CS472 WAP
Lecture 5: Introduction to JavaScript

Actions Supported by All the Laws of Nature

Key JavaScript Concepts

- Key JavaScript Concepts
- JavaScript Syntax
- Program Logic
- Advanced JavaScript Syntax

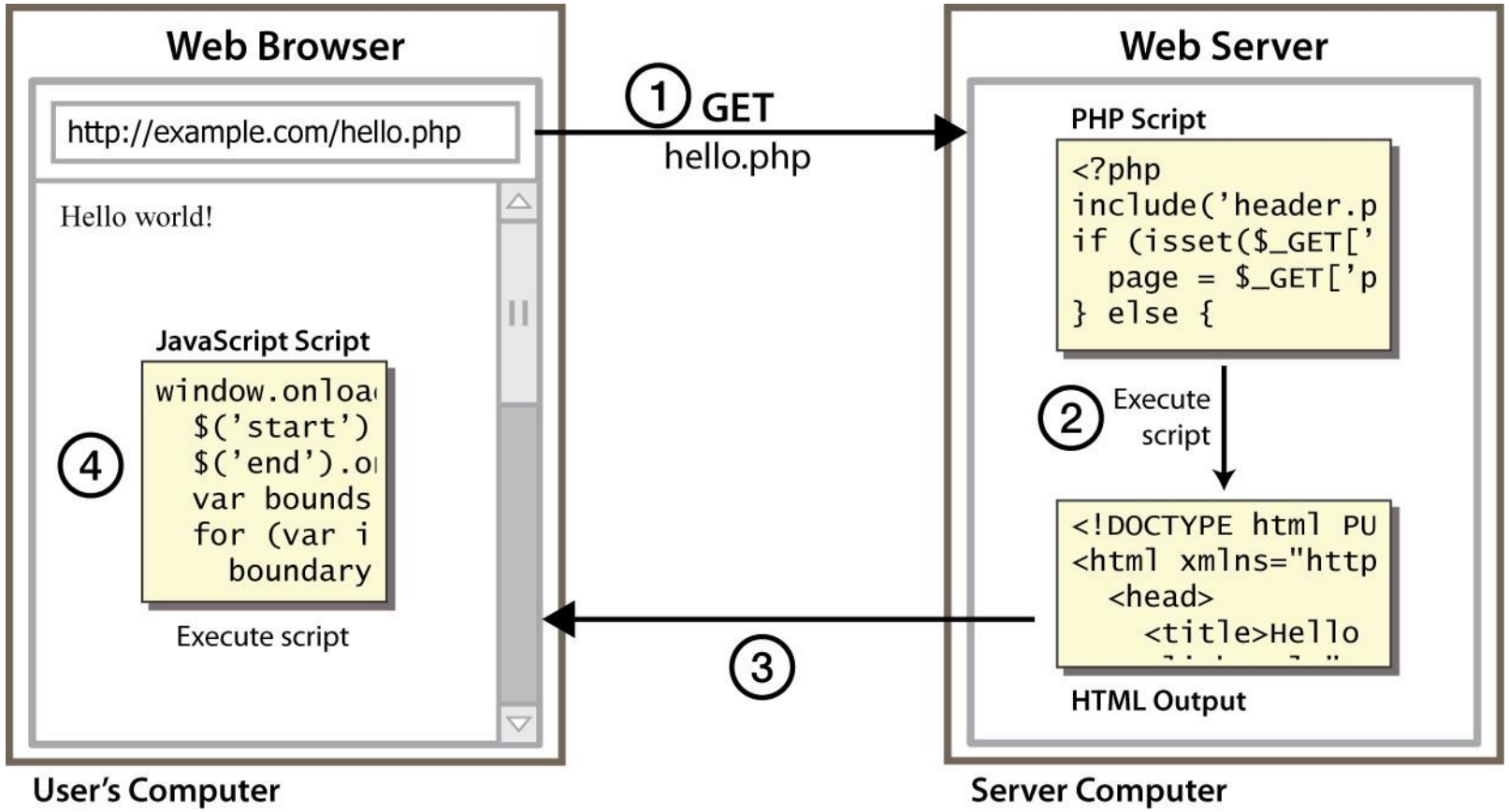
Main Point Preview

- JavaScript is a loosely typed language. It has types but does no compile time type checking. Programmers must be cautious of automatic type conversions, including conversions to Boolean types. It has a flexible and powerful array type as well as distinct types of null and undefined.

- **Science of Consciousness:** To be an effective JavaScript programmer one needs to understand the principles and details of the language. If our awareness is established in the source of all the laws of nature, then our actions will spontaneously be in accord with the laws of nature for a particular environment.

Client-side Scripting

- client-side script: code runs in browser *after* page is sent back from server
  - often this code manipulates the page or responds to user actions



Why use client-side programming?

- client-side scripting (JavaScript) benefits:
  - usability: can modify a page without having to post back to the server (faster UI)
  - efficiency: can make small, quick changes to page without waiting for server
  - event-driven: can respond to user actions like clicks and key presses

What is JavaScript?

- a lightweight programming language ("scripting language")

- used to make web pages interactive
  - insert dynamic text into HTML (ex: username)
  - react to events (ex: page load user click)
  - get information about a user's computer (ex: browser type)
  - perform calculations on user's computer (ex: form validation)

- NOT related to Java other than by name and some syntactic similarities

Your first JS file

To get JS Engine starts in your browser, all you need is to add the following code:

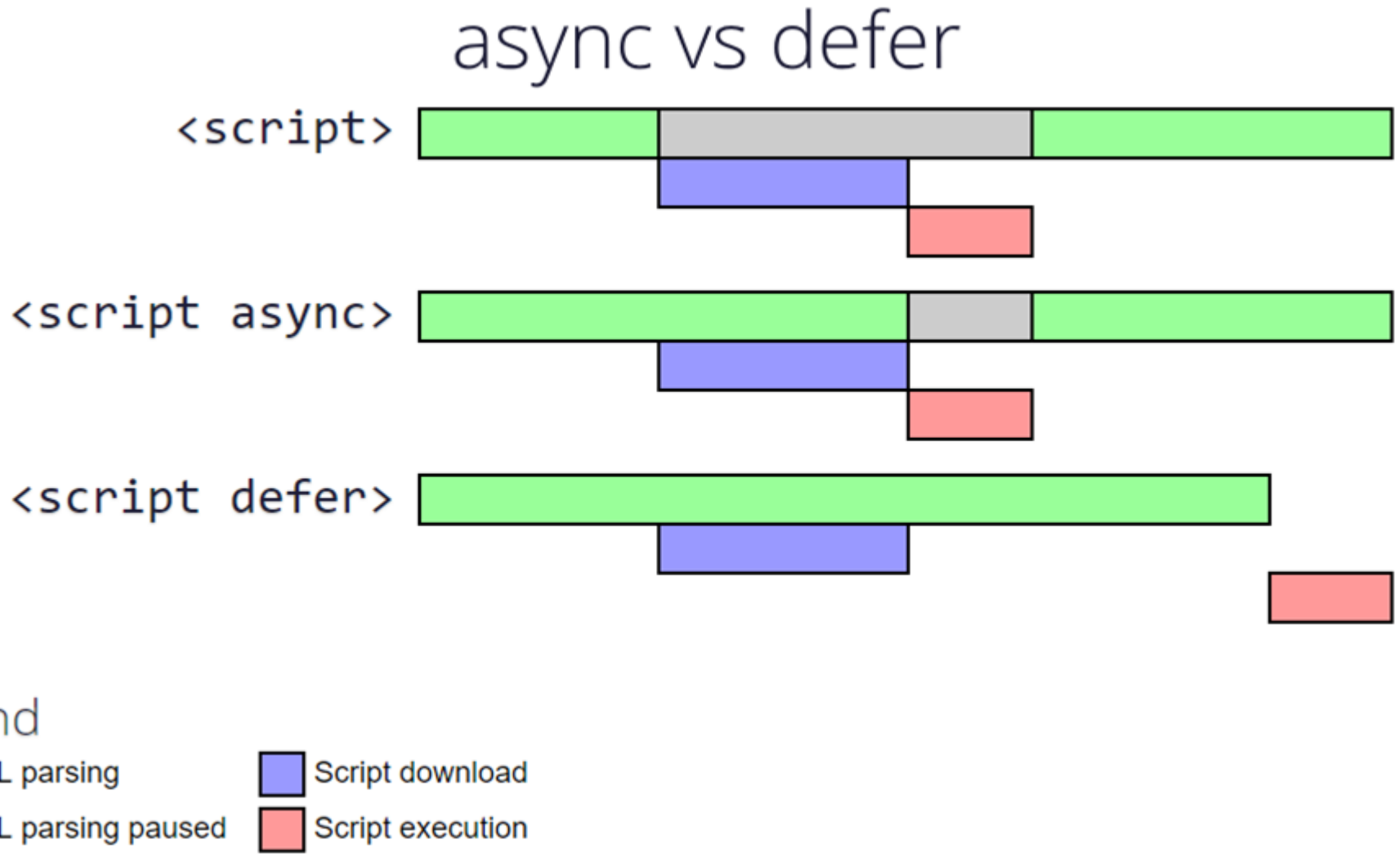
```
<script src="script.js" type="text/javascript"></script>
```

- The JS Engine will create all the **global objects** along with "**`this`**"
- All your code (variables and functions) will be attached to the global object **`window`**

What will happened if we have two or more scripts included? They all run as one single file!

```
<script src="script1.js" type="text/javascript"></script>
<script src="script2.js" type="text/javascript"></script>
```

Your first JS file



The modern mode, "use strict"

Strict mode **makes it easier to write "secure" JavaScript**.

Strict mode changes previously accepted "bad syntax" into real errors.

Examples

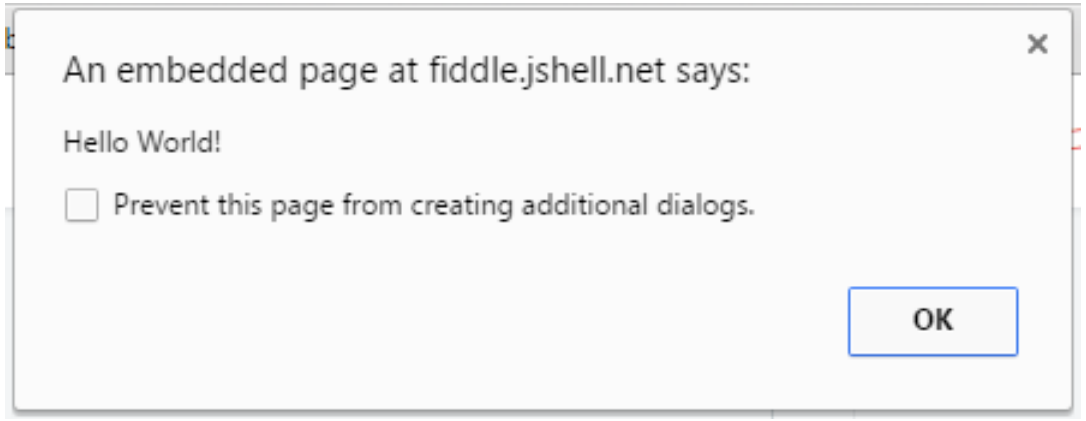
The strict mode in JavaScript does **not** allow following things:

1. Use of undefined variables
2. Use of reserved keywords as variable or function name
3. Duplicate properties of an object
4. Duplicate parameters of function
5. Assign values to read-only properties
6. Modifying arguments object
7. Octal numeric literals
8. with statement

Ref: https://www.tutorialsteacher.com/javascript/javascript-strict

A JavaScript statement: alert

```
alert("Hello World!");
```



- A JS command that pop up a dialog box with a message

8 basic data types in JavaScript

Seven primitive data types:

1. **number** for numbers of any kind: integer or floating-point.
2. **string** for strings.
    1. A string may have one or more characters, there's no separate single-character type.
3. **boolean** for true/false.
4. **null** for unknown values – a standalone type that has a single value null.
5. **undefined** for unassigned values – a standalone type that has a single value undefined.
6. **symbol** for unique identifiers.
7. **bigInt** for integers larger than 2**53

Non-primitive data type:

- **object** for more complex data structures.
- Arrays and functions are objects

- The **typeof** operator allows us to see which type is stored in a variable.
    - Two forms: typeof x or typeof(x).
    - Returns a string with the name of the type, like "string".
    - For null returns "object" – this is an error in the language, it's not actually an object.

