```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>closure</title>
        <script src="close.js"></script>
    </head>
    <body>
        <h1>closure</h1>
        <p>
        closure is ...<br>
            when a function is able to remember and access its scope even when being executed out of scope.
        </p>
    </body>
</html>
```

closure

closure is ...
when a function is able to remember and access its scope even when being executed out of scope.

```
window.onload=function(){

    function foo(){
        /*scope of foo*/
        var a=42;

        function bar(){
        /*scope of bar, which also can access foo()*/
            console.log(a);
        }
        return bar;
    }

    console.log(foo());

    foo();  /*bar() wird so übrigens nicht ausgeführt, sondern lediglich
    referenziert*/

    var barScope=foo();
    console.log(barScope);
    /*barScope enthält bar()*/

    barScope();
    /*jetzt wird bar() ausgeführt, obwohl wir gar nicht im scope von bar() sind*/

    /*das geht so nicht
    bar();*/

    /*das geht auch nicht
    console.log(a);*/

    /*Immer, wenn functions als values weitergereicht werden, nehmen sie ihren
    scope mit und wir haben closure*/

    function wrap(n){
        var localN=n;
        return function(){
            console.log('localN='+localN);
            return localN;}
    /*function() closes over wrap*/
    }

    var wrap1=wrap(2);
    console.log(wrap1);
    var wrap2=wrap(7);
    console.log(wrap2);
    wrap1(); //merkt sich 2 als localN
    wrap2(); //merkt sich 7 als localN

    /*Praxisbeispiel*/
    function multiplier(factor){
        console.log('factor='+factor);
        return function(number){
            console.log('number='+number);
            return number*factor;
        };
    }

    console.log(multiplier(2));
    /*2 wandert in die Variable factor*/

    var multiply=multiplier(2);
    console.log(multiply(5));
    /*alternativ*/
    console.log(multiplier(2)(5));
}
```