타입 변환과 단축 평가

타입 변환이란?

자바스크립트의 모든 값은 타입이 있다. 값의 타입은 다른 타입으로 개발자에 의해 의도적으로 변환할 수 있다. 또는 자바스크립트 엔진에 의해 암묵적으로 자동 변환될 수 있다. 개발자에 의해 의도적으로 값의 타입을 변환하는 것을 **명시적 타입 변환(Explicit coercion) 또는 타입 캐스팅(Type casting)**이라 한다.

```
var x = 10;

// 명시적 타입 변환
var str = x.toString(); // 숫자를 문자열로 타입 캐스팅한다.
console.log(typeof str); // string
```

동적 타입 언어인 자바스크립트는 개발자의 의도와는 상관없이 자바스크립트 엔진에 의해 암묵적으로 타입이 자동 변환되기도 한다. 이를 **암묵적 타입 변환(Implicit coercion) 또는 타입 강제 변환(Type coercion)**이라고 한다.

```
var x = 10;

// 암묵적 타입 변환
// 숫자 타입 x의 값을 바탕으로 새로운 문자열 타입의 값을 생성해 표현식을 평
var str = x + '';

console.log(typeof str, str); // string 10

// 변수 x의 값이 변경된 것은 아니다.
console.log(x); // 10
```

명시적 타입 변환도 마찬가지지만, 암묵적 타입 변환이 기존 값(위 예제의 경우, 변수 x의 값)을 직접 변경하는 것은 아니다. “변수”에서 살펴보았듯이 변수 값이 원시 타입의 값인 경우, 변수 값을 변경하려면 재할당을 통해 새로운 메모리 공간을 확보하고 그 곳에 원시 값을 저장한 후 변수명이 재할당된 원시 값이 저장된 메모리 공간의 주소를 기억하도록 해야 한다.

암묵적 타입 변환은 변수 값을 재할당해서 변경하는 것이 아니라 자바스크립트 엔진이 표현식을 에러없이 평가하기 위해 기존 값을 바탕으로 새로운 타입의 값을 만들어 단 한번 사용하고 버린다. 위 예제의 경우, 자바스크립트 엔진은 표현식 x + ‘’을 평가하기 위해 변수 x의 숫자 값을 바탕으로 새로운 문자열 값 ‘10’을 생성하고 이것으로 표현식 ‘10’ + ‘’를 평가한다. 이때 자동 생성된 문자열 ‘10’은 표현식의 평가가 끝나면 아무도 참조하지 않으므로 가비지 컬렉터에 의해 메모리에서 제거된다.

명시적 타입 변환은 타입을 변경하겠다는 개발자의 의지가 코드에 명백히 드러난다. 하지만 암묵적 타입 강제 변환은 자바스크립트 엔진에 의해 암묵적으로, 즉 드러나지 않게 타입이 자동 변환되기 때문에 타입을 변경하겠다는 개발자의 의지가 코드에 명백히 나타나지 않는다.

따라서 자신이 작성한 코드에서 암묵적 타입 변환이 발생하는지, 발생한다면 어떤 타입의 어떤 값으로 변환되는지, 그리고 타입 변환된 값으로 표현식은 어떻게 평가될 것인지 예측 가능해야 한다. 만약 예측하지 못하거나 예측한 내용이 결과와 일치하지 않는다면 버그를 생산할 가능성이 높아진다.

그렇다면 명시적 타입 변환 만을 사용하고 암묵적 타입 변환은 발생하지 않도록 코드를 작성하면 어떨까? 좋은 생각이긴 하지만 이러한 논리는 옳지 않다. 때로는 명시적 타입 변환보다 암묵적 타입 변환이 가독성 면에서 더 좋을 수도 있다. 예를 들어 자바스크립트 문법을 잘 이해하고 있는 개발자에게는 `(10).toString()`보다 `10 + ''`이 더욱 간결하고 이해하기 쉬울 수 있다.

중요한 것은 코드를 예측할 수 있어야 한다는 것이다. 동료가 작성한 코드를 정확히 이해할 수 있어야 하고 자신의 코드는 타인에 의해 쉽게 이해될 수 있어야 한다. 이를 위해 타입 변환이 어떻게 동작하는지 정확히 이해하고 사용하도록 하자.

