Summary

1. *What is Type Conversion?*
2. *Implicit Type Conversion (Type Coercion)*
3. *Explicit Type Conversion (Type Casting)*

In the world of JavaScript, Type Conversion is the process of ***transforming data from one type to another***, adapting to the ever-shifting needs of your code.

types: ***boolean***, ***number***, ***string***, ***bigInt***, ***symbol***, ***null***, ***undefined*** or ***Object***.

There are two main types of Type conversion in JavaScript:

1. ***Implicit Type Conversion*** — automatically done during code execution by JavaScript engine. It is usually done when some operation is done on operands of different data types.
2. ***Explicit Type Conversion*** — done manually by humans.

Implicit Type Conversion (Type Coercion)

Implicit Type Conversion or ***Type coercion*** in JavaScript is the ***automatic conversion of one data type to another***. This happens when an operator or function requires a certain type, but the actual data type is different.

```
let numberVar = 10;

let stringVar = "20";


let result = numberVar + stringVar

console.log(result); // Output: "1020"
```

JavaScript coerces the number into a string and performs ***string concatenation*** instead of numerical addition. The result is the string "1020."

Loose equality operator

JavaScript provides two equality operators: the double equality operator (`==`and `!=`), known as the ***loose equality operator***, and the triple equality operator (`===`and `!==`), known as the ***strict equality operator***. These operators are used to compare the equality of values.

The ***loose equality operator*** performs a loose check, focusing solely on whether values are equal. It converts operands to ***numbers*** unless they're ***null*** or ***undefined***. Of course, if operands are of the same type there is no need to convert them to numbers. Unlike the strict equality operator, it doesn't prioritize types — ***only the values themselves are considered***. Let's have a look:

```
let a = 10;
let b = '10';
a == b; // true - loose equality
```

What happens here? Well, coercion duh... When the loose equality operation is performed, the operand `b` is implicitly ***converted from string to number*** and then the comparation of both number values is done.

```
let bool1 = false;
let num1 = 0;
bool1 == num1; // true

let bool2 = true;
let num2 = 6;
bool2 == num2; // false - bool2 coerces to 1, since 1 != 6, the result is
false
```

Here, JavaScript will coerce the boolean `bool1` to a **number** in order to perform the loose equality check. The coercion rules are such that `false` is coerced to `0` during numeric operations. So, in this case `bool1` is loosely equal to `num1`, since the numeric values of both are `0`. In the second example, `bool2`is converted to number `1` and the result of loose equality is ***false***, since `1`is not equal to `6`. Here are some other logical examples:

```
'5' == 5;                   // true - '5' is converted to a number 5
5 == [5];                   // true - [5] is converted to a primitive value 5
5 == ['5'];                 // true - ['5'] is converted to a primitive value
5
0 == '';                    // true - '' is converted to a number 0
0 == '0';                   // true - '0' is converted to a number 0
false == 'false'           // false - 'false' is converted to a NaN, and false
is converted to 0
false == '0';               // true - false is converted to number 0 and '0'
also
new String('str') == 'str';// true - the first operand object is converted to
a primitive value 'str'
'' == '0';                  // false - the operands are not converted to
numbers, and strings '' and '0' are not equal
+0 == -0;                   // true
```

Here are some cases with null and undefined that you should keep in mind:

```
// have in mind that Loose equality operator converts operands to numbers
unless they're null or undefined
null == undefined;  // true
false == undefined; // false
false == null;      // false

// nothing is equal to NaN, not even NaN itself
NaN == NaN;         // false
NaN == false;       // false
NaN == undefined;   // false
NaN == null;        // false
```

And now some special cases:

```
[5] == [5]                // false - these arrays look the same but these are
not the same instances
{ a: 'a' } == { a: 'a' }  // false - again these two objects are not the same
instances
{} == {}                  // false - not same instances!
```

On the other side, the ***Strict equality operator*** or triple equality operator (`===` and `!==`) conducts a strict check, thoroughly examining both the values and their types. Unlike the loose equality operator, ***there is no type coercion happening here***, ensuring precise and expected results.

```
let a = 10;
let b = '10';
// strict equality:
a === b; // false
```

The values of operand `a` and operand `b` are the same, but their data types are different — number and string, so the strict equality operator returns false. There is no need for type coercion.

