LIKE LION

8강

기타

WHATEVER YOU WANT, MAKE IT REAL.

강사 정길용

LIKE LION

{

에러 핸들링

ESM (ECMAScript Modules)

defer, async

use strict

동기 방식과 비동기 방식

}

LIKE LION

ex08-01.ts 

Error 클래스

- 에러 정보를 표현하는 기본 클래스
- `new Error(message?: string, options?: ErrorOptions) => Error`
  - `message`: 에러 메세지
  - `options`: 원본 에러 객체. 중첩 에러를 만들때 cause 속성에 원본 에러를 지정하면 에러를 추적하기 용이함. ES2022에 추가됨

주요 속성

- name: 에러 이름. 생성한 에러명(Error, TypeError, ReferenceError, SyntaxError 등)
- message: 에러 설명. 생성자 함수에 전달한 `message` 문자열
- stack: 콜 스택 정보. 에러가 발생한 시점의 콜스택 정보가 들어있는 문자열
- cause: 중첩 에러를 만들때 지정한 원본 에러

```
function f1() {
  const err = new Error('에러 발생');
  console.log(err.name);
  console.log(err.message);
  console.log(err.stack);
}
function f2() {
  f1();
}
f2();
```

```
Error
에러 발생
Error: 에러 발생
    at f1 (C:₩febc13₩01.js₩JS₩workspace₩ch08₩ex08-01.ts:2:15)
    at f2 (C:₩febc13₩01.js₩JS₩workspace ₩ch08₩ex08-01.ts:8:3)
    at Object.<anonymous> (C:₩febc13₩01.js₩JS₩workspace ₩ch08₩ex08-
01.ts:10:1)
    ……
```

LIKE LION

ex08-02.ts

Error 클래스를 상속 받아서 세분화된 에러를 표현

TypeError

- 잘못된 타입 사용
- 예시: `null.fn(), undefined.fn()`
  - Uncaught TypeError: Cannot read properties of null (reading 'fn')
  - Uncaught TypeError: Cannot read properties of undefined (reading 'fn')

ReferenceError

- 선언되지 않은 변수 접근
- 예시: `console.log(x)`
  - Uncaught ReferenceError: x is not defined

SyntaxError

- 문법 오류
- 예시: `new Function('x', 'y', 'retrun x + y');`

사용자 정의 에러

- Error 클래스를 상속해서 구현

LIKE LION

ex08-03.ts

▶ try...catch 문

- 예외(에러)가 발생할 수 있는 코드를 안전하게 실행하기 위해서 사용
- try...catch문 없이 실행되는 코드에서 에러가 발생하면 JS 엔진은 콘솔에 에러 메세지와 콜스택 정보를 출력하며 프로그램 실행을 중단
- try...catch문을 사용할 경우 try 블럭 내에서 에러가 발생하면 catch 블럭이 실행된 후 catch 블럭 이후의 코드가 정상적으로 실행
- 선택적으로 finally 블럭을 추가하면 try 블럭의 모든 코드가 정상 실행되거나 에러가 발생해서 catch 블럭이 실행되거나 상관없이 항상 try...catch 문의 마지막에 실행해야 하는 코드를 작성할 수 있음

```
function f1() {
  try {
    const fn = new Function('x', 'y', 'retrun x + y'); // SyntaxError 발생
    // 에러가 발생하면 try 블럭의 나머지 코드는 실행되지 않음
    console.log(fn(10, 20));
  } catch(err) {
    console.error('에러 발생', err.message);
  } finally {
    console.log('try...catch 문의 마지막에 호출');
  }
  console.log('f1 함수 정상 종료'); // 에러가 발생해도 실행됨
}
```

```
에러 발생 SyntaxError: Unexpected identifier 'x'
try...catch 문의 마지막에 호출
f1 함수 정상 종료
```

LIKE LION

ex08-04.ts

▶ throw 문

- 개발자가 직접 에러를 발생시키는 문법
- 에러를 직접 처리하지 않고 함수를 호출한 쪽으로 전달할때 주로 사용
- 에러가 throw 되면 프로그램 실행이 중지되고 가까운 catch 블럭으로 이동

```
function divide(x: number, y: number) {
  if(y === 0) {
    throw new Error('0으로 나눌 수 없습니다.');
  }
  return x / y;
}
function f1() {
  try {
    const result1 = divide(10, Math.round(Math.random()));
    console.log(result1);
  } catch(err) {
    console.error('에러 발생', err.message);
  }
  const result2 = divide(10, 2);
  console.log(result2);
}
```

```
에러 발생 0으로 나눌 수 없습니다.
5
```

LIKE LION

ex08-05/

▶ 모듈이란?

