Loops: while 과 for statement

개발을 하다 보면 여러 동작을 반복해야 하는 경우가 종종 생긴다.
상품 목록에서 상품을 차례대로 출력하거나 숫자를 1부터 10까지 하나씩 증가시키면서 동일한 코드를 반복 실행해야 하는 경우이다. 반복문(loop) 을 사용하면 동일한 코드를 여러 번 반복할 수 있다.

❖ 'while' 반복문

while 반복문의 문법은 다음과 같다.

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

condition(조건)이 truthy 이면 반복문 본문의 코드가 실행된다.

아래 반복문은 조건 i < 3을 만족할 동안 i를 출력해준다.

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

반복문 본문이 한 번 실행되는 것을 반복(iteration) 이라고 부른다. 위 예시에선 반복문이 세 번의 iteration을 만든다.

i++가 없었다면 이론적으로 반복문이 영원히 반복되었을 것이다. 그로나 Browser는 이런 무한 반복을 멈추게 해주는 실질적인 수단을 제공한다. Server-side JavaScript도 이런 수단을 제공해 줌으로 무한으로 반복되는 프로세스를 죽일 수 있다.

반복문 조건엔 비교뿐만 아니라 모든 종류의 expression(표현식), variable(변수)가 올 수 있다. 조건은 while에 의해 평가되고, 평가 후엔 bolean value(불린값)으로 변경된다.

아래 예시에선 while (i != 0)을 짧게 줄여 while (i)로 만들어보았다.

```
let i = 3;
while (i) { // i가 0이 되면 조건이 falsy가 되므로 반복문이 멈춘다.
  alert( i );
  i--;
}
```

본문이 한 줄이면 curly brace(중괄호)를 쓰지 않아도 된다. 반복문 본문이 한 줄짜리 문이라면 중괄호 {…}를 생략할 수 있다.

```
let i = 3;
while (i) alert(i--);
```

❖ 'do…while' 반복문

do..while 문법을 사용하면 condition을 반복문 본문 아래로 옮길 수 있다.

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

이때 본문이 먼저 실행되고, 조건을 확인한 후 조건이 truthy인 동안엔 본문이 계속 실행된다.

[예시]

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

do..while 문법은 조건이 truthy 인지 아닌지에 상관없이, 본문을 최소한 한번이라도 실행하고 싶을 때만 사용해야 한다. 대다수의 상황에선 do..while보다 while(…) {…}이 적합하다.

❖ 'for' 반복문

for 반복문은 while 반복문보다는 복잡하지만 가장 많이 쓰이는 반복문이다.
문법은 다음과 같다.

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

for문을 구성하는 각 요소가 무엇을 의미하는지 알아보자. 아래 반복문을 실행하면 i가 0부터 3이 될 때까지(단, 3은 포함하지 않음) alert(i)가 호출된다.

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

이제 for문의 구성 요소를 하나씩 살펴보자.

☞ 구성 요소

begin	let i = 0	반복문에 진입할 때 단 한 번 실행된다.
condition	i < 3	반복마다 해당 조건이 확인된다. false이면 반복문을 멈춘다.
body	alert(i)	condition이 truthy일 동안 계속해서 실행된다.
step	i++	각 반복의 body가 실행된 이후에 실행된다.

일반적인 반복문 algorithm은 다음과 같다.

```
begin을 실행함
→ (condition이 truthy이면 → body를 실행한 후, step을 실행함)
→ (condition이 truthy이면 → body를 실행한 후, step을 실행함)
→ (condition이 truthy이면 → body를 실행한 후, step을 실행함)
→ ...
```

begin이 한 차례 실행된 이후에, condition 확인과 body, step이 계속해서 반복 실행된다.

반복문을 처음 배우면, 위 예시를 실행했을 때 어떤 과정을 거쳐 alert 창이 출력되는지 종이에 적어가며 공부해보자. 이렇게 하면 반복문을 쉽게 이해할 수 있다.

