L1) Coercion → Type Conversion

$2 + 3 = 5$

$2 + "3" = 23$

$"1" - 1 = 0$

L2) Definition of Coercion :

Whenever we do a operation, based on the i/p we can actually convert the i/p for operation

↓

we can convert type of input

Two types of casting :

1) Explicit : This conversion can be manually done by us
2) Implicit : Automatically converts the type.
   ↳ coercion

L3) Abstract Operation : These operations are NOT a part of the ECMA Script language, they are defined here to solely to aid the specifications of the semantics of the ECMA Script.

We as developer cannot use these operation directly.

(5) ToNumber Abstract Operation :

To Number converts argument to a value of type Number.

The abstract operation ToNumber converts *argument* to a value of type Number according to Table 10:

Table 10: ToNumber Conversions

Argument Type	Result
Undefined	Return **NaN**.
Null	Return **+0**.
Boolean	If *argument* is **true**, return 1. If *argument* is **false**, return **+0**.
Number	Return *argument* (no conversion).
String	See grammar and conversion algorithm below.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ? ToPrimitive(*argument*, hint Number). 2. Return ? ToNumber(*primValue*).

10 - null = 10 - 0 = 10

10 - undefined = 10 - NAN = NAN

(6) ToNumber on String :

ToNumber applied to Strings applies the following grammar to the input String interpreted as a sequence of UTF-16 encoded code points (6.1.4). If the grammar cannot interpret the String as an expansion of *StringNumericLiteral*, then the result of ToNumber is **NaN**.

```
// toNumber on strings

console.log(1 - "453"); // -452
console.log(1 - "43rizon"); // 1 - NaN => NaN
```

Assignmnt

```
// Assignment why its -9?
console.log(1 - "0xa"); // -9
```

1 - "0xa"

↳ hexadecimal ⇒ 10 decimal

⇒ 1 - 10

⇒ -9

(7)

12.8.4.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* - *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lnum* be ? ToNumber(*lval*).
6. Let *rnum* be ? ToNumber(*rval*).
7. Return the result of applying the subtraction operation to *lnum* and *rnum*. See the note below 12.8.5.

(8)

7.1.1 ToPrimitive (*input* [, *PreferredType*])

The abstract operation ToPrimitive takes an *input* argument and an optional argument *PreferredType*. The abstract operation ToPrimitive converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to the following algorithm:

1. Assert: *input* is an ECMAScript language value.
2. If Type(*input*) is Object, then
   a. If *PreferredType* is not present, let *hint* be `"default"`.
   b. Else if *PreferredType* is hint String, let *hint* be `"string"`.
   c. Else *PreferredType* is hint Number, let *hint* be `"number"`.
   d. Let *exoticToPrim* be ? GetMethod(*input*, @@toPrimitive).
   e. If *exoticToPrim* is not **undefined**, then
      i. Let *result* be ? Call(*exoticToPrim*, *input*, « *hint* »).
      ii. If Type(*result*) is not Object, return *result*.
      iii. Throw a **TypeError** exception.
   f. If *hint* is `"default"`, set *hint* to `"number"`.
   g. Return ? OrdinaryToPrimitive(*input*, *hint*).
3. Return *input*.

-X-

L9) To Primitive (input, Preferred Type)

↳ optional

↳ Convert the input to non-object type.

<u>preferred Type</u> : In case where i/p can be converted to multiple type we take the decision using this.

<u>To Primitive Cases</u> :

Case 1 : Preferred Type is NOT present

hint ← "default"

Case 2 : Preferred Type is a string

hint ← "string"

Case 3 : Preferred Type is a number

hint ← "number"

default → number

Ordinary To Primitive :

Ordinary To Primitive (input, hint)