Comparison operators

As you already know, comparison operators are used to compare two values and they are: >, <, ≥ and ≥. They coerce values to ***numbers***.

```
let a = 5;
let b = '10';
a > b; // false
a < b; // true
```

Explanation: JavaScript performs implicit type conversion when comparing `a` and `b`. So `b` is converted to a number for the comparison. Now, it compares the numerical values of `a` and `b`. Result: a > b is ***false*** because 5 is not greater than 10. Similarly, a < b is true because 5 is less than 10.

Logical operators

Logical operators are used to determine if an expression evaluates to ***true*** or ***false***. Operators are: **&&** (AND), || (OR) and **!** (NOT). They coerce values to ***boolean*** type.

```
let num = 10;
let str = "Hello";
num && str; // true
// Implicit conversion: num is converted to true, str is converted to true
```

Unary operators

Unary operators are operators that only operate on one operand. One common unary operator in JavaScript is the `+` operator, which is used for converting its operand to a number. Unary operators are: +, ++, -, - -.

```
let str = '42';
let num = +str; // Converts string '42' to number 42

let value = '5';
let result = -value; // Converts string '5' to negative number -5

let num2 = '5';
num2++; // Converts string '5' to number and increments it, so now it's a
number 6
```

Arithmetic operators

They are used to perform arithmetic operations over

operands: + (addition), — (subtraction), * (multiplication) and / (division). Arithmetic operands coerce operands to ***numbers***, except for + operator (if one operand is of type string, the operand + is used as concatenation and other operand is coerced to string).

```
let num = 10; // num is a number
let str = '5'; // str is a string containing the character '5'

let result = num * str;

// Implicit conversion: str is converted to number, so the result is 50
```

When the `*` operator is encountered, JavaScript attempts to perform a mathematical operation. Since the operand `str` is a string and the other `num` is a number, JavaScript implicitly coerces the string to a number. So `str` is converted to number 5, and the result of multiplication is number 50.

Bitwise operators

They treat operands as sets of 32-bit binary digits, so the operations are done on individual bits. They coerce operands to ***numbers***. They work on the binary representation of integers. If an operand is not an integer, it will be converted to one before the operation. Bitwise operators are: | (bitwise or), & (bitwise and), ^ (xor) and ~ (bitwise not).

```
let a = '5';
let b = 2;

let result = a | b;  // '5' is coerced to 5 before the bitwise OR operation
console.log(result); // Output: 7 (binary OR of 5 and 2)
```

Can I prevent Type Coercion somehow in JavaScript?

Yes, don't worry! Shielding your code from type coercion in JavaScript involves vigilance regarding variable and operand data types. Utilize type conversion functions when needed. Opt for the ***strict equality operators*** (=== and !==) instead of their loose counterparts (== and !=) during value comparisons. If you remember, strict equality operators ensure that compared values are not only identical in value but also share the same type, preventing implicit type coercion!

Explicit Type Conversion (Type Casting)

Explicit type conversion in JavaScript, also known as ***Type Casting***, is the process of ***manually converting a value from one data type to another***. This is done using built-in functions or operators that are specifically designed for type conversion. You use ***Type Constructors*** to explicitly convert types. Unlike implicit type conversion, which happens automatically during certain operations, explicit type conversion gives you control over the conversion process.

There are 3 types of explicit conversion:

1. Conversion to Boolean
2. Conversion to Number
3. Conversion to String

Conversion to Boolean

Here are examples of explicit conversion to Boolean data type

using **Boolean(value)** constructor:

```
Boolean(0);          // false
Boolean('');         // false
Boolean(null);       // false
Boolean(undefined);  // false
Boolean(NaN);        // false
Boolean(-0);         // false
Boolean(false);      // false
-------------------------------
Boolean(1);          // true
Boolean(-10);        // true
Boolean('apple');    // true
Boolean([]);         // true
Boolean({ a: 'a' }); // true - every non-primitive value is converted to true
Boolean({});         // true - even the empty object
Boolean(Symbol());   // true
Boolean(Symbol('x'));// true
```

Conversion to Number

Here are examples of explicit conversion to Number data type

using **Number(value)** constructor:

```
Number(0);             // 0
Number('');            // 0
Number(null);          // 0
Number(NaN);           // NaN
Number(undefined);     // NaN
Number(true);          // 1
Number(false);         // 0
Number(Symbol());      // TypeError
Number('   5    ');    // 5
Number(' ');           // 0
Number('\n');          // 0
Number('-1/2');        // NaN
Number('105.4');       // 105.4
```

