LESSONS 6 & 7 OBJECTS

Knowledge is the Wholeness of Knower, Known, and Process of Knowing

Slides based on material from https://javascript.info licensed as CC BY-NC-SA.
As per the CC BY-NC-SA licensing terms, these slides are under the same license.

Wholeness: (Almost) everything in JavaScript is an object. Objects encapsulate state information in properties and associated behavior in methods. Science of Consciousness: Objects are an example of the common model-view-controller pattern of computer science, which closely corresponds to knower-known-process of knowing in our own consciousness. The state information contained in the properties is a model. The methods define the controller. The property names are the interface or view.

Main Points

1. Objects and properties: Whole is greater than the sum of the parts
   1. Object literals
   2. Dot and square bracket notations
   3. Property shorthand
   4. For .. in
2. Objects are references: self referral awareness (name of an object versus the true object, Self referral vs relative identity)
   1. Const objects can be modified
   2. Garbage collection : automation in administration
      1. Reachability
      2. Mark and sweep
3. Object methods, this :
   1. Method if property with function as value
   2. This in method references object
   3. Functions can be moved between objects
4. Constructor functions, new operator – creating many copies of same object
   1. Convention start with capital
   2. Convention call with new
   3. Creates {} and assigns to this
   4. Returns this (or object)

Main Point Preview: Objects and properties

Objects and properties can be created very simply using the object literal syntax. Properties can be added after the object is created simply by assigning a value to a property. Science of Consciousness: This is an example of accomplishing something in a very simple manner. Natural law takes the path of least action.

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.
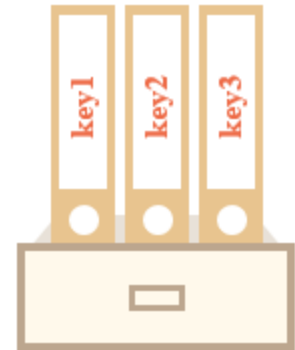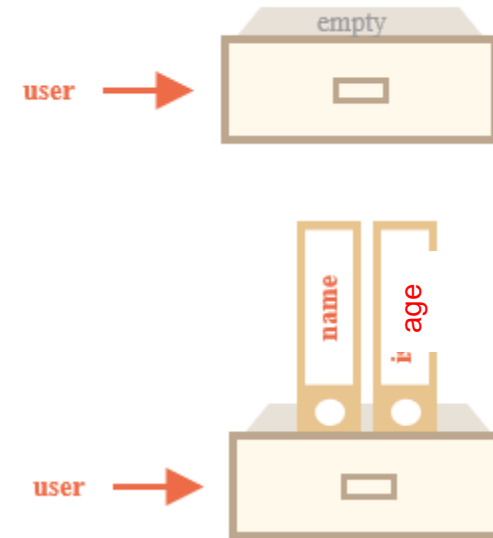
Object literals and properties

let user = {}; // "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:
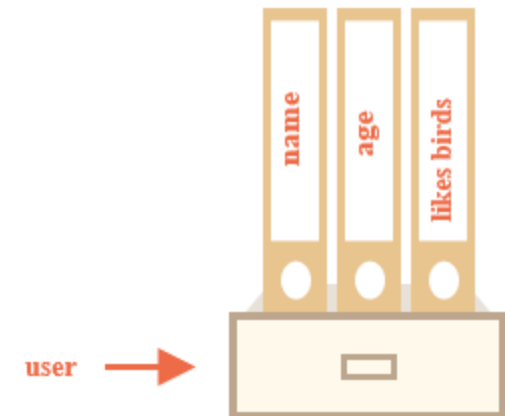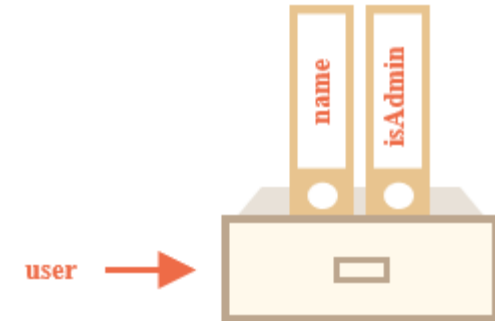alert(user.name); // John
alert(user.age); // 30