7.1.1.1 OrdinaryToPrimitive (*O*, *hint*)

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

1. Assert: Type(*O*) is Object.
2. Assert: Type(*hint*) is String and its value is either **"string"** or **"number"**.
3. If *hint* is **"string"**, then
   a. Let *methodNames* be « **"toString"**, **"valueOf"** ».
4. Else,
   a. Let *methodNames* be « **"valueOf"**, **"toString"** ».
5. For each *name* in *methodNames* in List order, do
   a. Let *method* be ? Get(*O*, *name*).
   b. If IsCallable(*method*) is **true**, then
      i. Let *result* be ? Call(*method*, *O*).
      ii. If Type(*result*) is not Object, return *result*.
6. Throw a **TypeError** exception.

hint

"string"

- ["toString", "valueOf"]

↳ method Name

"number"

- ["valueOf", "toString"]

↳ method Name

-

```
let obj = {
    toString() {
        // by default gives [Object object]
        return "Let's learn JAVASCRIPT"; // still you can override
    },

    valueOf() {
        // by default it returns the same object

    }
}
```

Coercion in Addition Operator

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*). → No hint here
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   a. Let *lstr* be ? ToString(*lprim*).
   b. Let *rstr* be ? ToString(*rprim*).
   c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*.

+ "10" ⇒ 10 // Unary Operator

```
let obj = {};

console.log(obj); // {}

console.log("18" + obj); // 18[object Object]

console.log(18 + obj); // 18[object Object]
```

Intro to To Boolean

Argument Type	Result
Undefined	Return **false**.
Null	Return **false**.
Boolean	Return *argument*.
Number	If *argument* is **+0**, **-0**, or **NaN**, return **false**; otherwise return **true**.
String	If *argument* is the empty String (its length is zero), return **false**; otherwise return **true**.
Symbol	Return **true**.
Object	Return **true**.

[undefined, null, NaN, +0, -0, false] ⇒ they are falsy value

```
let x = 10;
console.log(!x); // false

let y = undefined;
console.log(!y); // true
```

Equality Operator :

== → abstract equality

=== → strict equality

Both abstract & strict equality check for type. but what they do with it depends.

Abstract Equality & Strict Equality:

Abstract Equality:

If type of (x) is same as type of (y)
⇒ then works as strict equality

If type is NOT same

x = null y = undefined
then return true

x = undefined y = null
then return true

if type of x = Number type of y = String
then converts y → toNumber
then again does the comparsion

if type of x = String type of y = number
then convert x → toNumber
again does the comparsion

if type of x = Boolean
then convert x → toNumber (0 → 1, 1 → 0)
then do comparsion

if type of y = Boolean
then convert y → to Number

if type of x is either String, Number, Symbol and type of y is Object
then convert y → toPrimitive then compare

if type of x is object and type of y is String, number, symbol
then convert x → toPrimitive then compare

If none of above rules follow then return false

05-First Pillar of JS > 06_abstractEqualityDemo.js

```javascript
console.log(null == undefined); // true
console.log(12 == "12"); // true
console.log(false == "0"); // true
/**
 *  x -> boolean -> toNumber -> false -> 0
 *  x = 0, y = 0, x == y
 *  y -> toNumber -> 0
 *  0 == 0 -> true
 */

console.log(null == false); // false (last condition)
```

```javascript
let obj = {
  x: 10,
  valueOf() {
    return 100;
  },
};

console.log(99 == obj); // false
console.log(100 == obj); // true
```

Strict Equality :

If typeof (x) is different from typeof (y)
then return false

If type(x) is Number then

If x is NAN, return false (don't care value of y)

If y is NAN, return false (don't care value of x)

05-First Pillar of JS > 07_strictEqualityDemo.js

```
console.log(NaN === NaN); // false
```

x → NaN then return false.