- 코드를 파일 단위로 분리하고, 불필요한 전역 오염 없이 재사용할 수 있게 해줌
- 파일 자체가 독립된 스코프를 가지므로 모듈내에서 선언한 변수는 전역변수가 아닌 모듈변수로써 모듈 내부에서만 접근 가능
- 모듈 구성 요소(변수, 함수, 클래스, 타입 별칭, 인터페이스 등)를 명시적으로 내보내기(export) 하면 다른 모듈에서 사용 가능
- 다른 모듈에서 export한 값을 참조하려면 import 구문을 사용
- 브라우저에서 모듈을 사용하려면 <script> 태그에 type="module" 속성을 추가

math.ts

```ts
export function plus(a: number, b: number) { return a + b; }
export function minus(a: number, b: number) { return a - b; }
export default function multiply(a: number, b: number) { return a * b; }
```

index.ts

```ts
import { plus, minus } from './math.js';
plus(2, 3);
minus(2, 3);
```

index.html

```html
<script type="module" src="index.js"></script>
```

LIKE LION

ex08-05/ 

export

- 모듈 구성 요소(변수, 함수, 클래스, 타입 별칭, 인터페이스 등)를 외부로 내보내 다른 모듈에서 사용할 수 있게 해주는 키워드

Named Export

- 내보내기할 각 구성 요소 앞에 export 키워드 지정하거나 export { plus, minus } 형태로 선언과 분리해서 따로 작성 가능
- export는 여러번 사용 가능
- import 시 중괄호 안에 정확한 구성 요소명을 사용
    - `export function plus(a: number, b: number) { return a + b; }`
    - `export function minus(a: number, b: number) { return a - b; }`
    - `import { plus, minus } from './math.js';`

Default Export

- 내보내기할 구성 요소 앞에 export default 키워드 지정
- export default는 모듈 내에서 한번만 사용 가능
- import 시 이름은 자유롭게 지정 가능
    - `export default function multiply(a: number, b: number) { return a * b; }`
    - `import MyMath from './math.js';`

LIKE LION

ex08-05/ 

import
- 다른 모듈에서 export한 구성 요소를 가져올 때 사용하는 키워드

Named Import
- export로 내보낸 구성 요소를 중괄호로 감싸서 가져옴
- 이름이 정확히 일치 해야 하며 필요한 것만 선택해서 import 가능
- 별칭 사용 가능
  - `import { plus as add, minus } from './math.js';`

Default Import
- export default 로 내보낸 구성 요소는 중괄호 없이 자유롭게 이름을 지정해서 import 가능
  - `import MyMath from './math.js';`

Mixed Import
- Named Import와 Default Import를 같이 사용 (일관성과 가독성 저하로 권장하지 않음)
  - `import MyMath, { plus, minus } from './math.js';`

Type Import
- 타입 별칭이나 인터페이스를 export 했을 경우 import 시 type 키워드 추가 (생략 가능)
  - `import { type Member } from './math.js';`

ESM (ECMAScript Modules)

- ES2015에 도입된 자바스크립트의 공식 모듈 표준
- 내보내기: export 키워드 사용
- 가져오기: import 키워드 사용

CJS (CommonJS)

