JavaScript - Questions

Vincent Erich
10384081

November 6, 2015

Questions

1. The == operator is the 'equal to' operator. It is a binary operator, and an expression containing this operator evaluates to true if both arguments (values) are equal to each other (it evaluates to false otherwise), for example:

   ```
   console.log(10 == 10);        console.log("string" == "string");
   // -> true                   // -> true

   console.log(7 == 10);         console.log("not" == "equal");
   // -> false                  // -> false
   ```

   The examples above show expressions where both arguments (values) are of the same type (number or string). However, when the types differ, JavaScript uses a complicated set of rules to determine what to do. In most cases, it tries to convert one of the arguments to the other argument's type, for example:

   ```
   console.log("5" == 5);        console.log("" == false);
   // -> true                   // -> true
   ```

   If you do not want any automatic type conversions to happen, you can use the === operator. The === is also a binary operator; it behaves like the == operator, except that it does not use automatic type conversions if the two arguments (values) are not of the same type. That is, it tests whether an argument is precisely equal to the other, for example:

   ```
   console.log("5" === 5);       console.log("" === false);
   // -> false                  // -> false
   ```

2. If you create a function in JavaScript and the function contains a local variable, this variable is recreated every time the function is called, for example:

   ```
   function add() {
       var counter = 0;
       return counter += 1;
   }

   console.log(add());
   // -> 1
   console.log(add());
   // -> 1
   console.log(add());
   // -> 1
   // The result is not 3. Every time the function 'add()' is called,
   // the local variable 'counter' is recreated (and thus its value is
   // set to 0), causing the return value to always be 1.
   ```

   In JavaScript, it is possible to treat functions as values. This makes it possible to create a function (e.g., `add()`) that returns another function that accesses `add()`'s local variable(s), and assign this (returned) function to a variable (e.g. `add_1`), for example:

   ```
   function add() {
       var counter = 0;
       return function() { counter += 1; return counter; };
   }

   var add_1 = add();
   console.log(add_1());
   // -> 1
   console.log(add_1());
   // -> 2
   console.log(add_1());
   // -> 3
   ```

   The example above shows that it is possible to reference to a specific instance of local variables in an enclosing function. This is called closure.

3. Higher-order functions are functions that operate on other functions. There are several ways in which a function can operate on another function (i.e., be a higher-order function). In the previous question, the second `add()` function returns another function, so the `add()` function is a higher-order

function (it operates on another function). It is also possible to pass a function as an argument to another function, making the latter a higher-order function. Furthermore, it it also possible to create a function that changes another function, making the first a higher-order function.

4. A query selector is a (selector) string that can be used to find all the elements in a document or element node that match the (selector) string. The (selector) string contains special selector syntax.

   The following line of JavaScript uses a query selector to find all the <p> elements in `document`:

   ```
   var array_with_p_elements = document.querySelectorAll("p");
   ```