Chap9 타입 변환과 단축 평가

9-1 타입 변환이란?

- 기존 원시 값을 사용해 다른 타입의 새로운 원시 값을 생성하는 것
- 타입 변환 종류
  - 암묵적 타입 변환 (= 타입 강제 변환)
  - 명시적 타입 변환 (= 타입 캐스팅)

9-2 암묵적 타입 변환

- 표현식을 평가할 때 코드의 문맥에 부합하지 않는 상황에서 가급적 에러를 발생시키지 않도록 암묵적 타입 변환을 통해 표현식을 평가한다.

▼ 문자열 타입으로 변환 예시

```
// 숫자 타입
0 + ''          // → "0"
-0 + ''         // → "0"
1 + ''          // → "1"
-1 + ''         // → "-1"
NaN + ''        // → "NaN"
Infinity + ''   // → "Infinity"
-Infinity + '' // → "-Infinity"

// 불리언 타입
true + ''  // → "true"
false + '' // → "false"

// null 타입
null + '' // → "null"

// undefined 타입
undefined + '' // → "undefined"

// 심벌 타입
(Symbol()) + '' // → TypeError: Cannot convert a Symbol value to a string

// 객체 타입
({}) + ''            // → "[object Object]"
Math + ''            // → "[object Math]"
[] + ''              // → ""
[10, 20] + ''        // → "10,20"
(function(){}) + '' // → "function(){}"
Array + ''           // → "function Array() { [native code] }"
```

▼ 숫자 타입으로 변환 예시

```
// 문자열 타입
+''       // → 0
+'0'      // → 0
+'1'      // → 1
+'string' // → NaN

// 불리언 타입
+true     // → 1
+false    // → 0

// null 타입
+null     // → 0
```

```
// undefined 타입
+undefined // → NaN

// 심벌 타입
+Symbol() // → TypeError: Cannot convert a Symbol value to a number

// 객체 타입
+{}              // → NaN
+[]              // → 0
+[10, 20]        // → NaN
+(function(){}) // → NaN
```

▼ 불리언 타입으로 변환 예

```
// 아래의 조건문은 모두 코드 블록을 실행한다.
if (!false)     console.log(false + ' is falsy value');
if (!undefined) console.log(undefined + ' is falsy value');
if (!null)      console.log(null + ' is falsy value');
if (!0)         console.log(0 + ' is falsy value');
if (!NaN)       console.log(NaN + ' is falsy value');
if (!'')        console.log('' + ' is falsy value');
```

▼ Truthy / Falsy

- Truthy : 참 같은 값
- Falsy : 거짓 같 값

Falsy Values

- false
- 0
- "" or ' ' (Empty string)
- null
- undefined
- NaN

Truthy Values

- Everything else that is not falsy
- true
- '0' (0 in a string)
- ' ' (space in a string)
- 'false' (false in a string)
- [] (empty array)
- {} (empty object)
- function () {} (empty function)

9-3 명시적 타입 변환

- 개발자의 의도에 따라 명시적으로 타입을 변경하는 방법

- 방법
  - 생성자 함수(String, Number, Boolean)을 new 연산자 없이 호출하는 방법
  - 빌트인 메서드 사용하는 방법
  - 암묵적 타입 변환을 이용하는 방

▼ 문자열 타입으로 변환 예시

1. `String 생성자 함수`를 new 없이 호출하는 방법
2. `Object.prototype.toString` 메서드를 이용하는 방법
3. `문자열 연결 연산자`를 이용하는 방법

```
// 1. String 생성자 함수를 new 없이 호출하는 방법
// 숫자 타입 -> 문자열 타입
String(1);              // '1'
String(NaN);            // 'NaN'
String(Infinity)        // 'Infinity'

// 불리언 타입 -> 문자열 타입
String(true);           // 'true'
String(false);          // 'false'

// 2. Object.prototype.toString 메서드를 이용하는 방법
// 숫자 타입 -> 문자열 타입
(1).toString();         // '1'
(NaN).toString();       // 'NaN'
(Infinity).toString(); // 'Infinity'

// 불리언 타입 -> 문자열 타입
(true).toString();      // 'true'
(false).toString();     // 'false'

// 3. 문자열 연결 연산자를 이용하는 방법
// 숫자 타입 -> 문자열 타입
1 + '';                 // '1'
NaN + '';               // 'NaN'
Infinity + '';          // 'Infinity'
```

```
// 불리언 타입 -> 문자열 타입
true + '';              // 'true'
flase + '';             // 'false'
```