Conversion to String

Here are examples of explicit conversion to String data type using **String(value)** constructor:

```
String(0);              // '0'
String('');             // ''
String(true);           // 'true'
String(NaN);            // 'NaN'
String(undefined);      // 'undefined'
String(null);           // 'null'
String(Symbol());       // 'Symbol()'
String(Symbol('x'));    // 'Symbol(x)'
String(-105.4);         // '-105.4'
String([]);             // ''
String({});             // '[object Object]'
String({ a: 'a' });     // '[object Object]'
```

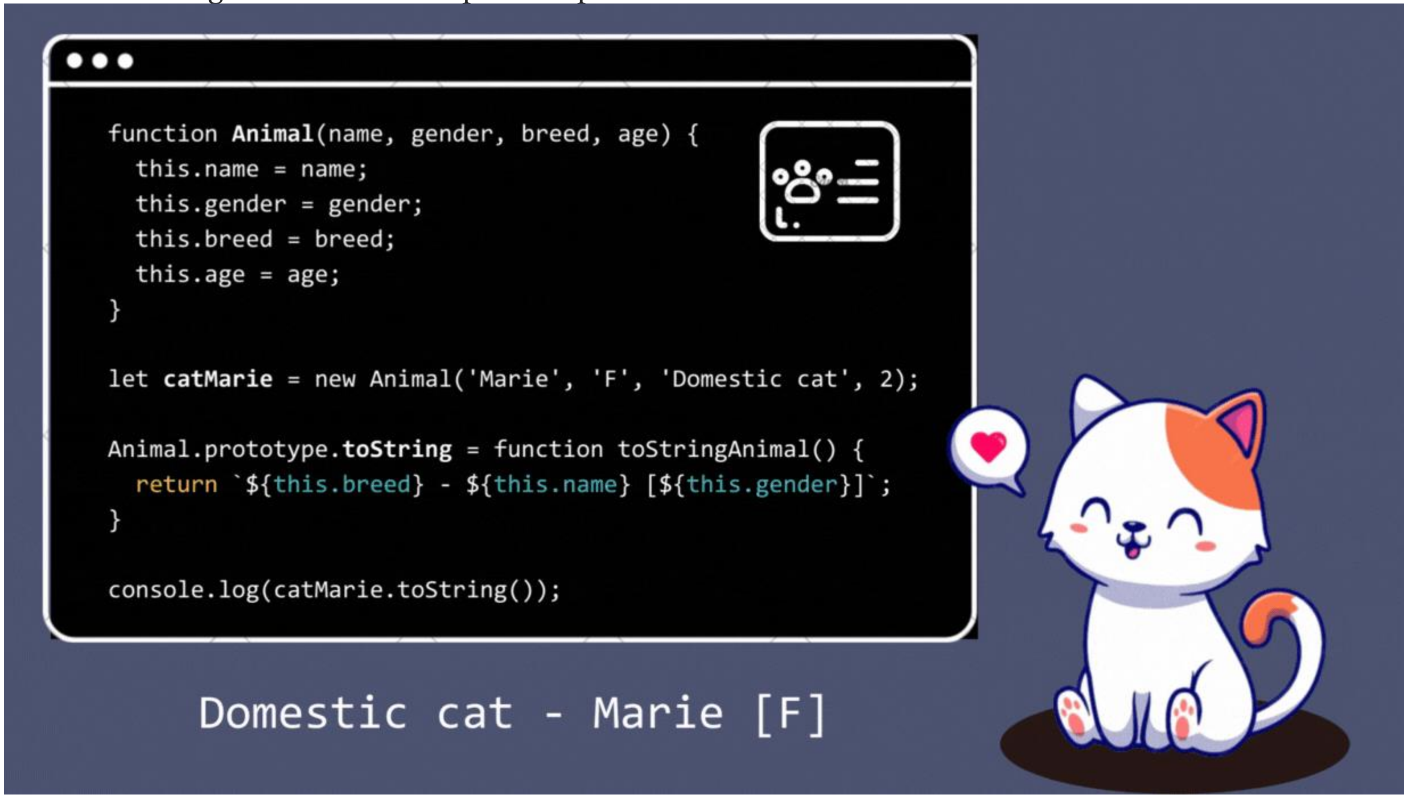
There are some specific functions that JavaScript provides us so we can do the type conversion:

- **value.toString([*radix*])** method — coverts the given ***number*** `value`and returns its corresponding string representation. The optional argument `radix` represents the base of number value and it can take values from 2 to 36 (default is 10).

```
let num = 5;
let str = num.toString();
console.log(str); // Output: '5'

let binStr = num.toString(2);
console.log(binStr); // Output: '101'
```

Object.toString() method — returns a string representation of an Object, and that's the `[object Object]` string. Mmm, that's not helpful at all, right? Well, the **toString()** method is meant to be ***overridden*** to accomplish the desired ***Custom Type Conversion*** logic. Let's have a simple example:



```
function Animal(name, gender, breed, age) {
  this.name = name;
  this.gender = gender;
  this.breed = breed;
  this.age = age;
}

let catMarie = new Animal('Marie', 'F', 'Domestic cat', 2);

Animal.prototype.toString = function toStringAnimal() {
  return `${this.breed} - ${this.name} [${this.gender}]`;
}

console.log(catMarie.toString());
```

```
Domestic cat - Marie [F]
```

Example of overridden toString() method.

Some Objects have the **toString()** method already overridden. In the previous example, **value.toString([*radix*])** method is actually an overridden **Object.toString()** method but for ***Number*** type. String object has its own **toString()** method overridden, also Boolean too. Let's have a quick look:

```
const numObj = new Number(5);
const strNum = numObj.toString();
console.log(numObj); // Number { 5 }
console.log(strNum); // '5'
------------------------------------------
const strObj = new String('example');
const strValue = strObj.toString();
console.log(strObj);     // String { 'exampe' }
console.log(strValue);   // 'example'
------------------------------------------
```

```
const boolObj = new Boolean(true);
const strBool = boolObj.toString();
console.log(boolObj); // Boolean { true }
console.log(strBool); // 'true'
```

- **parseInt(string [, *radix*])** — parses a `string` argument of type string and returns an **Intiger** value with a specified `radix` which is optional argument that can have values from 2 to 36 (defaults to 10). If the `string` argument can't be converted to an Integer, the result is a NaN value.

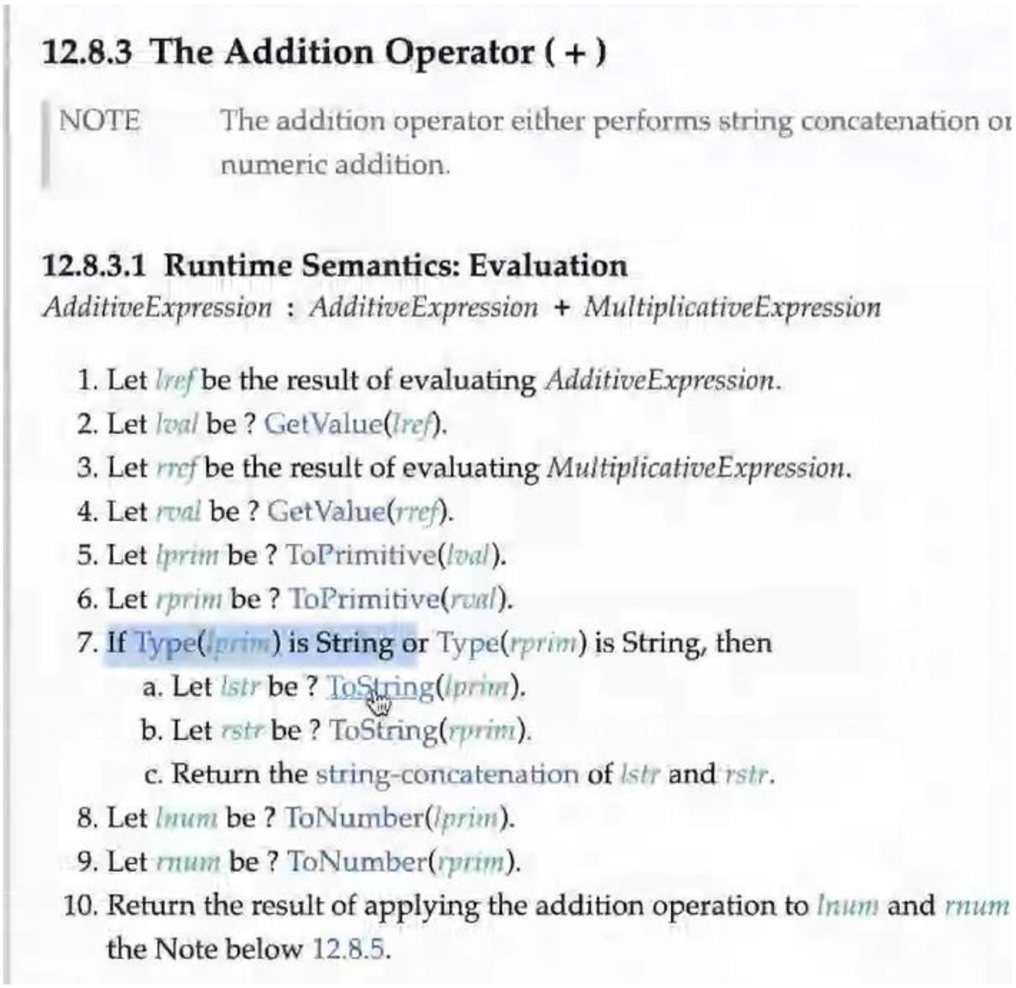
```
parseInt('5');          // 5
parseInt('-5');         // -5
parseInt('100');        // 100
parseInt('    12  ');   // 12 - whitespaces are ignored
parseInt('1.25');       // 1 - rounded to Integer
parseInt('101', 2);     // 5
parseInt('080');        // 80 - leading zeros are ignored
parseInt('example 5');  // NaN - can't be converted to Integer
```