정확히 어떤 과정을 거치는지는 아래 예시에서 확인할 수 있다.

```
// for (let i = 0; i < 3; i++) alert(i)

// begin을 실행함
let i = 0
// condition이 truthy이면 → body를 실행한 후, step을 실행함
if (i < 3) { alert(i); i++ }
// condition이 truthy이면 → body를 실행한 후, step을 실행함
if (i < 3) { alert(i); i++ }
// condition이 truthy이면 → body를 실행한 후, step을 실행함
if (i < 3) { alert(i); i++ }
// i == 3이므로 반복문 종료
```

☞ inline variable declaration(인라인 변수 선언)

지금까진 'counter' 변수 i를 반복문 안에서 선언하였다. 이런 방식을 'inline' 변수 선언이라고 부른다. 이렇게 선언한 변수는 반복문 안에서만 접근할 수 있다.

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i);    // Error: i is not defined
```

inline variable declaration 대신, 정의되어있는 변수를 사용할 수도 있다.

```
let i = 0;

for (i = 0; i < 3; i++) { // 기존에 정의된 변수 사용
  alert(i);  // 0, 1, 2
}

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

❖ 구성 요소 생략하기

for문의 구성 요소를 생략하는 것도 가능하다.
반복문이 시작될 때 아무것도 할 필요가 없으면 begin을 생략하는 것이 가능하다.

예시를 살펴보자.

```
let i = 0; // i를 선언하고 값도 할당하였다.
```

```
for (; i < 3; i++) { // 'begin'이 필요하지 않기 때문에 생략하였다.
  alert( i ); // 0, 1, 2
}
```

step 역시 생략할 수 있다.

```
let i = 0;

for (; i < 3;) {
  alert( i++ );
}
```

위와 같이 for문을 구성하면 while (i < 3)과 동일해진다.

모든 구성 요소를 생략할 수도 있는데, 이렇게 되면 무한 반복문이 만들어진다.

```
for (;;) {
  // 끊임 없이 본문이 실행된다.
}
```

for문의 구성요소를 생략할 때 주의할 점은 두 개의 ; 세미콜론을 꼭 넣어주어야 한다는 점이다. 하나라도 없으면 문법 에러가 발생한다.

❖ Breaking the loop(반복문 빠져나오기)

대개는 반복문의 조건이 falsy가 되면 반복문이 종료된다.
그런데 특별한 지시자인 break를 사용하면 언제든 원하는 때에 반복문을 빠져나올 수 있다.

아래 예시의 반복문은 사용자에게 일련의 숫자를 입력하도록 안내하고, 사용자가 아무런 값도 입력하지 않으면 반복문을 '종료'한다.

```
let sum = 0;

while (true) {
  let value = +prompt("숫자를 입력하세요.", '');

  if (!value) break; // (1)
  sum += value;
```

```
}
alert( '합계: ' + sum );
```

(1)로 표시한 줄에 있는 break는 사용자가 아무것도 입력하지 않거나 Cancel 버튼을 눌렀을 때 활성화된다. 이때 반복문이 즉시 중단되고 제어 흐름이 반복문 아래 첫 번째 줄로 이동한다. 여기선 alert가 그 첫 번째 줄이 된다.

반복문의 시작 지점이나 끝 지점에서 조건을 확인하는 것이 아니라 본문 가운데 혹은 본문 여러 곳에서 조건을 확인해야 하는 경우, 'infinite loop(무한 반복문) + break' 조합을 사용하면 좋다.

❖ Continue to the next iteration(다음 반복으로 넘어가기)

continue 지시자는 break의 'lighter version'이다. continue는 전체 반복문을 멈추지 않는다. 대신에 현재 실행 중인 iteration을 멈추고 반복문이 다음 iteration을 강제로 실행시키도록 한다(조건을 통과할 때).
continue는 현재 반복을 종료시키고 다음 반복으로 넘어가고 싶을 때 사용할 수 있다.