If x is same Number value as y
return true.
10 === 10 ⇒ True

If x = 0 and y = -0
return true

If x = -0 and y = 0
return true

Return false

If Same Value Non Number (x, y)

7.2.12 SameValueNonNumber (*x*, *y*)

The internal comparison abstract operation SameValueNonNumber(*x*, *y*), where neither *x* nor *y* are Number values, produces **true** or **false**. Such a comparison is performed as follows:

1. Assert: Type(*x*) is not Number.
2. Assert: Type(*x*) is the same as Type(*y*).
3. If Type(*x*) is Undefined, return **true**.
4. If Type(*x*) is Null, return **true**.
5. If Type(*x*) is String, then
   a. If *x* and *y* are exactly the same sequence of code units (same length and same code units at corresponding indices), return **true**; otherwise, return **false**.
6. If Type(*x*) is Boolean, then
   a. If *x* and *y* are both **true** or both **false**, return **true**; otherwise, return **false**.
7. If Type(*x*) is Symbol, then
   a. If *x* and *y* are both the same Symbol value, return **true**; otherwise, return **false**.
8. If *x* and *y* are the same Object value, return **true**. Otherwise, return **false**.

Coercion Special Cases

08_coercionCase.js 1, U

05-First Pillar of JS > 08_coercionCase.js

```js
// toString -> "" + value
console.log("" + 0);   // 0 -> "0"
console.log("" + (-0)); // -0 -> "0"

console.log("" + []); // [] -> ""
console.log("" + {}); // {} -> ""

console.log("" + [1, 2, 3, 4]); // 1, 2, 3
console.log("" + [null, undefined]); // ,
console.log("" + [1, 2, null, 4]); // 1,2,,4


// toNumber
console.log(0 - "010"); // -10
console.log(0 - "o10"); // NaN
console.log(0 - "O10"); // NaN

console.log(0 - 010); // -8 // octal number
console.log(0 - "0xb"); // -11 //hexadecimal number
console.log(0 - 0xb); // -11

console.log(0 - []); // 0
console.log([] - 1); // -1
console.log([""] - 1); // -1
```

05-First Pillar of JS > 09_coercionExamples.js

```
console.log(1 < 2 < 3);
// (1 < 2) = true -> (true < 3) -> (1 < 3) -> true
console.log(3 > 2 > 1);
// (3 > 2) = true -> (true > 1) -> (1 > 1) -> false
```

NaN : Not a Number :

11_NaNDemo.js U

05-First Pillar of JS > 11_NaNDemo.js > ...

```
console.log(Number("123")); // 123
console.log(Number("abcd")); // NaN
console.log(Number("0xa")); // 10

let x = NaN;

console.log(x == NaN); // false

console.log(isNaN(x)); // true

// isNaN converts the incoming input to a number
console.log(isNaN("Rizon")); // true

console.log(Number.isNaN("Rizon")); // false

console.log(Number.isNaN(x)); // true
```

Negative Zero :

12_negativeZero.js U

05-First Pillar of JS > 12_negativeZero.js > customSign

```
let x = -0;
console.log(x === 0); // true

console.log(Object.is(x, -0)); // T
console.log(Object.is(x, 0)); // F

console.log(Math.sign(-3)); // -1
console.log(Math.sign(2)); // 1
console.log(Math.sign(-0)); // -0
console.log(Math.sign(0)); // 0
```

```javascript
/**
 * Homework : Can we write a custom function that can give us
 *            sign of a number properly ?
 */

function customSign(num) {
  if (num === 0 || num === -0) {
    return Object.is(num, -0) ? -1 : 0;
  }

  return Math.sign(num);
}

// Test the customSign function
console.log(customSign(-3)); // -1
console.log(customSign(2)); // 1
console.log(customSign(-0)); // -1
console.log(customSign(0)); // 0
```

Boxing

```
> 1.toString();
Uncaught SyntaxError: Invalid or unexpected token
> "abc".toString();
<· 'abc'
⚠ ▶VueBackbone: Adding new Model attributes after binding is not supported, provi
> undefined.toString();
Uncaught TypeError: Cannot read properties of undefined (reading 'toString')
    at <anonymous>:1:11
> (undefined).toString();
Uncaught TypeError: Cannot read properties of undefined (reading 'toString')
    at <anonymous>:1:13
> (1).toString();
<· '1'
```

1, abc, undefined ⇒ primitive data type

but 1.toString ⇒ error

(1).toString ⇒ 1

Here comes the concept of Boxing concept.

Boxing: Your primitive J.S types get converted to non-primitive when you try to do boxing.