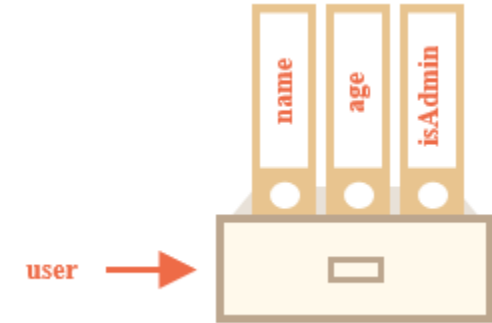# Adding and removing properties

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
```

Exercise

- ➢ Create 3 objects, student1, student2, student3
  - ➢ property studentId : s101, s102, s103 respectively
  - ➢ property quizAnswers: [1, 1, 2,4], [2, 1, 2,2], [3, 1, 3,4] respectively

- ➢ push the students into an array, quiz

- ➢ write a function, gradeQuiz, that takes the quiz array and an array of correct answers, e.g., [3,1,2,4] and returns an array that has scores for each student. Their score is the number of correct answers.
  - ➢ expect   [3, 2, 3]
  - ➢ use a helper function to compute the score for a given student

Square bracket notation

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

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

➢ Square brackets can obtain the property name from any expression
  ➢ e.g. 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 (if enter "name")
```

Computed properties

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

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {
  [fruit]: 5, // the name of the property is taken from the variable fruit
};
alert( bag.apple ); // 5 if fruit="apple"
```

➤ 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

Exercise

```
const numbers = {
    one: 1,
    two: 22,
    three: 333,
    four: 444
}


const properties = ["one", "two", "three", "four"];


//write a for loop to print out the values of the properties in numbers.  You must use the properties
array to access the values.  You cannot directly write console.log(numbers.one)  …
```

Property value shorthand

➢ often use existing variables as values for property names.

```
function makeUser(name, age) {
  return {
    name: name,
    age: age
    // ...other properties
  };
}
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
    // ...
  };
}
```

➢ Can you find where we used this shorthand notation to simplify the exports in yesterday's assignment?

Property value shorthand

```
let x = 5;
let y = {x};
console.log(y);  →  {x : 5}


function sum(arr){ let tot = 0;  for …  return tot; }
exports = {sum };


exports.sum === ??


exports.sum([1,2,3] ) === ??
```

Existence check

➢It is possible to access a property that doesn't exist!

   ➢<mark>Accessing a non-existing property just returns undefined.</mark>

   ➢Can test whether the property exists check for undefined:

Typo in property name might be hard to detect

```
let user = {};


console.log( user.noSuchProperty === 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
```

null versus undefined for values

➢ Usually, the strict comparison "=== undefined" works to check existence

  ➢ a special case when it fails, but "in" works correctly.

  ➢ when an object property exists, but stores undefined:

```
let obj = {
  myProp: undefined
};
console.log( obj. myProp ); // it's undefined, so - no such property?
console.log( " myProp" in obj ); // true, the property does exist!
```

➢ Situations like this happen very rarely, because undefined is usually not assigned.

  ➢ programmers assign null for "unknown" or "empty" values.

  ➢ in operator rarely used

  ➢ Good illustration of distinction of undefined vs null in JavaScript

Main Point : Objects and properties

Objects and properties can be created very simply using the object literal syntax. Properties can be added after the object is created simply by assigning a value to a property. Science of Consciousness: This is an example of accomplishing something in a very simple manner. Natural law takes the path of least action.

Main Point Preview: Object bindings hold references

Variables bound to primitive types contain actual values. Variables bound to objects contain references to memory locations that store the objects. Science of Consciousness: Variables are identifiers. They are not the actual values. When we transcend thought during the practice of our TM Technique we experience deeper values of our Self beyond our surface identity.

Access by reference

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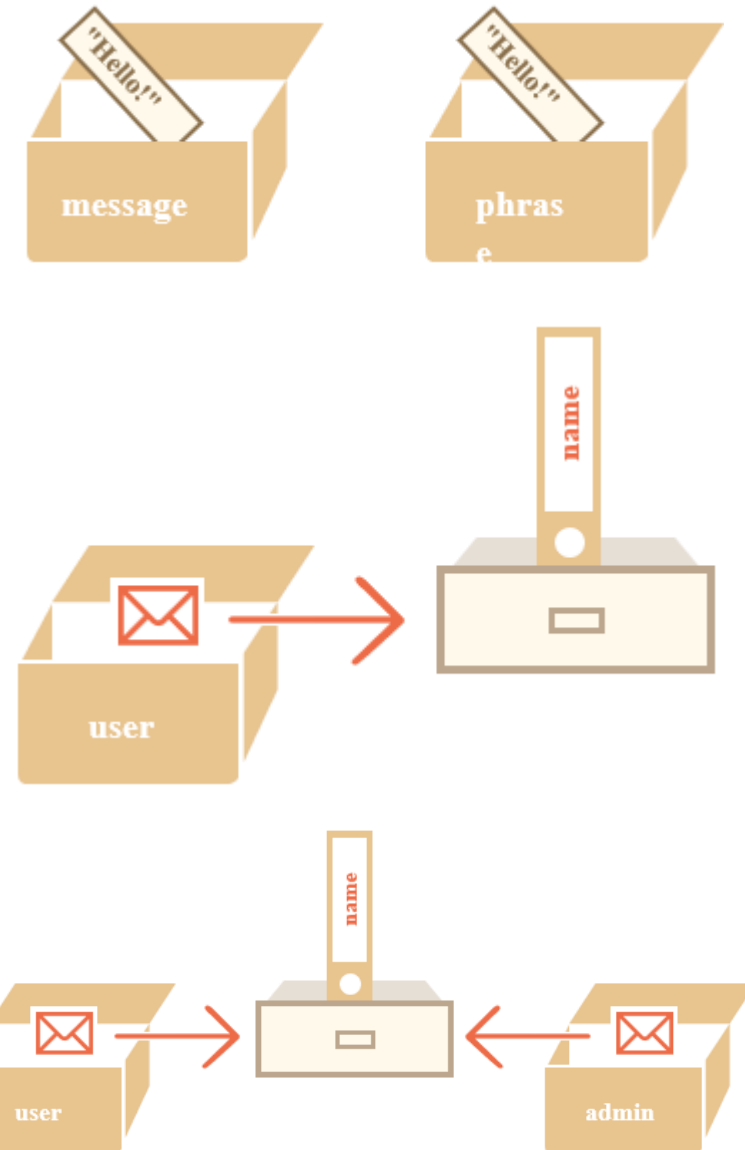
```
let message = "Hello!";
let phrase = message;
```

➢ An object 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
```

Access by reference

➢can use any variable to access the cabinet and modify its contents

➢**there is only one object**
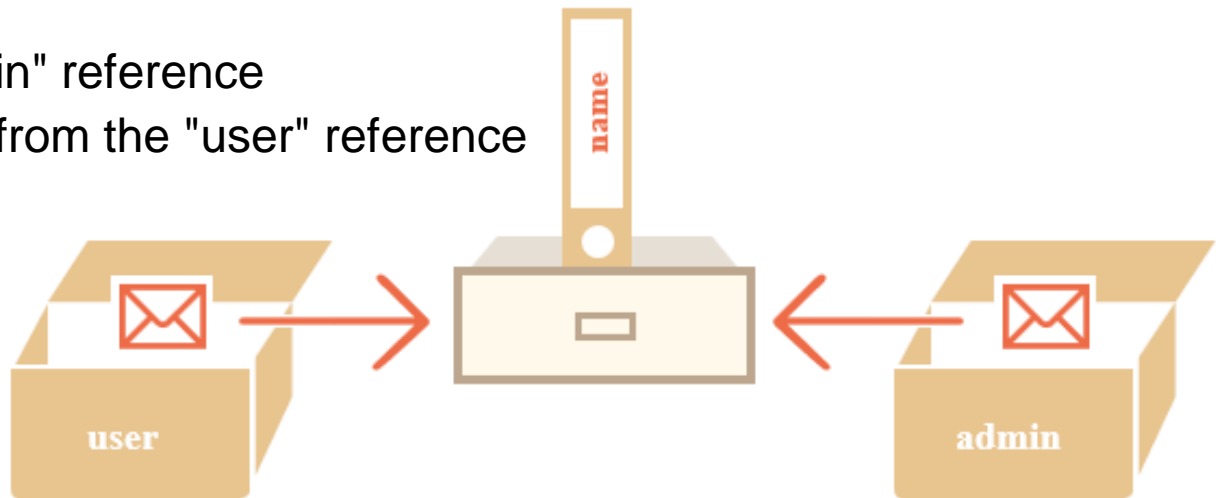 ➢If have two keys and use one of them (admin) to get into it.
  ➢later use the other key (user) we will see changes

    let user = { name: 'John' };
    let admin = user;
    admin.name = 'Pete'; // changed by the "admin" reference
    alert(user.name); // 'Pete', changes are seen from the "user" reference

