This chapter points out some common JavaScript mistakes.

Accidentally Using the Assignment Operator

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

This if statement returns false (as expected) because x is not equal to 10:

let x = 0;  
if (x == 10)

This if statement returns true (maybe not as expected), because 10 is true:

let x = 0;  
if (x = 10)

This if statement returns false (maybe not as expected), because 0 is false:

let x = 0;  
if (x = 0)

An assignment always returns the value of the assignment.

Expecting Loose Comparison

In regular comparison, data type does not matter. This if statement returns true:

let x = 10;  
let y = "10";  
if (x == y)

In strict comparison, data type does matter. This if statement returns false:

let x = 10;  
let y = "10";  
if (x === y)

It is a common mistake to forget that switch statements use strict comparison:

This case switch will display an alert:

let x = 10;  
switch(x) {  
  case 10: alert("Hello");  
}

This case switch will not display an alert:

let x = 10;  
switch(x) {  
  case "10": alert("Hello");  
}

ADVERTISEMENT

Confusing Addition & Concatenation

**Addition** is about adding **numbers**.

**Concatenation** is about adding **strings**.

In JavaScript both operations use the same + operator.

Because of this, adding a number as a number will produce a different result from adding a number as a string:

let x = 10;  
x = 10 + 5;       // Now x is 15  
  
let y = 10;  
y += "5";        // Now y is "105"

When adding two variables, it can be difficult to anticipate the result:

let x = 10;  
let y = 5;  
let z = x + y;     // Now z is 15  
  
let x = 10;  
let y = "5";  
let z = x + y;     // Now z is "105"

Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits **Floating point numbers** (Floats).

All programming languages, including JavaScript, have difficulties with precise floating point values:

let x = 0.1;  
let y = 0.2;  
let z = x + y            // the result in z will not be 0.3

To solve the problem above, it helps to multiply and divide:

Example

let z = (x \* 10 + y \* 10) / 10;       // z will be 0.3

Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

Example 1

let x =  
"Hello World!";

But, breaking a statement in the middle of a string will not work:

Example 2

let x = "Hello  
World!";

You must use a "backslash" if you must break a statement in a string:

Example 3

let x = "Hello \  
World!";

Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

if (x == 19);  
{  
  // code block   
}

Breaking a Return Statement

It is a default JavaScript behavior to close a statement automatically at the end of a line.

Because of this, these two examples will return the same result:

Example 1

function myFunction(a) {  
  let power = 10   
  return a \* power  
}

Example 2

function myFunction(a) {  
  let power = 10;  
  return a \* power;  
}

JavaScript will also allow you to break a statement into two lines.

Because of this, example 3 will also return the same result:

Example 3

function myFunction(a) {  
  let  
  power = 10;   
  return a \* power;  
}

But, what will happen if you break the return statement in two lines like this:

Example 4

function myFunction(a) {  
  let  
  power = 10;   
  return  
  a \* power;  
}

The function will return undefined!

Why? Because JavaScript thought you meant:

Example 5

function myFunction(a) {  
  let  
  power = 10;   
  return;  
  a \* power;  
}

Explanation

If a statement is incomplete like:

let

JavaScript will try to complete the statement by reading the next line:

power = 10;

But since this statement is complete:

return

JavaScript will automatically close it like this:

return;

This happens because closing (ending) statements with semicolon is optional in JavaScript.

JavaScript will close the return statement at the end of the line, because it is a complete statement.

Never break a return statement.

Accessing Arrays with Named Indexes

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** use **numbered indexes**:

Example

const person = [];  
person[0] = "John";  
person[1] = "Doe";  
person[2] = 46;  
person.length;       // person.length will return 3  
person[0];           // person[0] will return "John"

In JavaScript, **objects** use **named indexes**.

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.

After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

Example:

const person = [];  
person["firstName"] = "John";  
person["lastName"] = "Doe";  
person["age"] = 46;  
person.length;      // person.length will return 0  
person[0];          // person[0] will return undefined

Ending Definitions with a Comma

Trailing commas in object and array definition are legal in ECMAScript 5.

Object Example:

person = {firstName:"John", lastName:"Doe", age:46,}

Array Example:

points = [40, 100, 1, 5, 25, 10,];

WARNING !!

Internet Explorer 8 will crash.

JSON does not allow trailing commas.

JSON:

person = {"firstName":"John", "lastName":"Doe", "age":46}

JSON:

points = [40, 100, 1, 5, 25, 10];

Undefined is Not Null

JavaScript objects, variables, properties, and methods can be undefined.

In addition, empty JavaScript objects can have the value null.

This can make it a little bit difficult to test if an object is empty.

You can test if an object exists by testing if the type is undefined:

Example:

if (typeof myObj === "undefined")

But you cannot test if an object is null, because this will throw an error if the object is undefined:

Incorrect:

if (myObj === null)

To solve this problem, you must test if an object is not null, and not undefined.

But this can still throw an error:

Incorrect:

if (myObj !== null && typeof myObj !== "undefined")

Because of this, you must test for not undefined before you can test for not null:

Correct:

if (typeof myObj !== "undefined" && myObj !== null)