아래 반복문은 continue를 사용해 홀수만 출력한다.

```
for (let i = 0; i < 10; i++) {

  // 조건이 참이라면 남아있는 본문은 실행되지 않는다.
  if (i % 2 == 0) continue;

  alert(i); // 1, 3, 5, 7, 9가 차례대로 출력됨
}
```

i가 짝수이면 continue가 본문 실행을 중단시키고 다음 iteration이 실행되게 한다(i가 하나 증가하고, 다음 반복이 실행됨). 따라서 alert 함수는 인수가 홀수일 때만 호출된다.

⊙ continue는 nesting(중첩)을 줄이는 데 도움을 준다.

홀수를 출력해주는 예시는 아래처럼 생길 수도 있다.

```
for (let i = 0; i < 10; i++) {
  if (i % 2) {
    alert( i );
  }

}
```

기술적인 관점에서 봤을 때, 이 예시는 위쪽에 있는 예시와 동일하다. continue를 사용하는 대신 코드를 if 블록으로 감싼 점만 다르다.

그런데 이렇게 코드를 작성하면 부작용으로 nesting level(중첩 레벨)(중괄호 안의 alert 호출)이 하나 더 늘어난다. if 안의 코드가 길어진다면 전체 가독성이 떨어질 수 있다.

⊙ '?' 오른쪽엔 break나 continue가 올 수 없다.

표현식이 아닌 문법 구조(syntax construct)는 삼항 연산자 ?에 사용할 수 없다는 점을 항상 유의한다. 특히 break나 continue 같은 지시자는 ternary operator(삼항 연산자)에 사용하면 안된다.

아래와 같은 조건문이 있다고 가정한다.

```
if (i > 5) {
  alert(i);
} else {
  continue;
}
```

물음표를 사용해서 위 조건문을 아래와 같이 바꾸려는 시도를 할 수 있을 것이다.

```
(i > 5) ? alert(i) : continue; // 여기에 continue를 사용하면 안된다.
```

이런 코드는 문법 에러를 발생시킨다.

이는 물음표 연산자 ?를 if문 대용으로 쓰지 말아야 하는 이유 중 하나이다.

❖ break/continue를 위한 Label

여러 개의 nested loop(중첩 반복문)을 한 번에 빠져나와야 하는 경우가 종종 생기곤 한다.
i와 j를 반복하면서 prompt 창에 (0,0)부터 (2,2)까지를 구성하는 좌표 (i, j)를 입력하게 해주는 예시를 살펴보자.

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`(${i},${j})의 값`, '');
    // 여기서 멈춰서 아래쪽의 `완료!`가 출력되게 하려면 어떻게 해야 할까요?
  }
```

```
}

alert('완료!');
```

사용자가 Cancel 버튼을 눌렀을 때 반복문을 중단시킬 방법이 필요하다.
input 아래에 평범한 break 지시자를 사용하면 안쪽에 있는 반복문만 빠져나올 수 있다. 이것만으론 충분하지 않다(중첩 반복문을 포함한 반복문 두 개 모두를 빠져나와야 하기 때문이다). 이럴 때 label을 사용할 수 있다.
label 은 반복문 앞에 콜론과 함께 쓰이는 identifier(식별자) 이다.

```
labelName: for (...) {
  ...
}
```

반복문 안에서 break <labelName> 문을 사용하면 label에 해당하는 반복문을 빠져나올 수 있다.

```
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`(${i},${j})의 값`, '');
    // 사용자가 아무것도 입력하지 않거나 Cancel 버튼을 누르면 두 반복문 모두를 빠져나온다.
    if (!input) break outer; // (1)
    // 입력받은 값을 가지고 무언가를 함
  }
}
alert('완료!');
```

위 예시에서 break outer는 outer라는 label이 붙은 반복문을 찾고, 해당 반복문을 빠져나오게 해준다. 따라서 제어 흐름이 (1)에서 alert('완료!')로 바로 바뀝다.

label을 별도의 줄에 써주는 것도 가능하다.