암묵적 타입 변환

자바스크립트 엔진은 표현식을 평가할 때 문맥, 즉 컨텍스트(Context)에 고려하여 암묵적 타입 변환을 실행한다. 아래 예제를 살펴보자.

```
// 표현식이 모두 문자열 타입이여야 하는 컨텍스트
'10' + 2                // '102'
`1 * 10 = ${ 1 * 10 }` // "1 * 10 = 10"

// 표현식이 모두 숫자 타입이여야 하는 컨텍스트
5 * '10' // 50

// 표현식이 불리언 타입이여야 하는 컨텍스트
!0 // true
if (1) { }
```

이처럼 표현식을 평가할 때 문맥, 즉 컨텍스트에 부합하지 않는 다양한 상황이 발생할 수 있다. 이때 자바스크립트는 가급적 에러를 발생시키지 않도록 암묵적 타입 변환을 통해 표현식을 평가한다.

암묵적 타입 변환이 발생하면 문자열, 숫자, 불리언과 같은 원시 타입 중 하나로 타입을 자동 변환한다. 타입별로 암묵적 타입 변환이 어떻게 발생하는지 살펴보자.

문자열 타입으로 변환

아래의 예제를 살펴보자.

```
1 + '2' // "12"
```

위 예제의 `+` 연산자는 피연산자 중 하나 이상이 문자열이므로 문자열 연결 연산자로 동작한다. 문자열 연결 연산자의 역할은 문자열 값을 만드는 것이다. 따라서 문자열 연결 연산자의 피연산자는 문맥, 즉 컨텍스트 상 문자열 타입이여야 한다.

자바스크립트 엔진은 문자열 연결 연산자 표현식을 평가하기 위해 문자열 연결 연산자의 피연산자 중에서 문자열 타입이 아닌 피연산자를 문자열 타입으로 암묵적 타입 변환한다.

연산자 식의 피연산자(피연산자도 표현식이다) 만이 암묵적 타입 변환의 대상이 되는 것은 아니다. 앞서 언급했듯이 자바스크립트 엔진은 표현식을 평가할 때 문맥, 즉 컨텍스트에 부합하도록 암묵적 타입 변환을 실행한다. 예를 들어 ES6에서 도입된 템플릿 리터럴의 문자열 인터폴레이션(String Interpolation)은 표현식의 평가 결과를 문자열 타입으로 암묵적 타입 변환한다.

```
console.log(`1 + 1 = ${1 + 1}`); // "1 + 1 = 2"
```

자바스크립트 엔진은 문자열 타입 아닌 값을 문자열 타입으로 암묵적 타입 변환을 수행할 때 아래와 같이 동작한다.

```
// 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"
// 심볼 타입
(Symbol()) + ''     // TypeError: Cannot convert a Symbol val
// 객체 타입
```

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

숫자 타입으로 변환

아래의 예제를 살펴보자.

```
1 - '1'    // 0
1 * '10'   // 10
1 / 'one'  // NaN
```

위 예제의 연산자는 모두 산술 연산자이다. 산술 연산자의 역할은 숫자 값을 만드는 것이다. 따라서 산술 연산자의 피연산자는 문맥, 즉 컨텍스트 상 숫자 타입이여야 한다.

자바스크립트 엔진은 산술 연산자 표현식을 평가하기 위해 산술 연산자의 피연산자 중에서 숫자 타입이 아닌 피연산자를 숫자 타입으로 암묵적 타입 변환한다. 이때 피연산자를 숫자 타입으로 변환할 수 없는 경우는 산술 연산을 수행할 수 없으므로 NaN을 반환한다.

피연산자를 숫자 타입으로 변환해야 할 컨텍스트는 산술 연산자 뿐만이 아니다. 아래 예제를 살펴보자.

```
'1' > 0   // true
```

비교 연산자의 역할은 불리언 값을 만드는 것이다. > 비교 연산자는 피연산자의 크기를 비교하므로 피연산자는 컨텍스트 상 숫자 타입이여야 한다. 자바스크립트 엔진은 비교 연산자 표현식을 평가하기 위해 비교 연산자의 피연산자 중에서 숫자 타입이 아닌 피연산자를 숫자 타입으로 암묵적 타입 변환한다.

