LESSON 6
RECURSION

Self Referral Awareness

Main Points

1. Recursion and stack
    1. Recursive data structures: linked lists and graphs
2. Rest parameters and spread operator

Main Point Preview: Recursion

Recursion is when a function calls itself. Recursion is useful when a task can be simplified into an easy action plus a simpler variant of the same task.
Science of Consciousness: Repetition, looping, is the basis of computing. All repetition in computing can be implemented through recursion. Recursion is an example of self referral. Self referral awareness is the basis of manifest existence.

Two ways of thinking

pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16

Iterative thinking: the for loop:

```
function pow(x, n) {
  let result = 1;
  // multiply result by x n times in the loop
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
alert( pow(2, 3) ); // 8
```

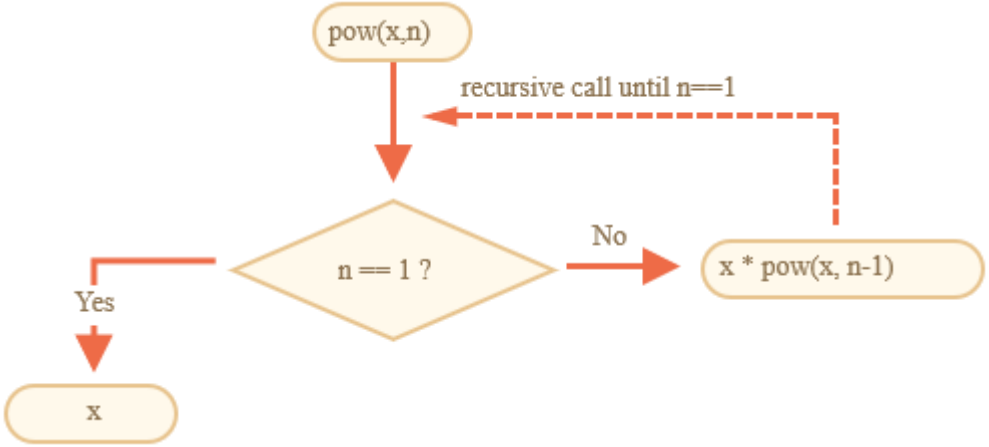
Recursive thinking: reduce the task and call self:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
alert( pow(2, 3) ); // 8
```

Recursive thinking

- If n == 1, then everything is trivial.
  - base case of recursion,
  - immediately produces the obvious result: pow(x, 1) equals x.
- Otherwise, we can represent pow(x, n) as x * pow(x, n - 1)
  - called a recursive step (and also a reduction step)
  - transform the task into a simpler action (multiplication by x) and a simpler call of the same task

pow(2, 4) = 2 * pow(2, 3)
pow(2, 3) = 2 * pow(2, 2)
pow(2, 2) = 2 * pow(2, 1)
pow(2, 1) = 2



execution context and stack

- information about execution of a running function is stored in its execution context.
  - internal data structure with details about execution of a function:
    - where the control flow is now,
    - current variables,
    - value of this and few other internal details.
  - One function call has exactly one execution context associated with it.

- When a function makes a nested call, the following happens:
  - current function is paused.
  - execution context associated with it is remembered in execution context stack.
  - The nested call executes.
  - After it ends, old execution context is retrieved from the stack,
    - outer function is resumed from where it stopped.

Function call and class stack frames (from 301)

```
// Output?

function A(){
    console.log("A is called");
    console.log("Before B is called");
    B();
    console.log("After B is called")
}

function B(){
    console.log("B is called");
    console.log("Before C is called");
    C();
    console.log("After C is called");
}

function C(){
    console.log("C is called");
}
A();
console.log("After A is called");
```

Example: Lets draw an execution stack (from 301)

