02

Javascript -1

기본 문법
제어문

들어가기 전에

Statement(서술문) 와 Expression(표현식)

- Statement - 진술, 서술, 서술문
  - 실행 가능한(Executable) 최소의 독립적인 코드
- Expression - 식, 수식, 표현식
  - Statement의 부분 집합
  - 평가(Evaluation)을 통해 하나의 ‘값’ 이 되는 코드
  - 산술 연산, 문자열 연산, 논리 연산, 함수 호출 등

기본 문법

기본 문법

- 파일 확장자는 js
- 파일의 첫줄부터 실행된다.
- 대문자와 소문자가 구별된다.
- 주석은 // (한 줄 주석) 과 /* */ (여러 줄 주석)를 사용한다.
  - 여러 줄 주석의 중첩은 허용하지 않는다. `/* /* */ */` 는 에러가 발생한다.

기본 문법

- 문장 끝에 세미콜론(;)은 생략할 경우 자동으로 삽입된다. (Automatic Semicolon Insertion)
  - 다음 줄의 코드가 이번 줄의 코드와 확실히 연결되지 않을 때
  - 다음 줄의 코드가 } 로 시작할 때
  - 파일의 마지막
  - 다음 statement가 한 줄에 단독으로 사용될 때: return, break, throw, continue

기본 문법

- ASI 를 주의해야 할 때

```
1  function hello(){
2      return
3      {
4          message:'Hello'
5      }
6  }
7
8  console.log(hello())
```

```
1  function hello(){
2      return {
3          message:'Hello'
4      }
5  }
6
7  console.log(hello())
```

return 문이 한 줄에 단독으로 사용되면서
return; 로 처리되어 객체가 반환되지 못함

기본 문법

- ASI 를 주의해야 할 때

```
1   const str = 'Hello'
2   const message = '^^ ' + str
3   [0] + [1]
4   console.log(message)
```

```
1   const str = 'Hello'
2   const message = '^^ ' + str[0] + [1]
3   console.log(message)
```

다음 줄 코드와 연결해도 문법 상 이상이 없으면 아래 코드와 연결되어 처리될 수 있다.

변수

변수

- 이름을 가진 값 저장 공간
  - 이름과 값으로 이루어진다.
  - 선언과 값 할당 동작이 있다.
    - 선언과 동시에 값을 할당 할 수 있고 선언만 해 둔 다음 값을 나중에 할당할 수 있다.
    - 값을 여러 번 할당하면 변수, 한 번만 할당할 수 있으면 상수라 한다.
  - 값에 따라 타입을 가진다.

변수의 타입

- 원시 타입 (Primitive type) - 불변 타입

Type	값의 범위	참조
boolean	true/false	
null	null	값이 없음을 나타내는 특수한 객체
undefined	undefined	값이 할당되지 않은 변수
number	8Byte 부동소숫점 값	double, +infinite, -infinite, NaN
biging	number를 넘어서는 값	정수 끝에 n을 추가
string	16비트 부호 없는 정수값의 집합	문자열. ‘’또는 “” 로 문자열을 감싼다.
Symbol		다른 값과 중복되지 않는 고유한 값

변수 선언과 할당

- 변수이름 = 값　(예: value = ‘user’ 또는 value=1)
  - 동작은 하지만 추천하지 않는 방법. 변수의 선언과 값 할당이 모호해 진다.
- var 변수이름 = 값 (예: var value=‘user’ 또는 var value=1)
  - ES6 이전 방식의 변수 선언
  - 중복 선언이 가능하다.
  - 그 외 다른 언어와 많이 다른 성격을 보이므로 주의한다.

```
1   var value=1
2   var value=2 // 중복 선언
```

변수 선언과 할당

- let, const: ES6 부터 지원
  - 다른 언어와 유사한 특징을 가지는 변수 선언법.
  - 같은 범위 내 중복 선언 불가능, { } 을 범위로 가진다. Hoisting 되지 않는다.
  - let: 값을 여러번 할당, const: 상수

```
1   let value=1
2   //let value=2 // 중복 선언 시 에러 발생
3   value=2 // 새로운 값 할당은 가능
```

```
1   const value=2
2   // value=3 // 상수라서 값 수정 불가능
```

변수 선언과 할당

- var, let은 선언과 할당을 분리할 수 있다.
  - 값이 할당되기 전에는 undefined 이다.
- const 는 선언하면서 바로 할당해야 한다.

```
1   let value; // 선언만
2   console.log(value); // undefined
3
4   value=3;
5   console.log(value);
```

```
1   const value; // error. const는 선언하면서 할당해야 한다.
2   console.log(value);
```