Variables and types

```
var name = expression;
let name = expression; (ES6)
const name = expression; (ES6)

var age = 32;
let weight = 127.4;
const clientName = "Connie Client";
```

- variables are declared with the `var/let/const` keyword (case sensitive)
- types are not specified, but JS does have types ("loosely typed")
  - Number, Boolean, String, Null, Undefined, Symbol, Object
  - can find out a variable's type by calling typeof

dynamic (loose) typing

- Dynamic typing

- JavaScript is a loosely typed or a dynamic language. Variables in JavaScript are not directly associated with a specific value type, and any variable can be assigned (and re-assigned) values of all types:

```
let   foo = 42;     // foo is now a number
foo = 'bar';         // foo is now a string
foo = true;        // foo is now a boolean
```

Assigning values to a variable

- When variables are declared, their default value is 'undefined’

```
let total;
console.log(total) //undefined

total = 5;
console.log(total) // 5

total = total + 10;
console.log(total); // 15
```

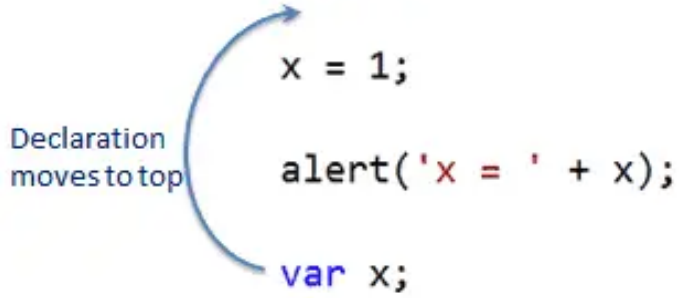
Scope of variables

- The scope of a variable determines how long and where a variable can be used.
- When const or let keywords are used, scope is within the block

```
let x = 5;
console.log(x);
if(x==5){
      let y = 2*x;
      console.log(y);
      console.log(x); // x is accessible here.
}
console.log(x);
console.log(y); // y is not accessible here.
```

- Declare y using `var` keyword in above code and see the change in output.

Hoisting



Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope.

Variables defined with **var** have function-scoped or global-scoped if declared outside function.
Variables defined with **const, let** have block-scoped.

Variables defined with **var** are hoisted to the top and can be initialized at any time.

Variables defined with **let**, **const** are also hoisted to the top of the block, but not initialized.

> Meaning: Using a let variable before it is declared will result in a ReferenceError.
> The block of code is aware of the variable, but it cannot be used until it has been declared.

null and undefined

- special values `null` and `undefined` are special types as well.
- **`null`** is not a “reference to a non-existing object” or a “null pointer” like in Java
  - special value which represents “nothing”, “empty” or “value unknown”.
- The meaning of **`undefined`** is “value is not assigned”.
  - If a variable is declared, but not assigned, then its value is `undefined`.
- Programmers assign null; the compiler assigns undefined

```
let name = null;
let age;
console.log(name,age) // null, undefined
```

Constants (ES6)

- also known as "immutable variables"
  - cannot be re-assigned new content.
- only makes the variable itself immutable, not its assigned content
  - object properties can be altered
  - array elements can be altered

- ```
  const pi = 3.1415926;
  pi = 3.14; //error
  ```
  ```

Number

- The *number* type represents both integer and floating-point numbers.
- Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: `Infinity`, `-Infinity` and `NaN`.

```
let n = 123;
n = 12.345;
console.log( 1 / 0 ); // Infinity
console.log( "not a number" / 2 ); // NaN
```

- `BigInt` type was added to the language to represent integers of arbitrary length.

Number Type

```
let enrollment = 99;
let medianGrade = 2.8;
let credits = 5 + 4 + (2 * 3);
```

- integers and real numbers are the same type (no int vs. double)
- same operators: + - * / % ++ -- = += -= *= /= %=
- similar precedence to Java
- many operators auto-convert types: "2" * 3 is 6
  - What is “2” + 3 ?

Numeric Conversion

- Numeric conversion happens in mathematical functions and expressions automatically.
  - For example, when division / is applied to non-numbers:

```
alert( "6" / "2" ); // 3, strings are converted to numbers
```

  - We can use the Number(value) function to explicitly convert a value to a number:

```
let str = "123.33";
let num = Number(str); // becomes a number 123.33
num = parseFloat(str); // becomes a number 123.33
num = parseInt(str); // becomes a number 123
```

  - Favor Number unless have a specific need for parseInt/parseFloat

Numeric conversion rules

Value	Becomes…
undefined	NaN
null	0
true and false	1 and 0
string	Whitespaces from the start and end are removed. If the remaining string is empty, the result is 0. Otherwise, the number is “read” from the string. An error gives NaN.

```
console.log( Number("   123   ") ); // 123
console.log( Number("123z") );      // NaN (error reading a number at "z")


console.log( Number(true) );        // 1
console.log( Number(false) );       // 0
```

Boolean Type

The `boolean` type has only two values: `true` and `false`.

```
let isLazy = false;
let isHealthy = true;
```

```
let iLikeWebApps = true;
if ("web dev is great") { /* true */ }
if (0) { /* false */ }
```

- any value can be used as a Boolean
  - "falsey" values: false, 0, 0.0, NaN, empty String(""), null, and undefined
  - "truthy" values: anything else, include objects

- !! Idiom – gives boolean value of any variable
  - const x=5;
  - console.log(!x);
  - console.log(x);
  - console.log(!!x);

Boolean Conversion

- Boolean conversion is the simplest one.
  - It happens in logical and relational operations (covered later)
  - But can also be performed explicitly with a call to Boolean(value).
- The conversion rule:
  - Values that are intuitively “empty”, like 0, an empty string, null, undefined, and NaN, become false.
  - Other values become true.

```
console.log( Boolean(1) ); // true
console.log( Boolean(0) ); // false

console.log( Boolean("hello") ); // true
console.log( Boolean("") ); // false
```

Logical Operators

- >, <, >=, <=, &&, ||, ! ==, !=, ===, *!*==
- most logical operators automatically convert types:
  - 5 < "7" is true
  - 42 == 42.0 is true
  - "5.0" == 5 is true
- === and !== are **strict** equality tests; checks both type and value
  - "5.0" === 5 is false
- Always use **strict** equality

Comments (same as Java)

```
// single-line comment
/* multi-line comment */
```

- identical to Java's comment syntax
- recall: 4 comment syntaxes
  - HTML: <!-- *comment* -->
  - CSS/JS/PHP: /* *comment* */
  - Java/JS/PHP: // *comment*
  - Python: # *comment*

String Type

```
let str = "Hello";
let str2 = 'Single quotes are ok too (but we prefer double)’;
let phrase = `can embed another ${str} in a backtick quote`; //backtick
```

```
let s = "Connie Client";
let fName = s.substring(0, s.indexOf(" "));  // "Connie"
let len = s.length; // 13
```

- methods: charAt, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase
  - charAt returns a one-letter String (there is no char type)

- length property (not a method as in Java)
- concatenation with + : 1 + 1 is 2, but "1" + 1 is "11"

More about String

- escape sequences behave as in Java: \' \" \& \n \t \\
- to convert between numbers and Strings:

```
let count = 10;
let s1 = "" + count; // "10"
let s2 = count + " bananas, ah ah ah!"; // "10 bananas, ah ah ah!"
const n1 = parseInt("42 is the answer"); // 42
const n2 = parseFloat("booyah"); // NaN
```

- to access characters of a String, use [*index*] or charAt:

```
const firstLetter = s[0];
let firstLetter = s.charAt(0);
let lastLetter = s.charAt(s.length - 1);
```

Searching for a substring

- look for the substr in str,
  - starting from the given position pos,
  - returns the position where the match was found or -1 if nothing can be found.