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) // ??
```

Exercise: draw the graphic representation (boxes and file cabinets) for a and b and pt1 and pt2
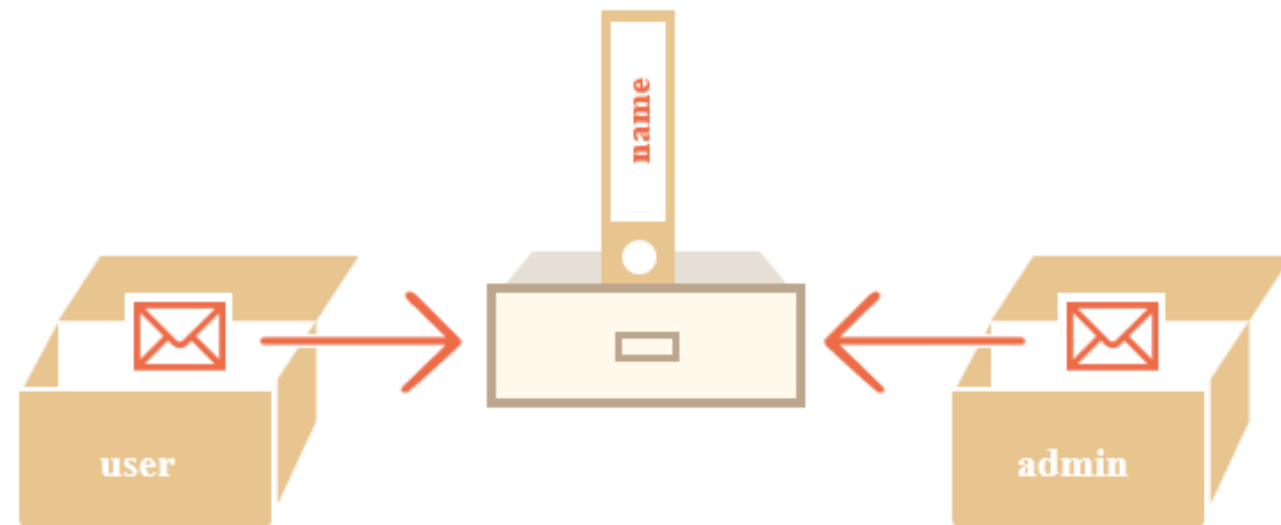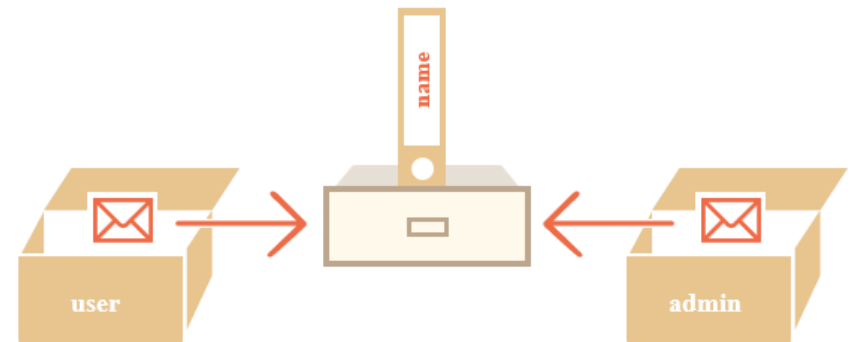
An object declared as const can be changed

```
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"
};
```

many other kinds of objects in JavaScript

➢ What we've studied in this chapter is called a "plain object", or just Object.

➢ Array to store ordered data collections,
➢ Date to store the information about the date and time,
➢ Error to store the information about an error.
➢ Functions are objects
➢ Modules are objects
➢ …And so on.

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 (versus iterable objects—e.g., arrays)

