Chapter 86: Tilde ~

The ~ operator looks at the binary representation of the values of the expression and does a bitwise negation operation on it.

Any digit that is a 1 in the expression becomes a 0 in the result. Any digit that is a 0 in the expression becomes a 1 in the result.

Section 86.1: ~ Integer

The following example illustrates use of the bitwise NOT (~) operator on integer numbers.

```
let number = 3;
let complement = ~number;
```

Result of the `complement` number equals to -4;

Expression	Binary value	Decimal value
3	`00000000 00000000 00000000 00000011`	3
~3	`11111111 11111111 11111111 11111100`	-4

To simplify this, we can think of it as function `f(n) = -(n+1)`.

```
let a = ~-2; // a is now 1
let b = ~-1; // b is now 0
let c = ~0;  // c is now -1
let d = ~1;  // d is now -2
let e = ~2;  // e is now -3
```

Section 86.2: ~~ Operator

Double Tilde `~~` will perform bitwise NOT operation twice.

The following example illustrates use of the bitwise NOT (~~) operator on decimal numbers.

To keep the example simple, decimal number `3.5` will be used, cause of it's simple representation in binary format.

```
let number = 3.5;
let complement = ~number;
```

Result of the `complement` number equals to -4;

Expression	Binary value	Decimal value
3	`00000000 00000000 00000000 00000011`	3
~~3	`00000000 00000000 00000000 00000011`	3
3.5	`00000000 00000011.1`	3.5
~~3.5	`00000000 00000011`	3

To simplify this, we can think of it as functions `f2(n) = -(-(n+1) + 1)` and `g2(n) = -(-(integer(n)+1) + 1)`.

f2(n) will leave the integer number as it is.

```
let a = ~~-2; // a is now -2
let b = ~~-1; // b is now -1
let c = ~~0;  // c is now 0
let d = ~~1;  // d is now 1
let e = ~~2;  // e is now 2
```

g2(n) will essentially round positive numbers down and negative numbers up.

```
let a = ~~-2.5; // a is now -2
let b = ~~-1.5; // b is now -1
let c = ~~0.5;  // c is now 0
let d = ~~1.5;  // d is now 1
let e = ~~2.5;  // e is now 2
```

Section 86.3: Converting Non-numeric values to Numbers

~~ Could be used on non-numeric values. A numeric expression will be first converted to a number and then performed bitwise NOT operation on it.

If expression cannot be converted to numeric value, it will convert to `0`.

`true` and `false` bool values are exceptions, where `true` is presented as numeric value `1` and `false` as `0`

```
let a = ~~"-2";     // a is now -2
let b = ~~"1";      // b is now -1
let c = ~~"0";      // c is now 0
let d = ~~"true";   // d is now 0
let e = ~~"false";  // e is now 0
let f = ~~true;     // f is now 1
let g = ~~false;    // g is now 0
let h = ~~"";       // h is now 0
```

Section 86.4: Shorthands

We can use `~` as a shorthand in some everyday scenarios.

We know that `~` converts `-1` to `0`, so we can use it with `indexOf` on array.

indexOf

```
let items = ['foo', 'bar', 'baz'];
let el = 'a';
```

```
if (items.indexOf('a') !== -1) {}

or

if (items.indexOf('a') >= 0) {}
```

can be re-written as

```
if (~items.indexOf('a')) {}
```

Section 86.5: ~ Decimal

The following example illustrates use of the bitwise NOT (~) operator on decimal numbers.

To keep the example simple, decimal number `3.5` will be used, cause of it's simple representation in binary format.

```
let number = 3.5;
let complement = ~number;
```

Result of the `complement` number equals to -4;

Expression	Binary value	Decimal value
3.5	`00000000 00000010.1`	3.5
~3.5	`11111111 11111100`	-4

To simplify this, we can think of it as function `f(n) = -(integer(n)+1)`.

```
let a = ~-2.5; // a is now 1
let b = ~-1.5; // b is now 0
let c = ~0.5;  // c is now -1
let d = ~1.5;  // c is now -2
let e = ~2.5;  // c is now -3
```