Introduction to Coercion

Whenever we do operation, based on the operands and operator we can convert the type of operands from one type to another type for operation.

There are two types of conversion in JavaScript.

* Explicit Type Conversion (also known as type casting)
  + This Occurs when we intentionally convert a value from one type to another using built-in functions or operators. For example:



* In the example above, the string “5” is explicit coerced to a number using **`parseInt ()`** .
* Implicit Type Conversion (also known as coercion):
  + This happens when JavaScript automatically converts a value from one type to another, usually during operations. This can sometimes lead to unexpected result if not understood properly. For Example:



444

* + In this case, the concatenation of a string and a number occurs because the `+` operator is used, and JavaScript implicitly coerces the number `y` to a string perform the concatenation.

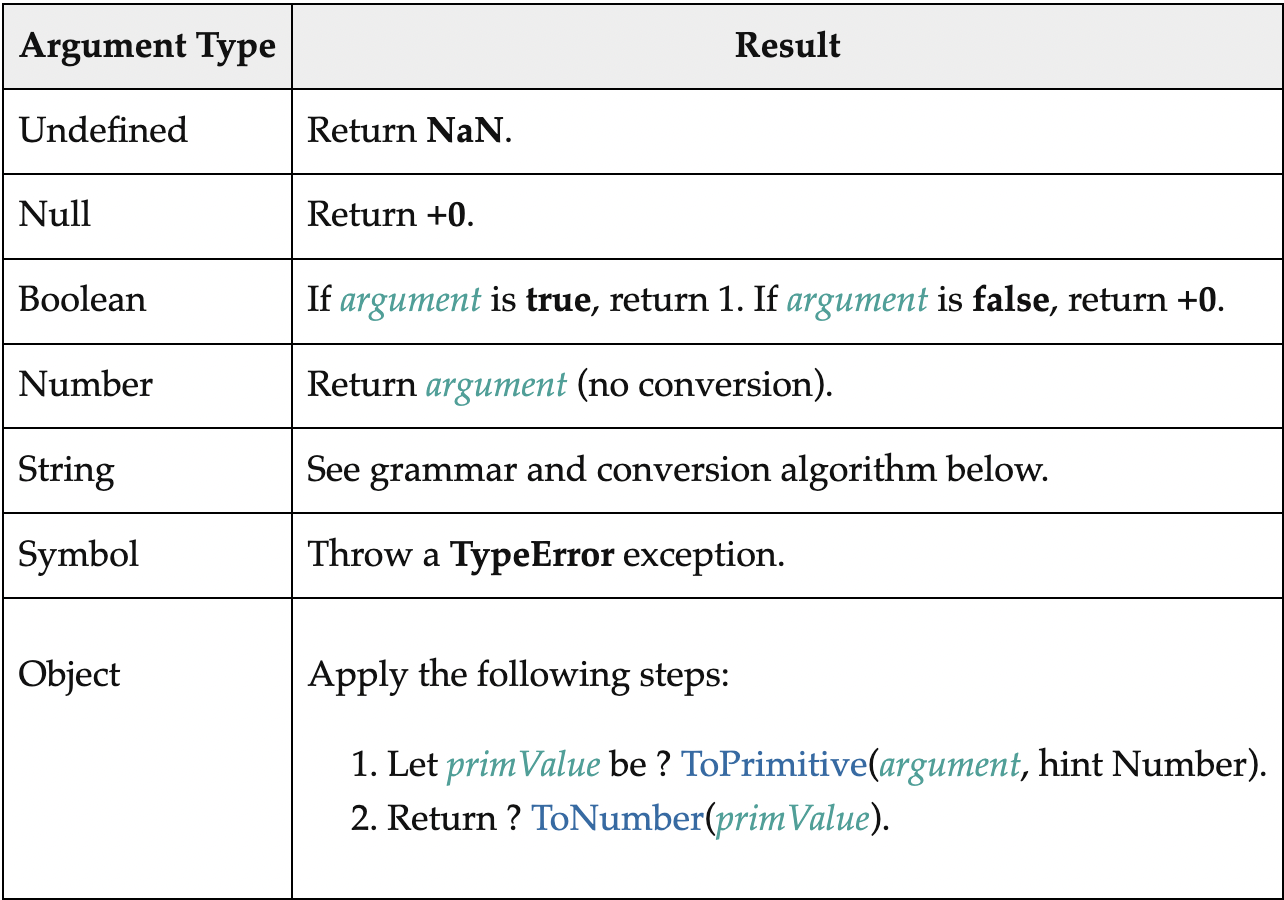
Abstract Operations:

There are some set of algorithms, that is present in ECMAScript docs, but they are not available for the usage in ECMAScript i.e., we as a developer cannot use those operations.

They are mentioned in the docs to aid (help) the documentation only. In ECMAScript docs, there are a lot of things that are done by the language internally. To explain these internal details of how and what language is doing, we have abstract operation mentioned in the docs.

ToNumber:

The abstract operation ToNumber converts argument to a value of type Number according to



The Subtraction Operator (-):

[AdditiveExpression](https://262.ecma-international.org/10.0/" \l "prod-AdditiveExpression): [AdditiveExpression](https://262.ecma-international.org/10.0/#prod-AdditiveExpression) - [MultiplicativeExpression](https://262.ecma-international.org/10.0/#prod-MultiplicativeExpression)

1. Let *lref* be the result of evaluating [AdditiveExpression](https://262.ecma-international.org/10.0/#prod-AdditiveExpression).
2. Let *lval* be ? [GetValue](https://262.ecma-international.org/10.0/" \l "sec-getvalue)(*lref*) { get the value of left operand & store it inside lval }
3. Let *rref* be the result of evaluating [MultiplicativeExpression](https://262.ecma-international.org/10.0/#prod-MultiplicativeExpression).
4. Let *rval* be ? [GetValue](https://262.ecma-international.org/10.0/" \l "sec-getvalue)(*rref*). { get the value of right operand & store it inside rval }
5. Let *lnum* be ? [ToNumber](https://262.ecma-international.org/10.0/" \l "sec-tonumber)(*lval*) .
6. Let *rnum* be ? [ToNumber](https://262.ecma-international.org/10.0/" \l "sec-tonumber)(*rval*).
7. Return the result of applying the subtraction operation to *lnum* and *rnum*.

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 favors that type.

1. [Assert](https://262.ecma-international.org/10.0/#assert): *input* is an [ECMAScript language value](https://262.ecma-international.org/10.0/#sec-ecmascript-language-types).
2. If [Type](https://262.ecma-international.org/10.0/#sec-ecmascript-data-types-and-values)(*input*) is Object, then
   1. If *PreferredType* is not present, let *hint* be **"default"**.
   2. Else if *PreferredType* is hint String, let *hint* be **"string"**.
   3. Else *PreferredType* is hint Number, let *hint* be **"number"**.
   4. Let *exoticToPrim* be ? [GetMethod](https://262.ecma-international.org/10.0/#sec-getmethod)(*input*, @@toPrimitive).
   5. If *exoticToPrim* is not undefined, then
      1. Let *result* be ? [Call](https://262.ecma-international.org/10.0/#sec-call)(*exoticToPrim*, *input*, « *hint* »).
      2. If [Type](https://262.ecma-international.org/10.0/#sec-ecmascript-data-types-and-values)(*result*) is not Object, return *result*.
      3. Throw a TypeError exception.
   6. If *hint* is **"default"**, set *hint* to **"number"**.
   7. Return ? [OrdinaryToPrimitive](https://262.ecma-international.org/10.0/#sec-ordinarytoprimitive)(*input*, *hint*).
3. Return *input*.

# OrdinaryToPrimitive ( O, hint )

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