```
function funX(a, b) {
    let c;
    c = 5;
    funY(a * c, "yes");
}

function funY(x, y) {
    let z;
    z = "I can see the sea";
    console.log("What is on the stack here?");
}

function main() {
    let a;
    let b;
    a = "Hello";
    funX(3, a);
    b = "World";
}

main();
```

Exercise: Draw the execution stack

```
function funA(a,n){
    let something;
    something = "something.";
    funB(something, n);
}

function funB(a,b){
    let thing;
    thing = "a thing.";
    console.log("What is on the stack when we're here?");
}

function main(){
    let test;
    let n;
    test = "Hello";
    n = 5;
    funA(n, 10);
}

main();
```

execution context and stack for pow(2, 3)

```
1.  function pow(x, n) {
2.    if (n == 1) {
3.      return x;
4.    } else {
5.      return x * pow(x, n - 1);
6.    }
7.  }
```

{ x: 2, n: 1, at line 1 }

Function call	Exec context	Recursive call	return
Pow(2,1)	{ x: 2, n: 1, at line 1 }		2
Pow(2,2)	{ x: 2, n: 2, at line 5 }	Pow(2,1)	2 * 2
Pow(2,3)	{ x: 2, n: 3, at line 5 }	Pow(2,2)	2 * 4

When recursive calls are appropriate

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);  }}
```

```
function pow(x, n) {
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;}
```

- Contexts take memory.
  - A loop-based algorithm uses less memory
  - Clarity is generally more important than efficiency
  - Any recursion can be rewritten as a loop.

- When recursive calls are appropriate
  - When problems have a natural recursive structure and solution
  - E.g., Tree and list data structures.

Recursive traversals

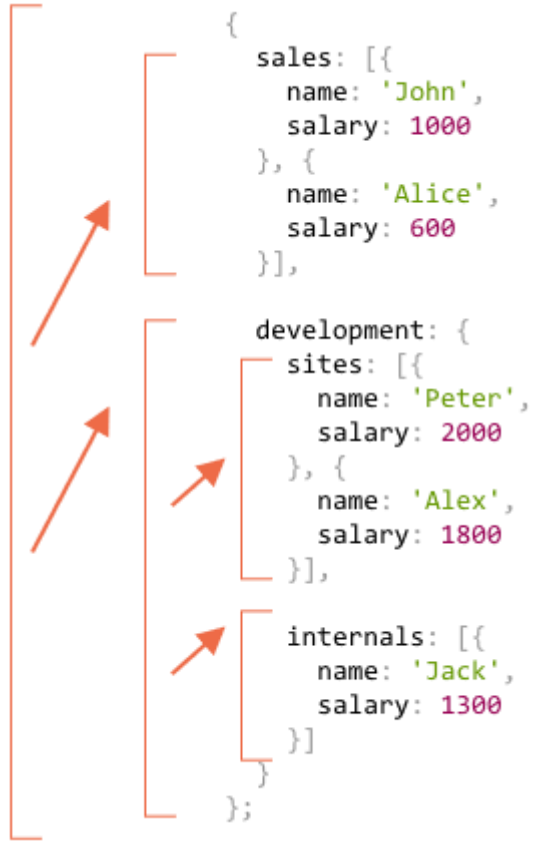
- want a function to get the sum of all salaries.
  - departments may have subdepartments which may have subsubdepartments, …
  - looping: would need loops within loops … (could be arbritrary depth)

- recursive algorithm
  - "simple" department with an array of people
    - sum the salaries in a simple loop.
  - object with N subdepartments
    - N recursive calls to get the sum for each of the subdeps and combine the results

```
let company = {
  sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 600 }],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800 }],  //subdepartments
    internals: [{name: 'Jack', salary: 1300}]
  }
};
```

Recursive traversals 2

```
function sumSalaries(department) {
  if (Array.isArray(department)) { // case (1)
    return department.reduce((prev, current) => prev + current.salary, 0); // sum the array
  } else { // case (2)
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // recursively call for subdepartments, sum the results
    }
    return sum;  }}
console.log(sumSalaries(company)); // 6700
```



- The code is short and easy to understand (hopefully?).
  - power of recursion. It also works for any level of subdepartment nesting
- easily see the principle:
  - for an object {...} subcalls are made,
  - while arrays [...] are the “leaves” of the recursion tree, they give immediate result.
- Note that the code uses smart features that we’ve covered before:
  - Method arr.reduce explained in the chapter Array methods to get the sum of the array.
  - Loop for(val of Object.values(obj)) to iterate over object values:
    - Object.values returns an array of them

Recursive structures

- A recursive (recursively-defined) data structure is a structure that replicates itself in parts.