▼ 숫자 타입으로 변환 예시

1. `Number 생성자 함수`를 new 연산자 없이 호출하는 방법
2. `parseInt, parseFloat 함수`를 사용하는 방법(문자열만 숫자 타입으로 변환 가능)
3. `+ 단항 산술 연산자`를 이용하는 방법
4. `산술 연산자`를 이용하는 방법

```
// 1. Number 생성자 함수를 new 연산자 없이 호출하는 방법
// 문자열 타입 -> 숫자 타입
Number('0');        // 0
Number('-1');       // -1
Number('10.53');    // 10.53

// 불리언 타입 -> 숫자 타입
Number(true);       // 1
Number(false);      // 0

// 2. parseInt, parseFloat 함수를 사용하는 방법(문자열만 숫자 타입으로
// 문자열 타입 -> 숫자 타입
parseInt('0');      // 0
parseInt('-1');     // -1
parseInt('10.53'); // 10.53

// 3. + 단항 산술 연산자를 이용하는 방법
// 문자열 타입 -> 숫자 타입
+ '0';              // 0
+ '-1';             // -1
+ '10.53';          // 10.53

// 불리언 타입 -> 숫자 타입
+ true              // 1
+ false             // 0

4. * 산술 연산자를 이용하는 방법
```

```
// 문자열 타입 -> 숫자 타입
'0' * 1;            // 0
'-1' * 1;           // -1
'10.53' * 1;        // 10.53

// 불리언 타입 -> 숫자 타입
true * 1;           // 1
false * 1;          // 0
```

▼ 불리언 타입으로 변환 예시

1. `Boolean 생성자 함수`를 new 연산자 없이 호출하는 방법
2. `!부정 논리 연산자`를 두번 사용하는 방법

```
// 1. Boolean 생성자 함수를 new 연산자 없이 호출하는 방법
//모타입 -> 불리언 타입
Boolean('X');       // true
Boolean('');        // flase
Boolean('false');   // true

// 숫자 타입 -> 불리언 타입
Boolean(0);         // flase
Boolean(1);         // true
Boolean(NaN);       // flase
Boolean(Infinity);  // true

// null 타입 -> 불리언 타입
Boolean(NaN);       // false

// undefined 타입 -> 불리언 타입
Boolean(undefined); // false

// 객체 타입 -> 불리언 타입
Boolean({});        // true
Boolean([]);        // true

// 2. !부정 논리 연산자를 두번 사용하는 방법
// 문자열 타입 -> 불리언 타입
```

```
!!'X';              // true
!!'';               // flase
!!'false';          // true

// 숫자 타입 -> 불리언 타입
!!0;                // flase
!!1;                // true
!!NaN;              // flase
!!Infinity;         // true

// null 타입 -> 불리언 타입
!!NaN;              // false

// undefined 타입 -> 불리언 타입
!!undefined;        // false

// 객체 타입 -> 불리언 타입
!!{};               // true
!![];               // true
```

9-4 단축 평가

▼ 논리 연산자를 사용한 단축 평가

- 논리 연산의 결과를 결정하는 피연산자를 타입 변환하지 않고 그대로 반환

```
'Cat' && 'Dog' // 'Dog'
```

- 단축 평가 사용이 유용할 때
  - 객체를 가리키기를 기대하는 변수가 null 또는 undefined가 아닌지 확인하고 프로퍼티를 참조할 때

```
var elem = null;
// elem이 null이나 undefined와 같이 Flasy 값이면 elem으로 평
// elem이 Truthy 값이면 elem.value 값으로 평가됩니다.
var value = elem && elem.value; // null
```

▼ object.value

`Object.values()` 메소드는 전달된 파라미터 객체가 가지는 (열거 가능한) 속성의 값들로 이루어진 배열을 리턴한다.