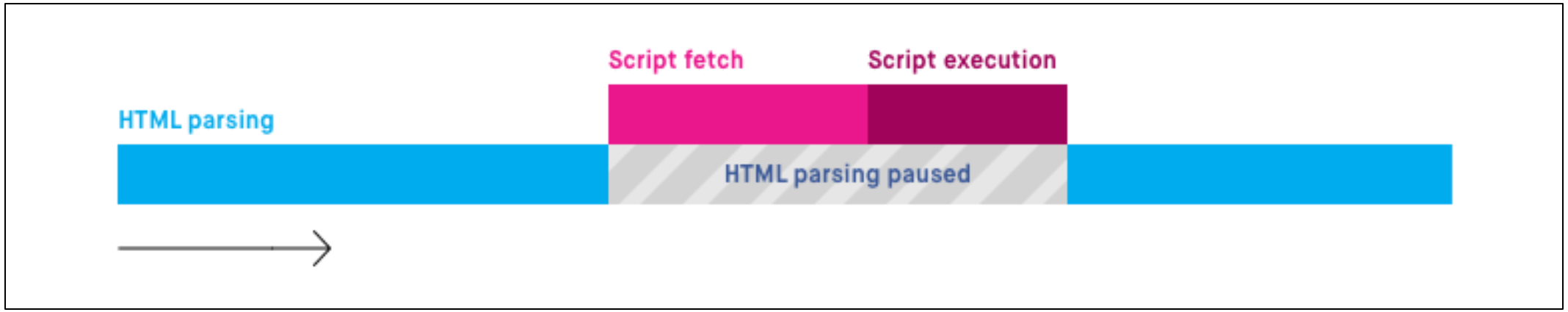
- Node.js에서 모듈화를 위해 만들어진 비표준 방식
- CommonJS: 브라우저 밖에서 실행되는 자바스크립트를 위한 표준
- 내보내기: module.exports 속성 사용
- 가져오기: require() 함수 사용
- Node.js 12 부터 package.json에 "type": "module" 속성을 추가하거나 .mjs 확장자로 만든 파일은 ESM 방식이 적용됨

Node.js에서 js, cjs, mjs 확장자

- .js: package.json에 "type": "module" 속성이 있을 경우 ESM 방식으로, 해당 속성이 없을 경우 CJS 방식으로 사용
- .cjs: "type": "module" 속성과 상관 없이 CJS 방식의 모듈 사용
- .mjs: "type": "module" 속성과 상관 없이 ESM 방식의 모듈 사용

▶ 기본 (속성 없음)

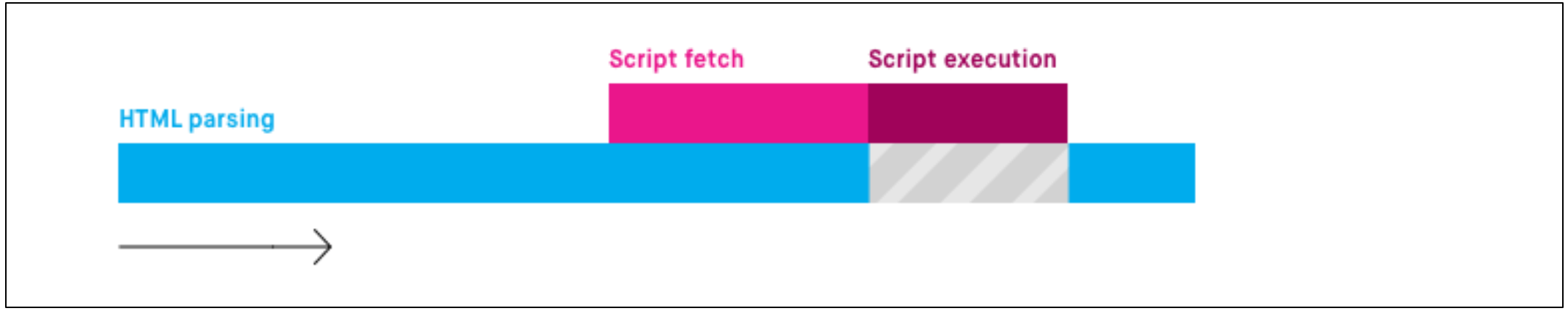
- HTML 파싱을 중지하고 스크립트 다운로드
- 다운로드가 완료되면 스크립트를 즉시 실행
- 스크립트 실행이 완료되면 HTML 파싱을 재개
- 스크립트 실행 시간동안 페이지 렌더링 지연이 발생할 수 있음



이미지 출처: https://wp-kama.com/2289/difference-between-async-and-defer-in-the-script-tag

