LESSON 3
OBJECTS

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

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

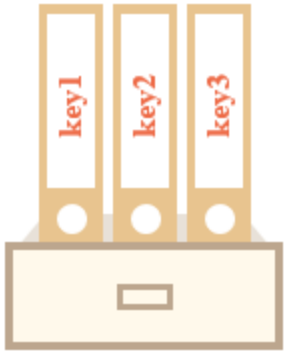
Objects and properties can be created very simply using the object literal syntax. Properties can be added after the object is created simply by assigning a value to a property. Science of Consciousness: This is an example of accomplishing something in a very simple manner. Natural law takes the path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

Objects: the basics

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

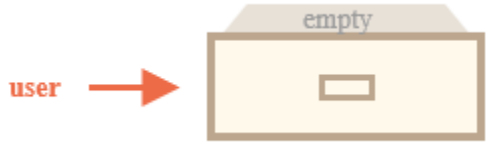
- objects store keyed collections of data and functions.

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

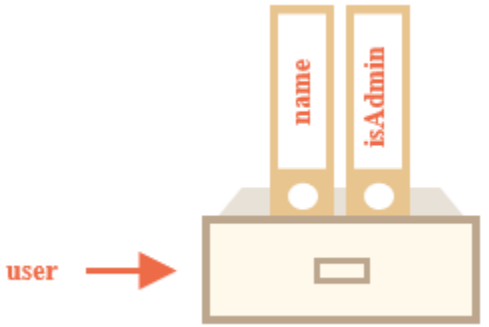


Object literals and properties

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



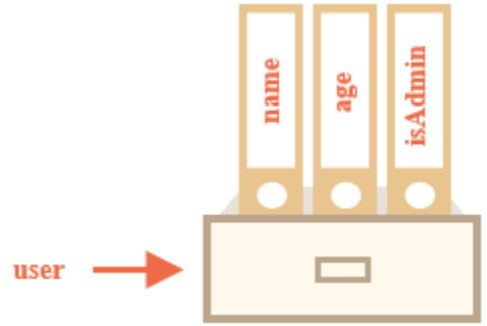

```
let user = {     // an object
  name: "John",  // by key "name" store value "John"
  age: 30        // by key "age" store value 30
};
```



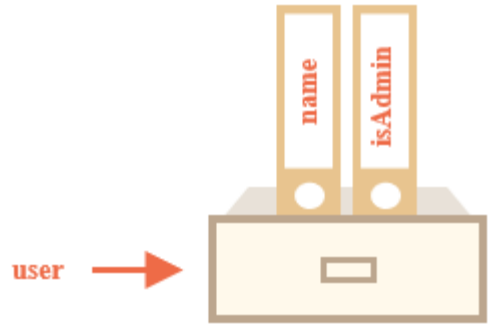

```
// get property values of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

Adding and removing properties

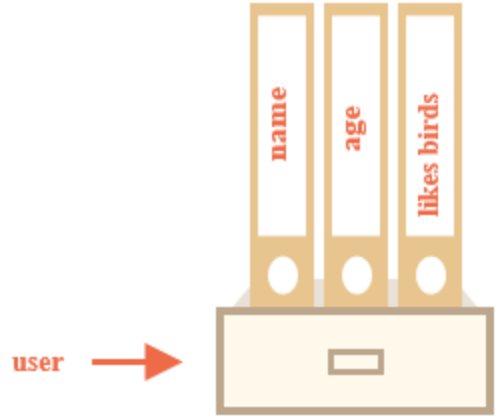
```
user.isAdmin = true;  // properties can be dynamically created
```




```
delete user.age;
```




```
let user = {
  name: "John",
"likes birds": true  // multiword property name must be quoted
};
user.age = 30;  //dynamic creation
```



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 (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

Property value shorthand (not on test)

- 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
    // ...
  };
}
```

Existence check (not on test)

- It is possible to access a property that doesn’t exist!
  - Accessing a non-existing property just returns undefined.
  - common way to test whether the property exists is get it and compare vs undefined:

```
let user = {};

alert( user.noSuchProperty === undefined ); // true means "no such property"
```

- special operator "in" 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 };

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

null versus undefined for values (not on test)

