DATA PROCESSING AND VALIDATION USING JAVASCRIPT

SHYAM SUNDAR DEBSARKAR

JIMMY ABU AL DENIEN

Informatics (Masters)

Winter term (15/16)

Javascript Seminar

Data Processing

- Basic manipulation and preparation of data using JavaScript for further analysis and visualization.

- Purpose of Data Processing is to focus on

– *Find data*
– *Filter data*
– *Manage data collection*
– *CRUD order (create, Read, Update and Delete)*

Data Processing Tasks

- Reading in Data
- Combining Data
- Summarizing data
- Iterating and reducing
- Working with strings
- Regular expressions
- Working with time
- Checking data assumptions

Reading in data

- The FileReader object lets web applications to read contents of files.
- It uses File or Blob objects to specify the file or data to be read.
- FileReader includes four methods for reading a file - .*readAsBinaryString (), .readAsText(), .readAsDataURL() and .readAsArrayBuffer().*
- Constructor: var reader = new FileReader();

FileReader example: The function below reads one file and displays some of the file's properties like – name, type and size.

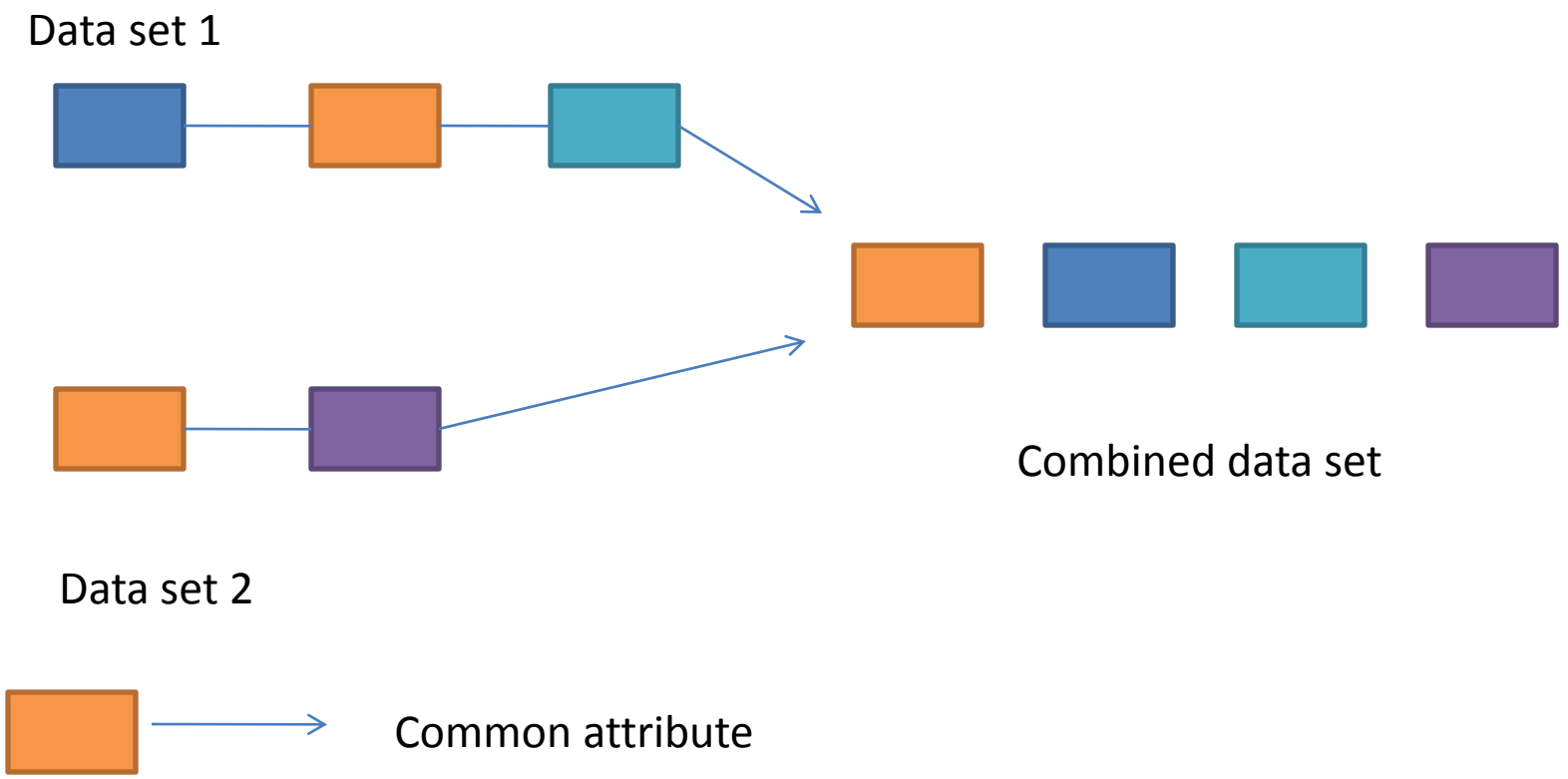
```
10  function readSingleFile(evt) {
11      var f = evt.target.files[0];
12
13    if (f) {
14      var r = new FileReader();
15      r.onload = function(e) {
16          var contents = e.target.result;
17        alert( "Got the file "
18               +"name: " + f.name + "n"
19               +"type: " + f.type + "n"
20               +"size: " + f.size + " bytesn"
21                     );
22      }
23      r.readAsText(f);
24    } else {
25      alert("Failed to load file");
26    }
27  }
28
29  document.getElementById('fileinput').addEventListener('change', readSingleFile, false);
30  </script>
```

