JS → Operators

add → 2 integer values → (2 + 3) → addition operator

↳ 12 (number) + 13 (number)

here because both the operands are numbers, + acts as arithmetic operator.

12 (number) + "13" (string)

↓ concatenation

"1213"

numbers

1 + 1 → 2

"1" + 1 → "11"

"1" − 1 → 0

"1" → string

− → minus

1 → number

{ x magic
x automatic
x wicked }

official docs

typecasty ⟷ type conversion Coercion → type conversion

whenever we do an operation, based on the input we can actually convert the input for operation.

↓

we can convert type of input.

↓

explicit typecasting ← this conversion can be manually done by us.

or

implicit typecasting ← the language based on some certain rules automatically converts the types.

also known as Coercion

Abstract Operations

1→ These are some set of algorithms, that is present in the ecmascript docs, but they are not available for usage in ecmascript
i.e. we as developers cannot use these operation directly.

2→ They are mentioned in the docs to aid (help) the documentation only.

In the ecma docs there are a lot of things that are done by the language internally. To explain these internal details of how & what lang is doing, we have abstract ops mentioned in the docs.

7 Abstract Operations

These operations are not a part of the ECMAScript language; they are defined here to solely to aid the specification of the semantics of the ECMAScript language. Other, more specialized abstract operations are defined throughout this specification.

7.1 Type Conversion

intenwell, automatically

The ECMAScript language implicitly performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operations. The conversion abstract operations are polymorphic; they can accept a value of any ECMAScript language type. But no other specification types are used with these operations.

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

To Number

9:41 AM Tue 9 Jan — 262.ecma-international.org — Private — 100%

7.1.3 ToNumber (*argument*)

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*).

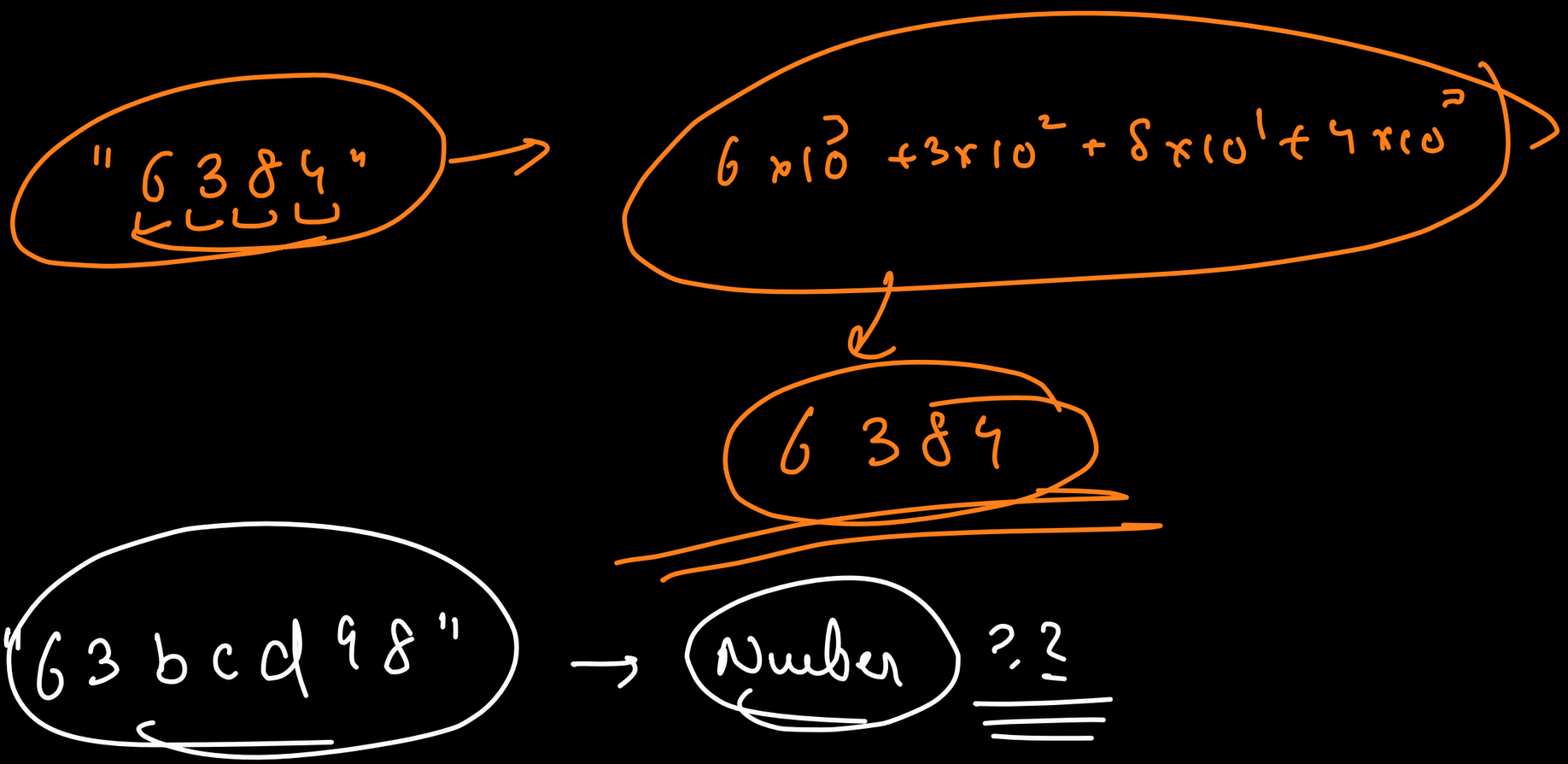
a b c

10 - null
↳ 10 - 0
↳ 10

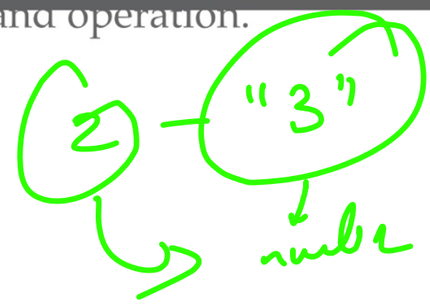
10 - undefined
↳ 10 - NaN
↳ NaN

10 - true
↳ 10 - 1
↳ 9

→ why only these rules ??



using the logical-or operation instead of the logical-and operation.



12.8.4 The Subtraction Operator (–)

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.

left operand → get the value of left operand & store it inside lval

right operand → get the value of right oprn & store it in rval

→ JS converts left operand to a num & store in lnum

12.8.5 Applying the Additive Operators to Numbers

The + operator performs addition when applied to two operands of numeric type, produci the sum of the operands. The – operator performs subtraction, producing the difference of