```
let str = 'Widget with id';
console.log( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning
console.log( str.indexOf('widget') ); // -1, not found, the search is case-sensitive
console.log( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)
```

- includes, startsWith, endsWith
  - str.includes(substr, pos) returns true/false depending on whether str contains substr within.
  - right choice if we need to test for the match, but don't need its position:

```
console.log( "Widget with id".includes("Widget") ); // true
console.log( "Hello".includes("Bye") ); // false
```

if/else statement (same as Java)

```
if (condition) {
  statements;
} else if (condition) {
  statements;
} else {
  statements;
}
```

- Identical structure to Java's if/else statement
- JavaScript allows almost anything as a *condition*

for loop (same as Java)

```
for (initialization; condition; update){
 statements;
}

let sum = 0;
for (let i = 0; i < 100; i++) {
 sum = sum + i;
}

const s1 = "hello";
let s2 = "";
for (let i = 0; i < s.length; i++) {
 s2 += s1[i] + s1[i];
} // s2 stores "hheellllloo"
```

while loops (same as Java)

```
while (condition) {
  statements;
}

do {
  statements;
} while (condition);
```

- break and continue keywords also behave as in Java

Function declaration (function statement)

- The first line of function is called the signature, and it includes the keyword `function`, the function name and the optional parameter list.

```
function sum(num1, num2){
  return num1+num2;
}
```

```
function greet(){
  console.log("Hi, from a function");
}
```

- The statements inside a function are called the body of a function.
  - Function returns `undefined`, when return is not explicit.

Calling a function

- To call a function, write a function name followed by a set of parentheses;
- optionally passing matching arguments for the corresponding parameters.

```
let total = sum(5,5); // call to function sum

greet(); // call to function greet
```

Parameters vs Arguments

- Function parameters are the names of variables in the function definition.
- Function arguments are the actual values passed to the function

```
// function sum has two parameters num1 and num2
function sum(num1, num2){
  return num1+num2;
}

let total = sum(5,10); // arguments 5 and 10 for num1 and num2 respectively
```

Default values

- If an argument is not provided for a parameter, then its value becomes undefined.
- If we want to use a “default” value instead, then we can specify it after `=`

```
function sum(num1=0, num2=0){
  return num1+num2;
}
```

- What would be the result of calling `sum()` if default parameters were not assigned?
  - Is it even a valid call?

Returning a value

- A function can return a value to the calling code
- The directive `return` can be any place of the function.
  - When the execution reaches it, the function stops, and the value is returned to the calling code.
  - There may be many occurrences of `return` in a single function.
  - It is also possible to use `return` without a value. That causes the function to exit immediately.
- A function with an empty `return` or without it returns `undefined`

```
function oddEven(num){
  if (!num) return;
  if(num%2==0) return "Even";
  else return "Odd"
}
```

Local variables

- A variable declared inside a function is only visible inside that function.

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable
  console.log( message );
}
showMessage(); // Hello, I'm JavaScript!
console.log( message ); // <--
 Error! The variable is local to the function
```

Outer variables

- A function can access an outer variable as well, for example:

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

- function has full access to the outer variable. It can modify it as well.
- Avoid if possible
  - Breaks encapsulation
  - Sometimes necessary (closures)

Variable Shadowing

- If a same-named variable is declared inside the function, then it *shadows* the outer one.
  - For instance, in the code below the function uses the local `userName`. The outer one is ignored:
  - Shadowing is generally a bad practice since it can confuse humans

```
let userName = 'John’;

function showMessage() {
  let userName = "Bob"; // declare a local variable
  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

showMessage();

alert( userName ); // John, unchanged
```

Lexical scope in JavaScript (ES6+)

- From ES6, in JavaScript every block ({}) defines a scope
  - Via let and const

```
let x = 10;                                          Global Scope

function main() {
 let x;                                              Block Scope
 console.log("x1: " + x);
 if (x > 0) {
   let x = 30;          Block Scope
   console.log("x2: " + x);
 }

 x= 40;
 let f = function(x) { console.log("x3: " + x); }
 f(50);
}

main();
```

Function Expressions

- Can be Anonymous function
  - Widely used in JS with event handlers

```
const square = function(number) { return number * number };
const x = square(4) // x gets the value 16
```

- Can also have a name to be used inside the function to refer to itself //NFE (Named Function Expression)

```
const factorial = function fac(n) {
        return n < 2 ? 1 : n * fac(n - 1)
 };
console.log(factorial(3));
```

- Basically, a function expression is same syntax as a declaration, just used where an expression is expected

Function Signature

- If a function is called with missing arguments(less than declared), the missing values are set to : `undefined`

```
function f(x) {
 console.log("x: " + x);
}
f();
f(1);
f(2, 3);
```

No overloading!

```
function log() {
	console.log("No Arguments");
}
function log(x) {
	console.log("1 Argument: " + x);
}
function log(x, y) {
	console.log("2 Arguments: " + x + ", " + y);
}
log();
log(5);
log(6, 7);
```

- Why? Functions are objects!

`arguments` Object

The **arguments** object is an Array-like object corresponding to the arguments passed to a function.

```
function findMax() {
 let i;
 let max = -Infinity;
 for (i = 0; i < arguments.length; i++) {
      if (arguments[i] > max) {
            max = arguments[i];
      }
 }
 return max;
}
let max1 = findMax(1, 123, 500, 115, 66, 88);
let max2 = findMax(3, 6, 8);
```

Rest parameters (ES6)

rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function

```
function sum(x, y, ...more) {
    //"more" is array of all extra passed params
    let total = x + y;
    if (more.length > 0) {
        for (let i = 0; i < more.length; i++) {
            total += more[i];
        }
        console.log(total);
        return total;
    }
}
sum(5, 5, 5);
sum(7, 7, 7, 7, 7);
```

Arrow functions (ES6)

- Arrow functions can be a shorthand for an anonymous function in callbacks

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
          // equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter:
(singleParam) => { statements }
singleParam => { statements }