```
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
```

CAUTION
WATCH YOUR STEP

easy to confuse with for.. of

Exercise: Objects as "associative arrays"

- Create 3 objects, student1, student2, student3
  - property studentId : s101, s102, s103 respectively
  - property quiz answers: [1, 1, 2,4], [2, 1, 2,2], [3, 1, 3,4] respectively

- push the students into an array, quiz

- write a function, collectAnswers(questionNum) that will go through the array of student quiz answers, "quiz", and will collect the answers for all of the students for the nth question. For example, collectAnswers(3) will return an array with the answers of all students to question 3: [2, 2, 3] in the above example

Memory management in JavaScript is automatic and invisible

➢ primitives, objects, functions all take memory

➢ There's a background process in the JavaScript engine that is called garbage collection
  - ➢ monitors all objects and removes those that become unreachable.
  - ➢ "reachable" values are those that are accessible or usable somehow.

➢ base set of inherently reachable values called roots, that cannot be deleted
  - ➢ Global variables
  - ➢ Variables and parameters for functions on the current chain of nested calls.

➢ Any value is reachable if it's reachable from a root by a reference or by a chain of references
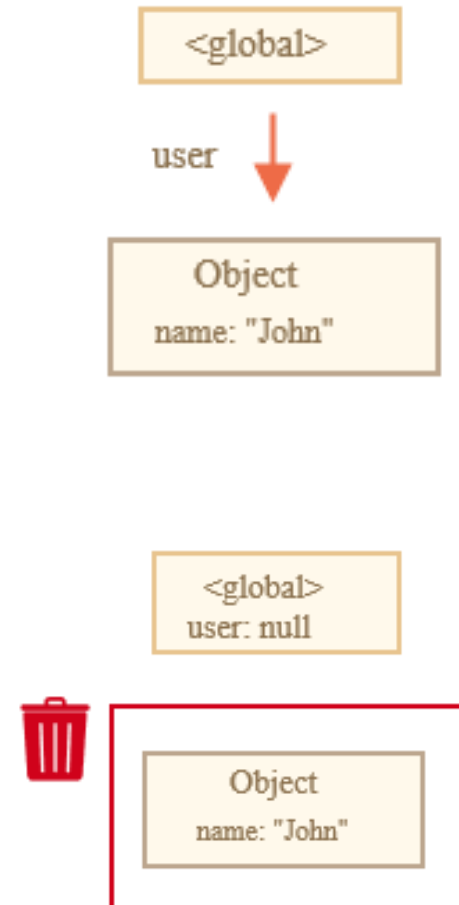
A simple example

// user has a reference to the object
let user = {
 name: "John"
};

user = null;

➢ If the value of user is overwritten, the reference is lost
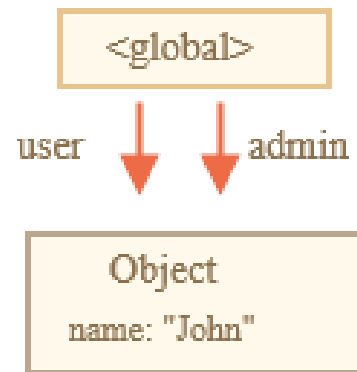 ➢ Unreachable
  ➢ Garbage collected

Two references

```
// user has a reference to the object
let user = {
  name: "John"
};

let admin = user;
user = null;
```

➢ Still have a reference from admin
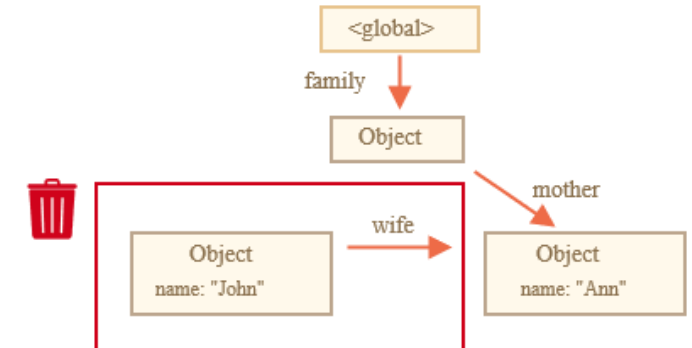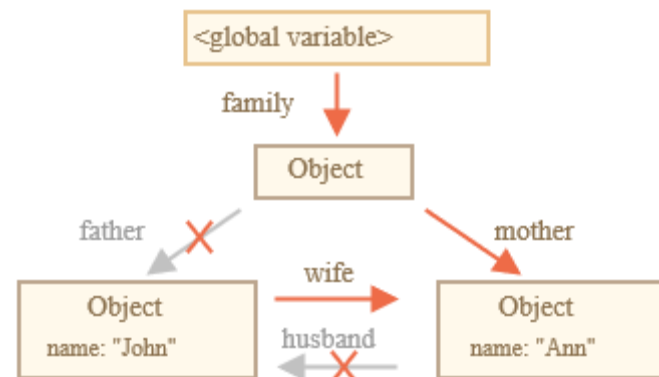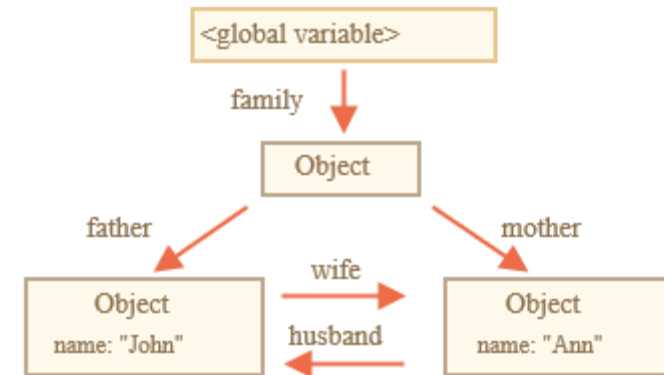 ➢ reachable
  ➢ Not garbage collected

Interlinked objects