- 함수 매개변수에 기본값을 설정할 때

```
// 단축 평가를 사용한 매개변수의 기본값 설정
function getStringLength(str) {
  str = str || '';
  return str.length;
}

getStringLength();     // 0
getStringLength('Hi'); // 2

// ES6의 매개변수의 기본값 설정
function getStringLength(str = '') {
  return str.length;
}

getStringLength();     // 0
getStringLength('Hi'); // 2
```

▼ 옵셔널 체이닝 연산자

- ES11(ECMAScript2020)에서 도입 후

`옵셔널 체이닝(optional chaining)` 연산자 ?.는 좌항의 피연산자가 null 또는 undefined인 경우 undefined를 반환하고, 그렇지 않으면 우항의 프로퍼티 참조를 이어간다.

- 옵셔널 체이닝을 사용함으로써 객체의 특정 속성이 존재하는지 매번 확인할 필요 없이 안전하게 접근할 수 있습니다. 즉 에러없이 undefined를 반환하게 함으로써 안전하게 접근 가능하다.

```
var elem = null;

// elem이 null 또는 undefined이면 undefined를 반환하고, 그렇지 않
var value = elem?.value;
console.log(value); // undefined
```

```
//옵셔널 체이닝x
var value2=elem.value;
console.log(var2);//런타임 에러 발생
```

- ES11(ECMAScript2020)에서 도입 전

  `논리 연산자&&` 를 사용한 단축 평가를 이용하여 null또는 undefined인지 확인했다.

```
var elem=null;
var value=elem&&elem.value;
console.log(value);
```

그런데 좌항 피연산자가 false로 평가되는 Falsy값이면 좌항 피연산자를 그대로 반환한다.

```
var str = '';

// 문자열 길이(length)를 참조합니다.
var length = str && str.length;

// 문자열 길이(length)를 참조하지 못합니다.
console.log(str); // ''
```

그렇기 때문에 `null`, `undefined`가 아닌지 확인하고 프로퍼티를 참조할 때는 `옵셔널 체이닝 연산자`를 사용하는 것이 좋습니다.

▼ null 병합 연산자

ES11(ECMAScript2020)에서 도입된 `null 병합 연산자` **??는 좌항의 피연산자가 `null` 또는 `undefined`인 경우 우항의 피연산자를 반환하고, 그렇지 않으면 좌항의 피연산자를 반환합니다.**

```
// 좌항의 피연산자가 null 또는 undefined이면 우항의 피연산자를 반
var foo = null ? 'default string';
console.log(foo); // 'default string'
```

null 병합 연산자 ??가 도입되기 이전에는 논리 연산자 ||를 사용한 단축 평가를 통해 변수에 기본값을 설정했습니다. 논리 연산자 ||를 사용한 단축 평가의 경우 좌항

의 피연산자가 Flasy 속성이면 우항의 피연산자를 반환하게 되는데, 이때 Falsy 값인 0또는 ''도 기본값으로서 유효하다면 예기치 않은 동작이 발생할 수 있습니다.

```
// 좌항의 피연산자가 Falsy 속성이 아니면 좌항의 피연산자를 그대로
var foo = '' || 'default string';
console.log(foo); // 'default string'
```

하지만 null 병합 연산자 ??를 사용하면 좌항의 피연산자가 Falsy 속성이라도 null 또는 undefined가 아니라면 좌항의 피연산자를 그대로 반환합니다.

```
// 좌항의 피연산자가 null, undefined가 아니면 좌항의 피연산자를
var foo = '' ?? 'default string';
console.log(foo); // ''
```

정리)

null 병합 연산자는 좌항이 null 또는 undefined이면 우항을 반환.

논리연산자 단축 평가를 이용하면 똑같이 좌항이 null,undefiend이면 우항을 반환하는데 좌항이 Falsy여도 우항을 반환

그러나null 병합 연산자는 Falsy값이라도 null, undefined이 아니니깐 좌항을 반환한다.