Combining Data

- Sometimes, combination of data sets is required because they contain complementary information.

- Different ways of Combining or *merging* data :
  - *Combine data sets by one or more common attributes*
  - *Add together rows from different data sets*
  - *Combine attributes from different data sets*

Combine data sets by one or more common attributes



(Continuing)

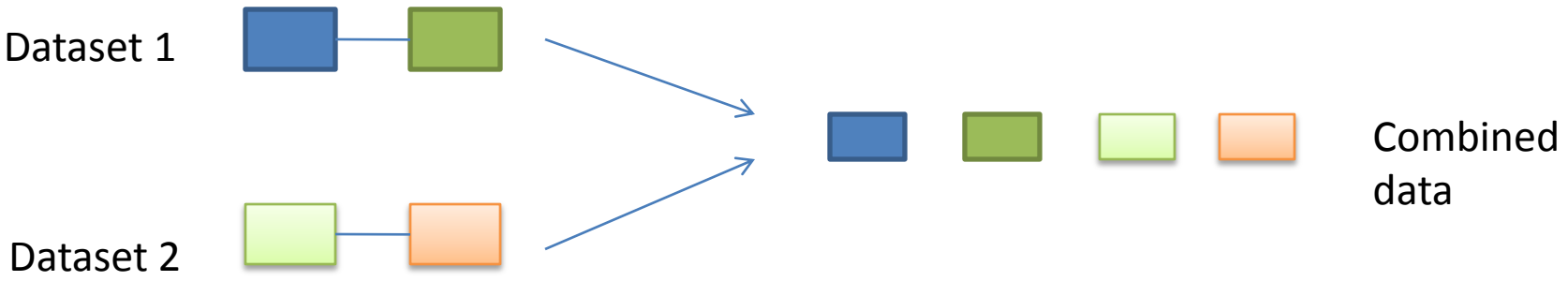
```
var articles = [{"id": 1,"name": "vacuum cleaner","brand_id": 2},
                { "id": 2,"name": "washing machine","brand_id": 1}];

var brands = [{"id": 1,"name": "SuperKitchen"
}, {"id": 2,"name": "HomeSweetHome"}];

articles.forEach(function(article) {
    var result = brands.filter(function(brand) {
        return brand.id == article.brand_id;
    });
    delete article.brand_id;
    article.brand = (result[0] != undefined) ? result[0].name : null;
});
//document.write(articles[0].brand);
for(i=0;i<articles.length;i++){
console.log(articles[i].brand,articles[i].name,articles[i].id,articles[i].brand_id);
}
```

```
HomeSweetHome vacuum cleaner 1 undefined
SuperKitchen washing machine 2 undefined
>
```