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

- Situations like this happen very rarely, because undefined is usually not assigned.
  - mostly use null for “unknown” or “empty” values.
  - In operator rarely used
  - Good illustration of distinction of undefined vs null in JavaScript

The “for…in” loop

- To walk over all keys of an object, there exists a special form of the 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
  alert( key );  // name, age, isAdmin
  // values for the keys
  alert( user[key] ); // John, 30, true
```

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.

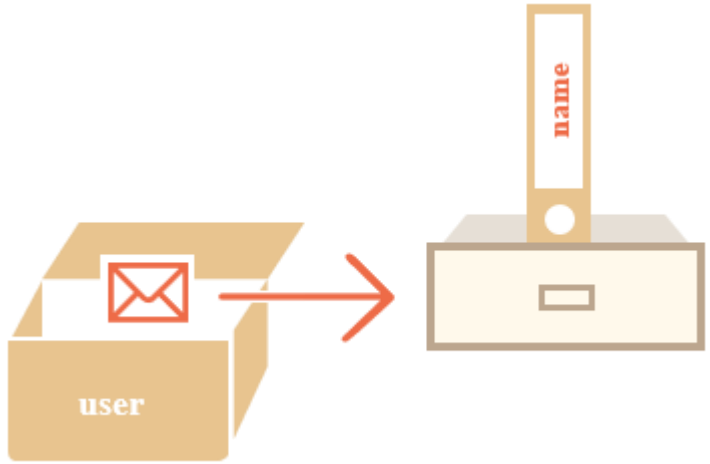
Copying by reference

- 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;
```

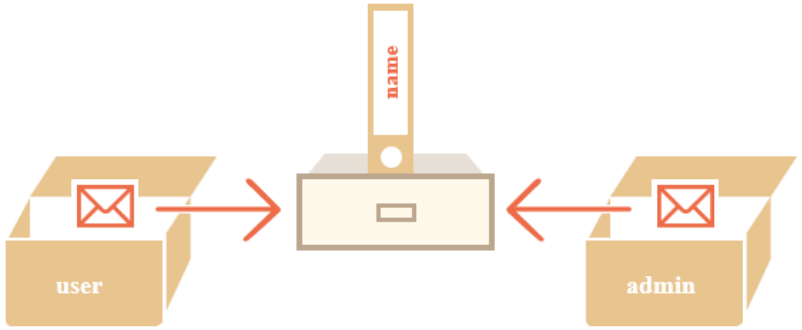
- An object variable stores not the object itself, but its “address in memory”, in other words “a reference” to it.

```
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
```