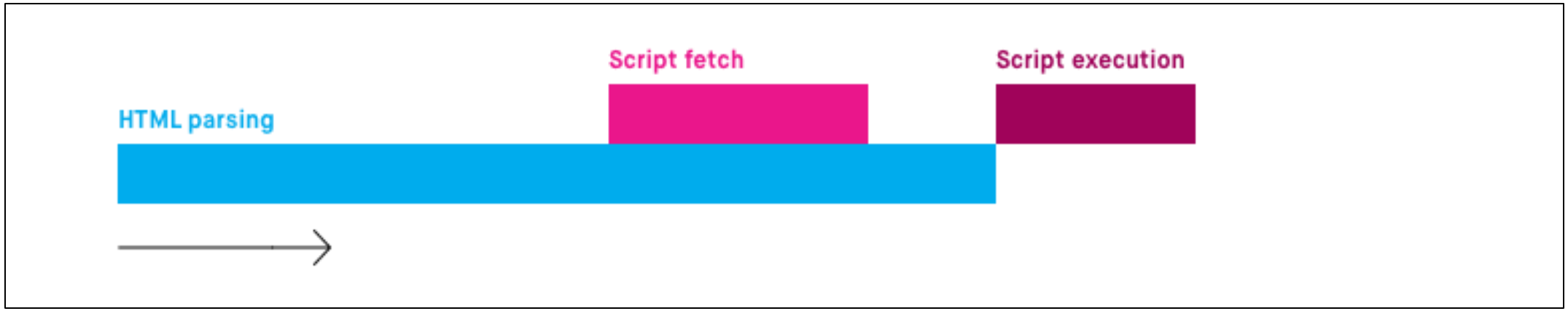
async

- HTML 파싱과 병렬로 스크립트 다운로드
- 다운로드가 완료되면 HTML 파싱을 멈추고 스크립트를 즉시 실행
- async 스크립트가 여러개 있을 경우 작성 순서와 상관없이 다운로드 완료된 스크립트 먼저 실행
  - DOMContentLoaded 이벤트 발생 전, 후 아무때나 실행될 수 있음
  - 스크립트간 실행 순서가 보장되지 않음
- 외부 스크립트에만(src 속성이 있는 경우) 적용됨
- 방문자 수 카운트, 접속 통계 기록 등 페이지 내의 스크립트와 독립적인 기능 사용시 주로 사용