Add together rows from different data sets



- _.zip

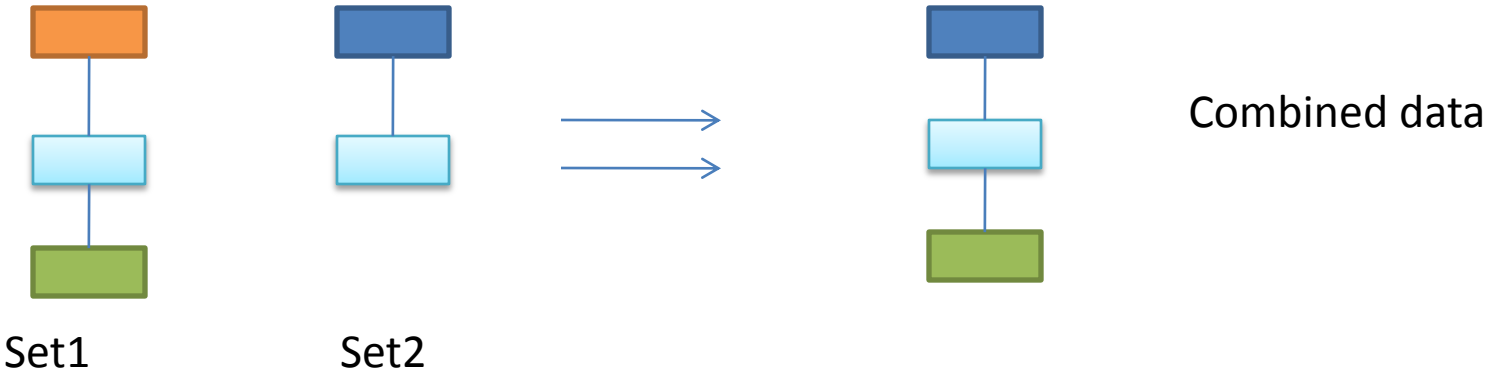
```
_.zip(['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]);

=> [["moe", 30, true], ["larry", 40, false], ["curly", 50, false]]
```

- _.extend

```
_.extend({name: 'moe'}, {age: 50});
=> {name: 'moe', age: 50}
```

Combine attributes from different data sets



- Two or more data sets that contain attributes describing the same observations, or conceptual entities, and they need to be combined.
- jQuery.js has extend method for this purpose.

(Continuing)

```
var object1 = {
   red: 0,
   blue: {lightblue:24, darkblue:12},
   black:17
};

var object2= {
   white: 200,
   blue: {lightblue:40}
};

var object3= {
   yellow:234,
   white:150
   };

var targetResult = $.extend({},object1,object2,object3);
console.log(targetResult);
```

```
                                                    playing.ht
▼Object {red: 0, blue: Object, black: 17, white: 150,
yellow: 234} i
    black: 17
  ▼blue: Object
      lightblue: 40
    ▶__proto__: Object
    red: 0
    white: 150
    yellow: 234
  ▶__proto__: Object
```

Summarizing data

- After data loading, we can find out the required data from the data set.
- Some of the important methods of **Math** object are
- .min([*value1*[, *value2*[, ...]]])
  .max([*value1*[, *value2*[, ...]]])
  .pow(*base*, *exponent*)
  .floor(*x*)
  .ceil(*x*)

Max & Min

```
var data = [
  {"city":“ACity", "population":652405},
  {"city":“BCity", "population":8405837},
  {"city":“CCity”,"population":645966},
  {"city":“DCity", "population":467007}
]; function finder(cmp, arr, attr) {
    var val = arr[0][attr];
    for(var i=1;i<arr.length;i++) {
        val = cmp(val, arr[i][attr])
    }
    return val;
}alert(finder(Math.max, data, "population"));
alert(finder(Math.min, data, "population"));
```

Iterating Over and Reducing Data

- Most of the functions used to summarize data, need to iterate over the entire dataset to generate their results.