```
outer:
for (let i = 0; i < 3; i++) { ... }
```

continue 지시자를 label과 함께 사용하는 것도 가능하다. 두 가지를 같이 사용하면 label이 붙은 반복문의 다음 iteration이 실행된다.

⊙ label은 마음대로 'jump'할 수 있게 해주지 않는다.

label을 사용한다고 해서 원하는 곳으로 마음대로 jump 할 수 있는 것은 아니다.
아래 예시처럼 label을 사용하는 것은 불가능하다.

```
break label; // 아래 for 문으로 점프할 수 없다.

label: for (...)
```

break와 continue는 반복문 안에서만 사용할 수 있고, label은 반드시 break이나 continue 지시자 위에 있어야 한다.

요약하면, 지금까지 3 종류의 반복문에 대해 살펴보았다.

☞ while - 각 반복이 시작하기 전에 조건을 확인한다.
☞ do..while - 각 반복이 끝난 후에 조건을 확인한다.
☞ for (;;) - 각 반복이 시작하기 전에 조건을 확인한다. 추가 세팅을 할 수 있다.

'무한' 반복문은 보통 while(true)를 써서 만든다. 무한 반복문은 여타 반복문과 마찬가지로 break 지시자를 사용해 멈출 수 있다.

현재 실행 중인 반복에서 더는 무언가를 하지 않고 다음 반복으로 넘어가고 싶다면 continue 지시자를 사용할 수 있다.

반복문 앞에 label을 붙이고, break/continue에 이 label을 함께 사용할 수습니다. label은 중첩 반복문을 빠져나와 바깥의 반복문으로 갈 수 있게 해주는 유일한 방법이다.

[과제]

☞ 반복문의 마지막 값

★ **중요도:** 3

아래 코드를 실행했을 때 얼럿 창에 마지막으로 뜨는 값은 무엇일까? 이유도 함께 설명해보자.

```
let i = 3;

while (i) {
  alert( i-- );
}
```

[해답]

답: 1

```
let i = 3;

while (i) {
  alert( i-- );
}
```

반복이 하나씩 끝날 때마다 i는 1씩 줄어든다. while(i)은 i = 0일 때 멈춘다.
따라서 전체 반복문은 아래 순서를 따라 실행된다.

```
let i = 3;
alert(i--); // 3이 출력되고 i는 2로 줄어든다.
alert(i--) // 2가 출력되고 i는 1로 줄어든다.
alert(i--) // 1이 출력되고 i는 0으로 줄어든다.
// i가 0이 되었기 때문에 while(i)는 종료된다.
```

☞ while 반복문의 출력값 예상하기

★ 중요도: 4

while 반복문이 순차적으로 실행될 때마다 alert 창에 어떤 값이 출력될지 예상해보자.
아래 두 예시는 같은 값을 출력할까?

전위형 증가 연산자를 사용한 경우(++i):

```
let i = 0;
while (++i < 5) alert( i );
```

후위형 증가 연산자를 사용한 경우(i++):

```
let i = 0;
while (i++ < 5) alert( i );
```

[해답]

이 문제는 비교 연산자와 후위/전위형 연산자를 함께 사용하는 경우 어떤 차이가 있는지 보여준다.
전위형 증가 연산자를 사용한 경우엔 1부터 4까지 출력된다.

```
let i = 0;
while (++i < 5) alert( i );
```

++i는 i를 먼저 증가시키고 새로운 값을 반환하기 때문에 첫 번째 while 반복문에선 1과 5를 비교(1 < 5)하고, 얼럿 창엔 1이 출력된다.
1에 이어서 2, 3, 4…이 출력된다. i 앞에 ++가 붙어있기 때문에 5는 항상 증가 이후의 값과 비교된다.
i = 4 이후에 i의 값이 5로 증가하면 while(5 < 5)안의 비교가 실패하기 때문에 반복문은 멈춘다.
따라서 5는 출력되지 않는다.

후위형 증가 연산자를 사용한 경우엔 1부터 5까지 출력됩니다.