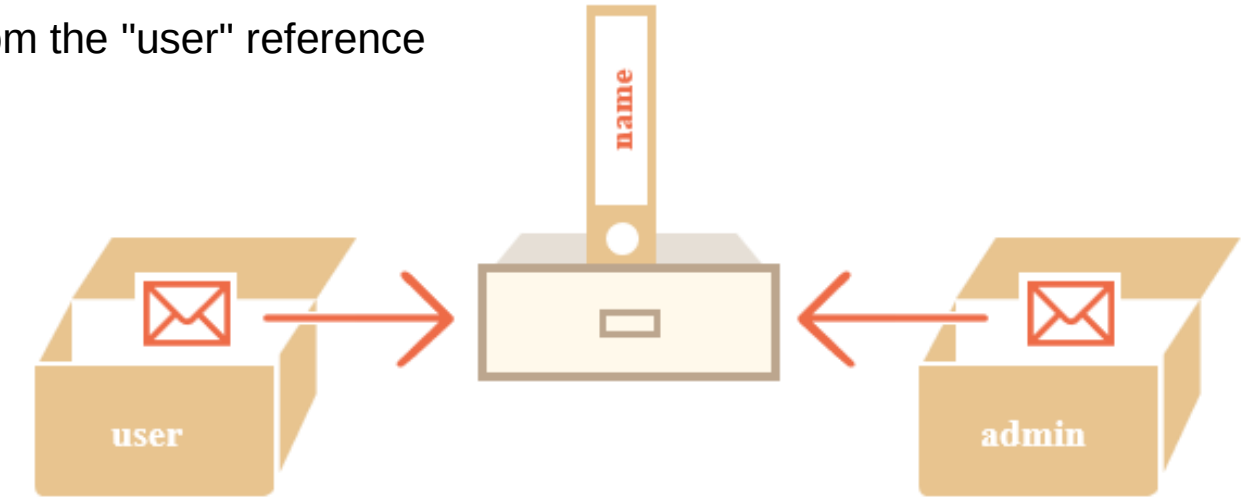


Copying by reference

- 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
```



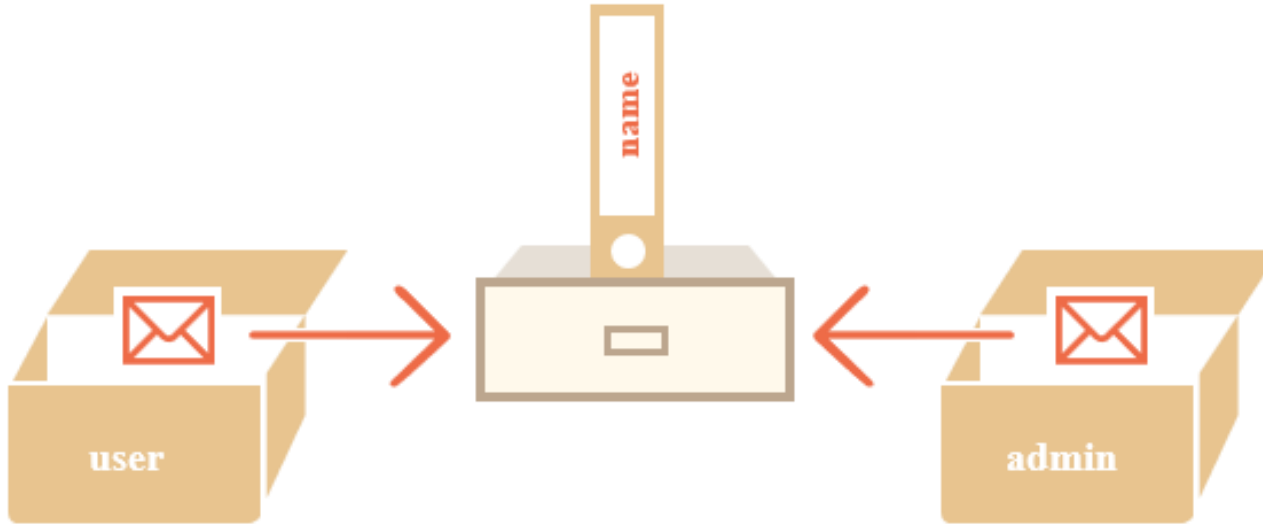
Two objects are equal only if they are the same object

```
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) // ??
```



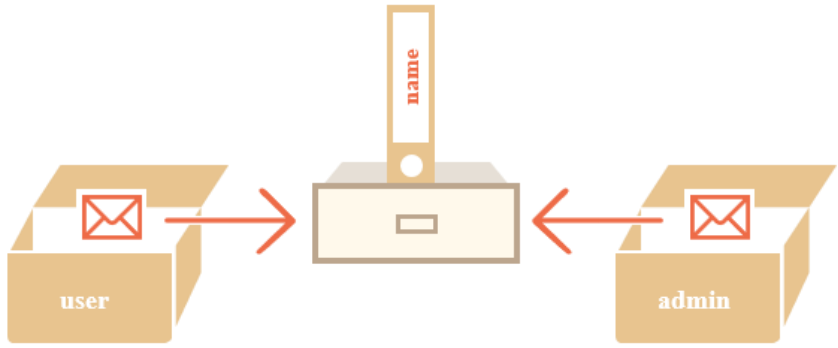
An object declared as const can be changed

```
const user = {
  name: "John"
};
user.age = 25; // (*)
alert(user.age); // 25
User.name = ‘Fred’;
Alert(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.
- …And so on.

- Exercises:
  - Hello object
  - Check for emptiness
  - Constant objects?
  - Sum object properties
  - Multiply numeric properties by 2

Memory management in JavaScript is automatic and invisible

- primitives, objects, function all take memory

  - The main concept of memory management in JavaScript is reachability.
    - “reachable” values are those that are accessible or usable somehow.
    - They are guaranteed to be stored in memory.