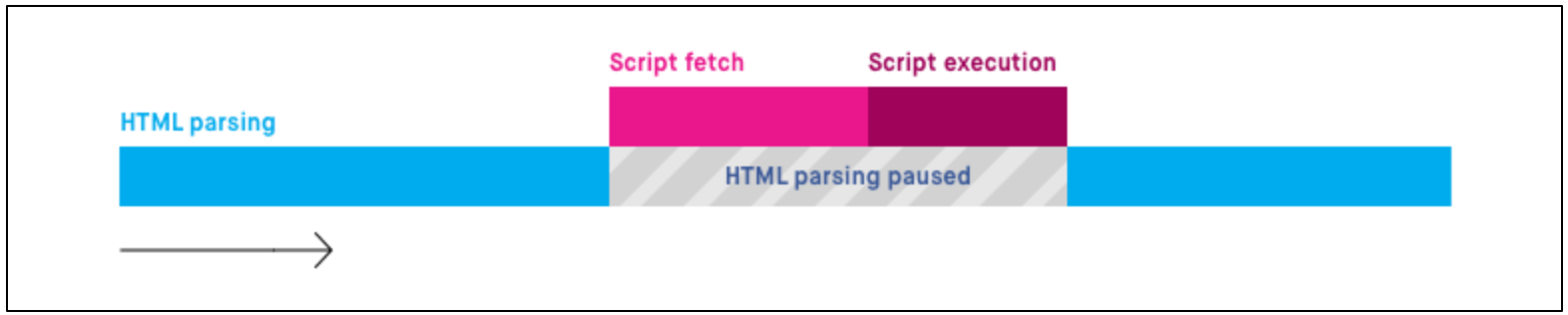


defer

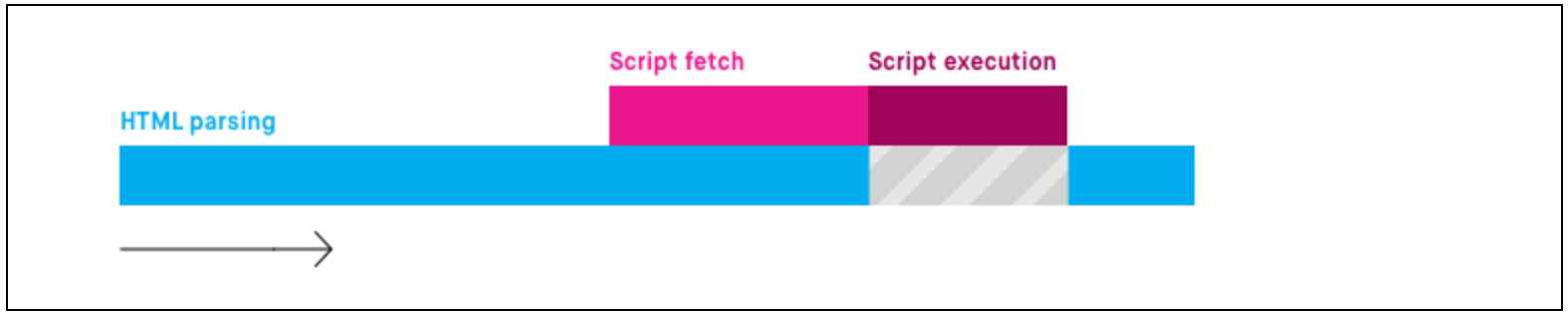
- HTML 파싱과 병렬로 스크립트 다운로드
- 다운로드가 완료되어도 HTML 파싱이 완료될 때까지 기다렸다가 실행
  - DOMContentLoaded 이벤트 발생 직전에 실행
- defer 스크립트가 여러개 있을 경우 작성 순서대로 순차 실행
- 외부 스크립트에만(src 속성이 있는 경우) 적용됨
- type="module" 속성이 있으면 기본이 defer 속성으로 동작
- 페이지 렌더링이 우선시 되는 경우 주로 사용



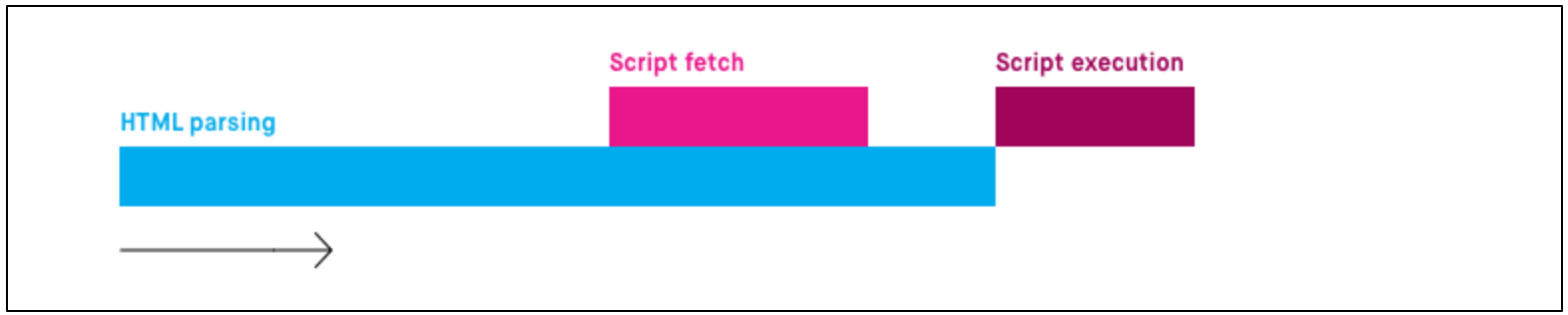
▶ 기본 (속성 없음)



▶ async



▶ defer



▶ use strict란?

- 자바스크립트의 문제점으로 지적되어온 몇가지 문법적 특징들을 새로운 버전이 나오면서 보완 했지만 하위 호환을 위해서 기본이 비활성화 되어 있음
- 이를 활성화 시키기 위해 코드나 함수의 맨 위에 "use strict" 지시자를 사용
  - 코드 실수 예방
  - 예상치 못한 버그 감소
  - 미래 JS 문법과 호환되는 코드 작성
- 클래스와 모듈의 내부에서는 자동으로 적용됨
- 타입스크립트 tsconfig.json 파일에 "strict": true 나 "alwaysStrict": true 로 설정하면 컴파일된 js 파일에 "use strict" 자동으로 추가

```
// 기본 모드
function test(a, a) {
  console.log(this); // window
  x = 10; // window.x = 10;
  console.log(x); // 10
  console.log(a + a); // 40
}
test(10, 20);
```