```
let i = 0;
while (i++ < 5) alert( i );
```

후위 증가 연산자를 적용하면 i++는 i를 증가시키긴 하지만 기존 값을 반환한다. 따라서 첫 번째 while 반복문에선 0과 5를 비교(0 < 5)한다. 이 점이 전위 증가 연산자와의 차이이다.
그런데 alert 문은 조건문과 별개의 문이므로 alert 창엔 1이 출력된다. i는 이미 증가한 이후이기 때문이다.
1이 출력된 이후에 2, 3, 4…가 이어서 출력된다.
i = 4일 때 잠시 생각을 가다듬어 보자. 전위 증가 연산자(++i)를 사용하면 값이 먼저 증가하기 때문에 5와 5를 비교하게 되는데, 여기선 후위 증가 연산자(i++)를 사용하고 있으므로 i는 증가하지만 기존 값인 4가 비교에 사용된다. 따라서 while(4 < 5)가 되고, 해당 조건은 true이므로 하단 블록이 실행되어 alert 창이 뜨게 된다.
다음 반복문은 while(5 < 5)이므로 마지막 출력되는 값은 5가 된다.

☞ 'for' 반복문의 출력값 예상하기

★ 중요도: 4

for 반복문이 순차적으로 실행될 때마다 얼럿 창에 어떤 값이 출력될지 예상해보자.
아래 두 예시는 같은 값을 출력할까?

후위형 증가 연산자를 사용한 경우(i++):

```
for (let i = 0; i < 5; i++) alert( i );
```

전위형 증가 연산자를 사용한 경우(++i):

```
for (let i = 0; i < 5; ++i) alert( i );
```

[해답]

두 경우 모두 0부터 4까지 출력된다.

```
for (let i = 0; i < 5; ++i) alert( i );
for (let i = 0; i < 5; i++) alert( i );
```

for문의 algorithm을 떠올려보면 쉽게 추론할 수 있는 문제이다.
(1) 모든 작업이 시작되기 전 일단 i = 0 이다.
(2) i < 5 조건을 만족하는지 확인한다.
(3) 위 조건이 true이면 반복문의 본문 alert(i)가 실행되고, 그 이후 i++가 실행된다.

i++는 위 algorithm의 두 번째 단계(조건 확인)와 별개로 실행된다. 전혀 다른 구문이기 때문이다.
증가 연산자가 반환하는 값은 (2) 에서 쓰이지 않기 때문에 i++와 ++i에 차이가 없다.

☞ for 반복문을 이용하여 짝수 출력하기

★ 중요도: 5

for 반복문을 이용하여 2부터 10까지 숫자 중 짝수만을 출력해보자.

[해답]

```
for (let i = 2; i <= 10; i++) {
  if (i % 2 == 0) {
    alert( i );
  }
}
```

나머지 연산자 %를 사용하면 짝수인지를 확인할 수 있다.

☞ 'for' 반복문을 'while' 반복문으로 바꾸기

★ 중요도: 5

for 반복문을 while 반복문으로 바꾸되, 동작 방식에는 변화가 없도록 해보자. 출력 결과도 동일해야 한다.

```
for (let i = 0; i < 3; i++) {
  alert( `number ${i}!` );
}
```

[해답]

```
let i = 0;
while (i < 3) {
  alert( `number ${i}!` );
  i++;
}
```

☞ 사용자가 유효한 값을 입력할 때까지 prompt 창 띄우기

★ 중요도: 5

사용자가 100보다 큰 숫자를 입력하도록 안내하는 prompt 창을 띄워보자. 사용자가 조건에 맞지 않은 값을 입력한 경우 반복문을 사용해 동일한 prompt 창을 띄워준다.
사용자가 100을 초과하는 숫자를 입력하거나 취소 버튼을 누른 경우, 혹은 아무것도 입력하지 않고 확인 버튼을 누른 경우엔 더는 prompt 창을 띄워주지 않아도 된다.
사용자가 오직 숫자만 입력한다고 가정하고 답안을 작성하도록 해보자. 숫자가 아닌 값이 입력되는 예외 상황은 처리하지 않아도 된다.

[해답]