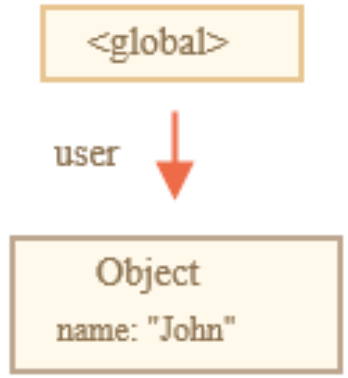
  - base set of inherently reachable values called roots, that cannot be deleted
    - Global variables
    - Variables and parameters for functions on the current chain of nested calls.

  - Any value is reachable if it’s reachable from a root by a reference or by a chain of references

  - There’s a background process in the JavaScript engine that is called garbage collection
    - monitors all objects and removes those that become unreachable.

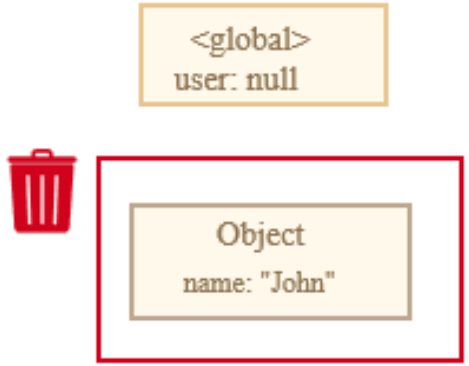
A simple example

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




```
user = null;
```

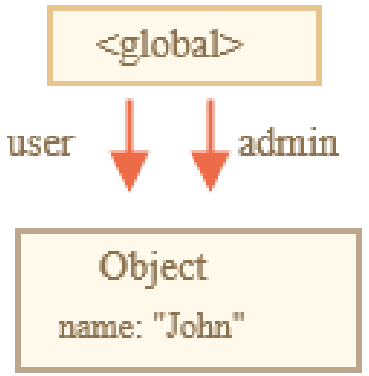
- If the value of user is overwritten, the reference is lost
  - Unreachable
  - Garbage collected



Two references

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

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

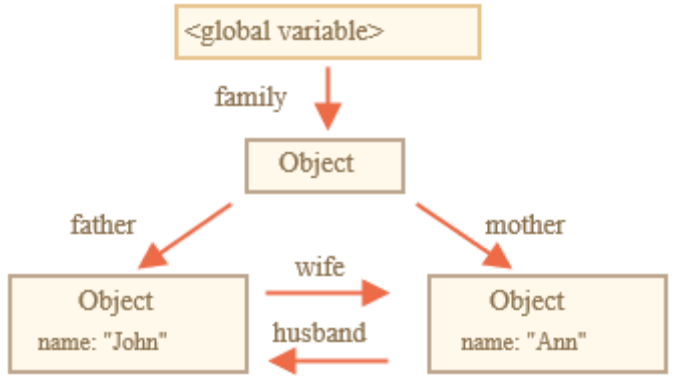


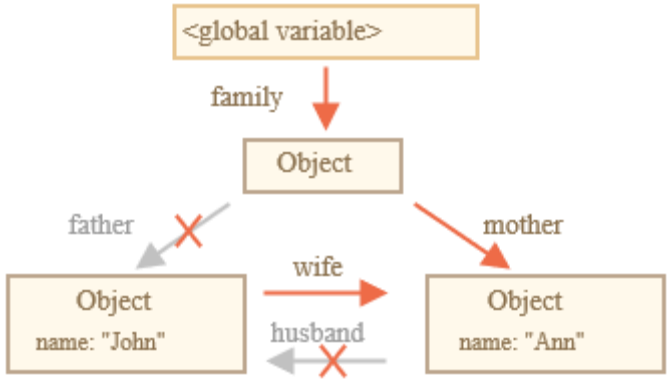
- Still have a reference from admin
  - reachable
  - Not garbage collected

