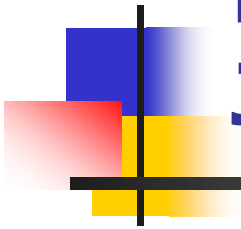
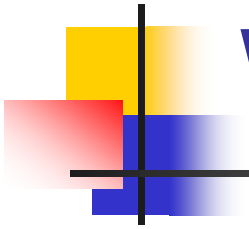


IF2032
Pemrograman Berorientasi Objek
Javascript - Function

Achmad Imam Kistijantoro (imam@informatika.org)
Yani Widyani (yani@stei.itb.ac.id)
Februari 2009
Informatika – STEI – ITB

Sumber: Object-Oriented JavaScript, Create scalable, reusable high-quality JavaScript applications, and libraries, **Stoyan Stefanov, 2008**



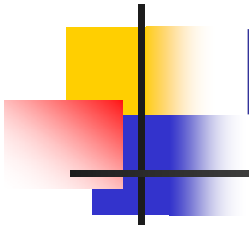
What is a function

- Functions allow you group together some code
  Example:

```
function sum(a, b) {
    var c = a + b;
    return c;
}
```

- Calling a function:

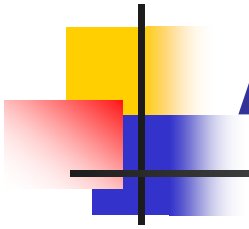
```
>>> var result = sum(1, 2);
>>> result;
3
```



Parameters

- A function may not require any parameters, but if it does and you forget to pass them, JavaScript will assign the value undefined

```
>>> sum(1)
NaN

>>> sum(1, 2, 3, 4, 5)
3
```

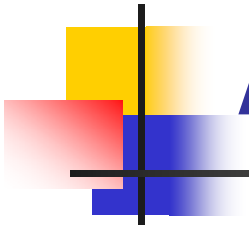


Arguments array

```
>>> function args() { return arguments; }
>>> args();
[]
>>> args( 1, 2, 3, 4, true, 'ninja');
[1, 2, 3, 4, true, "ninja"]
```

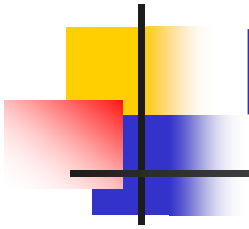
- By using the arguments array you can improve the sum() function to accept any number of parameters and add them all up.

```
function sumOnSteroids() {
var i, res = 0;
var number_of_params = arguments.length;
for (i = 0; i < number_of_params; i++) {
res += arguments[i];
}
return res;
```



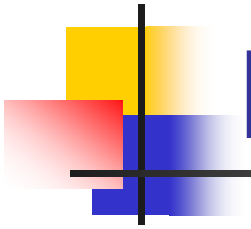
Arguments array (2)

```
>>> sumOnSteroids(1, 1, 1);
3

>>> sumOnSteroids(1, 2, 3, 4);
10

>>> sumOnSteroids(1, 2, 3, 4, 4, 3, 2, 1);
20

>>> sumOnSteroids(5);
5

>>> sumOnSteroids();
0
```



Pre-defined functions

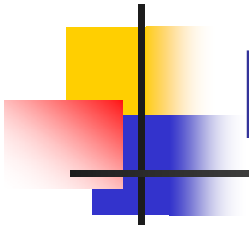
- parseInt()
- parseFloat()
- isNaN()
- isFinite()
- Encode/Decode URIs
- eval()
- alert()



parseInt()

```
>>> parseInt('123')
123

>>> parseInt('abc123')
NaN

>>> parseInt('1abc23')
1

>>> parseInt('123abc')
123
```

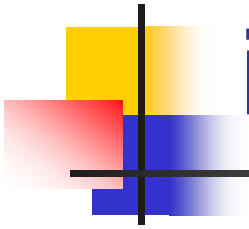
```
>>> parseInt('FF', 10)
NaN

>>> parseInt('FF', 16)
255

>>> parseInt('0377', 10)
377

>>> parseInt('0377', 8)
255
```



parseFloat()

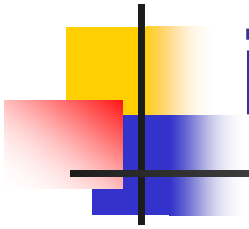
```
>>> parseFloat('123')
123

>>> parseFloat('1.23')
1.23

>>> parseFloat('1.23abc.00')
1.23

>>> parseFloat('a.bc1.23')
NaN

>>> parseFloat('a123.34')
NaN
```

```
>>> parseFloat('12a3.34')
12

>>> parseFloat('123e-2')
1.23

>>> parseFloat('123e2')
12300

>>> parseInt('1e10')
1
```



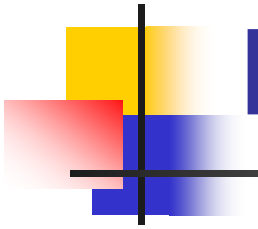
isNaN()

```
>>> isNaN(NaN)
true

>>> isNaN(123)
false

>>> isNaN(1.23)
false

>>> isNaN(parseInt('abc123'))
true

>>> isNaN('1.23')
false

>>> isNaN('a1.23')
true
```



isFinite()

```
>>> isFinite(Infinity)
false

>>> isFinite(-Infinity)
false

>>> isFinite(12)
true

>>> isFinite(1e308)
true

>>> isFinite(1e309)
false
```

