JavaScript 치트시트

1. 변수, 타입, 연산자

- 변수 선언:
  - `var`: 함수 스코프. 호이스팅(hoisting) 발생. 재선언 가능. (사용 지양)
  - `let`: 블록({}) 스코프. 재할당 가능.
  - `const`: 블록 스코프. 재할당 불가능. (객체나 배열의 내용은 변경 가능)
- 데이터 타입:
  - 원시: `String`, `Number`, `Boolean`, `null`, `undefined`, `Symbol`, `BigInt`
  - 객체: `Object` (`Array`, `Function`, `Date`, `RegExp` 등 포함)
- 연산자:
  - 비교: `==` (동등, 타입 변환 O), `===` (일치, 타입 변환 X) - `===` 사용 권장.
  - 논리: `&&`, `||`, `!`
  - Nullish Coalescing: `a ?? b` (a가 `null` 또는 `undefined`일 때 b 반환, 그 외에는 a)
  - Optional Chaining: `obj?.prop?.method()` (중간에 `null`이나 `undefined`가 있으면 에러 대신 `undefined` 반환)

2. 제어 흐름

- `if...else`, `switch`, `for`, `while`, `do...while`
- `for...in`: 객체의 열거 가능한 속성(key)을 순회. (상속된 속성 포함 가능)
- `for...of`: 이터러블 객체(`Array`, `String`, `Map`, `Set` 등)의 값(value)을 순회.

3. 함수

- 함수 선언식: `function myFunction() {}` (호이스팅 O)
- 함수 표현식: `const myFunction = function() {};` (호이스팅 X)
- 화살표 함수 (Arrow Function):
  - `const add = (a, b) => a + b;`
  - `this`를 바인딩하지 않음 (상위 스코프의 `this`를 그대로 사용).
  - `arguments` 객체가 없음.
- Default Parameters: `function greet(name = "Guest") {}`
- Rest Parameters: `function sum(...numbers) {}` (나머지 인수들을 배열로 받음)
- Spread Syntax: `...` (배열이나 객체를 펼침)
  - `const arr2 = [...arr1];` (배열 복사)
  - `const obj2 = {...obj1};` (객체 복사)

4. 객체와 프로토타입

- 객체 리터럴: `const person = { name: "John", age: 30 };`
- Computed Property Names: `const prop = "name"; const obj = { [prop]: "John" };`
- 프로토타입 (Prototype): 모든 객체는 프로토타입이라는 다른 객체를 참조하며, 프로토타입의 속성과 메서드를 상속받음.
- 생성자 함수 (Constructor Function):

```
function Person(name) { this.name = name; }
Person.prototype.greet = function() { console.log("Hi, " + this.name); };
const person1 = new Person("Jules");
```

- 클래스 (ES6+): 프로토타입 상속을 더 명확하게 표현하는 문법적 설탕(Syntactic Sugar).

```
class Person {
  constructor(name) { this.name = name; }
  greet() { console.log("Hi, " + this.name); }
}
class Developer extends Person { /* ... */ }
```

5. 배열 고급 메서드

- 변형 메서드 (Mutator methods): `push`, `pop`, `shift`, `unshift`, `splice`, `sort`, `reverse`
- 접근자 메서드 (Accessor methods): `slice`, `concat`, `join`, `indexOf`
- 반복 메서드 (Iteration methods):
  - `forEach(callback)`: 각 요소에 대해 콜백 실행.
  - `map(callback)`: 각 요소에 콜백을 적용한 **새 배열** 반환.
  - `filter(callback)`: 콜백이 `true`를 반환하는 요소로 **새 배열** 반환.
  - `reduce(callback, initialValue)`: 누적 계산 값 반환.
  - `find(callback)`: 콜백이 `true`를 반환하는 첫 번째 **요소** 반환.
  - `findIndex(callback)`: 콜백이 `true`를 반환하는 첫 번째 **인덱스** 반환.
  - `some(callback)`: 조건을 만족하는 요소가 하나라도 있으면 `true`.
  - `every(callback)`: 모든 요소가 조건을 만족하면 `true`.

6. 비동기 처리

- 콜백 (Callback): 비동기 작업이 끝난 후 실행될 함수. (콜백 헬 문제 발생 가능)
- Promise: 비동기 작업의 최종 완료 또는 실패를 나타내는 객체.
  - 상태: `pending`, `fulfilled`, `rejected`
  - 메서드: `.then()`, `.catch()`, `.finally()`
  - `Promise.all(promises)`: 모든 프로미스가 성공하면 성공.
  - `Promise.race(promises)`: 가장 먼저 완료되는 프로미스의 결과를 따름.
- async/await (ES2017): Promise를 동기 코드처럼 보이게 하는 문법.

```
async function fetchData() {
  try {
    const response = await fetch('api/data');
    if (!response.ok) throw new Error('Network response was not ok.');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Fetch error:', error);
  }
}
```

7. 모듈 시스템 (ES6 Modules)

- 내보내기 (Export):
  - `export const name = "Jules";` (이름 있는 내보내기)
  - `export default function() { ... }` (기본 내보내기, 파일당 하나만 가능)
- 가져오기 (Import):
  - `import { name } from './module.js';`
  - `import myFunc from './module.js';`
  - `import * as myModule from './module.js';`
- HTML에서 사용: `<script type="module" src="./main.js"></script>`

8. 구조 분해 할당 (Destructuring Assignment)

- 배열: `const [first, second] = [1, 2];`
- 객체: `const { name, age } = { name: "Jules", age: 30 };`
- 이름 변경: `const { name: personName } = person;`
- 기본값: `const { city = "Unknown" } = person;`

9. this 키워드

`this`가 가리키는 값은 함수가 호출되는 방식에 따라 결정됨.