- **parseFloat(string)** — parses a `string` argument of type string and returns a Floating point number. If the `string` can't be converted to Floating point number, the result is a NaN value.

```
parseFloat('5');                // 5
parseFloat('-1.25');            // -1.25
parseFloat('     12.24   ');    // 12.24 - whitespaces are ignored
parseFloat('001.25');           // 1.25 - leading zeros are ignored
parseFloat('-001.25');          // -1.25
parseFloat('sum: 1050.49');     // NaN - can't be converted to Floating point
number
```

Conclusion

In conclusion, navigating the nuances of JavaScript's type conversion and coercion is crucial for writing robust and predictable code. Understanding how values are converted, both implicitly and explicitly, empowers developers to handle data effectively. Whether you're converting between types for operations or dealing with comparisons,



12.8.3 The Addition Operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

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).
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
    the Note below 12.8.5.
```

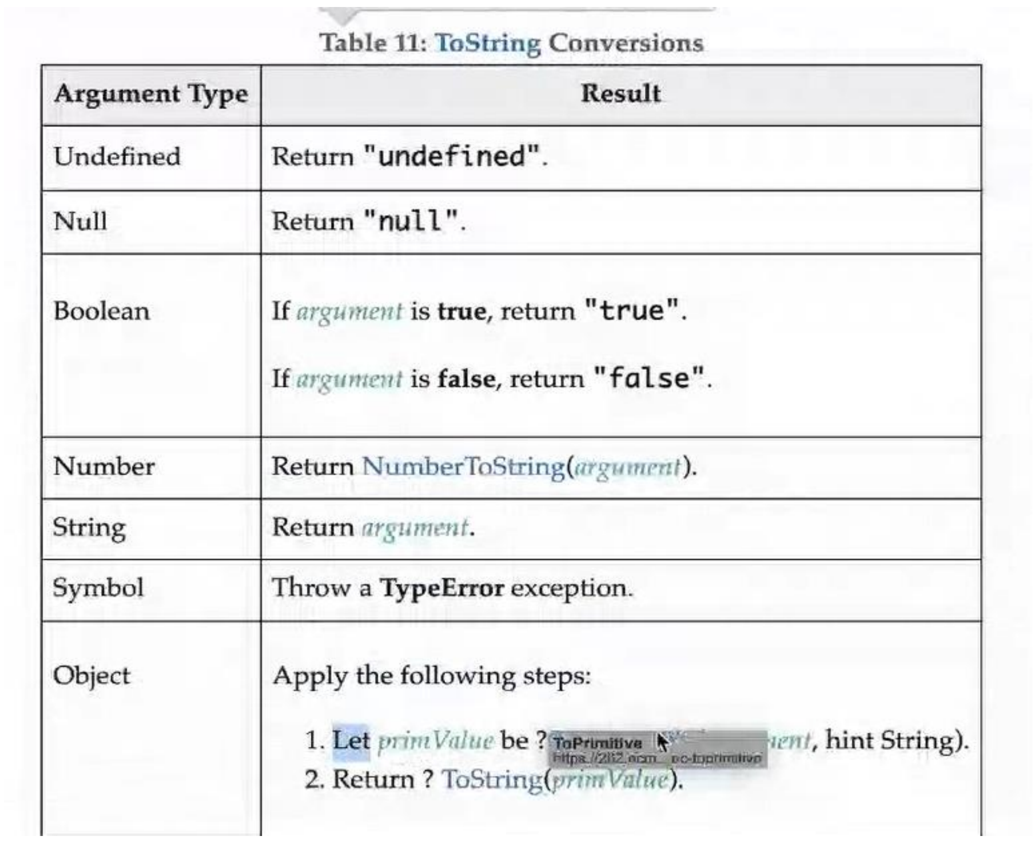
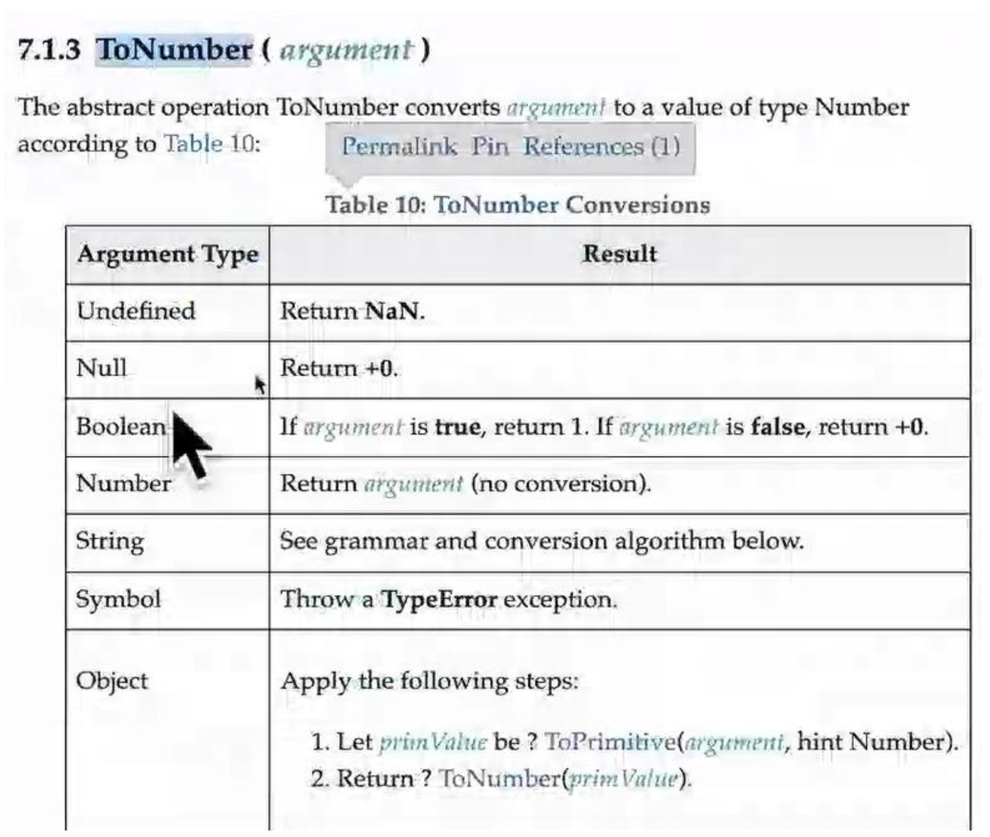


Table 11: ToString Conversions

Argument Type	Result
Undefined	Return `"undefined"`.
Null	Return `"null"`.
Boolean	If *argument* is **true**, return `"true"`. If *argument* is **false**, return `"false"`.
Number	Return NumberToString(*argument*).
String	Return *argument*.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ? ToPrimitive https://262.ecm... ec-toprimitive *ent*, hint String). 2. Return ? ToString(*primValue*).



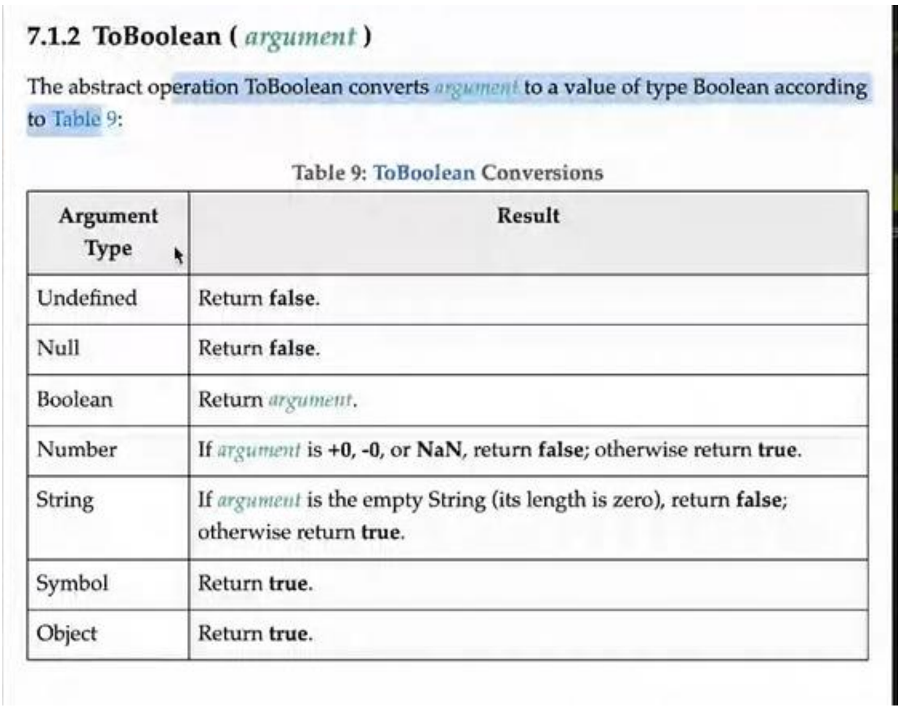
7.1.3 ToNumber (*argument*)

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

Permalink Pin References (1)

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*).

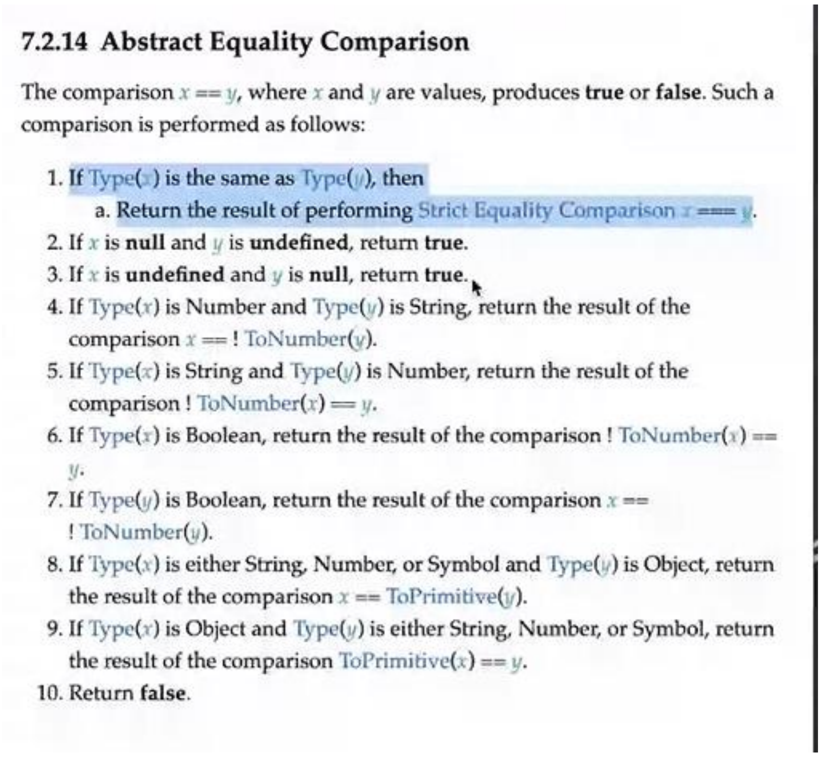


7.1.2 ToBoolean (*argument*)

The abstract operation ToBoolean converts *argument* to a value of type Boolean according to Table 9:

Table 9: ToBoolean Conversions

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**.



7.2.14 Abstract Equality Comparison

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is the same as Type(*y*), then
   a. Return the result of performing Strict Equality Comparison *x* === *y*.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String, return the result of the comparison *x* == ! ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number, return the result of the comparison ! ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ! ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ! ToNumber(*y*).
8. If Type(*x*) is either String, Number, or Symbol and Type(*y*) is Object, return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(*x*) == *y*.
10. Return **false**.