

Coercion

Every language has got some mechanism for

type interconversion. for example → in Java if we

do → `System.out.println("Hello" + 22);`

the above code will print → `Hello22` i.e. it did
convert 22 from number to String & then did concatenation.

Type Interconversion

implicit



when the language
automatically converts
types

explicit



when we manually
Command to convert
types

Similarly this concept exist in JS also.

for JS things are a bit more tricky as JS
handle wide variety of cases.

Coercion → Type later Conversion

Abstract Operations → These are operations/functions which are not available for end users to use.

But JS internally uses it & these are mentioned in the official docs to actually aid the documentation.

⇒ ToString

⇒ ToNumber

⇒ ToBoolean

⇒ ToPrimitive

etc.

we cannot directly call them.
But few JS operations like
'-' (subtraction) , '+' (addition)
etc internally calls them & hence
we can mimic them using these
operations.

ToNumber

We can use '-' operation to mimic ToNumber.

$a - b$
 \swarrow \searrow
lval rval

2 variables $\rightarrow a, b$

```
lnum = ToNumber (a);  
rnum = ToNumber (b);  
return lnum - rnum;
```

Subtraction always
converts both the
operands to a
Number.

"" → 0

"0" → 0

"-0" → -0

" 009 " → 9

"3.145" → 3.145

"0." → 0

".0" → 0

" ." → NaN

"\$ " → NaN

"0xaf" → 175

↓ this a valid
hexadecimal
number, So JS can
parse it

"ab32" → NaN

true → 1

false → 0

null → 0

undefined → NaN

$[" "] \rightarrow 0$

$["0"] \rightarrow 0$

$[" -0"] \rightarrow -0$

$[\text{null}] \rightarrow 0$

$[\text{undefined}] \rightarrow 0$

$[1, 2, 3] \rightarrow \text{NaN}$

$[[[]]] \rightarrow 0$

$[] \rightarrow 0$

To String

We can use '+' operator to mimic ToString

a + b
↓ ↓
lval rval

lprim = ToPrimitive (lval)

rprim = ToPrimitive (rval)

if (lprim is a String or rprim is a String)

we do string concatenation

lnum ⇒ ToNumber (lprim)

rnum ⇒ ToNumber (rprim)

return lnum + rnum.

null → "null"

undefined → "undefined"

true → "true"

false → "false"

3.145 → "3.145"

0 → "0"

~0 → "0"

[] → ""

[1,2,3] → "1,2,3"

[null, undefined] → ", "

[[],[],[]] → "..."

[...] → "..."

ToPrimitive

internally calls
OrdinaryToPrimitive
tries to

This function takes an input argument and converts it into a non object type (primitive type). If it

can't convert it can throw error.

It takes one more optional parameter called as

preferred type. If we have ^{2 or} more than 2 values that

can become ans, this optional argument helps us.

→ if preferred type is not given, hint = "default".

new variable use
initialize.

→ else if preferred type is "String", hint = "String"

→ else preferred type is "Number", hint = "Number"

if hint is "default" then hint = "Number"

hint → String

↓

noString()

↓

valueOf()

hint → Number

↓

valueOf()

↓

noString()

hint = String



toString()



result is
non object



return result



result is object



valueOf



result is
non object



return result



result is
object



throw error

hint = number



value Of

result is
non object



return result

result is object



toString

result is
non object



return result

result is
Object



throw error

valueOf and toString() are not abstract operations. That means we can call them.

By default toString() on an object returns
'[object Object]' ←

By default valueOf → on an object return same Object.

for array → value of → same array

toString → print array without brackets.

$\{x:10\}.\text{valueOf}() \rightarrow \text{same object}$

$\{x:10, \text{valueOf}() \{x:10\}.\text{valueOf}()\}$ \rightarrow 2

overriding valueOf