- 전역 컨텍스트: `window` (브라우저) 또는 `global` (Node.js).
- 일반 함수 호출: `strict mode`에서는 `undefined`, 아니면 전역 객체.
- 메서드 호출: `obj.method()` -> `this`는 `obj`.
- 생성자 함수: `new Person()` -> `this`는 새로 생성된 인스턴스.
- 화살표 함수: `this`를 갖지 않음. 상위 스코프의 `this`를 참조.
- 명시적 바인딩: `.call()`, `.apply()`, `.bind()` 메서드로 `this`를 직접 설정.

10. ES6+ 모던 JavaScript 기능

모듈 시스템 심화

```
// Named exports
export const apiUrl = 'https://api.example.com';
export function fetchData() { /* ... */ }
export class ApiClient { /* ... */ }

// Default export
export default class App { /* ... */ }

// Re-exports
export { fetchData as getData } from './api.js';
export * from './utils.js';

// Dynamic imports
const module = await import('./module.js');
```

```
const { default: App } = await
import('./App.js');
```

클래스 심화

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    // 정적 메서드
    static create(name) {
        return new Animal(name);
    }

    // 게터와 세터
    get species() {
        return this._species ||
'Unknown';
    }

    set species(value) {
        this._species = value;
    }

    // 프라이빗 필드 (ES2022)
    #secret = 'hidden';

    // 프라이빗 메서드
    #privateMethod() {
        return this.#secret;
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }

    // 메서드 오버라이딩
    speak() {
        return `${this.name} barks!`;
    }
}
```

Symbol과 이터레이터

```
// Symbol 생성
const sym1 = Symbol('description');
const sym2 = Symbol.for('global');

// 이터레이터 프로토콜
const iterable = {
    [Symbol.iterator]() {
        let count = 0;
        return {
            next() {
                if (count < 3) {
                    return { value:
count++, done: false };
                }
                return { done: true };
            }
        };
    }
};

// 제너레이터 함수
function* numberGenerator() {
    yield 1;
    yield 2;
    yield 3;
}

const gen = numberGenerator();
console.log(gen.next().value); // 1
```

Proxy와 Reflect

```
// Proxy로 객체 동작 가로채기
const target = { name: 'John', age:
30 };
const proxy = new Proxy(target, {
    get(obj, prop) {
        console.log(`Getting ${prop}`);
        return obj[prop];
    },
    set(obj, prop, value) {
        console.log(`Setting ${prop} to
${value}`);
        obj[prop] = value;
        return true;
    }
});

// Reflect로 메타 프로그래밍
const obj = { x: 1, y: 2 };
Reflect.set(obj, 'z', 3);
console.log(Reflect.get(obj, 'z')); // 3
```

WeakMap과 WeakSet

```
// WeakMap - 키가 약한 참조
const wm = new WeakMap();
const obj = {};
wm.set(obj, 'private data');

// WeakSet - 약한 참조로 객체 저장
const ws = new WeakSet();
ws.add(obj);
console.log(ws.has(obj)); // true
```

템플릿 리터럴 고급 사용법

```
// 태그된 템플릿 리터럴
function highlight(strings, ...values) {
    return strings.reduce((result,
string, i) => {
        const value = values[i] ?
`<mark>${values[i]}</mark>` : '';
        return result + string + value;
    }, '');
}

const name = 'John';
const age = 30;
const html = highlight`Hello ${name},
you are ${age} years old.`;

// SQL 쿼리 안전성 검사
function sql(strings, ...values) {
    // SQL 인젝션 방지를 위한 값 검증
    const safeValues = values.map(val =>
{
        if (typeof val === 'string') {
            return `'${val.replace(/'/g,
"''")}'`;
        }
        return val;
    });
    return strings.reduce((result,
string, i) =>
        result + string + (safeValues[i]
|| ''), '');
}
```

함수형 프로그래밍 패턴

```
// 커링 (Currying)
const add = (a) => (b) => a + b;
const add5 = add(5);
console.log(add5(3)); // 8

// 파이프라인
const pipe = (...fns) => (value) =>
fns.reduce((acc, fn) => fn(acc), value);

const addOne = x => x + 1;
const multiplyByTwo = x => x * 2;
const square = x => x * x;

const pipeline = pipe(addOne,
multiplyByTwo, square);
console.log(pipeline(3)); // 64