// A function with no parameters requires parentheses:
() => { statements }
```

Arrow functions

- Can be used in the same way as function expressions
  - More succinct, advantageous for short anonymous callbacks -- “one liners”
  - Fix language issue involving inner functions (will discuss with objects)

```
//function expression
let sum = function (a, b) { return a + b; };

//equivalent arrow function
let sum = (a, b) => a + b;

//only one argument, then parentheses around parameters can be omitted
let doubleV = x => 2 * x;

//no arguments, parentheses should be empty (but they should be present):
let sayHi = () => alert("Hello!");

//if body has { } brackets then must use return:
let sum = (a, b) => { return a + b; }
```

Arrow Functions Example

```
function multiply(num1, num2) {
      return num1 * num2;
}
var output = multiply(5, 5);
console.log("output: " + output);

var multiply2 = (num1, num2) => num1 * num2;
var output2 = multiply2(5, 5);
console.log("output2: " + output2);
```

Declaring an Array

- Using array literal syntax

```
const numbers = [];
const fruits = ["Apple", "Banana", "Mango"];
```

- can also be created using new keyword

```
const numbers = new Array(6);
```

- generally, use literal syntax

Using an Array

- Indices start with zero.
- get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

- replace an element

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

- add a new one

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

Size of an array

- built-in property, `length` represents current size of array
- total count of elements in array is its `length`

```
let numbers = []
console.log(numbers.length); // 0
numbers = [1,2,3];
console.log(numbers.length) // 3
```

Looping through an array

- Traditional way to cycle array items is the for loop over indexes:

```
let fruits = ["Apple", "Orange", "Pear"];

for (let i = 0; i < fruits.length; i++) {
  alert( fruits[i] );
}
```

- But for arrays there is another form of loop, `for..of`:

```
for (let element of fruits) {
  alert( element );
}
```

- The for..of doesn't give access to the index of the current element, just its value, but in most cases, that's enough.
  - shorter
  - avoids bugs that often occur from index errors at the end points
  - Favor for..of as default loop over arrays unless really need index

`toString`

- Arrays have their own implementation of `toString` method that returns a comma-separated list of elements.
- For instance:

```
let arr = [1, 2, 3];

console.log( arr ); // [1,2,3]
console.log( arr.toString() === '1,2,3' ); // true
```

Add/Remove elements To/From the beginning

- built in methods to add/remove elements to/from beginning of array.
  - `shift`: extracts the first element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];
console.log( fruits.shift() ); // remove Apple and alert it
console.log( fruits ); // Orange, Pear
```

  - `unshift`: add the element to the beginning of the array

```
let fruits = ["Orange", "Pear"];
fruits.unshift('Apple');
console.log( fruits ); // Apple, Orange, Pear
```

Add/Remove elements To/From the end

- add/remove elements to/from the end of the array.
  - `pop`: extracts the last element of the array and return it.

```
let fruits = ["Apple", "Orange", "Pear"];
console.log( fruits.pop() ); // remove "Pear" and log it
console.log( fruits ); // Apple, Orange
```

  - `push`: append element to the end of the array.

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
console.log( fruits ); // Apple, Orange, Pear
```

  - The call `fruits.push("Peach")` is equal to `fruits[fruits.length] = "Peach"`

Array methods summary

- To add/remove elements:
  - push(...items) – adds items to the end,
  - pop() – extracts an item from the end,
  - shift() – extracts an item from the beginning,
  - unshift(...items) – adds items to the beginning.
  - splice(pos, deleteCount, ...items) – at index pos delete deleteCount elements and insert items.
  - slice(start, end) – creates a new array, copies elements from position start till end (not inclusive) into it.
  - concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.
- To search among elements:
  - indexOf/lastIndexOf(item, pos) – look for item starting from position pos, return the index or -1 if not found.
  - includes(value) – returns true if the array has value, otherwise false.
  - find/filter(func) – filter elements through the function, return first/all values that make it return true.
  - findIndex is like find, but returns the index instead of a value.
- To iterate over elements:
  - forEach(func) – calls func for every element, does not return anything.
- To transform the array:
  - map(func) – creates a new array from results of calling func for every element.
  - sort(func) – sorts the array in-place, then returns it.
  - reverse() – reverses the array in-place, then returns it.
  - split/join – convert a string to array and back.
  - reduce(func, initial) – calculate a single value over the array by calling func for each element and passing an intermediate result between the calls.

splice

- The arr.splice(str) method is a swiss army knife for arrays.
  - It can do everything: insert, remove and replace elements.

```
arr.splice(index [, deleteCount, elem1, ..., elemN])
```

- It starts from the position index:
  - removes deleteCount elements and then
  - inserts elem1, ..., elemN at their place.
  - Returns the array of removed elements.

deletion:

```
let arr = ["I", "study", "JavaScript"];
arr.splice(1, 1); // from index 1 remove 1 element
console.log( arr ); // ["I", "JavaScript"]
```

splice (2)

remove 3 elements and replace them with the other two:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");
console.log(arr) // now ["Let's", "dance", "right", "now"]
```

splice returns the array of removed elements:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
// remove 2 first elements
let removed = arr.splice(0, 2);
console.log(removed); // "I", "study" <-- array of removed elements
```

insert the elements without any removals.

```
let arr = ["I", "study", "JavaScript"];
// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");
console.log(arr); // "I", "study", "complex", "language", "JavaScript"
```

concat

- returns new array that includes values from other arrays and additional items
  - accepts any number of arguments – either arrays or values.
  - result is a new array containing items from arr, then arg1, arg2 etc.
  - If an argument argN is an array, then all its elements are copied.
  - Otherwise, the argument itself is copied. arr.concat(arg1, arg2...)

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert(arr.concat([3, 4])); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert(arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert(arr.concat([3, 4], 5, 6)); // 1,2,3,4,5,6
```

map/filter/find/reduce are “pure” functions

- Important principle of “functional” programming

- Pure functions have no side effects
  - Do not change state information
  - Do not modify the input arguments

- Take arguments and return a new value

Iterate: forEach

- run a function for every element of the array.
  - result of the function (if it returns any) is thrown away and ignored
  - Intended for some side effect on each element of the array
    - print or alert or post to database

```
arr.forEach(function(item, index, array) {
 // ... do something with item
});
```

- shows each element of the array

```
 // for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(function(element){console.log(element)} );

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
     console.log (`${item} is at index ${index} in ${array}`);
});
```

filter

- Apply function to each item in array and return new array of all that pass the filter

```
let results = arr.filter(function(item, index, array) {
     // if true item is pushed to results and the iteration continues
     // returns empty array if nothing found
});
```

```
let users = ["John", "Pete", "Mary"]

// returns array
let someUsers = users.filter(item => !item.startsWith("J"));

console.log(someUsers); // 2
```

map

- one of the most useful and often used.
- calls function for each element and returns new array of results
- “map onto”
  - mathematical functions "map" a domain to a range
  - the passed function maps each array element to a transformed element

```
let result = arr.map(function(item, index, array) {
 // passed function returns the new value in place of each item
});
```

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
console.log(lengths); // [5,7,6]
```

sort(fn)

- default sort order is ascending and converts all arguments to strings
- sorts the array in place, changing its element order.
- returns sorted array, but the returned value is usually ignored, as arr itself is modified.