- A company department is:
  - Either an array of people.
  - Or an object with departments.

- In an HTML document, an HTML-tag may contain a list of:
  - Text pieces.
  - HTML-comments.
  - Other HTML-tags (that in turn may contain text pieces/comments or other tags etc).

Recursive structures

```
let node4 = {
    name: "label",
    value: "Name",
    children: null
};

let node5 = {
    name: "input",
    value: "this was typed by a user",
    children: null
};

let node3 = {
    name: "p",
    value: "This is text in the a paragraph",
    children: null
};

let node2 = {
    name: "div",
    value: null,
    children: [node4, node5]
};

let node1 = {
    name: "body",
    children:  [node2, node3],
    value: null,
};
```

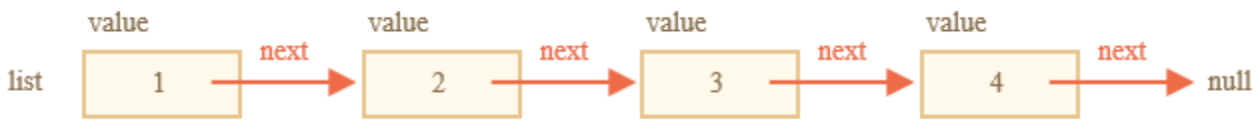
Linked list

- "Linked list" recursive structure might be better than array in some cases
  - problem with arrays.
    - "delete element" and "insert element" operations are expensive.
    - , arr.unshift(obj) operation must renumber all elements to make room for a new obj
      - if the array is big, it takes time.
    - Same with arr.shift().
  - The only structural modifications that do not require mass-renumbering are those that operate with the end of array:
    - arr.push/pop.
  - an array can be slow for big queues, when we must work with the beginning.
    - choose a linked list.
  - if need fast insertion/deletion,
    - choose a linked list.

Linked list definition

- The linked list element is recursively defined as an object with:
  - value.
  - next property referencing the next linked list element or null if that's the end.

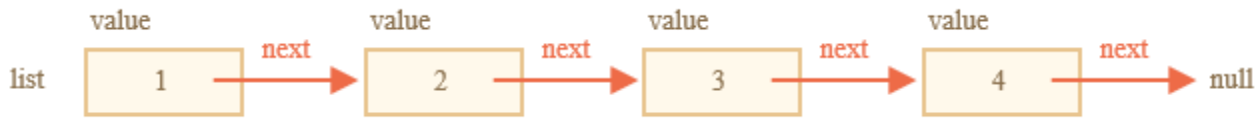
```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };
```



Linked list definition 2

- Same linked list, built node by node

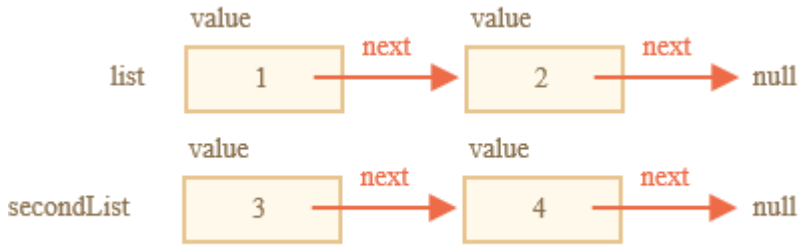
```
let el4 = {
    value: 4,
    next: null};

let el3 = {
    value: 3,
    next: el4};

let el2 = {
    value: 2,
    next: el3};

let list = {
    value: 1,
    next: el2};
```



Linked list operations

- easily split into multiple parts

```
let secondList = list.next.next;
list.next.next = null;
```