```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;
  return {
    father: man,
    mother: woman
  }}
```

```
let family = marry({  name: "John"},
{  name: "Ann"});
```

```
delete family.father;
delete family.mother.husband;
```

Unreachable island

> possible that the whole island of interlinked objects becomes unreachable
>> John and Ann are still linked, both have incoming references. But that's not enough
>> former "family" object has been unlinked from the root
>> no reference to it any more
>> whole island becomes unreachable and will be removed

family = null
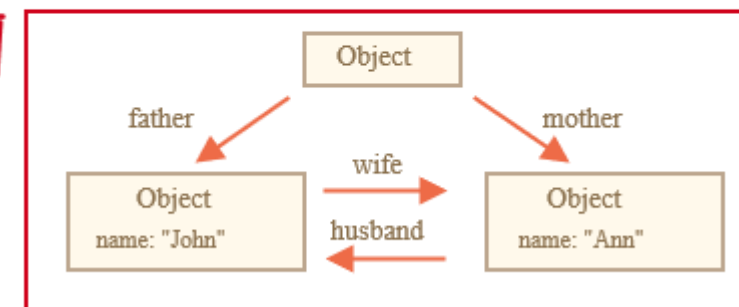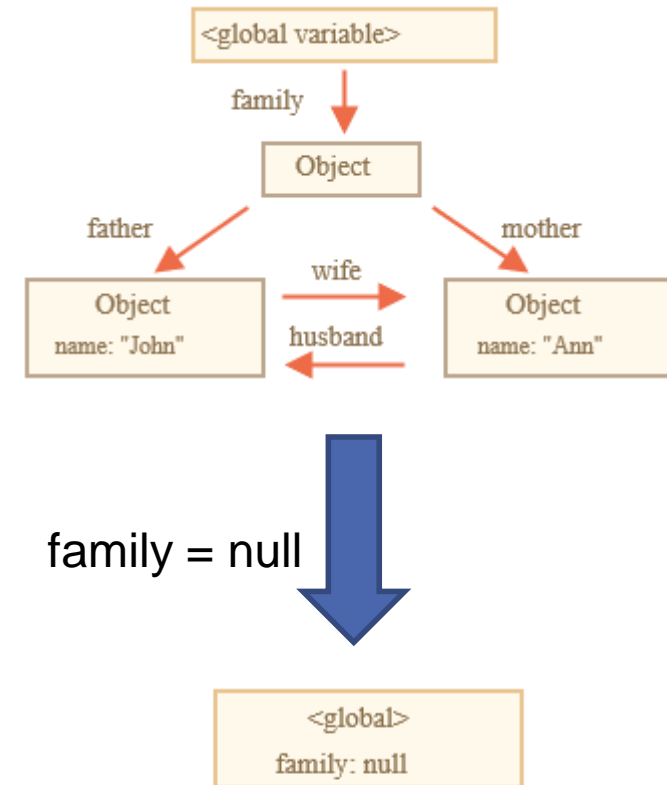
basic garbage collection algorithm is called "mark-and-sweep"

"garbage collection" steps are regularly performed

➢ The garbage collector "marks" roots

➢ Then visits and "marks" all references from them.

➢ Then it visits marked objects and marks *their* references.

   ➢ All visited objects are remembered, so as not to visit the same object twice in the future.

➢ …And so on until there are no unvisited references

   ➢ reachable from the roots

➢ All objects except marked ones are removed

basic garbage collection algorithm is called "mark-and-sweep"
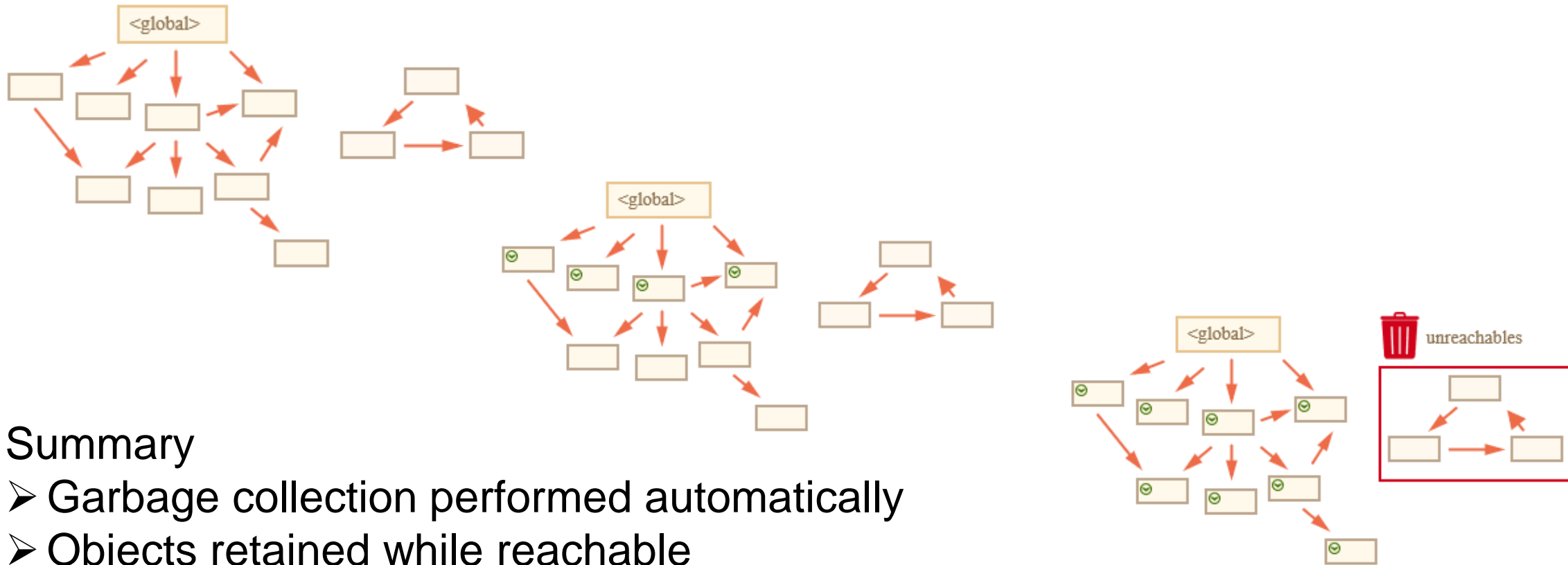
Summary
- Garbage collection performed automatically
- Objects retained while reachable
- Being referenced not same as being reachable
  - a pack of interlinked objects can become unreachable

Main Point : Object bindings hold references

Variables bound to primitive types contain actual values. Variables bound to objects contain references to memory locations that store the objects. Science of Consciousness: Variables are identifiers. They are not the actual values. When we transcend thought during the practice of our TM Technique we experience deeper values of our Self beyond our surface identity.