```
let arr = [2, 1, 15];
// the method reorders the content of arr
arr.sort();
console.log(arr); // [1, 15, 2]
```

To use our own sorting order, we need to supply a (comparator) function as the argument of arr.sort().

```
function compareNumeric(a, b) {
    if (a > b) return 1;    //a comes after b if 1
    if (a == b) return 0;
    if (a < b) return -1;   //a comes before b if -1
}

let arr = [2, 1, 15];
arr.sort(compareNumeric);
console.log(arr);   // [1, 2, 15]
```

reduce

calculate a single value based on the array.

```
let value = arr.reduce(function(accumulator, item, index, array) {
      // ...
}, [initial]);
```

The function is applied to all array elements one after another and “carries on” its result to the next call.

accumulator – is the result of the previous function call, equals initial the first time (if initial is provided).

item – is the current array item.

index – is its position.

array – is the array.

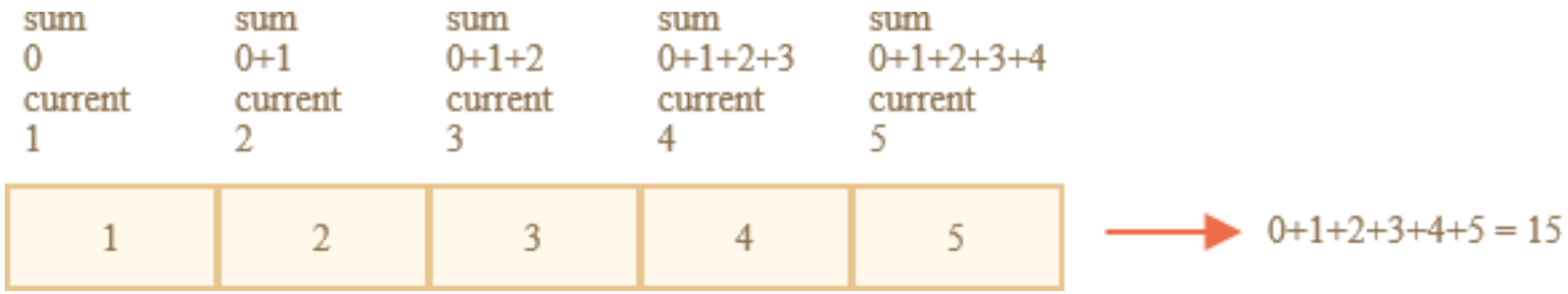
- first argument is the “*accumulator*” that stores the combined result of all previous execution.
  - at the end it becomes the result of reduce.

reduce continue…

How to get a sum of an array in one line:

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce(function (sum, current) { return sum + current; }, 0);
let result2 = arr.reduce((sum, current) => sum + current, 0);
console.log(result); // 15
console.log(result2); // 15
```

- On the first run, sum is the initial value = 0, and current is first array element = 1
- On the second run, sum = 1, we add the second array element (2) to it and return.
- On the 3rd run, sum = 3 and we add one more element to it, and so on…



Array methods: map, filter, reduce

```
//functional programming: map, filter, reduce can replace many loops
const a = [1, 3, 5, 3, 3];

//map all elements in an array to another set of values
const b = a.map(function (elem, i, array) { return elem + 3; })   // [4,6,8,6,6]

//select elements based on a condition
const c = a.filter(function (elem, i, array) {
    return elem !== 3;
});     //[1,5]

//find first element or index of first element satisfying condition
const d = a.find(function (elem) { return elem > 1; });           //3
const e = a.findIndex(function(elem) {return elem > 1;});         //1

//find a cumulative or concatenated value based on elements across the array
const f = a.reduce(function (prevVal, elem, currentIndex, array) {
        return prevVal + elem;
});     //15
```

Arrow Functions Example

Returns an Array containing all the array elements that pass the test. If no elements pass the test it returns an empty array.

```
const a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];

const a3 = a.filter(s => s.length > 7);
const a4 = a.find(s => s.length > 7);
const a5 = a.findIndex(s => s.length > 7);
```

Multidimensional arrays

- Arrays can have items that are also arrays.
  - We can use it for multidimensional arrays, for example to store matrices:

```
let myJSmatrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
  ];

  console.log(   myJSmatrix[1][1] ); // 5, the central element
```

Accessing elements with inner loop

```
for (let i = 0; i <  myJSmatrix.length; i++) {
    for (let j = 0; j <  myJSmatrix[i].length; j++) {
        console.log(myJSmatrix[i][j]);
    }
}
```

Never add a newline between return and the value

- **Semicolons** are (technically) '**optional**'
- For a long expression in return, it might be tempting to put it on a separate line, like this:

```
return
 (some + long + expression + or + whatever * f(a) + f(b))
```

- That doesn't work, because JavaScript assumes a semicolon after return. That'll work the same as:
  - effectively becomes an empty return

```
return;
 (some + long + expression + or + whatever * f(a) + f(b))
```

- If we want the returned expression to wrap across multiple lines, we should start it at the same line as return. Or at least put the opening parentheses there as follows:

```
return (
      some + long + expression + or + whatever * f(a) + f(b)
 )
```

Spread operator (ES6)

The same … notation can be used to unpack iterable elements (array, string, object) rather than pack extra arguments into a function parameter.

```
alert(Math.max(3, 5, 1)); // 5

let arr = [3, 5, 1];
alert(Math.max(arr)); // NaN
Math.max(arr[0], arr[1], arr[2]) //5

let arr = [3, 5, 1];
alert(Math.max(...arr));
//5 (spread turns array into a list of arguments)
```

Spread Operator syntax – for array

```
let initialNumbers = [0, 1, 2];
let newNumber = 15;
let updatedNumbers = [...initialNumbers, newNumber];
console.log(updatedNumbers); //[0,1,2,15]

//Example use-case for Spread syntax: Concatenate arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
let arr3 = arr1.concat(arr2); // arr3: [1,2,3,4,5,6]

//Use spread syntax:
let arr4 = [...arr1, ...arr2]; // arr4: [1,2,3,4,5,6]
```

Spread operator (ES6)

The same … notation can be used to unpack iterable elements (array, string, object) rather than pack extra arguments into a function parameter.

```
var a, b, c, d, e;
a = [1, 2, 3];
b = "dog";
c = [42, "cat"];

// Using the concat method.
d = a.concat(b, c); // [1, 2, 3, "dog", 42, "cat"]

// Using the spread operator.
e = [...a, b, ...c]; // [1, 2, 3, "dog", 42, "cat"]
```

What is destructuring assignment?