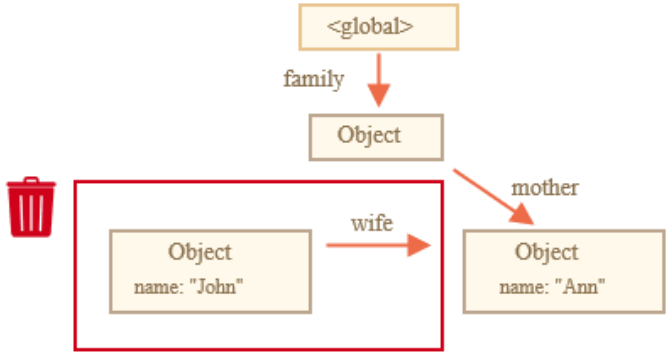
Interlinked objects

```
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;
```

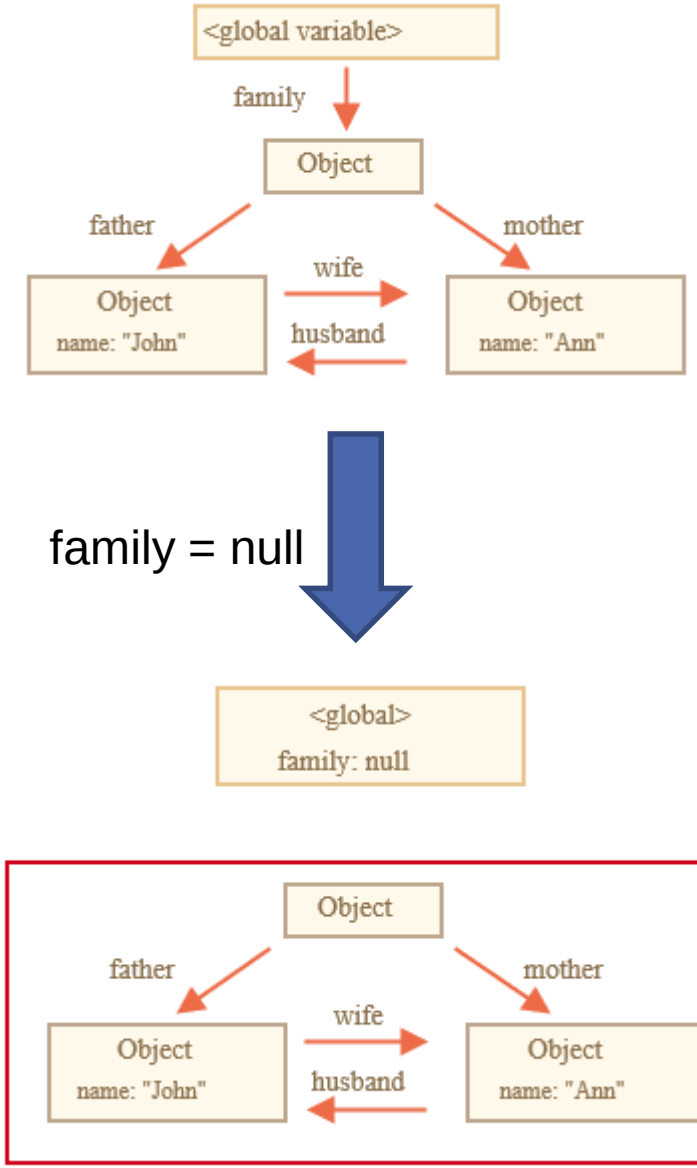






Unreachable island

- 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

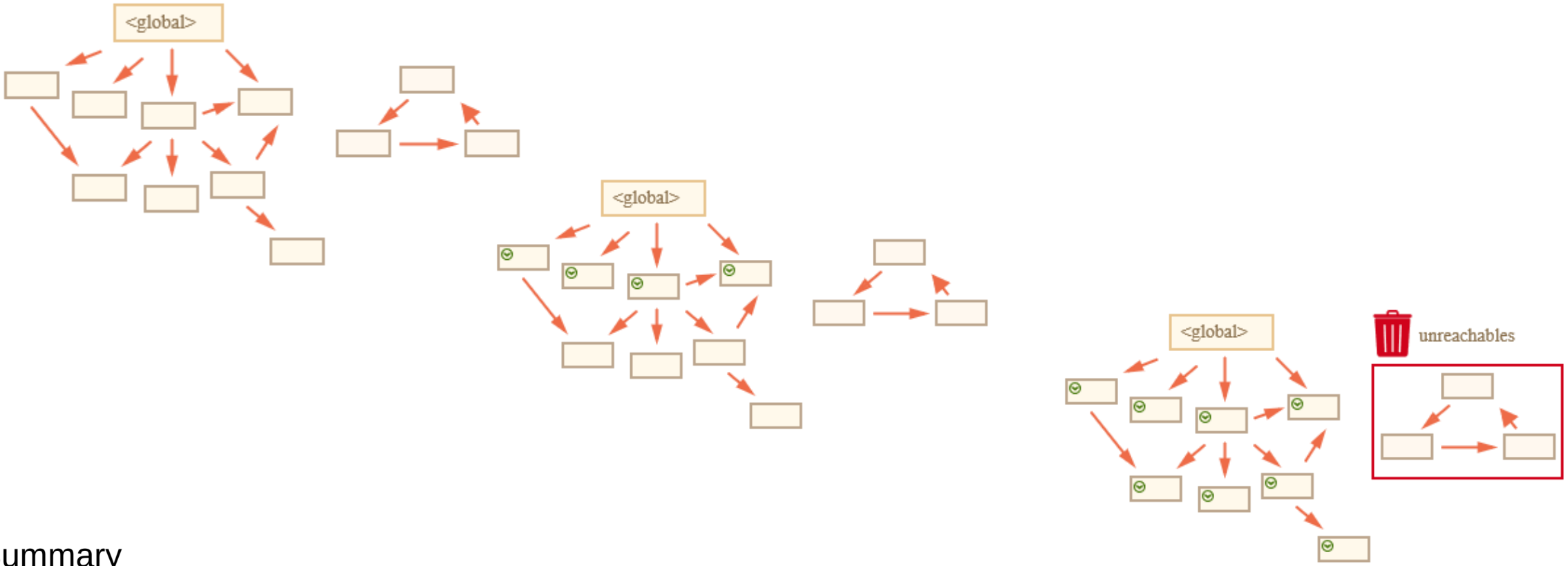


basic garbage collection algorithm is called “mark-and-sweep”

“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.

Symbol (optional section)

- Symbol is a primitive type for unique identifiers.
- Symbols are created with Symbol() call with an optional description (name).
- Symbols are always different values, even if they have the same name.

- Symbols have two main use cases:

- “Hidden” object properties.
  - can “covertly” hide something into objects that we need, but others should not see
    - add a property into an object that “belongs” to another script or a library
    - create a symbol and use it as a property key.
  - does not appear in for..in
  - won’t be accessed directly, because another script does not have our symbol.
    - protected from overwrite.

- system symbols used by JavaScript which are accessible as Symbol.*.
  - We can use them to alter some built-in behaviors.
  - For instance, later in the tutorial we’ll use Symbol.iterator for iterables, Symbol.toPrimitive to setup object-to-primitive conversion and so on.

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*.

shorter syntax for methods in an object literal

```
// these objects do the same
user = {
  sayHi: function() {
    alert("Hello");
  }
};

// method shorthand (avoid this during CS303)
user = {
  sayHi() { // same as "sayHi: function()"
    alert("Hello");
  }
};
```

“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
```

Calling without an object: this == undefined

- We can even call the function without an object at all:

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

- In this case this is undefined in strict mode. If we 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 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.

Exercises

- Using “this” in object literal
- Do the Create a calculator exercise in VSCode
  - create a Mocha test and run it in VSCode
  - you can view the solution in the book and then have it run in VSCode
    - You do not need to use the sinon.stub and prompt.onCall methods
    - But ok to use them if you like
    - Or explicitly set the fields in the test
- Chaining
  - hint: ladder.up().down() will execute the chain from left to right
    - first executes ladder.up() which returns a value (maybe undefined)
    - then attempts to execute a .down() method on an object returned by ladder.up()

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

- Result of new User("Jack") is same as

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

- if we want to create other users, can call new User("Ann"), new User("Alice") etc
  - shorter than using literals every time
  - easy to read
  - For CS303 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 it instead of this
  - Create new Calculator
  - Create new Accumulator

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 by it. 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.