UNIVERSITÄT BERN

Übung Open Data:

Einführung Web Programmierung und verwendete Tools

Termin 4, 12. März 2015

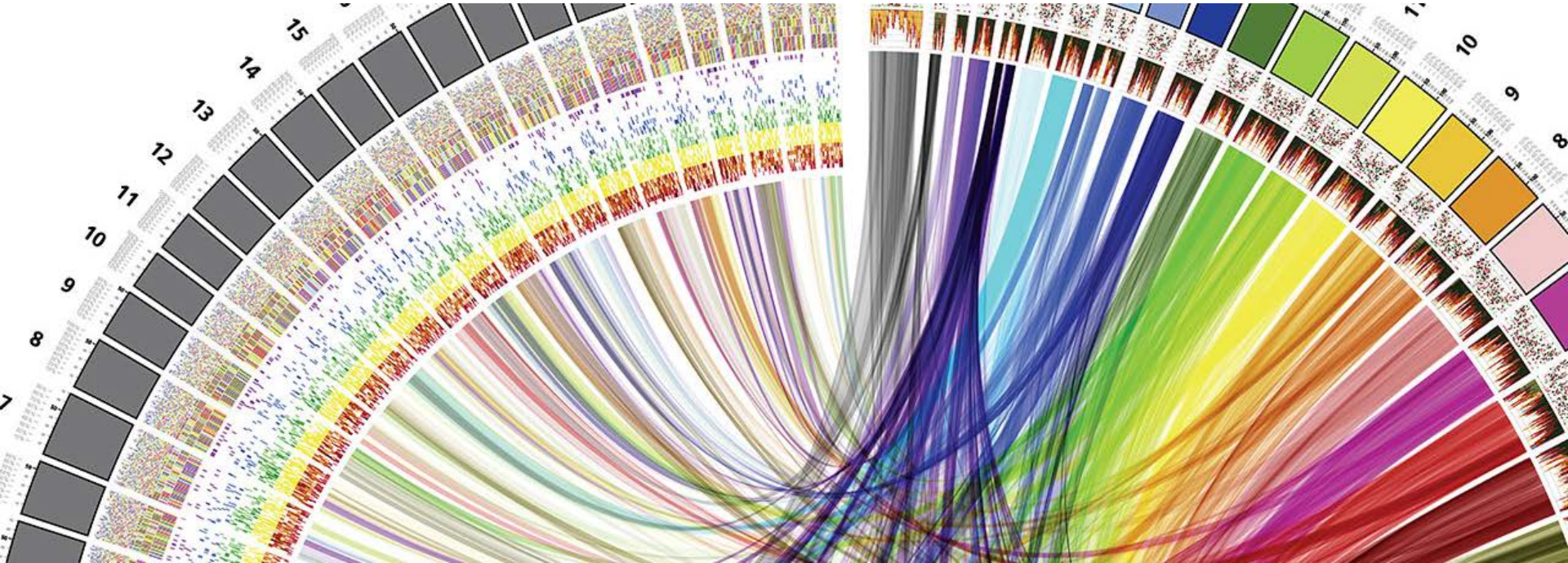
Dr. Matthias Stürmer und Prof. Dr. Thomas Myrach

Universität Bern, Institut für Wirtschaftsinformatik
Abteilung Informationsmanagement
Forschungsstelle Digitale Nachhaltigkeit

