Promise

0) Promise필요성

자바스크립트는 시간이 오래 걸리는 작업(타이머, 네트워크, 파일 등)을 **비동기**로 처리. 이때 **결과가 '나중에' 오기 때문에 단순 `return` 으로 값을 받을 수 없다.** 그래서 "결과가 준비되면 무엇을 할지"를 구조적으로 표현하는 도구가 필요, 그 대표가 **Promise**

1) Promise

미래에 성공/실패로 끝날 작업을 나타내는 객체

* 성공하면 상태 `fulfilled`, 실패하면 `rejected`
* 처음 상태는 `pending`
* 결과(값/에러)를 받으려면 `.then(...)`, `.catch(...)`, `.finally(...)` 사용

2) 예제

```
<script>
// 오래 걸리는 작업 (타이머)
function getNumber(){
  setTimeout(() => {
    return 10;
   //그냥 return은 외부에 전달되지 않음
  }, 500);
}

console.log("시작");
const result = getNumber(); // undefined
console.log("받은 값:", result);
console.log("끝");
</script>
```

```
<script>
    const p = new Promise((resolve, reject) => {
        setTimeout( () => {  resolve("10"); }, 200)
    });
    console.log("시작");
    p.then(n => console.log("받은값", n));
    console.log("끝");
</script>
```

2-1. 즉시 성공하는 Promise

```
<script>
const p = new Promise((resolve, reject) => {
  resolve("성공 값"); // 바로 성공
});

p.then(v => console.log("then:", v)); // then: 성공 값
</script>
```

2-2. 타이머로 비동기 시뮬레이션

```
<script>
function delay(ms){
  return new Promise((resolve) => {
    setTimeout(() => resolve(ms + "ms 뒤 완료"), ms);
  });
}

delay(500)
```

```
  .then(msg => {
    console.log("1:", msg);
    return delay(800);
  })
  .then(msg => {
    console.log("2:", msg);
    return delay(1000);
  })
  .then(msg => console.log("3:", msg))
  .finally(() => console.log("끝"));
</script>
```

3) Promise의 상태(state)

- pending : 진행 중
- fulfilled : `resolve(value)` 호출로 성공
- rejected : `reject(error)` 호출 또는 `throw` 로 실패

```
<script>
const ok = new Promise((resolve) => resolve(123));
ok.then(v => console.log("성공:", v)); // 성공: 123

const bad = new Promise((_, reject) => reject(new Error("문제 발생")));
bad.catch(e => console.log("실패:", e.message)); // 실패: 문제 발생
</script>
```

4) then / catch / finally

```
<script>
new Promise((resolve) => {
  setTimeout(() => resolve("□ 준비 완료"), 700);
})
.then(v => {
  console.log("then 1:", v);
  return v + " + 콜라"; // 다음 then으로 전달
})
.then(v2 => {
  console.log("then 2:", v2);
  // throw new Error("빨대 없음"); // 주석 해제하면 catch로 이동
})
.then(v3 =>{
    console.log("then 3:", v3);
})
.catch(err => {
  console.log("catch:", err.message);
})
.finally(() => {
```

```
    console.log("항상 실행(finally)");
});
</script>
```

- then: 성공 시 실행, **값을 return**하면 다음 then으로 전달
- catch: 실패 시 실행(어디서 에러가 나도 여기로 모임)
- finally: 성공/실패와 무관하게 마지막에 실행(값 전달 X)

5) 콜백을 Promise로 감싸기 (프로미스화)

전통적인 콜백 API를 **Promise**로 바꾸면 체인/await 사용이 쉬워집니다.

```
<script>
// 콜백 기반 비동기 함수(가상)
function maina(cb){
  setTimeout(() => cb(null, 10), 300);
}

// 1) 콜백 -> Promise 래핑
function getNumber(){
  return new Promise((resolve, reject) => {
    maina((err, num) => {
      if(err) reject(err); else resolve(num);
    });
  });
}

// 2) 사용
getNumber()
  .then(n => n * 2)
  .then(n2 => console.log("결과:", n2)) // 20
  .catch(console.error);
</script>
```

6) async / await (가장 읽기 쉬운 문법)

`async/await`은 Promise를 **동기처럼 읽게 해주는 문법 설탕입니다.**

```
<script>
function delay(ms){
  return new Promise(res => setTimeout(res, ms));
}

function fetchValue(){
```

```
  return new Promise(res => setTimeout(() => res(7), 400));
}

async function run(){
  console.log("시작");
  await delay(500);                    // 0.5초 대기
  const v = await fetchValue();    // 값이 올 때까지 대기
  console.log("받은 값:", v);       // 7
  console.log("끝");
}
run().catch(console.error);
</script>
```

> 규칙: `await`는 *항상 Promise를 기다린다 `async` 함수 내부에서만 사용 가능.

