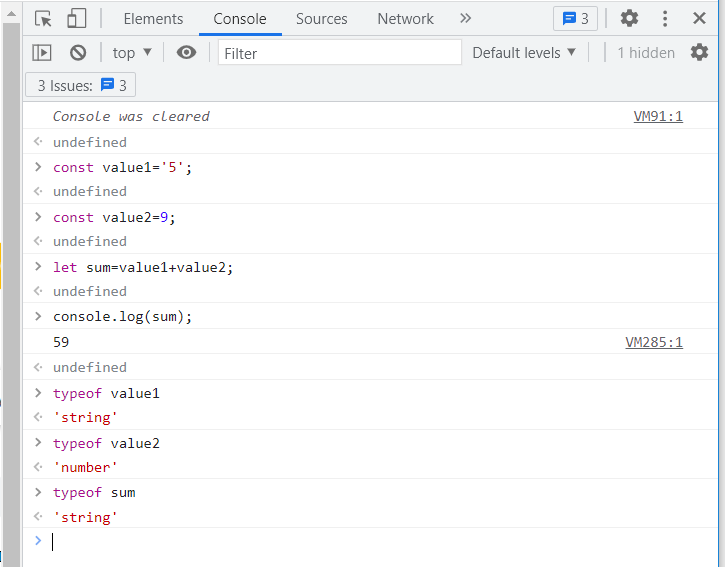
**Chapter 6: Type Coercion Madness**

**Type Coercion:**

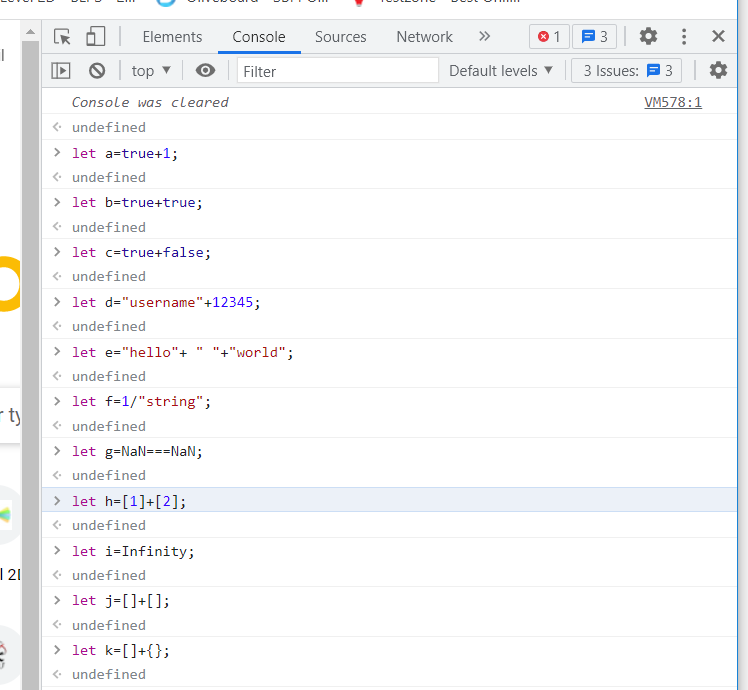
Type coercion is the automatic or implicit conversion of values from one data type to another.

For Eg:



In the above example, the variable value1 is declared as constant and 5 is assigned as a string. For variable value2, it is declared as constant and 9 is assigned as a number to it. A variable sum is declared as let and the total of value1 and value2 is assigned to it. Since one variable is string and the other is number , javascript will coerce the number 9 to string and instead of adding two variables it will concatenate them and produce the result as 59 which is a string.

Examples of type coercion



Their output will be:



In the variable a , true will be taken as 1, thus true+1 will be treated as 1+1= 2.

In b, true+true will be treated as 1+1 which is equal to 2.

In c, true+false will be treated as 1+0, which is equal to 1.

In d, string + number, number will be taken as string thus it will be concatenated instead of addition.

In e, string + string, + operator will concatenate when used with strings.

In f, / operator should be used with number.

In g, equating NaN to itself will give the result false.

In h, [1]+[2] will be considered as strings and thus they will be concatenated.