// 메모이제이션
const memoize = (fn) => {
    const cache = new Map();
    return (...args) => {
        const key =
JSON.stringify(args);
        if (cache.has(key)) {
            return cache.get(key);
        }
        const result = fn(...args);
        cache.set(key, result);
        return result;
    };
};

const expensiveFunction = memoize((n) =>
{
    console.log('Computing...');
    return n * n;
});
```

비동기 프로그래밍 고급 패턴

```
// Promise.allSettled - 모든 Promise 완료
대기
const promises = [
    fetch('/api/user'),
    fetch('/api/posts'),
    fetch('/api/comments')
];

Promise.allSettled(promises)
    .then(results => {
        results.forEach((result, index)
=> {
            if (result.status ===
'fulfilled') {
```

```
                console.log(`Promise
${index} succeeded:`, result.value);
            } else {
                console.log(`Promise
${index} failed:`, result.reason);
            }
        });
    });

// Promise.race - 가장 빠른 Promise 대기
const timeoutPromise = new Promise((_,
reject) =>
    setTimeout(() => reject(new
Error('Timeout')), 5000)
);

Promise.race([fetch('/api/data'),
timeoutPromise])
    .then(data => console.log('Data
received:', data))
    .catch(error =>
console.log('Error:', error));

// Async Iterator
async function* asyncGenerator() {
    for (let i = 0; i < 3; i++) {
        await new Promise(resolve =>
setTimeout(resolve, 1000));
        yield i;
    }
}

(async () => {
    for await (const value of
asyncGenerator()) {
        console.log(value);
    }
})();
```

모던 배열 메서드

```
// Array.flat() - 중첩 배열 평탄화
const nested = [1, [2, 3], [4, [5, 6]]];
console.log(nested.flat(2)); // [1, 2,
3, 4, 5, 6]

// Array.flatMap() - map + flat 조합
const words = ['hello world',
'javascript is awesome'];
const letters = words.flatMap(word =>
word.split(' '));
console.log(letters); // ['hello',
'world', 'javascript', 'is', 'awesome']

// Array.from() - 유사 배열을 배열로 변환
const nodeList =
document.querySelectorAll('div');
const divArray = Array.from(nodeList,
node => node.textContent);

// Array.of() - 배열 생성
const numbers = Array.of(1, 2, 3, 4, 5);

// Array.includes() - 포함 여부 확인
const fruits = ['apple', 'banana',
'orange'];
console.log(fruits.includes('banana')); //
true
```

객체 고급 기능

```
// Object.entries()와
Object.fromEntries()
const obj = { a: 1, b: 2, c: 3 };
const entries = Object.entries(obj); //
[['a', 1], ['b', 2], ['c', 3]]
const newObj =
Object.fromEntries(entries); // { a: 1,
b: 2, c: 3 }

// Object.assign()과 스프레드 연산자
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };
const merged =
{ ...target, ...source }; // { a: 1, b:
3, c: 4 }

// Object.freeze() - 객체 불변화
const frozen = Object.freeze({ x: 1, y:
2 });
// frozen.x = 3; // 에러 발생 (strict
mode에서)

// Object.seal() - 객체 밀봉
const sealed = Object.seal({ x: 1, y:
2 });
sealed.x = 3; // 가능
// sealed.z = 4; // 에러 발생
```

에러 처리 고급 패턴

```
// 커스텀 에러 클래스
class ValidationError extends Error {
    constructor(message, field) {
        super(message);
        this.name = 'ValidationError';
        this.field = field;
    }
}

// 에러 경계 (Error Boundary)
function withErrorBoundary(Component) {
    return class extends React.Component
{
        constructor(props) {
            super(props);
            this.state = { hasError:
false };
        }

        static
getDerivedStateFromError(error) {
            return { hasError: true };
        }

        componentDidCatch(error,
errorInfo) {
            console.error('Error
caught:', error, errorInfo);
        }

        render() {
            if (this.state.hasError) {
                return <h1>Something
went wrong.</h1>;
            }
            return <Component
{...this.props} />;
        }
    };
}
```

성능 최적화 기법

```
// 디바운싱 (Debouncing)
function debounce(func, wait) {
    let timeout;
    return function
executedFunction(...args) {
        const later = () => {
            clearTimeout(timeout);
            func(...args);
        };
        clearTimeout(timeout);
        timeout = setTimeout(later,
wait);
    };
}

// 스로틀링 (Throttling)
function throttle(func, limit) {
    let inThrottle;
    return function(...args) {
        if (!inThrottle) {
            func.apply(this, args);
            inThrottle = true;
            setTimeout(() => inThrottle
= false, limit);
        }
    };
}

// 가상 스크롤링
class VirtualScroll {
    constructor(container, items,
itemHeight) {
        this.container = container;
        this.items = items;
        this.itemHeight = itemHeight;
        this.visibleCount =
Math.ceil(container.clientHeight /
itemHeight);
        this.startIndex = 0;
        this.endIndex =
this.visibleCount;
    }

    render() {
        const visibleItems =
this.items.slice(this.startIndex,
this.endIndex);
        // 렌더링 로직
    }
}
```