▶ 엄격 모드에서 적용되는 주요 문법

- 변수를 선언 없이 사용하면 오류 발생
- 함수를 호출할때 this가 window가 아닌 undefined가 됨
- 중복 매개변수 사용 금지
- 미래에 사용될 수 있는 키워드는 식별자로 사용 못함
  - implements , interface, package
  - private, protected, public 등

```
// 엄격 모드
"use strict";
function test(a, a) { // ❌ SyntaxError
  console.log(this); // undefined
  x = 10; // ❌ ReferenceError
}
test(10, 20);
```

LIKE LION

ex08-06.ts

▶ 일반 함수

- 일반 함수는 호출되면 내부의 코드가 순차적으로 실행된 뒤 값을 반환
- 함수를 호출한 코드는 해당 함수가 반환될 때까지 기다리며, 반환된 이후에 다음 코드가 실행됨
- 따라서 코드 실행 순서가 명확하며, 흐름의 일관성이 보장됨 (동기 방식)

```
function f1(){
  console.log('\t\t3. f1 호출됨.');
  console.log(`\t\t4. 작업중...`);
  console.log('\t\t5. f1 리턴됨.');
}

function test(){
  console.log('\t2. test 호출됨.');
  f1(); // f1()이 리턴될 때까지 기다림
  console.log('\t6. test 리턴됨');
}

console.log('1. 작업 시작.');
test(); // test()가 리턴될 때까지 기다림
console.log('7. 작업 종료');
```

```
1. 작업 시작.
        2. test 호출됨.
                3. f1 호출됨.
                4. 작업중...
                5. f1 리턴됨.
        6. test 리턴됨
7. 작업 종료.
```

LIKE LION

ex08-07.ts

▶ 비동기 함수

- 비동기 함수는 호출되면 작업을 바로 시작하지만, 결과가 준비될 때까지 기다리지 않고 곧바로 리턴됨
- 비동기 함수는 나중에 결과를 전달할 것을 약속하며, 호출한 코드는 그 사이에 다른 작업을 계속 진행
- 따라서 코드 실행 순서가 고정되지 않고, 효율적인 작업 분산이 가능함 (비동기 방식)

```
function f1(){
  console.log('\t\t3. f1 호출됨.');
  const delay = Math.random()*1000*10;
  console.log(`\t\t4. ${delay.toFixed()}ms 동안 작업중...`);
  setTimeout(() => {
    console.log('\t\t\t?. f1 작업 완료.', delay);
  }, delay);
  console.log('\t\t5. f1 리턴됨.');
}
function test(){
  console.log('\t2. test 호출됨.');
  f1();
  f1();
  console.log('\t6. test 리턴됨');
}
console.log('1. 작업 시작.');
test();
console.log('7. 작업 종료.');
```

```
1. 작업 시작.
        2. test 호출됨.
                3. f1 호출됨.
                4. 6857ms 동안 작업중...
                5. f1 리턴됨.
                3. f1 호출됨.
                4. 2297ms 동안 작업중...
                5. f1 리턴됨.
        6. test 리턴됨
7. 작업 종료.
                        ?. f1 작업 완료. 2296.751882600838
                        ?. f1 작업 완료. 6856.743768633691
```

▶ 콜백 방식

ex08-07-02.ts, 03, 04, 05

- 비동기 작업이 끝난 뒤에 실행할 함수를 인자로 전달해서 결과를 처리
- 작업이 완료되면 미리 등록한 콜백 함수가 호출돼 결과나 에러를 전달받아 처리하는 방식

```
function f1(resolve: (result: string) => void){
  const delay = Math.random()*1000*10;
  setTimeout(() => {
    console.log('f1 작업 완료.', delay);
    resolve('f1의 작업 결과.'); // 작업 완료 후 콜백 함수 호출
  }, delay);
}

f1((result) => {
  console.log('f1의 작업이 완료된 후 호출.', result);
});
console.log('작업 종료.');
```

```
작업 종료 .
f1 작업 완료. 6039.799201385707
f1의 작업이 완료된 후 호출. f1의 작업 결과.
```

