**Primitive types:**

1.Undefined – does not have properties

2.Null - does not have properties

3.String

4.Boolean

5.Number

**Non primitive type** = **Object type** (including functions, array)

The main difference between primitive and object is that primitive **values** are immutable, it is not possible to change their value, not their reference variable name.

var a = 10; // 10

a.str(2) – does not change

a=typeof a

there are tree values. And it does not change value of variable a.

but in objects it is totally different. Objects are mutable, it is possible to modify or add or delete their values.

Var person1={“name”:”Huseyn”};

Var person2 = {“name”:”Huseyn”};

Person2=person1;

Person2.name=”Huseyn2”;

console.log(person2.name) // as expected Huseyn2.

But console.log(person1.name)// Huseyn2. Because objects are mutable values. They are bound with reference types.

In above mentioned example, everything (values, property names, and order) are the same. But when JS compares it gives false. : console.log(person1===person2) // false.

**Global object**

When the JavaScript interpreter starts (or whenever a web browser loads a new page), it creates a new global object and gives it an initial set of properties that define:

• global properties like undefined, Infinity, and NaN

• global functions like isNaN(), parseInt() alert()

• constructor functions like Date(), RegExp(), String(), Object(), and Array() (§3.8.2)

• global objects like Math and JSON (§6.9)

**Wrapper object**

Var a=”123”;

Console.log(a.length) // 3

Strings are not objects, though, so why do they have properties? Whenever you try to refer to a property of a string s, JavaScript converts the string value to a temporarily object as if by calling **new String(s)**. Once it has created, it is discarded.

Properties of primitive types are only readonly, it is not possible to add or remove a property of these.

Var a=”123”;

a.testproperty=”test”;

Console.log(typeof a) // string

Console.log(a.testproperty) // undefined.

Because it is already discarded. And as I mentioned above it is only readonly.

Var b= new String(“asdad”);

b.test=”123123”;

console.log(b.test) // 123123

Console.log(typeof b) // object.

For b object it is possible to add new properties, because already it is not acting as string, acting as Object.

**Function scope and Hoisting**

Unlike other programming languages, JS does not have block scope, instead of this it has function scope. So,

Block scope: Each block of code within curly braces has its own scope, and variables are not visible outside of the block in which they are declared.

Function scope: variables are visible within the function in which they are defined and within any functions that are nested within that function.

function test(o) {

var i = 0;// i is defined throughout function

if (typeof o == "object") {

var j = 0; // j is defined everywhere, not just block

for(var k=0; k < 10; k++) { // k is defined everywhere, not just loop

console.log(k); // print numbers 0 through 9

}

console.log(k); // k is still defined: prints 10

}

console.log(j); // j is defined, but may not be initialized

}

So, all variables(I,j,k,) are accessible in everywhere within function. It is function scope.

This means that variables are even visible before they are declared. But of course, if they are not declared, they will return *undefined*

*Hoisting*

Console.log(a); undefined, not error.

Var a=10;

Hoisting creates **all possible variable** in the top of a function.

**Variables As Properties**

When we create a global variable in Javascript, it is actually part(property) of global object (this, window). As we know, it is possible to delete property of an object with simply a delete command. That’s why it is possible to delete global variable because of this is part of global object.

But for local variables it is not possible to delete them.

var truevar = 1; // A properly declared global variable, nondeletable.

fakevar = 2; // Creates a deletable property of the global object.

this.fakevar2 = 3; // This does the same thing.

delete truevar // => false: variable not deleted and it does not give any error. Just could not delete.

delete fakevar // => true: variable deleted

delete this.fakevar2 // => true: variable deleted