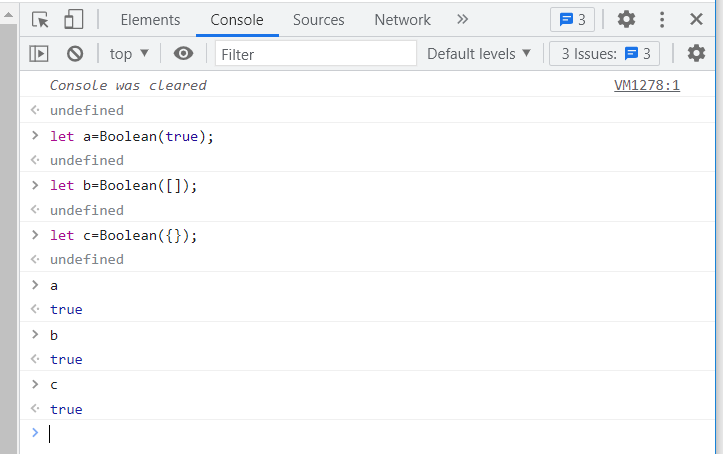
In i, the value remains as infinity. When given in double quotes or single quotes it will consider as string else it will be considered as number.

In j, []+[] will be considered as string “”

In k, when object {} +array [] is added it will consider both as objects.

**Type coercion in constructors:**

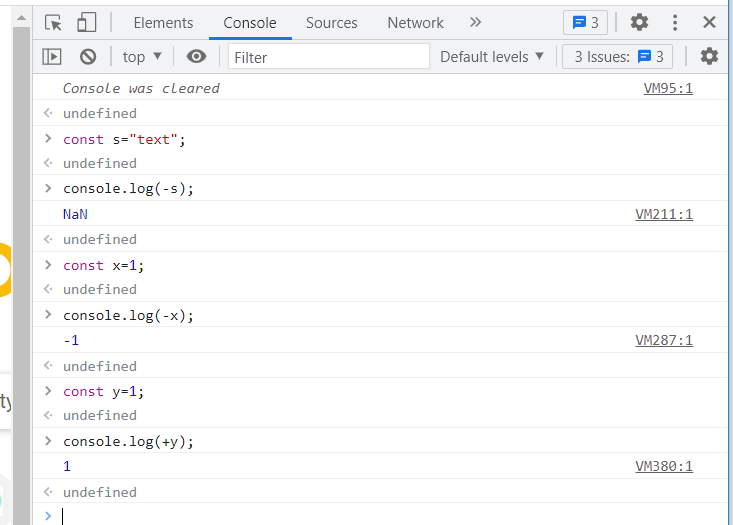
Coercion also takes place when we provide initialization value to a type constructor.



Type Coercion:

In type coercion []==[] is false and let a=[]; a==a; is true.

This is because in javascript == operator tests by reference and not by value. And []==![] is true.



The unary operator(+, -) cannot be applied to string

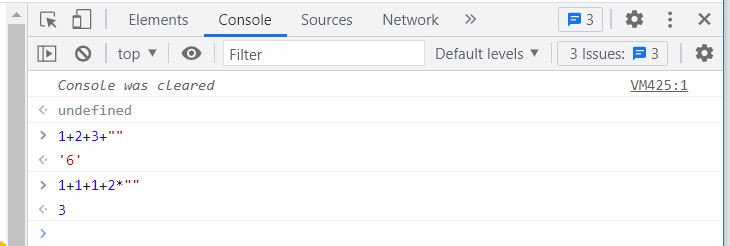
Note:

* The arithmetic operator is evaluated from left to right
* The assignment operator is evaluated from right to left.

Adding Multiple Values:

When we add multiple values the addition takes place from left to right.