Promise

- 비동기 작업의 성공 또는 실패 결과를 나중에 전달하기 위한 객체(ES2015에 추가)
- 어떤 함수가 Promise 객체를 반환한다면 현재 작업을 처리 중이며 작업이 처리 완료되는 미래에 어떤 값을 준비해서 전달할 것이라는 약속

Promise 생성자 함수

- Promise 객체 생성에 사용
- executor: 비동기로 처리할 작업을 가진 함수
  - resolve: 작업이 성공적으로 완료 되었을 때 호출할 함수. 인자값은 작업 결과를 전달하는데 사용
  - reject: 작업이 실패 했을 때 호출할 함수. 인자값은 실패 사유를 전달하는데 사용

```
new Promise<T>(
  executor: (
    resolve: (value: T) => void,
    reject: (reason?: any) => void
  ) => void
)
```

Promise 객체 반환

- Promise 객체를 반환하는 함수는 비동기 함수가 됨

Promise 생성자 함수의 executor 함수 작성

- executor 함수에서 처리할 작업 구현
- 작업 성공 시 resolve()를 호출하고 인자값으로 작업 결과 전달
- 작업 실패 시 reject()를 호출하고 인자값으로 실패 사유 전달

```
function f1(){
  return new Promise<string>((resolve, reject) => {
    const delay = Math.random()*1000*10;
    setTimeout(() => {
      console.log('f1 작업 완료.', delay);
      if(delay < 5000) {
        resolve('f1의 작업 결과.');
      }else{
        reject('f1 작업중 에러 발생.');
      }
    }, delay);
  });
}
```

ex08-08.ts, 08-08-02

▶ Promise 객체의 메서드

- then(onfulfilled?: value => (void | Promise),
  onrejected?: reason => (void | Promise)): Promise
  - onfulfilled: resolve()가 호출될 때 실행되는 함수
    - value: 비동기 함수에서 작업 성공 시 호출한 resolve()에 전달한 인자값
  - onrejected: reject()가 호출될 때 실행되는 함수
    - reason: 비동기 함수에서 작업 실패 시 호출한 reject()에 전달한 인자값
  - 리턴값: 새로운 Promise가 반환되어 체인 방식으로 호출이 가능
    - onfulfilled나 onrejected 가 Promise를 반환하는 함수일 경우 여러 비동기 함수를 순차적으로 호출하는데 사용

```
function test(){
  f1().then((result) => {
    console.log('f1의 작업이 완료된 후 호출.', result);
  }, (reason) => {
    console.error('f1의 작업이 실패한 후 호출.', reason);
  }).then(()=>{}).then(()=>{}).then(()=>{});
}
```

▶ Promise 객체의 메서드

ex08-08-03.ts, 08-08-04

- `catch(onrejected?: reason => (void | Promise)): Promise`
  - `onrejected`: then()의 `onrejected`와 동일
  - then()의 두번째 인자인 `onrejected`에서 처리하지 않은 에러는 catch()에서 처리됨

- `finally(onfinally?: () => void): Promise`
  - `onfinally`: Promise의 성공 실패와 상관없이 항상 호출되는 함수

```
function test(){
  f1().then((result) => {
    console.log('4. 첫번째 f1의 작업이 완료된 후 호출.', result);
  }).then(f1).then((result) => {
    console.log('5. 두번째 f1의 작업이 완료된 후 호출.', result);
  }).catch((reason) => {
    console.error('f1의 작업이 실패한 후 호출.', reason);
  }).finally(() => {
    console.log('f1의 성공, 실패와 상관없이 항상 호출.');
  });
}
```

▶ `Promise.all(values: Array<Promise>): Promise`

- 여러개의 Promise를 동시에 실행한다.
- 모두 성공하면 각 Promise의 결과들을 모은 배열을 가진 fulfilled 상태의 새로운 Promise 반환
- 하나라도 실패하면 즉시 reject되고 가장 먼저 실패한 사유를 가진 rejected 상태의 새로운 Promise 반환