- Special syntax that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
const numbers = [10, 20];
let [a, b] = numbers;
console.log(a);
console.log(b);
```

- Benefits:
  - ‘Syntactic sugar’ to replace the following:
    - `let a = numbers[0];`
    - `let b = numbers[1];`

Destructuring assignment

- Unwanted elements of the array can also be thrown away via an extra comma:

```
const [first, , third] = ["foo", "bar","baz","foo"];
console.log(first);
console.log(third);
```

Creating objects via object literal

Syntax:

```
const objRef = {
    'fieldName': value,
    ...
    'fieldName': value
};

const pt = {
    'x': 4,
    'y': 3
};
alert(pt.x + ", " + pt.y);
```

- in JavaScript, you can create a new object without creating a class
- the above is like a Point object; it has fields named x and y
- the object does not belong to any class; it is the only one of its kind, a singleton
  - typeof(pt) === "object"

JavaScript objects

- Objects in JavaScript are like associative arrays
- the keys can be any string, different from variable names.
- you do not need quotes if the key is a valid JavaScript identifier
- values can be anything, including functions
- you can add keys dynamically using associative array or the . syntax
- object properties that have functions as their value are called ‘methods’

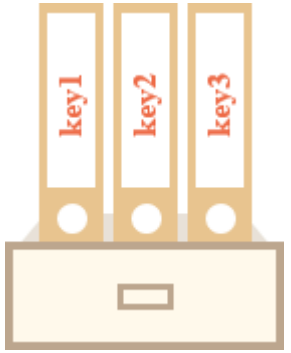
```
const x = {
  'a': 97,
  'b': 98,
  'c': 99,
  'd': 199,
  'mult': function(a, b) {
        return a * b;
  }
};
```

Objects: the basics

- primitive data types contain only a single thing
  - String, Number, Boolean, (BigInt, Symbol, undefined)

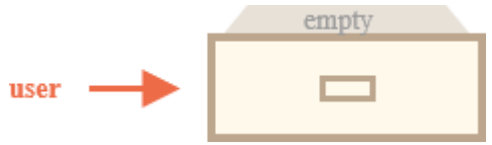
- objects store keyed collections of data and functions.

- imagine an object as a cabinet with signed files.
  - Every piece of data is stored in its file by the key.



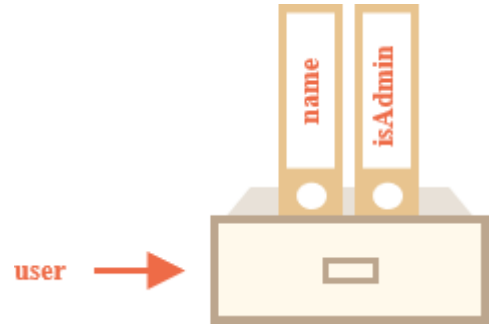
Object literals and properties

```
let userE = {};  // "object literal" syntax
```




```
let user = {          // an object
    name: "John",     // by key "name" store value "John"
    age: 30           // by key "age" store value 30
};

// get property values of the object:
alert(user.name); // John
alert(user.age); // 30
```



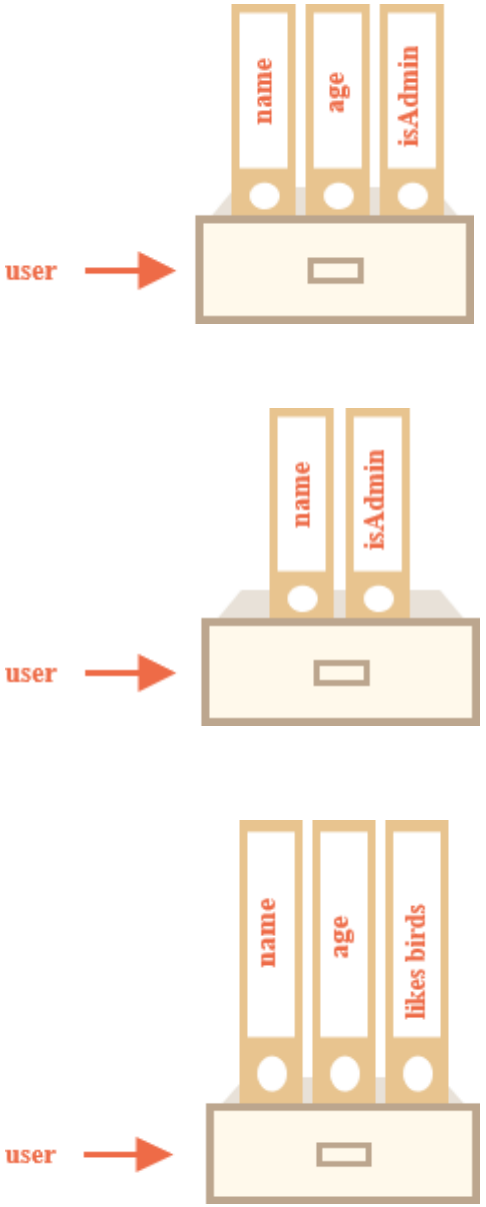
Adding and removing properties

```
user.isAdmin = true;
// properties can be dynamically created


delete user.age;


let user = {
       name: "John",
       "likes birds": true
       //multiword property name must be quoted
};

user.age = 30;  //dynamic creation
```



Square bracket notation

```
// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];


//Square brackets can obtain the property name from any expression like from a variable
let user = {
    name: "John",
    age: 30
};

let key = prompt("What do you want to know about the user?", "name");


// access by variable
alert( user[key] );   // John
```

Computed properties

- Variables or expressions can appear in square brackets and are called *computed properties*

```
let fruit = "apple";
let bag = {
    [fruit]: 5, // the name of the property is apple
};
alert(bag.apple); // 5
```

- can use more complex expressions inside square brackets:

```
let fruit = 'apple';
let bag = {
    [fruit + 'Computers']: 5    // bag.appleComputers = 5
};
```

- Square brackets more powerful than dot notation
  - also, more cumbersome
  - most of the time the dot is used

Property value shorthand

- often use existing variables as values for property names.