Main Methods	Functions	Syntax
forEach()	Executes a provided function once per array element.	*arr*.forEach(*callback*[, *thisArg*])
Mapping	Creates a new array by calling a provided function on every element in this array.	arr.map(callback[, thisArg]).
Filtering	Select a subset of the data	*arr*.filter(*callback*[, *thisArg*])
Sorting	Specifies a function that defines the sort order	*arr*.sort([*compareFunction*])

forEach()

- Example:

```
var data = [{"city":"ACity", "population":652405,"land_area":83.9},
            {"city":"BCity", "population":8405837, "land_area":302.6},
            {"city":"CCity", "population":645966, "land_area":48.3},
            {"city":"DCity", "population":467007, "land_area":315} ]; var count = 0;
data.forEach(function(d) {
console.log(d.city);
count = count + 1;
});   console.log(count);
```

⇒ ACity
BCity
CCity
DCity
Total Count = 4

Filtering

- Select a subset of the data using the built in filter method.
- *Syntax: arr*.filter(*callback*[, *thisArg*]).

- Example:

```
var large_land = data.filter(function(d) { return d.land_area > 200; });
console.log(large_land);
```

=>

[{"city":“BCity“, "land_area":302.6, “population":8405837},
{“city":“DCity“, "land_area":315, “population":467007}]

Sorting

- Syntax: *arr*.sort([*compareFunction*]).
- Example:

```
data.sort(function(a,b) {
  return b.population - a.population;
});

console.log(data);
```

=> {"city":"BCity", "population":8405837, "land_area":302.6},
{"city":"ACity", "population":652405,"land_area":83.9},
{"city":"CCity", "population":645966, "land_area":48.3},
{"city":"DCity", "population":467007, "land_area":315}

Reducing

- Applies a function against an accumulator and each value of the array to reduce it to a single value.
- Syntax : *arr*.reduce(*callback*[, *initialValue*]).
- Example:

```
var landSum = data.reduce(function(sum, d) {
return sum + d.land_area;
}, 0);
console.log(landSum);
```

=> 749.8.

Working with Strings

Basic methods	Function	Syntax
Stripping whitespaces	Remove whitespace from both sides of a string.	<string>.trim()
charAt(x)	Access a character in a string using index x.	return <string>.charAt(x)
Find	Finding out a substring or extracting pieces out of a string	<string>.indexOf(<substring>);
Replace	Replacing a character/substring within string.	<string>.replace(/<substring>/g, <new substring>);

Find and replace

- Finding out a substring or extracting pieces out of a string using indexOf() method.

```
var str = "Hello world, welcome to the universe.";
  var n = str.indexOf("welcome");
```

=> 13

- Replacing a character can be done using replace() method.

```
var str = "Mr Blue has a blue house and a blue car";
  var res = str.replace(/blue/g, "red");
```

=> Mr Blue has a red house and a red car

Regular Expressions

- Used to match certain patterns of strings within other strings.
- Useful tool for extracting *patterns* rather than exact strings.
- Two RegExp object methods (exec and test) and four String methods (match, search, replace and split) are used.
- Two types of syntax
    1. `var re = /ab+c/;`
    2. `var re = new RegExp("ab+c");`

Examples of regular expressions

- **Finding Strings**

```
var str = "how much wood would a  woodchuck chuck if a woodchuck could
    chuck wood";
   var regex = /wood/;
  if (regex.test(str)) {
  console.log("we found 'wood' in the string!");
  }
```

⇒ "we found 'wood' in the string!“

- **Replacing with regular expressions**

```
regex = /wood/g;
var newstr = str.replace(regex, "nun");
console.log(newstr);
```

=> "how much nun would a nunchuck chuck if a nunchuck could chuck nun"

Working with Time

- Data contain dates or times in an (mostly) arbitrary format and need to force that into an actual date.

```
var d = new Date ("2015-12");          // Tue Dec 01 2015 05:30:00
```

- We can also refer Date.js or moment.js libraries for methods to do this parsing.

E.g.
```
Date.parse('9/16/2015') ;          // Wed Sep 16 2015 00:00:00 GMT+0530
moment("12-25-1995", "MM-DD-YYYY");
                                   // Mon Dec 25 1995 00:00:00
```

- We can perform time modification.

Examples: moment().format().