▶ `Promise.any(values: Array<Promise>): Promise`

- 여러개의 Promise를 동시에 실행한다.
- 가장 먼저 성공한 Promise의 결과 반환
- 모두 실패하면 각 Promise의 실패 사유를 모은 배열을 errors 속성으로 가진 결과 반환

```
Promise.all([f1(), f1(), f1()]).then(result => {
  console.log('f1 작업이 모두 성공한 후 호출.', result);
}).catch(reason => {
  console.error('f1 작업이 하나라도 실패한 후 호출.', reason);
});
```

```
Promise.any([f1(), f1(), f1()]).then(result => {
  console.log('f1 작업중 가장 먼저 성공한 작업의 결과.', result);
}).catch(reason => {
  console.error('f1 작업이 모두 실패한 후 호출.', reason);
});
```

LIKE LION

ex08-08-05.ts

- `Promise.race(values: Array<Promise>): Promise`
  - 여러개의 Promise를 동시에 실행해서 성공, 실패 여부와 상관없이 가장 먼저 끝난(settled) Promise의 결과 반환
- `Promise.allSettled(values: Array<Promise>): Promise`
  - 여러개의 Promise를 동시에 실행해서 모든 Promise가 settled 된 후 결과 배열 반환
  - 실패해도 reject 되지 않음
  - 결과 배열에는 각 Promise의 상태(status: fulfilled | rejected)와 결과(value | reason: 결과 | 실패 이유)가 들어감

```
Promise.race([f1(), f1(), f1()]).then(result => {
  console.log('f1 작업중 가장 먼저 성공한 작업의 결과.', result);
}).catch(reason => {
  console.error('f1 작업중 가장 먼저 실패한 작업의 결과.', reason);
});
```

```
Promise.allSettled([f1(), f1(), f1()]).then(result => {
  console.log('모든 f1 작업 결과.', result);
});
```

```
모든 f1 작업 결과. [
  { status: 'fulfilled', value: 'f1의 작업 성공 결과. 212' },
  { status: 'rejected', reason: 'f1 작업 실패 사유. 581' },
  { status: 'rejected', reason: 'f1 작업 실패 사유. 711' }
]
```

LIKE LION

ex08-09.ts

▶ async/await

- Promise를 쉽게 다루기 위해 ES2017에 추가된 문법
- 콜백헬이나 then()의 복잡한 체인 방식을 사용하지 않고도 비동기 함수의 순차적인 호출이 가능해서 비동기 코드를 마치 동기 코드처럼 작성할 수 있음

▶ async 키워드

- 함수 선언부에 붙이는 키워드
- async 키워드가 붙은 함수는 자동으로 Promise 객체를 반환
- async 함수가 리턴한 값은 Promise의 resolve()에 값을 전달하는 효과
- async 함수가 throw한 값은 Promise의 reject()에 값을 전달하는 효과

```
function p1(){ // Promise
  return new Promise((resolve) => {
    resolve('p1 결과');
  });
}
async function a1(){ // async
  return 'a1 결과';
}
```

```
function p2(){ // Promise
  return new Promise((resolve, reject) => {
    reject('p2 에러');
  });
}
async function a2(){ // async
  throw 'a2 에러';
}
```

LIKE LION

ex08-09-02.ts, 08-10

▶ await 키워드

- Promise 객체 앞에 붙여서, Promise가 처리될 때까지(settled) 기다렸다가, 그 결과값을 반환해주는 키워드
- async 함수 안에서만 사용 가능
- 코드의 흐름이 동기함수를 호출하는 것과 비슷해서 가독성이 좋아짐

```
function test(){
  p1().then(result => {
    console.log('p1의 작업 결과.', result);
  }).then(a1).then(result => {
    console.log('a1의 작업 결과.', result);
  }).then(p2).then(result => {
    console.log('p2의 작업 결과.', result);
  }).then(a2).then(result => {
    console.log('a2의 작업 결과.', result);
  }).catch(error => {
    console.log('에러 발생.', error);
  });
}
```

```
async function test(){
  try {
    const result1 = await p1();
    console.log('p1의 작업 결과.', result1);
    const result2 = await a1();
    console.log('a1의 작업 결과.', result2);
    const result3 = await p2();
    console.log('p2의 작업 결과.', result3);
    const result4 = await a2();
    console.log('a2의 작업 결과.', result4);
  } catch(error) {
    console.log('에러 발생.', error);
  }
}
```