부(인수)
`slice(start, end)`	`start`부터 `end`까지(`end`는 미포함)	음수 허용
`substring(start, end)`	`start`와 `end` 사이	음수는 `0`으로 취급함
`substr(start, length)`	`start`부터 `length`개의 글자	음수 허용

문자열 비교하기

소문자는 대문자보다 항상 큽니다.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

```
alert( 'a' > 'Z' ); // true
```

https://unicode-table.com/en/

str.codePointAt(pos)

특정 글자의 코드포인트(유니코드의 위치) 반환

```
// 글자는 같지만 케이스는 다르므로 반환되는 코드가 다릅니다.
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
alert( "Ö".codePointAt(0) ); // 214
alert( "가".codePointAt(0) ); // 44032
```

String.fromCodePoint(code)

코드포인트로부터 글자 반환

```
let c = String.fromCodePoint(90)
alert( c ); // Z
c = String.fromCodePoint(44032)
alert( c ); // 가
```