예약어

- 변수, 함수, 라벨에서
- 이름으로 사용 못함

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Words marked with* are new in ECMAScript 5 and 6.

https://www.w3schools.com/js/js_reserved.asp

변수 선언과 할당 실습

- console.log(변수이름) : 값 출력
- typeof 변수이름: 변수의 타입
- typeof(변수이름): 변수의 타입

```
1  const val_boolean = true
2  const val_null = null
3  let val_undefined_1 // undefined
4  const val_undefined_2=undefined
5  const val_number_1 = 1
6  const val_number_2 = 1.1
7  const val_bigint = 1n
8  const val_string='abc'
9
10 console.log(typeof val_boolean ) // 이렇게 해도 되고
11 console.log(typeof(val_null)) // 함수처럼 불러도 된다.
12 console.log(typeof val_undefined_1)
13 console.log(typeof val_undefined_2)
14 console.log(typeof val_number_1)
15 console.log(typeof val_number_2)
16 console.log(typeof val_bigint)
17 console.log(typeof val_string)
```

변수 타입 - Symbol

- ES6에 도입, 중복되지 않는 고유한 값을 생성
  - Symbol('설명') 함수 호출로 생성, 설명 내용이 같더라도 내부적으로는 다른 값 생성
  - 파라미터로 넘어가는 문자열은 설명(description)으로 실제 생성되는 값을 확인 할 수는 없다.

```
1   const symbol1 = Symbol('a')
2   const symbol2 = Symbol('a')
3   const symbol3 = Symbol() // description은 선택사항
4
5   console.log(symbol1.description)
6   console.log(symbol1.toString())
7   console.log(symbol1===symbol2) // false
```

변수 타입 - Symbol

전역 Symbol

- Symbol.for(Key) : 전역 symbol 레지스트리에 접근. 해당 키를 가진 심볼이 있으면 반환, 없으면 만들어서 반환. 이를 통해 원하는 키로 심볼을 생성할 수 있다.

```
1   const symbolA = Symbol.for('A') // 없으니까 새로 생성
2   const symbolA2 = Symbol.for('A') // 생성해둔 심볼 반환
3   console.log(symbolA===symbolA2)
4   console.log(Symbol.keyFor(symbolA)) // A
```

- Symbol() 로 생성할 경우 키 지정을 할 수 없기 때문에 전역 레지스트리에서 관리하지 않는다.

String Template Literals

- 문자열을 정의할 때 ``(백틱)를 사용하면 표현식을 문장 사이에 삽입할 수 있다.
- 표현식은 ${expression} 과 같이 표현한다.

```
1  let i = 0;
2  const str = `value is ${i}`;
3  console.log(str);
```

```
1  let i = 0;
2  const str = `value is ${i+5}`;
3  console.log(str);
```

```
1  // 줄바꿈이 적용된다.
2  const str=`Hello,
3  Everyone`;
4
5  console.log(str);
```

연산자

기본 연산자

할당 연산자

	사용법	풀어쓰기	참고
할당	x = y		x 에 y 값을 할당. 할당 후 x 가 반환된다.
더하기 할당	x += y	x = x + y	빼기(-=), 곱하기(*=), 나누기(/=), 나머지(%=)
거듭제곱 할당	x **= y	x = x ** y	2 ** 3 => 8
좌측 시프트 할당	x <<= y	x = x << y	우측 시프트(>>=)
비트 AND	x &= y	x = x & y	비트 OR(|=) , 비트 XOR(^=)
논리 AND	x &&= y	x = x && y	OR(||=)
Nullish 할당	x ??= y	x ?? (x=y)	만약 x 가 null 이거나 undefined 라면 y 를 할당하라
참고**		x ?? y	왼쪽이 null 이거나 undefined 라면 오른쪽 항목을, 아니면 왼쪽을 반환

기본 연산자

산술 연산자

	사용법	참고
사칙연산		+, -, *, /
나머지	x % y	
거듭제곱	x ** y	2 ** 3 => 8
전위 단항 증가	++x	피연산자에 1을 더함
후위 단항 증가	x++	더하기 전의 피연산자 값을 반환하고 이후 1을 더함
전위 단항 감소	--x	피연산자에서 1을 뺀다
후위 단항 감소	x--	빼기 이전의 피연산자 값을 반환하고 이후 1을 뺀다.
단항 부정	-x	피연산자의 부호를 반대로 바꾼다.
단항 더하기	+x	피연산자가 숫자가 아닐 경우 숫자로 변환 시도. +true => 1