UNIVERSITÄT BERN

Agenda

1. Fortsetzung Einführung JavaScript Programmierung und SVG

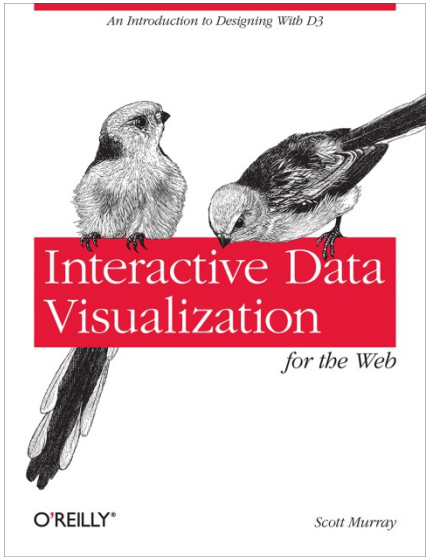


Agenda

Chapter 3. Technology Fundamentals:

http://chimera.labs.oreilly.com/books/1230000000345/ch03.html

1. Web Servers
2. Hypertext Markup Language HTML
3. Cascading Style Sheets CSS
4. **JavaScript**
5. Scalable Vector Graphics SVG



Referencing JavaScript files

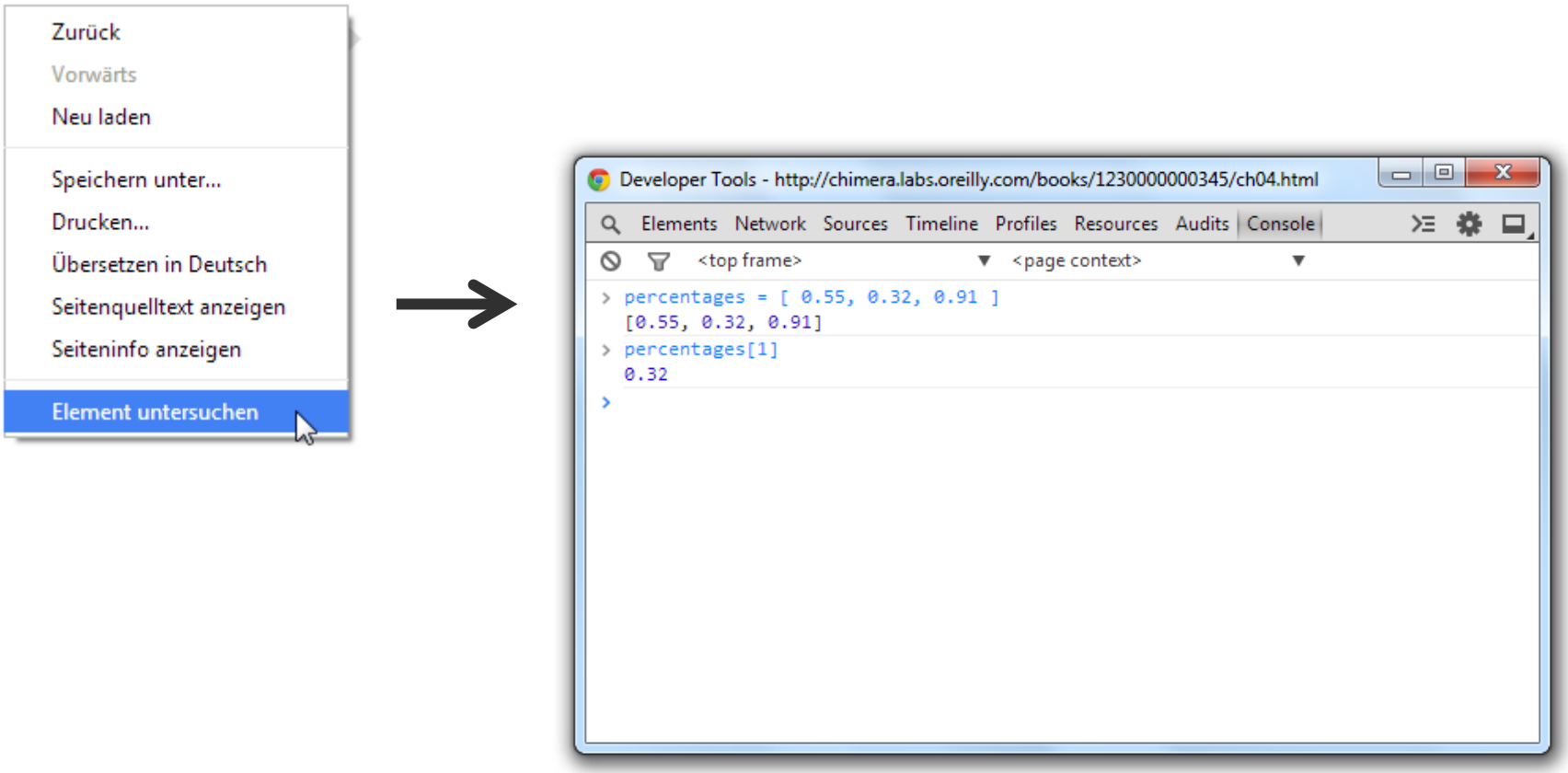
- Scripts can be included directly in HTML between two script tags:

```
<body>
    <script type="text/javascript">
        alert("Hello, world!");
    </script>
</body>
```