자바스크립트 엔진은 숫자 타입 아닌 값을 숫자 타입으로 암묵적 타입 변환을 수행할 때 아래와 같이 동작한다. + 단항 연산자는 피연산자가 숫자 타입의 값이 아니면 숫자 타입의 값으로 암묵적 타입 변환을 수행한다.

```
// 문자열 타입
+''             // 0
+'0'            // 0
+'1'            // 1
+'string'       // NaN
// 불리언 타입
+true           // 1
+false          // 0
// null 타입
+null           // 0
// undefined 타입
+undefined      // NaN
// 심볼 타입
+Symbol()       // TypeError: Cannot convert a Symbol value t
// 객체 타입
+{}             // NaN
+[]             // 0
+[10, 20]       // NaN
+(function(){}) // NaN
```

빈 문자열(''), 빈 배열([]), null, false는 0으로, true는 1로 변환된다. 객체와 빈 배열이 아닌 배열, undefined는 변환되지 않아 NaN이 된다는 것에 주의하자.

불리언 타입으로 변환

아래의 예제를 살펴보자.

```
if ('') console.log(x);
```

if 문이나 for 문과 같은 제어문의 조건식(conditional expression)은 불리언 값, 즉 논리적 참, 거짓을 반환해야 하는 표현식이다. 자바스크립트 엔진은 제어문의 조건식을 평가 결과를 불리언 타입으로 암묵적 타입 변환한다.

```
if ('')    console.log('1');
if (true)  console.log('2');
if (0)     console.log('3');
if ('str') console.log('4');
if (null)  console.log('5');

// 2 4
```

이때 자바스크립트 엔진은 불리언 타입이 아닌 값을 **Truthy 값(참으로 인식할 값) 또는 Falsy 값(거짓으로 인식할 값)**으로 구분한다. 즉, Truthy 값은 true로, Falsy 값은 false로 변환된다.

아래 값들은 제어문의 조건식과 같이 불리언 값으로 평가되어야 할 컨텍스트에서 false로 평가되는 False 값이다.

- false
- undefined
- null
- 0, -0
- NaN
- '' (빈문자열)

```
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');
```

Falsy 값 이외의 값은 제어문의 조건식과 같이 불리언 값으로 평가되어야 할 컨텍스트에서 모두 true로 평가되는 Truthy 값이다.

아래 예제는 Truthy/Falsy 값을 판별하는 함수다. 함수란 어떤 작업을 수행하기 위해 필요한 문들의 집합을 정의한 코드 블록이다. 함수는 이름과 매개변수를 갖으며 필요한 때에 호출하여 코드 블록에 담긴 문들을 일괄적으로 실행할 수 있다. 함수에 대해서는 나중에 자세히 살펴볼 것이다.

```
// 주어진 인자가 Falsy 값이면 true, Truthy 값이면 false를 반환한다.
function isFalsy(v) {
  return !v;
}

// 주어진 인자가 Truthy 값이면 true, Falsy 값이면 false를 반환한다.
function isTruthy(v) {
  return !!v;
}

// 모두 true를 반환한다.
console.log(isFalsy(false));
console.log(isFalsy(undefined));
console.log(isFalsy(null));
console.log(isFalsy(0));
console.log(isFalsy(NaN));
console.log(isFalsy(''));

console.log(isTruthy(true));
// 빈 문자열이 아닌 문자열은 Truthy 값이다.
console.log(isTruthy('0'));
console.log(isTruthy({}));
console.log(isTruthy([]));
```

명시적 타입 변환

개발자의 의도에 의해 명시적으로 타입을 변경하는 방법은 다양하다. 원래는 래퍼 객체를 생성하기 위해 사용하는 래퍼 객체 생성자 함수를 new 연산자 없이 호출하는 방법과 자바스크립트에서 제공하는 빌트인 메소드를 사용하는 방법, 그리고 앞에서 살펴본 암묵적 타입 변환을 이용하는 방법이 있다.

문자열 타입으로 변환

문자열 타입이 아닌 값을 문자열 타입으로 변환하는 방법은 아래와 같다.