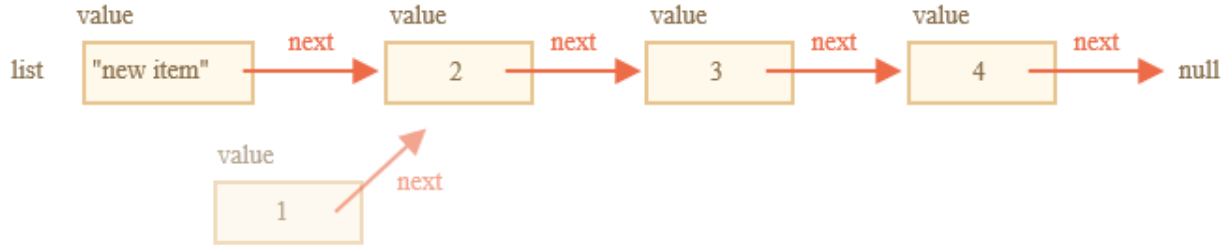
- How would you rejoin it?

- to prepend a new value, we need to update the head

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };
// prepend the new value to the list
list = { value: "new item", next: list };
```

- To remove a value from the middle, change next of the previous one

```
list.next = list.next.next;
```



Exercises

- Sum all numbers till the given one
- Calculate factorial
- Fibonacci numbers
- Output a single-linked list
- Output a single-linked list in the reverse order
  - hint: recall the recursive pow function
    - return x * pow(x, n – 1);
    - pow(x, n – 1) is called before the multiplication can happen
    - i.e., the multiplication operation happens after the recursive call
    - recall how the stack then unwinds
    - first multiplication is 2 * 2
    - then 2 * 4
    - then 2 * 8, etc
    - key point is that the “operation” is happening after the recursive call

Main Point: Recursion

Recursion is when a function calls itself. Recursion is useful when a task can be simplified into an easy action plus a simpler variant of the same task. *Science of Consciousness*: Looping is the basis of computing. All repetition in computing can be implemented through recursion. Recursion is an example of self referral. Self referral awareness is the basis of manifest existence.

Main Point Preview: Rest parameters and spread operator

Rest parameters and the spread operator are convenience syntax for, respectively, collecting function parameters into an array or assigning collection elements across a set of variables.

Function Signature

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

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

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? JavaScript ignores extra arguments and uses undefined for missing arguments. Last declaration overwrites earlier ones.

`arguments` Object

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

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

Exercise: write a function that can be called with any number of arguments and returns the sum of the arguments.

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
- ES6 compatible code, then rest parameters should be preferred.

```
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: " + total);
 return total;
}
sum(5, 5, 5);
sum(6, 6, 6, 6, 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”]
copyOfA = [...a]  //[1, 2, 3]

let str = "Hello";
alert( [...str] ); // H,e,l,l,o
```

Summary

- When we see "..."
  - can be rest parameters or spread operator
  - spread syntax "expands" an array into its elements
  - rest syntax collects multiple elements and "condenses" them into a single element

- ... In an assignment context then “rest parameters”
  - end of function definition parameters,
  - end of destructure assignment
  - gathers the rest of the list of arguments into an array.

- ... occurs in an evaluation or expression context then is “spread operator”
  - function call
  - array literal
  - expands an array into a list

- Use patterns:
  - Rest parameters create functions that accept any number of arguments.
  - spread operator
    - pass an array to functions that require multiple individual arguments
    - or assign array elements individually into another array – like concat
    - clone an array or object (shallow clone)

Main Point: Rest parameters and spread operator

Rest parameters and the spread operator are convenience syntax for, respectively, collecting function parameters into an array or assigning collection elements across a set of variables.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Self Referral Awareness

1. Recursive functions call themselves on successively smaller parts of a program.

2. In theory all of computing can be represented by recursion. In practice, recursive functions work best for problems involving recursive data structures such as trees and linked lists.

---

3. **Transcendental consciousness**. Is the experience of the Self awake to itself.
4. **Impulses within the transcendental field:** Awareness of Awareness is the most basic process of knowledge and produces a three in one structure of knower, known, and process of knowing.
5. **Wholeness moving within itself:** In unity one perceives the basic reality of Self referral awareness as the source of all elements, matter, and existence.