- or stored in a separate file with a .js suffix, and then referenced somewhere in the HTML (could be in the head, as shown here, or also just before the end of the closing body tag):

```
<head>
    <title>Page Title</title>
    <script type="text/javascript" src="myscript.js"></script>
</head>
```

JavaScript Developer Console



Variables

- Variables are containers for data. A simple variable holds one value:

```
var number = 5;
```

- In that statement, **`var`** indicates you are declaring a new variable, the name of which is number. The equals sign is an *assignment operator* because it takes the value on the right (5) and *assigns* it to the variable on the left (number).
- A variable is a datum, the smallest building block of data. The variable is the foundation of all other data structures, which are simply different configurations of variables.
- More examples:

```
var defaultColor = "hot pink";
var thisMakesSenseSoFar = true;
```

Arrays

- Keeping track of related values in separate variables is inefficient:

```
var numberA = 5;
var numberB = 10;
var numberC = 15;
var numberD = 20;
var numberE = 25;
```

- Rewritten as an array, those values are much simpler. Hard brackets [] indicate an array, and each value is separated by a comma:

```
var numbers = [ 5, 10, 15, 20, 25 ];
```

- You can access a value in an array by using *bracket notation*:

```
numbers[0]  //Returns 5
numbers[1]  //Returns 10
numbers[2]  //Returns 15
```

Arrays

- Arrays can contain any type of data, not just integers:

```
var percentages = [ 0.55, 0.32, 0.91 ];
var names = [ "Ernie", "Bert", "Oscar" ];

percentages[1]  //Returns 0.32
names[1]        //Returns "Bert"
```

- Although I don't recommend it, different types of values can even be stored within the same array:

```
var mishmash = [ 1, 2, 3, 4.5, 5.6, "oh boy",
"say it isn't", true ];
```

Objects

- Think of a JavaScript object as a custom data structure. We use curly brackets **{}** to indicate an object. In between the brackets, we include **properties** and **values**. A colon **:** separates each property and its value, and a comma separates each property/value pair:

```
var fruit = {
    kind: "grape",
    color: "red",
    quantity: 12,
    tasty: true
};
```

- To reference each value, we use dot notation, specifying the name of the property:

```
fruit.kind      //Returns "grape"
fruit.color     //Returns "red"
fruit.quantity  //Returns 12
fruit.tasty     //Returns true
```

Objects and Arrays

- You can combine these two structures to create arrays of objects, or objects of arrays, or objects of objects or, well, basically whatever structure makes sense for your dataset.
- Let's say we have acquired a couple more pieces of fruit, and we want to expand our catalog accordingly. We use hard brackets [] on the outside, to indicate an array, followed by curly brackets{} and object notation on the inside, with each object separated by a comma:

```
var fruits = [
    {
        kind: "grape",
        color: "red",
        quantity: 12,
        tasty: true
    },
    {
        kind: "kiwi",
        color: "brown",
        quantity: 98,
        tasty: true
    },
    {
        kind: "banana",
        color: "yellow",
        quantity: 0,
        tasty: true
    }
];
```