```
function makeUser(name, age) {
    return {
        name: name,
        age: age
    };
}
let user = makeUser("John", 30);
alert(user.name); // John
```

- In the example above, properties have the same names as variables. The use-case of making a property from a variable is so common, that there's a special property value shorthand to make it shorter.

```
function makeUser(name, age) {
    return {
        name, // same as name: name
        age     // same as age: age
    };
}
```

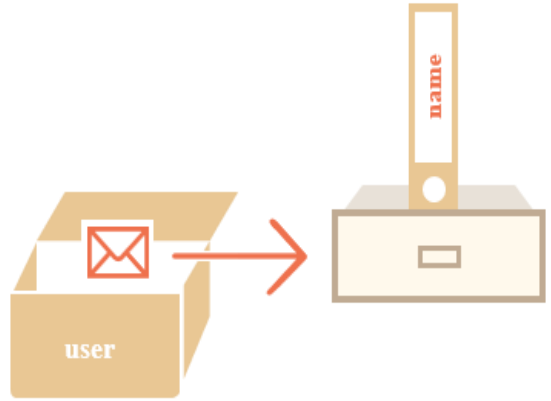
Variables as values versus references

- fundamental differences of objects vs primitives is that they are stored and copied “by reference”.
- Primitive values: strings, numbers, booleans – are assigned/copied “as a whole value”.

```
let message = "Hello!";
let phrase = message;
```

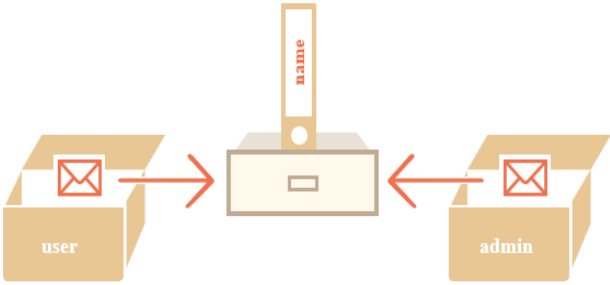
- A variable stores not the object itself, but its “address in memory”, in other words “a reference” to it.

```
let user = {
          name: "John"
};
```



- When an object variable is copied – the reference is copied, the object is not duplicated.

```
let user = { name: "John" };
let admin = user; // copy the reference
```



Existence check

- It is possible to access a property that doesn't exist!
  - Accessing a non-existing property just returns undefined.
  - Can test whether the property exists is if compare vs undefined:

```
let user = {};
console.log( user.age === undefined ); // true means "no such property"
```

- special operator "in" another way to check for the existence of a property
  - "key" in object
  - "key" must be a property name (or expression that evaluates to a property name)

```
let user = { name: "John", age: 30 };

console.log( "age" in user );    // true, user.age exists
console.log( "blabla" in user ); // false, user.blabla doesn't exist
```

Destructuring objects

- Destructuring on objects lets you bind variables to different properties of an object.
  - Order does not matter

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};
let { title, height, width } = options;
alert(title); // Menu
alert(width); // 100
alert(height); // 200
```

Destructure property to another name

- to assign a property to a variable with another name, set it using a colon

```
// { sourceProperty: targetVariable }
let { width: w, height: h, title } = options;

// width -> w
// height -> h
// title -> title

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

The “for…in” loop

- To walk over all keys of an object, there exists a special for loop:
  - different thing from the for .. of construct
    - Accesses property names instead of values
    - works with all objects

```
let user = {
    name: "John",
    age: 30,
    isAdmin: true
};

for (let key in user) {
    // keys
    console.log(key);   // name, age, isAdmin
    // values for the keys
    console.log(user[key]); // John, 30, true
}
```

easy to confuse with **for.. of**

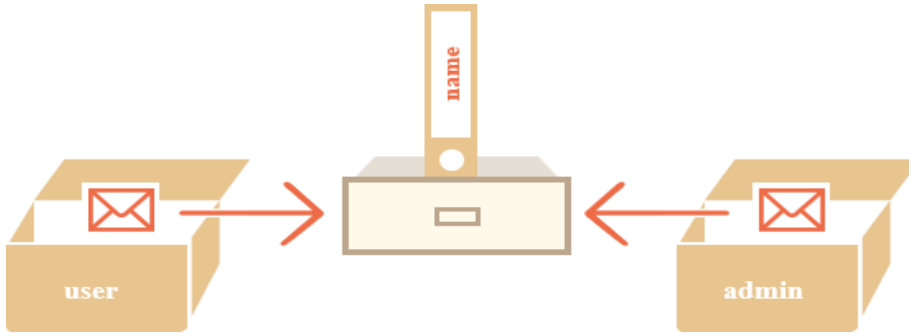
Object references == | === only if same object

```
let a = {};
let b = a;                  // copy the reference
alert(a == b);              // true, both variables reference the same object
alert( a === b );           // true
```

And here two independent objects are not equal, even though both are empty:

```
let a = {};
let b = {};                 // two independent objects
alert( a == b );            // false
```

```
let pt1 = {x:1 , y:2};
let pt2 = {x:1 , y:2}; // two independent objects
alert(pt1 == pt2); // ??
alert(pt1 === pt2) // ??
```



An object declared as const can be changed

```
const user = {
    name: "John"
};
user.age = 25;             // (*)
console.log(user.age);     // 25
user.name = 'Fred';
console.log(user.name);  //’Fred’
```

- But cannot be reassigned

```
const user = {
    name: "John"
};

// Error (can't reassign user)
user = {
    name: "Pete"
};
```

Methods

- Objects are usually created to represent entities of the real world, like users, orders and so on
- in the real world, a user can act
  - select something from the shopping cart, login, logout etc.
  - actions are represented in JavaScript by functions in properties.

```
let user = {
        name: "John",
        age: 30
};

user.sayHi = function () {
        alert("Hello!");
};

user.sayHi(); // Hello!
```

- used a Function Expression and assign to property user.sayHi

- A function that is the property of an object is called its *method*.

“this” in methods

- It’s common that an object method needs to access the information stored in the object to do its job.
  - the code inside user.sayHi() may need the name of the user.
  - To access the object, a method can use the **`this`** keyword.
  - The value of `this` is the object “before dot”, the one used to call the method.

```
let user = {
        name: "John",
        age: 30,
        sayHi() {
            // "this" is the "current object"
            alert(this.name);
        }
};

user.sayHi(); // John
```

shorter syntax for methods in an object literal

```
user = {
    sayHi: function () {
        console.log("Hello");
    }
};

// method shorthand
user = {
    sayHi() { // same as "sayHi: function()"
        console.log("Hello");
    }
};
```