Main Point Preview: Object methods, "this"

The behavior associated with an object is defined in methods, which are properties that have functions as their value. Methods refer to other object properties using the keyword 'this'. Since functions are first-class they can be passed between objects, in which case 'this' can refer to different objects at different times.

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:  function() {
    // "this" is the "current object"
    alert(this.name);
  }
};

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

Calling without an object: this == undefined

➢can call the function without an object at all:

```
function sayHi() {
  alert(this);
}
sayHi(); // undefined
```

➢ In this case this is undefined in strict mode.
  ➢ If try to access this.name, there will be an error.

➢ In non-strict mode the value of this in such case will be the global object
  ➢ historical behavior that "use strict" fixes.

➢ Usually such a call is a programming error.
  ➢ If there's this inside a function, it expects to be called in an object context.

Summary of methods and 'this'

➢Functions that are stored in object properties are called "methods".

➢Methods can reference the object as this.
  ➢The value of this is defined at run-time.
  ➢When a function is declared, it may use this, but that this has no value until the function is called.

➢A function can be copied between objects.

➢When a function is called in the "method" syntax: myObj.method(), the value of this during the call is myObj.
  ➢ "object before the dot" rule

Class exercise: practice with objects and this

➢ find the sayhi.js file in demos/d22methods implement the sayHi function

➢ find the this.js file in the same folder.
  ➢ what is the difference when running in strict mode versus not?

shorter syntax for methods in an object literal