Checking Data Assumptions

- Mistakes in data processing can be taken care by checking the assumptions about the shape and contents of data..
- Assertions :
- These tests can be created with assertions - functions that check the truthiness of a statement in code.

Example:

```
function assert(isTrue, message) {
  if(!isTrue) {
    console.log(message);
    return false;
  }
  return true;
}
```

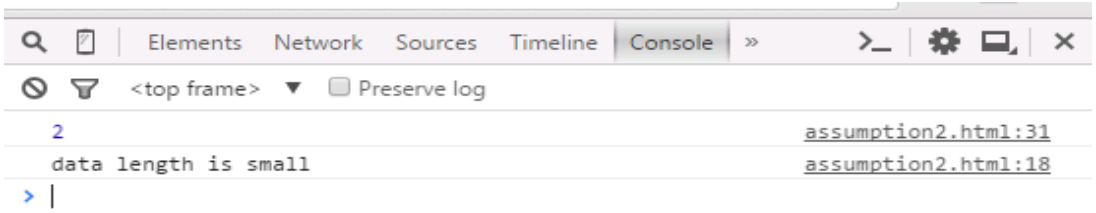
Data Shape Assumptions: Using the assert () function to check some assumptions about the shape of data.

```
var data = [{"name":23,
             "age":23,
             "student":true},
            {"name":"Sleepwalker",
             "age":NaN,
             "student":false}
];


function assert(isTrue, message) {
  if(!isTrue) {
    console.log(message);
    return false;

  } else{
  return true;}
}

function checkDataShape(data) {
  assert(data.length > 0, "data is empty");
  assert(data.length > 4, "data length is small");
}
console.log(data.length);
checkDataShape(data);
```



Data Content Assumptions : Checking the content of the data is correct or not.

```
var data = [{"name":23,
             "age":23,
             "student":true},
            {"name":"Sleepwalker",
              "age":NaN,
              "student":false}
];


function assert(isTrue, message) {
  if(!isTrue) {
    document.write(message);
    return false;

  } else{
  return true;}
}

function checkDataContent(d){
for(i=0;i<d.length;i++){


assert (typeof(d[i].name) == 'string',i+" no. element's name should be character, ");
assert (!isNaN(d[i].age), i+" no. element's age should be number, ");
}

}

checkDataContent(data);
```

0 no. element's name should be character, 1 no. element's age should be number,

Data Validation Approaches

- The process of ensuring that user input is clean, correct, and as expected.

- **Ways of validation**:
  - **Server side validation** → performed by a web **server**, after input has been submitted into the server.

  - **Client side validation** → performed by a web **browser**, before input is submitted into a web server. (we will discussed today).

Validation Techniques

- **JavaScript Validation API *(Without external frameworks)***
  - *Properties, Methods*
  - *Form Validation*
  - *Validation Using Regex*

- **External Frameworks**
  - *Parsley.js, Validate.js, Verify.js*

JavaScript Validation API

For every DOM control (*ex:Input*), the default JS API provides set of **methods** and **properties** that can be performed and checked.

```
<input id="id1" type="number" min="100" max="300">
```

- **checkValidity():** Returns **true** if an input element contains valid data.
- **setCustomValidity():** Sets the validationMessage property of an input element.
- **Properties:** validity, validationMessage, willValidate
  - **validity**: rangeOverflow, rangeUnderflow, tooLong, typeMismatch..etc
  - **willValidate**: Indicates if an input element will be validated.

Example

```
<input id="id1" type="number" min="100" max="300">
function myFunction() {
    var inpObj = document.getElementById("id1");
    if (inpObj.checkValidity() == false)
    { alert( inpObj.validationMessage);}

   if(inpObj .validity.rangeOverflow)
    { alert("Value too large"); }
```

JavaScript Form Validation

Validating HTML forms usually performed on submitting, example:

```
<form name="myForm“ onsubmit="return validateForm()“ method="post">
    <input type="text" name="fname">

var x = document.forms["myForm"]["fieldname"].value;
   if (x == null || x == "") {
   alert("Name must be filled out");
   return false; }
```

Here comes the HTML5 power, it let the browser understand and do the validation for you. No JS function or logic needed:

```
<input type="text" required>
```

Complex Constraints?