기본 연산자

비교 연산자 - true/false를 반환

	사용법	참고
같다	x == y	x 와 y의 값이 같을 때. 1 == '1' 도 true 임을 주의한다. 데이터 타입이 서로 다를 때는 숫자형으로 변환을 시도한다.
다르다	x != y	빼기(-=), 곱하기(*=), 나누기(/=), 나머지(%=)
일치한다	x===y	값과 타입이 모두 같을 때
불일치	x !== y	값 또는 타입이 다를 때
크다	x > y	문자열의 경우 유니코드 순으로 크다, 작다를 판별한다.
크거나 같다	x >= y	
작다	x < y	
작거나 같다	x <= y	

기본 연산자

비교 연산자 - true/false를 반환

- 비교 연산자 주의할 점

```
1   console.log( 1=='1' )               // true
2   console.log( 0 == false )           // true
3   console.log( '' == false )          // true
4   console.log( undefined == 0 )       // false
5   console.log( null == 0)             // false
6   console.log( undefined == null )    // true. 특별한 케이스.
```

조건문

if

- if (statement) { statements }
  - 괄호 안의 문장이 true 이면 { } 의 코드 블록이 실행된다.

```
1   const num1 = 10;
2   const num2 = 11;
3
4   if(num1 > num2) {
5       console.log('not executed');
6   }
7
8   if(num1 < num2) {
9       console.log('executed');
10  }
```

if-else

- if (statement) { statements } else { }
  - 괄호 안의 문장이 true 이면 { } 의 코드 블록이 실행된다.
  - false일 경우 else 의 코드 블록이 실행된다.

```
1  const num1 = 10;
2  const num2 = 11;
3
4  if(num1 > num2) {
5      console.log('if');
6  }
7  else { // num1 < num2 or num1==num2
8      console.log('else');
9  }
```

if-else if - else

- if (statement) { }
- else if (statement) { }
- else { }
  - 괄호 안의 문장이 true 이면 if의 의 코드 블록이 실행된다.
  - false일 경우 중 else if 의 문장이 true 이면 else if 의 코드 블록이 실행된다.
  - if, else if 에서 모두 false 인 경우 else의 코드 블록이 실행된다.

if-else if - else

```
1   const score=80;
2
3   if(score >= 90){
4       console.log('A');
5   } else if(score >= 80){ // score < 90 and score >= 80
6       console.log('B');
7   } else if(score >= 70){ // score < 80 and score >= 70
8       console.log('C');
9   } else if(score >= 60){ // score < 70 and score >= 60
10      console.log('D');
11  } else {                // score < 60
12      console.log('F');
13  }
```

삼항 연산자 '?'

- ? 연산자를 이용해 if/else 간단히 표기할 수 있다.
- condition ? statement1:statement2
- condition이 true라면 statement1을 반환
- false라면 statement2를 반환

```
1   const score=10;
2
3   let result;
4   if(score>5)
5       result=true;
6   else
7       result=false;
8
9   console.log(result);
10
11  // 위 if 문과 같은 동작
12  result = score > 5 ? true:false;
13
14  console.log(result);
```

switch

- 여러 조건을 가진 if-else if-else 를 대체할 수 있는 제어문

```
switch ( expression ) {
  case value1 : statements
                [break;]
  case value2 : statements
                [break;]
  [default: statements ]
}
```

switch

```
1  const date=0;
2
3  switch(date) {
4      case 0: console.log('SUN'); break;
5      case 1: console.log('MON'); break;
6      case 2: console.log('TUE'); break;
7      case 3: console.log('WED'); break;
8      case 4: console.log('THU'); break;
9      case 5: console.log('FRI'); break;
10     case 6: console.log('SAT'); break;
11     default: console.log('UNKNOWN')
12 }
```

```
1  const date=1;
2
3  switch(date) {
4      case 0: console.log('REST'); break;
5      case 1:
6      case 2:
7      case 3:
8      case 4:
9      case 5: console.log('WORK'); break;
10     case 6: console.log('REST'); break;
11     default: console.log('UNKNOWN')
12 }
```

break 를 만날 때 까지 계속 진행된다.