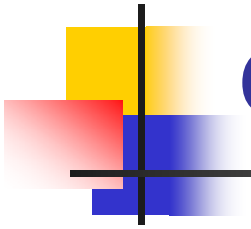


Encode/Decode URIs

- In a URL (Uniform Resource Locator) or a URI (Uniform Resource Identifier), some characters have special meanings.
- If you want to "escape" those characters, you can use the functions encodeURI() or encodeURIComponent().
  - The first one will return a usable URL,
  - The second one assumes you're only passing a part of the URL, like a query string for example, and will encode all applicable characters.

```
>>> var url = 'http://www.packtpub.com/scr ipt.php?q=this and that';
>>> encodeURI(url);
"http://www.packtpub.com/scr%20ipt.php?q=this%20and%20that"

>>> encodeURIComponent(url);
"http%3A%2F%2Fwww.packtpub.com%2Fscr%20ipt.php%3Fq%3Dthis%
20and%20that"
```

- The opposites of encodeURI() and encodeURIComponent() are decodeURI() and decodeURIComponent() respectively.
- Sometimes, in older code, you might see the similar functions escape() and unescape() but these functions have been deprecated and should not be used.



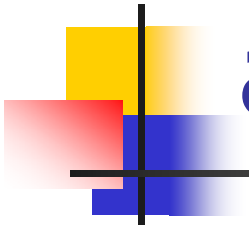
eval()

- eval() takes a string input and executes it as JavaScript code:

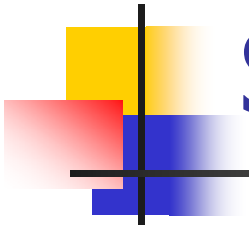
```
>>> eval('var ii = 2;')
>>> ii
2
```

- eval('var ii = 2;') is the same as simply var ii = 2;

- eval() can be useful sometimes, but should be avoided if there are other options
- "Eval is evil" is a mantra you can often hear from seasoned JavaScript programmers
- The drawbacks of using eval() are:
  - Performance—it is slower to evaluate "live" code, than to have the code directly in the script.
  - Security—JavaScript is powerful, which also means it can cause damage. If you don't trust the source of the input you pass to eval(), just don't use it.

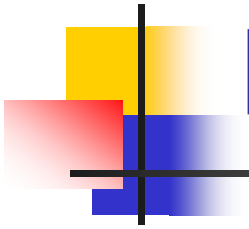


alert()



Scope of Variables

- variables in JavaScript are not defined in a block scope, but in a *function scope*
  - if a variable is defined inside a function, it's not visible outside of the function.
  - a variable defined inside an if or a for code block is visible outside the code block.
  - "global variables" describes variables you define outside of any function, as opposed to "local variables" which are defined inside a function



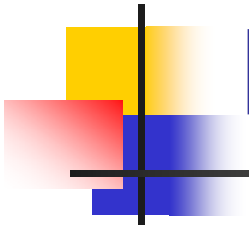
Functions are Data

- Functions in JavaScript are actually data
- The following two ways to define a function are exactly the same:

```
function f(){return 1;}
var f = function(){return 1;}
```

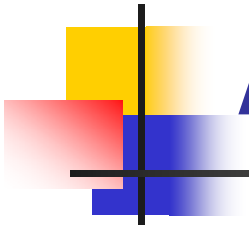
- The second way of defining a function is known as *function literal notation*
- When you use the typeof operator on a variable that holds a function value, it returns the string "function"

```
>>> function f(){return 1;}
>>> typeof f
"function"
```



Functions are Data (2)

```
>>> var sum = function(a, b) {return a + b;}
>>> var add = sum;
>>> delete sum

>>> typeof sum;
"undefined"

>>> typeof add;
"function"

>>> add(1, 2);
3
```



Anonymous functions

- It's ok to have pieces of data lying around your program:
  >>> "test"; [1,2,3]; undefined; null; 1;

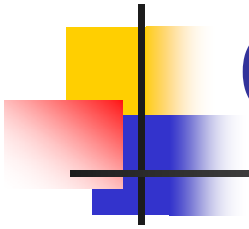
  → Anonymous data—anonymous because the data pieces are not assigned to any variable and therefore don't have a name

- Functions are like any other variable so they can also be used without being assigned a name:
  >>> function(a){return a;}

  → Anonymous function

- Two elegant uses for them:
  - You can pass an anonymous function as a parameter to another function. The receiving function can do something useful with the function that you pass.
  - You can define an anonymous function and execute it right away.