1. String 생성자 함수를 new 연산자 없이 호출하는 방법
2. Object.prototype.toString 메소드를 사용하는 방법
3. 문자열 연결 연산자를 이용하는 방법

```
// 1. String 생성자 함수를 new 연산자 없이 호출하는 방법
// 숫자 타입 => 문자열 타입
console.log(String(1));        // "1"
console.log(String(NaN));      // "NaN"
console.log(String(Infinity)); // "Infinity"
// 불리언 타입 => 문자열 타입
console.log(String(true));     // "true"
console.log(String(false));    // "false"

// 2. Object.prototype.toString 메소드를 사용하는 방법
// 숫자 타입 => 문자열 타입
console.log((1).toString());        // "1"
console.log((NaN).toString());      // "NaN"
console.log((Infinity).toString()); // "Infinity"
// 불리언 타입 => 문자열 타입
console.log((true).toString());     // "true"
console.log((false).toString());    // "false"

// 3. 문자열 연결 연산자를 이용하는 방법
// 숫자 타입 => 문자열 타입
```

```
console.log(1 + '');         // "1"
console.log(NaN + '');       // "NaN"
console.log(Infinity + ''); // "Infinity"
// 불리언 타입 => 문자열 타입
console.log(true + '');      // "true"
console.log(false + '');     // "false"
```

숫자 타입으로 변환

숫자 타입이 아닌 값을 숫자 타입으로 변환하는 방법은 아래와 같다.

1. Number 생성자 함수를 new 연산자 없이 호출하는 방법

2. parseInt, parseFloat 함수를 사용하는 방법(문자열만 변환 가능)

3. 단항 연결 연산자를 이용하는 방법

4. 산술 연산자를 이용하는 방법

```
// 1. Number 생성자 함수를 new 연산자 없이 호출하는 방법
// 문자열 타입 => 숫자 타입
console.log(Number('0'));     // 0
console.log(Number('-1'));    // -1
console.log(Number('10.53')); // 10.53
// 불리언 타입 => 숫자 타입
console.log(Number(true));    // 1
console.log(Number(false));   // 0

// 2. parseInt, parseFloat 함수를 사용하는 방법(문자열만 변환 가능)
// 문자열 타입 => 숫자 타입
console.log(parseInt('0'));       // 0
console.log(parseInt('-1'));      // -1
console.log(parseFloat('10.53')); // 10.53

// 3. + 단항 연결 연산자를 이용하는 방법
// 문자열 타입 => 숫자 타입
console.log(+'0');     // 0
console.log(+'-1');    // -1
console.log(+'10.53'); // 10.53
```

```
// 불리언 타입 => 숫자 타입
console.log(+true);    // 1
console.log(+false);   // 0

// 4. * 산술 연산자를 이용하는 방법
// 문자열 타입 => 숫자 타입
console.log('0' * 1);     // 0
console.log('-1' * 1);    // -1
console.log('10.53' * 1); // 10.53
// 불리언 타입 => 숫자 타입
console.log(true * 1);    // 1
console.log(false * 1);   // 0
```

불리언 타입으로 변환

불리언 타입이 아닌 값을 불리언 타입으로 변환하는 방법은 아래와 같다.

1. Boolean 생성자 함수를 new 연산자 없이 호출하는 방법

2. ! 부정 논리 연산자를 두번 사용하는 방법

```
// 1. Boolean 생성자 함수를 new 연산자 없이 호출하는 방법
// 문자열 타입 => 불리언 타입
console.log(Boolean('x'));       // true
console.log(Boolean(''));        // false
console.log(Boolean('false'));   // true
// 숫자 타입 => 불리언 타입
console.log(Boolean(0));         // false
console.log(Boolean(1));         // true
console.log(Boolean(NaN));       // false
console.log(Boolean(Infinity));  // true
// null 타입 => 불리언 타입
console.log(Boolean(null));      // false
// undefined 타입 => 불리언 타 입
console.log(Boolean(undefined)); // false
// 객체 타입 => 불리언 타입
```

```
console.log(Boolean({}));         // true
console.log(Boolean([]));         // true

// 2. ! 부정 논리 연산자를 두번 사용하는 방법
// 문자열 타입 => 불리언 타입
console.log(!!'x');       // true
console.log(!!'');        // false
console.log(!!'false');   // true
// 숫자 타입 => 불리언 타입
console.log(!!0);         // false
console.log(!!1);         // true
console.log(!!NaN);       // false
console.log(!!Infinity);  // true
// null 타입 => 불리언 타입
console.log(!!null);      // false
// undefined 타입 => 불리언 타입
console.log(!!undefined); // false
// 객체 타입 => 불리언 타입
console.log(!!{});        // true
console.log(!![]);        // true
```

