# JavaScript Primitive vs. Reference Values

**Summary**: in this tutorial, you'll learn about two different types of values in JavaScript including primitive and reference values.

JavaScript has two different types of values:

- Primitive values
- Reference values

Primitive values are atomic pieces of data while reference values are objects that might consist of multiple values.

## Stack and heap memory

When you declare variables, the JavaScript engine allocates the memory for them on two memory locations: stack and heap.

Static data is the data whose size is fixed at compile time. Static data includes:

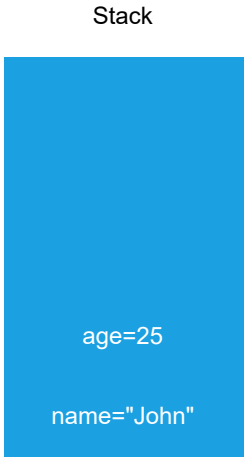
- Primitive values (null, undefined, boolean, number, string, symbol, and BigInt)
- Reference values that refer to objects.

Since static data has a size that does not change, the JavaScript engine allocates a fixed amount of memory space to the static data and stores it on the stack.

For example, the following declares two variables and initializes their values to a literal string and a number:

```
let name = 'John';
let age = 25;
```

Since `name` and `age` are primitive values, the JavaScript engine stores these variables on the stack as shown in the following picture:



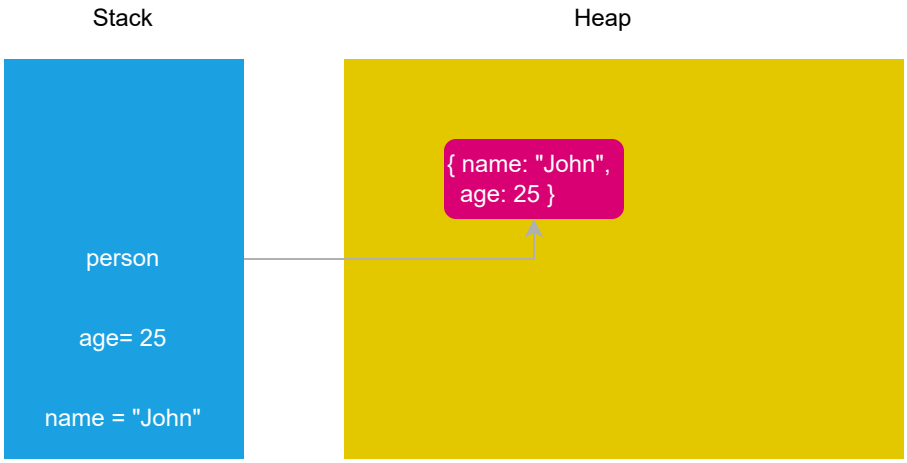

> Note that strings are objects in many programming languages, including Java and C#. However, strings are primitive values in JavaScript.

Unlike the stack, JavaScript stores objects (and functions) on the heap. The JavaScript engine doesn't allocate a fixed amount of memory for these objects. Instead, it'll allocate more space as needed.

The following example defines the `name`, `age`, and `person` variables:

```
let name = 'John';
let age = 25;

let person = {
  name: 'John',
  age: 25,
};
```

Internally, the JavaScript engine allocates the memory to these variables as shown in the following picture:




In this picture, JavaScript allocates memory on the stack for the three variables `name`, `age`, and `person`.

The JavaScript engine creates a new object on the heap memory and links the `person` variable on the stack memory to the object on the heap memory.

Because of this, we say that the `person` variable is a *reference* to an object.

## Dynamic Properties

A reference value allows you to add, change, or delete properties at any time. For example:

```
let person = {
  name: 'John',
  age: 25,
};

// add the ssn property
person.ssn = '123-45-6789';

// change the name
person.name = 'John Doe';

// delete the age property
delete person.age;


console.log(person);
```

Output:

```
{ name: 'John Doe', ssn: '123-45-6789' }
```

Unlike reference values, primitive value cannot have properties.

If you attempt to add a property to a primitive value, it won't take any effect. For example:

```
let name = 'John';
name.alias = 'Knight';

console.log(name.alias); // undefined
```

Output:

```
undefined
```

In this example, we add the `alias` property to the `name` primitive value. But when we access the `alias` property via the `name` primitive value, it returns `undefined` .

## Copying values

When you assign a primitive value from one variable to another, the JavaScript engine creates a copy of that value and assigns it to the variable. For example:

```
let age = 25;
let newAge = age;
```

In this example:

- First, declare a new variable `age` and initialize its value to `25` .
- Second, declare another variable `newAge` and assign the `age` to the `newAge` variable.

Behind the scenes, the JavaScript engine creates a copy of the primitive value `25` and assign it to the `newAge` variable.

The following picture illustrates the stack memory after the assignment:


On the stack memory, the `newAge` and `age` are separate variables. If you change the value of one variable, it won't affect the other.

For example:

```
let age = 25;
let newAge = age;

newAge = newAge + 1;
console.log(age, newAge);
```


When you assign a reference value from one variable to another, the JavaScript engine creates a reference so that both variables refer to the same object on the heap memory. This means that if you change one variable, it'll affect the other.

For example:

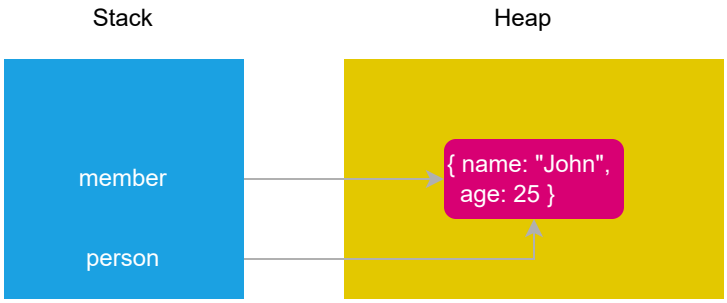
```
let person = {
  name: 'John',
  age: 25,
};

let member = person;

member.age = 26;

console.log(person);
console.log(member);
```

How it works.

First, declare a `person` variable and initialize its value with an object with two properties `name` and `age` :

```
let person = {
  name: 'John',
  age: 25,
};
```

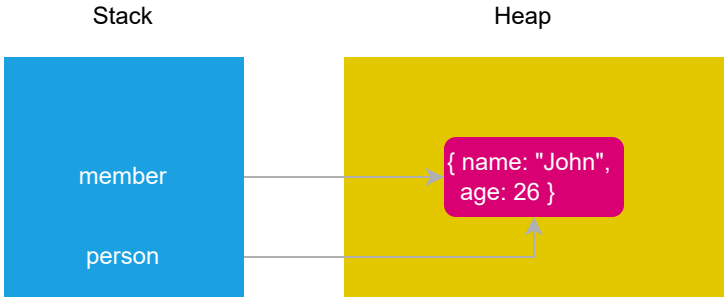
Second, assign the `person` variable to the `member` variable. In the memory, both variables reference the same object, as shown in the following picture:




```
let member = person;
```

Third, change the `age` property of the object via the `member` variable:

```
member.age = 26;
```




Since both `person` and `member` variables reference the same object, changing the object via the `member` variable is also reflected in the `person` variable.

## Quiz

## Summary

- Javascript has two types of values: primitive and reference values.
- You can add, change, or delete properties to a reference value, whereas you cannot do it with a primitive value.
- Copying a primitive value from one variable to another creates a separate value copy, meaning that changing the value in one variable does not affect the other.
- Copying a reference from one variable to another creates a reference so that two variables refer to the same object. This means that changing the object via one variable reflects in another variable.