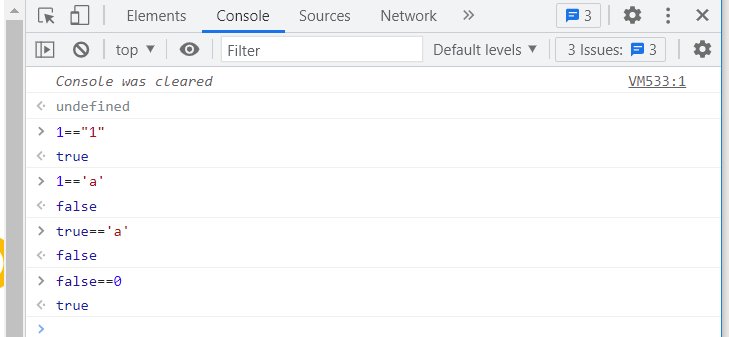
In certain cases some operators take precedence over others.



In the above example 1+2=3,3+3=6 and 6+”” will be numeric string of 6 i.e ‘6’.

In the next example 1+1+1+2\*””, here the multiplication operator has the firts priority the addition comes the next.

Type comparison:



**Operator precedence and associativity table:**

| **Precedence** | Operator type | Associativity | Individual operators |
| --- | --- | --- | --- |
| 18 | Grouping | n/a | ( … ) |
| 17 | Member Access | Left to right | … . … |
| COmputed Member Access | n/a | … [...] |
| New (with argument list) | n/a | New …(...) |
| Function call | n/a | … ( … ) |
| Optional chaining | Left to right | ?. |
| 16 | new(without argument list) | n/a | New … |
| 15 | Postfix increment | n/a | … ++ |
| Postfix decrement | n/a | … – |
| 14 | Logical NOT(!) | n/a | ! … |
| Bitwise NOT(~) | ~ … |
| Unary plus(+) | * … |
| Unary negation(-) | \_ … |
| Prefix increment | ++ … |
| Prefx decrement | – … |
| typeof | Typeof … |
| void | Void … |
| delete | Delete … |
| await | Await … |
| 13 | Exponentiation(\*\*) | Right to left | … \*\*... |
| 12 | Multiplication(\*) | Left to right | … \* … |
| Division(/) | … / … |
| Remainder(%) | … % … |
| 11 | Addition (+) | Left to right | … +... |
|  | Subtraction (-) | … - … |
| 10 | Bitwise left shift (<<) | Left to right | … << … |
| Bitwise right shift(>>) | … >> … |
| Bitwise unsigned right shift (>>>) | … >>> … |
| 9 | Less than (<) | Left to right | … < … |
| Less than or equal (<=) | … <= … |
| Greater than (>) | … > … |
| Greater than or equal (>=) | … >= ... |
| in | … in … |
| instanceof | … instanceof … |
| 8 | Equality(==) | Left to right | .. == .. |
| inequality(!=) | .. != .. |
| Strict Equality (===) | … === … |
| Strict Inequality(!==) | … !== .. |
| 7 | Bitwise AND (&) | Left to right | … & … |
| 6 | Bitwise XOR (^) | Left to right | … ^ … |
| 5 | Bitwise OR (|) | Left to right | … | … |
| 4 | Logical AND (&&) | Left to right | … && … |
| 3 | Logical OR(||) | Left to right | … || … |
| Nullishcoalescing operator(??) | Left to right | … ?? … |
| 2 | Assignment |  | ..=.. |
|  | ..+=.. |
|  | ..-=.. |
|  | …\*\*=... |
|  | ..\*=.. |
|  | ../=.. |
|  | ..%=.. |
|  | ..<<=.. |
|  | ..>>=.. |
|  | ..>>>=.. |
|  | ..&=.. |
|  | ..^=.. |
|  | ..|=.. |
|  | ..&&=.. |
|  | ..||=.. |
|  | ..??=.. |
| Conditional (ternary) operator | Right to left | … ? ... : .. |
| Arrow(=>) | n/a | … => … |
| yield | Yield … |
| yield\* | yield\* … |
| spread(..) | . . . … |
| 1 | comma/sequence | Left to right | … , … |

**L-value and R-value:**

Assignment Operator:

L=R here R is assigned to L

L+R here L value is added to R

Null vs undefined:

The null primitive is not an object, it doent have a built in constructor. its literal value is null. If a variable is not a assigned a value at the time of declaration its value will be undefined by default. At this time its good to assign null as its value.

The null keyword is used to assign a temporary default value to a variable before it is assigned with an actual object data.