```javascript
// these objects do the same
user = {
  sayHi: function() {
    console.log("Hello");
  }
};


// method shorthand (avoid this during CS305)
user = {
  sayHi() { // same as "sayHi: function()"
    console.log("Hello");
  }
};
```

Homework Exercise (in book)

Create a calculator

Create an object calculator with three methods (slight modification from book):

    setValues(operand1, operand2) takes two values and saves them as object properties.
    sum() returns the sum of saved values.
    mul() multiplies saved values and returns the result.

```
let calculator = {
  // ... your code ...
};

calculator.setValues(5, 10);
console.log( "expect 15 : ", calculator.sum() );
console.log("expect 50 : ", calculator.mul() );
```

Homework exercises

➢ Using "this" in object literal

➢ Chaining

    ➢ hint: ladder.up().down() will execute the chain from left to right

        ➢ first executes ladder.up() which returns a value

        ➢ then attempts to execute a .down() method on an object returned by ladder.up()

    ➢ always change alert to console.log for code to run in Node.js

Main Point: Object methods, "this"

The behavior associated with an object is defined in methods, which are properties that have functions as their value. Methods refer to other object properties using the keyword 'this'. Since functions are first-class they can be passed between objects, in which case 'this' can refer to different objects at different times.

Main Point Preview: **Constructor, operator "new"**

Constructor functions are helpful when we need to create many similar objects.

Constructor functions, operator "new"

➤ Object literal {...} syntax creates a single object.
  ➤ often need to create many similar objects,
    ➤ multiple users or menu items and so on.
  ➤ Use constructor functions and the "new" operator

➤ Constructor functions technically are regular functions.
➤ two conventions:
  ➤ start with capital letter
  ➤ executed only with "new" operator

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

new User(...) does the following steps:

1. A new empty object is created and assigned to this.
2. The function body executes. Usually it modifies this, adds new properties to it.
3. The value of this is returned.

➢In other words, new User(...) does something like:

```
function User(name) {
  // this = {};  (implicitly)

  // add properties to this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitly)}
```

Constructor vs object literal

➢ let user = new User("Jack") is same as

    let user = {
      name: "Jack",
      isAdmin: false
    };

➢to create other users, can call new User("Ann"), new User("Alice") etc

  ➢shorter than using literals every time
  ➢easy to read
  ➢For CS305 favor object literals
  ➢Constructors will become important with inheritance and classes

➢Exercises
  ➢Two functions – one object
    ➢Hint: If a function returns an object then new returns that object instead of 'this'
  ➢Create new Calculator
    ➢ use the modified version on the next page instead of the book version
  ➢Create new Accumulator

  ➢In class exercise: explain to partner exactly what 'new' operator does to a function call

Homework exercise

//use this slight modification of the exercise that is in the book for today's homework

// instead of a read method as in the book, your calculator should have a setValues method that sets the input values to be the values of
//two properties. sum and mul will access those properties for their calculations.

```
describe("test calculator with values 2, 3", function () {
    calculator.setValues(2, 3);  //values for the tests

    it("checks initial values 2 and 3", function () {
        assert.strictEqual(calculator.a, 2);
        assert.strictEqual(calculator.b, 3);
    });

    it("when 2 and 3 are entered, the sum is 5", function () {
        assert.strictEqual(calculator.sum(), 5);
    });

    it("when 2 and 3 are entered, the product is 6", function () {
        assert.strictEqual(calculator.mul(), 6);
    });


});
```

Main Point: **Constructor, operator "new"**

Constructor functions are helpful when we need to create many similar objects

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Knowledge Is the Wholeness of Knower, Known, and Process of Knowing

1. Objects contain properties and their values.

2. Large JavaScript programs use objects and their properties to break the program into smaller manageable components. It is the interaction of all the component objects that make the program work.

_____

3. **Transcendental consciousness**. State of wholeness.
4. **Impulses within the transcendental field:** Pure consciousness, by its own nature, is aware of itself. It thereby becomes the knower of itself through the self-referral process of knowing.
5. **Wholeness moving within itself:** In unity consciousness the value of wholeness is appreciated even in the components of knower, known, and process of knowing.