---

7) 에러 처리 패턴

7-1. then/catch 체인

```
<script>
function mightFail(){
  return new Promise((resolve, reject) => {
    setTimeout(() => Math.random() < 0.5 ? resolve("OK") : reject(new Error("FAIL")), 300);
  });
}

mightFail()
  .then(v => console.log("성공:", v))
  .catch(e => console.log("실패:", e.message))
  .finally(() => console.log("정리"));
</script>
```

7-2. async/await + try/catch

```
<script>
function mightFail(){
  return new Promise((resolve, reject) => {
    setTimeout(() => Math.random() < 0.5 ? resolve("OK") : reject(new Error("FAIL")), 300);
  });
}

async function run(){
  try {
    const v = await mightFail();
    console.log("성공:", v);
  } catch (e) {
    console.log("실패:", e.message);
  } finally {
    console.log("정리");
  }
```

```
}
run();
</script>
```

※ 참고하기 병렬방식, 직렬방식

```
<script>
        function work(id, ms){
          return new Promise(res => setTimeout(() => res(id), ms));
        }
        async function serial(){
          console.time("serial");
          const a = await work("A", 500);
          const b = await work("B", 600);
          console.timeEnd("serial"); // 약 1100ms
        }
        async function parallel(){
          console.time("parallel");
          const [a, b] = await Promise.all([work("A", 500), work("B", 600)]);
          console.timeEnd("parallel"); // 약 600ms
        }
        serial();
        parallel();
</script>
```

10-1. 예제함께하기

```
<!DOCTYPE html><meta charset="utf-8">
<body>
<div id="status"> 로딩 중..... </div>
<pre id="out"></pre>
<script>
function fakeFetch(){
  return new Promise((res,rej) => setTimeout(() => {
    res({ id:1, name:"hong" });
    // rej(new Error("오류발생"))
    }, 800) );
}

async function main(){
  const status = document.getElementById('status');
  const out = document.getElementById('out');
  try{
    const data = await fakeFetch();
    status.textContent = "    완료";
    out.textContent = JSON.stringify(data, null, 2);
  }catch(e){
    status.textContent = "    실패" + e;
  }
```

```
}
main();
</script>
</body>
```

예제 함께하기 2

```
<script>
function readUser(callback){
    setTimeout(() => callback(null, {id:1}), 300);
}


// Promise 래핑
const pReadUser = () => new Promise((res, rej) => readUser((e, v)=> e ? rej(e):res(v)));


// 리팩터링: async/await
(async () => {
  try{
    const user = await pReadUser();
    console.log(user);
  }catch(e){
    console.error(e);
  }
})();
</script>
</body>
```

문제 1. 프로미스를 통해 다음과 같이 출력되도록 하세요

```
then 1:  점심 준비
then 2:  점심 준비 + 숟가락
then 3:  점심 준비 + 숟가락 + 젓가락
항상 실행(finally)
```

```
<script>
new Promise((resolve) => {
  setTimeout(() => resolve(" 점심 준비 "), 700);
})
.then(v => {
  console.log("then 1:", v);
  return v + " + 숟가락";
})
.then(v2 => {
  console.log("then 2:", v2);
```

```
  // throw new Error("젓가락 없음"); // ← 주석 해제시 catch 실행
  return v2 + " + 젓가락";
})
.then(v3 => console.log("then 3:", v3))
.catch(err => console.log("catch:", err.message))
.finally(() => console.log("항상 실행(finally)"));
</script>
```

문제 2. 프로미스(async와 await를 활용하여 1초가 흐른디 랜덤함수를 발생하여 0.7보다 작으면 ok 그렇지 않으면 falil을 표시하도록 구현하세요) (try, catch, finally사용

```
실패: FAIL
정리
```

```
성공: OK
정리
```

```
<script>
function mightFail(){
  return new Promise((res, rej) => {
    setTimeout(() => Math.random() < 0.5 ? res("OK") : rej(new Error("FAIL")), 300);
  });
}

async function run(){
  try{
    const v = await mightFail();
    console.log("성공:", v);
  }catch(e){
    console.log("실패:", e.message);
  }finally{
    console.log("정리");
  }
}
run();
</script>
```