switch

- switch 의 괄호에는 계산식이 들어가도 된다.
- switch는 타입에 엄격하다.

```
1  const value=1;
2
3  switch( value%2 ) {
4      case 0: console.log('EVEN'); break;
5      case 1: console.log('ODD'); break;
6  }
```

```
1  const value='0';
2
3  switch(value) {
4      case 0:
5          console.log('case 0');
6          break;
7      default:
8          console.log('default');
9  }
```

반복문

for

- for(begin; condition; step) { statements }
  - begin: for 문을 시작하기 전의 초기화 문
  - condition: for문을 계속 수행할지 결정하는 논리 표현식. true로 판단되면 statements를 실행한다.
  - step: for 문을 한 번 수행할 때마다 수행되는 연산
  - begin을 실행 -> condition이 true이면 statements 실행 -> step 실행 -> condition이 true이면 statements 실행 -> step 실행…

for

```
1  // for의 초기화문에서 선언된 변순는
2  // for문 안에서만 사용할 수 있다.
3  for(let i=0; i < 3; i++){
4      console.log(i);
5  }
6
7  // console.log(i); // error
```

```
1  let i;
2  for(i=0; i < 3; i++){
3      console.log(i);
4  }
5
6  console.log(i); // 가능
```

```
1  let i=0;
2  // 할당까지 된 변수라면 초기화를 생략하고 바로 쓸 수 있다.
3  for( ; i < 3; i++){
4      console.log(i);
5  }
6
7  console.log(i); // 가능
```

while

- while(condition) { statements }
- 조건 확인 -> 실행 -> 조건 확인 -> 실행…
- 해당 조건이 true로 판단되는 동안 반복하여 statements 를 실행

```
1   let i=0;
2   while(i < 3) {
3       console.log(i);
4       i++;
5   }
```

- 무한히 실행되지 않도록 condition을 잘 관리해야 한다.

do while

- do { statements } while (condition);
- statements를 일단 실행하고 이후 조건을 이용하여 다시 실행할 것인지 결정한다.
- 실행 -> 조건 확인 -> 실행 -> 조건 확인…

```
1  let i=3;
2  do{
3      // i < 3 은 false이지만
4      // 일단 한 번은 실행한다.
5      console.log(i);
6      i++;
7  } while(i < 3);
```

반복문의 중첩

- 반복문은 중첩하여 작성가능하며 이 경우 반복 횟수는 외부 횟수 * 내부 횟수가 된다.

```
1  // i 는 2회
2  for(let i=0; i < 2; i++){
3      // 각 i 당 j 3회 반복
4      for(let j=0; j < 3; j++) {
5          console.log(i+','+j);
6      }
7  }
```

```
i=0   j=0
      j=1
      j=3

i=1   j=0
      j=1
      j=3
```

break

- 실행 중인 반복문을 종료한다.

```
let i=0;
while (true) {
    console.log(i);
    i++;
    if(i > 2) break;
}
```

```
// for문의 조건은 i < 10 이지만
// break로 조기 종료된다.
for(let i=0; i < 10; i++) {
    console.log(i);
    if(i > 2) break;
}
```

break

- 중첩된 반복문에서 사용할 경우 자신이 소속된 가장 내부의 반복문만 종료한다.

```
1  for(let i=0; i<3; i++) {
2      for(let j = 0; j < 5; j++) {
3          console.log(i+','+j);
4          // 내부 for 문만 종료된다.
5          if(j==2) break;
6      }
7  }
```

continue

- continue 이후의 statement를 수행하지 않고 반복문의 처음으로 돌아간다.
  - for: step 이 실행되고 condition을 체크한다.
  - while: condition 체크부터 다시 실행한다.

```
for(let i=0; i<7; i++) {
    if(i%2==1) continue;
    else console.log(i);
}
```

```
let i=-1;
while(i < 7) {
    i++;
    if(i%2==1){
        continue;
    }
    else{
        console.log(i);
    }
}
```

라벨

- 중첩된 반복문을 한번에 종료(break, continue) 해야 할 때

```
1  outer: // label
2  for(let i=0; i < 3; i++) {
3      inner: // label
4      for(let j=0; j < 3; j++) {
5          if(i + j > 2) break outer;
6          console.log(`${i} + ${j} = ${i+j}`);
7      }
8  }
```

- 라벨 이름은 예약어를 피해 만든다.

라벨

- continue는 반복문에서만 라벨을 사용할 수 있다.
- break는 코드블록에 라벨을 사용할 수 있다.

```
1  myBlock: {
2      let i=0;
3      console.log(i);
4      if(i==0) break myBlock;
5      console.log('Dead code...');
6  }
```