```
let num;

do {
  num = prompt("100을 초과하는 숫자를 입력해주세요.", 0);
} while (num <= 100 && num);
```

do..while반복문을 사용해 아래 두 조건이 모두 truthy인 경우 prompt 창이 뜨게 하면 된다.
num <= 100인지 확인하기. 100보다 작거나 같은 값을 입력한 경우 prompt 창이 떠야 한다.
num이 null이나 빈 문자열인지 확인하기. num이 null이나 빈 문자열이면 && num이 거짓이 되므로 while 반복문이 종료된다. 참고로 num이 null인 경우 num <= 100은 true가 되므로 두 번째 조건이 없으면 취소 버튼을 눌러도 반복문이 계속해서 실행된다. 따라서 위 두 조건을 모두 확인해야 한다.

☞ Prime number(소수) 출력하기

★ **중요도: 3**

prime numbe)는 자신보다 작은 두 개의 자연수를 곱하여 만들 수 없는 1보다 큰 자연수이다.
다시 말해서 1과 그 수 자신 이외의 자연수로는 나눌 수 없는 자연수를 소수라고 부른다.
5는 2나 3, 4로 나눌 수 없기 때문에 소수이다. 5를 이들 숫자로 나누면 나머지가 있기 때문이다.
2부터 n까지의 숫자 중 소수만 출력해주는 코드를 작성해보자.
n = 10이라면 결과는 2,3,5,7이 되어야 한다.
주의할 점은 작성한 코드는 임의의 숫자 n에 대해 동작해야 한다.

[해답]

소수를 판단하는 algorithm은 다양하다. 먼저 중첩 반복문을 사용한 algorithm을 살펴보자.

```
범위 내 모든 숫자 i에 대해서 {
  1과 i 사이에 제수가 있는지를 확인
  있으면 => 소수가 아님
  없으면 => 소수이므로 출력해줌
}
```

label을 사용해 위 algorithm을 구현한 코드는 다음과 같다.

```
let n = 10;

nextPrime:
for (let i = 2; i <= n; i++) { // 각 i에 대하여 반복문을 돌림

  for (let j = 2; j < i; j++) { // divisor(나눗수)를 찾음
    if (i % j == 0) continue nextPrime; // prime이 아니므로 다음 i로 넘어감
  }

  alert( i ); // prime
}
```

위에서 사용한 algorithm은 최적화할 부분이 많다. divisor를 2와 i의 square root(제곱근) 사이에서 찾으면 좀 더 나아진다.

아주 큰 n에 대해서 Quadratic sieve(2차 체)나 General number field sieve(수체 체)와 같이 좀 더 어려운 수학과 복잡한 algorithm을 이용해 소수 검색 algorithm을 개선할 수 있을 것이다.

Quadratic sieve는 어떤 큰 자연수 N을 prime factorization(소인수분해)하기 위해 사용되는 prime

factorization(소인수분해) algorithm으로, 양자컴퓨터가 상용화되었을 때 기준으로는 현재까지 발견된 algorithm 중에서 3번째로 빠른 algorithm이며, General number field sieve(수체 체)의 기본이 되어 General number field sieve(수체 체)보다 더 간단한 algorithm이다.

General number field sieve(수체 체) algorithm은 어떤 양의 정수 N을 빠르게 prime factorization(소인수분해)할 수 있는 prime factorization(소인수분해) algorithm이다.

prime factorization(소인수분해)는 1보다 큰 자연수를 소인수(소수인 인수)들만의 곱으로 나타내는 것 또는 합성수를 소수의 곱으로 나타내는 방법을 말한다.