Javascript Notes

TYPE COERCION

what is type coercion in javascript?

In JavaScript, type coercion is the process of converting a value from one data type to another data type. This can happen automatically by the JavaScript engine or it can be explicitly done in code.

For example, consider the following code snippet:

var x = 10;

var y = "5";

var z = x + y;

In this code, we have declared two variables x and y with the values 10 and "5", respectively. We then create a third variable z and set its value to the result of x + y.

Now, x is a number and y is a string. When we try to add them together, JavaScript will perform type coercion on the string y to convert it to a number so that it can perform the addition operation. The result of z will be 15, which is the numeric value of y added to x.

JavaScript has both implicit and explicit type coercion. Implicit coercion happens automatically when a value is used in a context where a different type is expected. Explicit coercion happens when we deliberately convert a value from one type to another using functions like Number(), String(), or Boolean().

It is important to be aware of type coercion in JavaScript as it can sometimes lead to unexpected behavior and bugs in your code.

why `y` will be converted and not the x ?

In JavaScript, when the + operator is used with a number and a string, the JavaScript engine will always try to convert the string to a number and then perform the addition operation. This is known as type coercion.

In the code snippet I provided earlier, x is already a number, so it doesn't need to be converted. On the other hand, y is a string, so it needs to be converted to a number before the addition operation can be performed.

JavaScript has a set of rules that it follows to determine how type coercion should be performed in different situations. These rules are often referred to as the "type coercion rules" or the "type conversion rules". Understanding these rules is important when writing JavaScript code, as it can help you avoid common pitfalls and bugs caused by unexpected type coercion.

EQUALITY

In JavaScript, the double equals (==) and triple equals (===) are comparison operators used to compare values. The main difference between the two is the type of comparison they perform.

The double equals (==) performs a loose or abstract comparison, which means that it checks if the values being compared are equal in value, but not necessarily in type. If the types are different, JavaScript will try to coerce one or both values to a common type before making the comparison. For example:

javascript

Copy code

console.log(5 == '5'); // true

console.log(true == 1); // true

console.log(null == undefined); // true

On the other hand, the triple equals (===) performs a strict comparison, which means that it checks if the values being compared are not only equal in value, but also in type. If the types are different, the comparison will always return false. For example:

javascript

Copy code

console.log(5 === '5'); // false

console.log(true === 1); // false

console.log(null === undefined); // false

It's generally recommended to use the triple equals (===) operator for comparisons, as it can help prevent unexpected behavior due to type coercion. However, there may be cases where a loose comparison (==) is more appropriate, such as when checking for null or undefined values.

In JavaScript, there are two types of equality operators: == (loose equality) and === (strict equality).

The **== (loose equality)** operator compares the values of two operands after attempting to perform type coercion if necessary. If the operands are of different types, JavaScript will try to convert one or both of the operands to the same type before making the comparison.

**For example:**

console.log(5 == "5"); // Output: true

In this example, the == operator compares the number 5 and the string "5". Since the two operands are of different types, JavaScript performs type coercion and converts the string "5" to the number 5. The resulting comparison is 5 == 5, which is true.

The === (strict equality) operator, on the other hand, compares the values of two operands without performing any type coercion. If the operands are of different types, the comparison will always return false. For example:

console.log(5 === "5"); // Output: false

In this example, the === operator compares the number 5 and the string "5". Since the two operands are of different types, the comparison immediately returns false without performing any type coercion.

In general, it's recommended to use the === (strict equality) operator in JavaScript, as it avoids potential issues that can arise from type coercion when using the == (loose equality) operator.

**SEMICOLON**

One thing to take notice of in all the examples you've seen so far is the use of semicolons ; at the end of each line. Semicolons make it clear where one statement ends and another begins. This is especially handy when multiple lines of code are written on the same line (which is valid JavaScript, but definitely not recommended!). For instance:

var totalAfterTax = 53.03 var tip = 8 // this is incorrect!

Uncaught SyntaxError: Unexpected token var

vs.

var totalAfterTax = 53.03; var tip = 8; // this is correct!

Not adding semicolons to the end of each line can cause bugs and errors in your programs. JavaScript does have ways to occasionally predict where semicolons should be, but just like how type coercion can result in some unexpected quirky behavior in JavaScript, it's good practice to not depend on it.

CONDITIONALS

The steps you code takes to solve a problem known as an algorithm.

IF – ELSE statement

A couple of important things to notice about if...else statements.

The value inside the **if** statement is always converted to true or false. Depending on the value, the code inside the if statement is run or the code inside the else statement is run, but not both. The code inside the if and else statements are surrounded by curly braces {...} to separate the conditions and indicate which code should be run.

TIP: When coding, sometimes you may only want to use an if statement. However, if you try to use only an else statement, then you will receive the error SyntaxError: Unexpected token else. You’ll see this error because else statements need an if statement in order to work. You can’t have an else statement without first having an if statement.

Else if statements

In JavaScript, you can represent this secondary check by using an extra if statement called an else if statement.

More Complex Problems

**Logical Operators**

console.log((3 != 6 % 3) && !(24 > 45) && (!false));

Yes, that is correct! The expression (3 != 6 % 3) && !(24 > 45) && (!false) evaluates to true, and the console.log() function will output the value of this expression to the console, which is true.

console.log((10 ==="10") && (1 <= 2));

No, that is not correct. The expression (10 === "10") && (1 <= 2) evaluates to false. The first part of the expression (10 === "10") is comparing the number 10 with the string "10". Since these two are of different types (a number and a string), they are not equal, so the result of this comparison is false. The second part of the expression (1 <= 2) is comparing the number 1 with the number 2. Since 1 is less than or equal to 2, the result of this comparison is true. When we use the logical AND operator && to join these two comparisons, we get false && true, which evaluates to false. Therefore, the console.log() function will output the value false to the console.

console.log((13 > -7 ) || (false == 0));

Yes, that is correct! The expression (13 > -7) || (false == 0) evaluates to true. In the first part of the expression, 13 is greater than -7, so the comparison 13 > -7 evaluates to true. In the second part of the expression, false is equal to 0, so the comparison false == 0 also evaluates to true. When we use the logical OR operator || to join these two comparisons, we get true || true, which evaluates to true. Therefore, the console.log() function will output the value true to the console.

console.log(!true);

Yes, that is correct! The expression !true evaluates to false. The exclamation mark (!) is the logical NOT operator, and it returns the opposite boolean value of the operand. In this case, the operand is true, so the opposite boolean value is false. Therefore, the console.log() function will output the value false to the console.

console.log(false && false);

Yes, that is correct! The expression false && false evaluates to false. The && operator is the logical AND operator. It returns true if and only if both of its operands are true. In this case, both of the operands are false, so the result is false. Therefore, the console.log() function will output the value false to the console.

console.log(true || false);

Yes, that is correct! The expression true || false evaluates to true. The || operator is the logical OR operator. It returns true if at least one of its operands is true. In this case, one of the operands is true (the value true), so the result is true. Therefore, the console.log() function will output the value true to the console.