When developing complex constrains the DOM properties or a simple validation JS function is not enough / efficient.

For example if we want provide a validation logic for ***(Alphabets, numbers and space(' ') no special characters min 3 and max 20 characters)*** *how would you do that?*

Parse the value and test it against each of the rules? Complex logic / code? Efficient? Bug free?

JavaScript Validation using Regular Expressions

To reduce the complexity of our validation logic and improve it's efficiency we can use **RegEx.**

Solution for: ***Alphabets, numbers and space(' ') no special characters min 3 and max 20 characters:***

```
var ck_name = /^[A-Za-z0-9 ]{3,20}$/;
if (!ck_name.test({nameFieldValue}))
{
    alert("Invalid value");
}
```

JavaScript Validation Frameworks Motivation

- **Don't Reinvent The Wheel**
  - Why write code that's already been written (better)?
- **Do More With Less Code**
  - Pre-Defined utility functions

JS frameworks define multiple attributes that get translated into special actions at runtime.

Parsley.js

Usage (some of what can be done):

- Form definition: **<form** id="demo-form" data-parsley-validate**>**

- Field validation on change event: **<input** type="text" data-parsley-trigger="change" **/>**

- Password matching: data-parsley-equalto="{FieldName}"

- Custom validation rules (custom logic):

DOM: **<input** type="text" data-parsley-userexist **/>**

JS: window.ParsleyValidator.addValidator('userexist', function(value, requirement) {//some code});

Verify.js

Usage (some of what can be done):

- Field validation as number, email, url or required (one or multiple): data-validate="{number,email,url,required}"

- validate using RegEx: data-validate="regex(abc,Must contain abc)"

- Custom Validation/function: data-validate="customValidationFcn"

Some other properties:

hideErrorOnChange→Hide error while the user is editing the field

beforeSubmit→Pre-form-submit hook/function. If returns true, form will submit

Validate.js:

- Introduces new object “FormValidator”
- Definition: var validator = new FormValidator ({formName}, fields, callback)
- Callback: A function that will fire after all validation rules are success.
- Fields: Array of the form fields that’s being constructed as
  - “name” - required →html field name,
  - “rules” – required →either a predefined or custom rule,
  - “depends” - optional→a function that runs before the field is validated, if it returns “false” the field will not be validated.

I don’t like:

Parsley.js: The documentation is not clear for the advanced usage. I think it can be done better than this.

Verify.js: Not a good option for big complicated applications in terms of features *(ex: no pre-defined validate on “change” event).*

Validate.js: Usability and readability are not good *(ex: no inline DOM rules definition)*, only basic pre-defined functionality.

I like:

Parsley.js: Big community, powerful ready features (just add a DOM attribute), code readability is very good.

Verify.js: Simple, ease-of-use, good readability and clear documentation. Would definitely use for simple application.

Validate.js: No dependencies (not even JQuery), JS util code written for you.

My Opinion

	JavaScript API	Parsley.js	Verify.js	Validate.js
	★ ★	★ ★ ★ ★	★ ★ ★	★ ★ ★
Rely on JQuery?	NO	YES	YES	NO
Useability	Fair	Very Good	Good	Fair
Productivity	Fair	Very Good	Good	Good
Perfomance	Excelent	Very Good	Very Good	Very Good
Functionality	Good	Very Good	Good	Good
Size	None	6.6 KB	25 KB	7 KB
Community	Excelent	Very Good	Good	Good
Documentation	Excelent	Good	Very Good	Good

References

Janko Jovanovic, Web Form Validation: Best Practices and Tutorials, http://www.smashingmagazine.com/2009/07/web-form-validation-best-practices-and-tutorials/

parsley.js, http://parsleyjs.org/

validate.js, http://rickharrison.github.io/validate.js/

verify.js, http://verifyjs.com/

https://developer.mozilla.org/en-US/docs/Web/JavaScript

http://www.html5rocks.com/en/tutorials/file/dndfiles/

http://www.slideshare.net/quentinadam/javascript-as-data-processing-language-html5-integration

https://lodash.com/docs

https://www.dashingd3js.com/table-of-contents

Thank you for your attention

Questions?