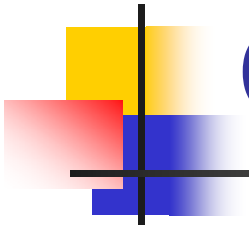
Callback functions



- Because a function is just like any other data assigned to a variable, it can be defined, deleted, copied, and *passed as an argument* to other functions

```
function invoke_and_add(a, b){
return a() + b();
}
function one() {
return 1;
}
function two() {
return 2;
}
>>> invoke_and_add(one, two);
3
```

- A is a callback function.
  - If A doesn't have a name, then you can say that it's an anonymous callback function.



Callback functions (2)

- Some of thhe benefits of the callback functions:
  - They let you pass functions without the need to name them (which means there are less global variables)
  - You can delegate the responsibility of calling a function to another function (which means there is less code to write)
  - They can help with performance

Callback - Example

```
function multiplyByTwo(a, b, c) {
var i, ar = [];
   for(i = 0; i < 3; i++) {
   ar[i] = arguments[i] * 2;
  }
return ar;
}
function addOne(a) {
return a + 1;
}
>>> multiplyByTwo(1, 2, 3);
[2, 4, 6]
>>> addOne(100)
101
>>> var myarr = [];
>>> myarr = multiplyByTwo(10, 20, 30);
[20, 40, 60]
>>> for (var i = 0; i < 3; i++) {myarr[i] =
      addOne(myarr[i]);}
>>> myarr
[21, 41, 61]
```

```
function multiplyByTwo(a, b, c, callback) {
var i, ar = [];
for(i = 0; i < 3; i++) {
ar[i] = callback(arguments[i] * 2);
}
return ar;
}
>>> myarr = multiplyByTwo(1, 2, 3, addOne);
[3, 5, 7]
>>> myarr = multiplyByTwo(1, 2, 3,
      function(a){return a + 1});
[3, 5, 7]
```

Anonymous functions are easy to change should the need arise:

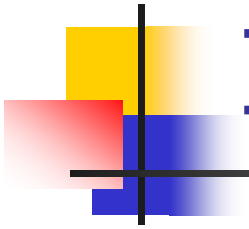
```
>>> myarr = multiplyByTwo(1, 2, 3,
      function(a){return a + 2});
[4, 6, 8]
```

Self-invoking functions

- Calling a function right after it was defined

```
(
function(){
alert('boo');
}
)()

(
function(name){
alert('Hello ' + name + '!');
}
)('dude')
```

- Using self-invoking anonymous functions will have some work done without creating global variables
  - You cannot execute the same function twice (unless you put it inside a loop or another function).
  - Best suited for one-off or initialization tasks.

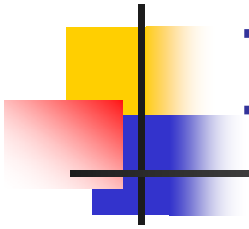


Inner (private) functions

```
function a(param) {
  function b(theinput) {
    return theinput * 2;
  };
  return 'The result is ' + b(param);
};
```

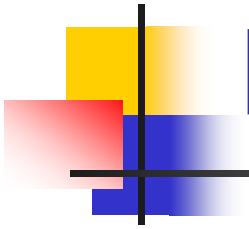
Using the function literal notation:

```
var a = function(param) {
var b = function(theinput) {
  return theinput * 2;
  };
return 'The result is ' + b(param);
};
```

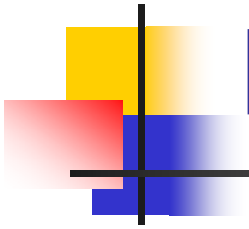


Inner (private) functions (2)

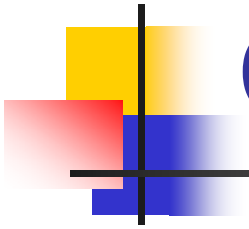
- The benefit of using private functions are as follows:
  - You keep the global namespace clean (smaller chance of naming collisions).
  - Privacy—you expose only the functions you decide to the "outside world", keeping to yourself functionality that is not meant to be consumed by the rest of the application.



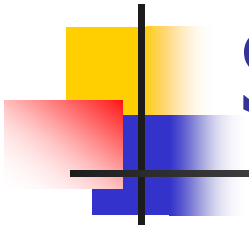
Functions that return functions



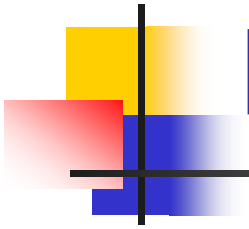
Functions, Rewrite Thyself!



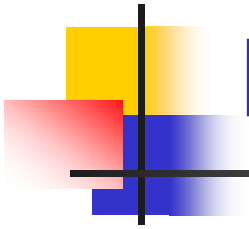
Closure



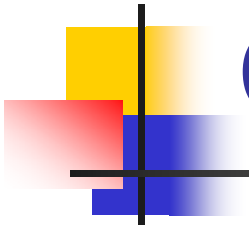
Scope Chain



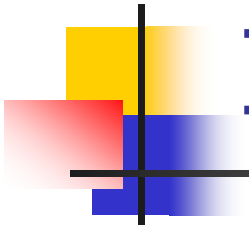
Lexical Scope



Breaking the Chain with Closure



Getter/Setter



Iterator