단축 평가

아래 예제를 살펴보자.

```
'Cat' && 'Dog' // “Dog”
```

논리곱 연산자 `&&`는 두개의 피연산자가 모두 `true`로 평가될 때 `true`를 반환한다. 대부분의 연산자가 그렇듯이 논리곱 연산자도 오른쪽에서 왼쪽으로 평가가 진행된다.

1. 첫번째 피연산자 ‘Cat’은 Truthy 값이므로 `true`로 평가된다. 하지만 이 시점까지는 위 표현식을 평가할 수 없다. 두번째 피연산자까지 평가해 보아야 위 표현식을 평가할 수 있다.

2. 두번째 피연산자 'Dog'은 Truthy 값이므로 `true`로 평가된다. 이때 두개의 피연산자가 모두 `true`로 평가되었다. 이때 논리곱 연산의 결과를 결정한 것은 두번째 피연산자 'Dog'다.

3. 논리곱 연산자 `&&`는 **논리 연산의 결과를 결정한 두번째 피연산자의 평가 결과 즉, 문자열 'Dog'를 그대로 반환한다.**

논리합 연산자 `||`도 논리곱 연산자 `&&`와 동일하게 동작한다.

```
'Cat' || 'Dog' // 'Cat'
```

논리합 연산자 `||`는 두개의 피연산자 중 하나만 `true`로 평가되어도 `true`를 반환한다. 대부분의 연산자가 그렇듯이 논리합 연산자도 오른쪽에서 왼쪽으로 평가가 진행된다.

1. 첫번째 피연산자 'Cat'은 Truthy 값이므로 `true`로 평가된다. 이 시점에 두번째 피연산자까지 평가해 보지 않아도 위 표현식을 평가할 수 있다.

2. 논리합 연산자 `||`는 **논리 연산의 결과를 결정한 첫번째 피연산자의 평가 결과 즉, 문자열 'Cat'를 그대로 반환한다.**

논리곱 연산자 `&&`와 논리합 연산자 `||`는 이와 같이 **논리 평가를 결정한 피연산자의 평가 결과를 그대로 반환한다.** 이를 **단축 평가(Short-Circuit evaluation)**라 부른다. 단축 평가는 아래의 규칙을 따른다.

단축 평가 표현식	평가 결과
true || anything	true
false || anything	anything
true && anything	anything
false && anything	false

```
// 논리합(||) 연산자
'Cat' || 'Dog'  // 'Cat'
false || 'Dog'  // 'Dog'
```

```
'Cat' || false  // 'Cat'

// 논리곱(&&) 연산자
'Cat' && 'Dog'  // Dog
false && 'Dog'  // false
'Cat' && false  // false
```

단축 평가는 아래와 같은 상황에서 유용하게 사용된다.

아직 살펴보지 않은 객체와 함수에 대한 내용이 나와서 혼란스러울 수 있겠다. 지금은 아래와 같은 단축 평가의 유용한 패턴이 있다는 정도로 이해하고 넘어가도 좋다. 객체와 함수에 대해서는 해당 장에서 자세히 살펴볼 것이다.

- 객체가 null인지 확인하고 프로퍼티를 참조할 때

```
var elem = null;

console.log(elem.value); // TypeError: Cannot read property '
console.log(elem && elem.value); // null
```

객체는 키(key)과 값(value)으로 구성된 프로퍼티(Property)들의 집합이다. 만약 객체가 null인 경우, 객체의 프로퍼티를 참조하면 타입 에러(TypeError)가 발생한다. 이때 단축 평가를 사용하면 에러를 발생시키지 않는다.

- 함수의 인수(argument)를 초기화할 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
```

함수를 호출할 때 인수를 전달하지 않으면 매개변수는 undefined를 갖는다. 이때 단축 평가를 사용하여 매개변수의 기본값을 설정하면 undefined로 인해 발생할 수 있는 에러를 방지할 수 있다.