Objects and Arrays

- To access this data, we just follow the trail of properties down to the values we want. Remember, **[] means array,** and **{} means object.** fruits is an array, so first we use bracket notation to **specify an array** index:

```
fruits[1]
```

- Next, each array element is an object, just add a dot and a property:

```
fruits[1].quantity   //Returns 98
```

- Access values in the fruits array of objects:

```
fruits[0].kind      ==  "grape"
fruits[0].color     ==  "red"
fruits[0].quantity  ==  12
fruits[0].tasty     ==  true
```

JSON

- JSON = JavaScript Object Notation:

```
{
    "kind": "grape",
    "color": "red",
    "quantity": 12,
    "tasty": true
}
```

- The only difference here is that our property names are now surrounded by double quotation marks "", making them string values.
- JSON objects, like all other JavaScript objects, can be stored in variables like so:

```
var jsonFruit = {
    "kind": "grape",
    "color": "red",
    "quantity": 12,
    "tasty": true
};
```

GeoJSON

```
{
    "type": "FeatureCollection",
    "features": [
        {
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [ 150.1282427, -24.471803 ]
            },
            "properties": {
                "type": "town"
            }
        }
    ]
}
```

cantons.json


```
[
    {
        "name": "Bern",
        "numbers": [
            {"name": "Einwohnerzahl", "value": 1408575},
            {"name": "Fläche", "value": 5959.1},
            {"name": "Wähleranteile in % FDP", "value": 8.7},
            {"name": "Wähleranteile in % CVP", "value": 2.1},
            {"name": "Wähleranteile in % SP", "value": 19.3},
            {"name": "Wähleranteile in % SVP", "value": 29.0},
            {"name": "Wähleranteile in % EVP/CSP", "value": 4.2},
            {"name": "Wähleranteile in % GLP", "value": 5.3},
            {"name": "Wähleranteile in % BDP", "value": 14.9},
            {"name": "Wähleranteile in % PdA/Sol.", "value": 0.3},
            {"name": "Wähleranteile in % GPS", "value": 9.4},
            {"name": "Wähleranteile in % kleine Rechtsparteien", "value": 3.7}
        ]
    },
        ...
```

Mathematical and Comparison Operators

```
1 + 2      //Returns 3
10 - 0.5   //Returns 9.5
33 * 3     //Returns 99
3 / 4      //Returns 0.75

==         //Equal to
!=         //Not equal to
<          //Less than
>          //Greater than
<=         //Less than or equal to
>=         //Greater than or equal to
```

Übung:

```
3 == 3
3 == 5
3 >= 3
3 >= 2
100 < 2
298 != 298
```

Control Structure: if()

- If the test between parentheses is **true,** then the code between the curly brackets is run. If the test turns up **false,** then the bracketed code is ignored, and life goes on. (Technically, life goes on either way.)

```
if (3 < 5) {
     Eureka! Three is less than five!";
}
```

- In the preceding example, the bracketed code will always be executed, because 3 < 5 is always true.if statements are more useful when comparing variables or other conditions that change.

Control Structure: for()

- You can use for loops to repeatedly execute the same code, with slight variations.
- They are so-called because they loop through the code *for* as many times as specified. First, the initialization statement is run. Then, the test is evaluated, like a mini if statement. If the test is true, then the bracketed code is run. Finally, the update statement is run, and the test is reevaluated.
- The most common application of a for loop is to increase some variable by 1 each time through the loop. The test statement can then control how many times the loop is run by referencing that value. (The variable is often named i, purely by convention, because it is short and easy to type.)

```
for (var i = 0; i < 5; i++) {
    console.log(i);  //Prints value to console
}
```

What arrays are made for()

- An array organizes lots of data values in one convenient place. Then **for()** can quickly “loop” through every value in an array and perform some action with it—such as, express the value as a visual form. D3 often manages this looping for us, such as with its magical **data()** method.

```
var numbers = [ 8, 100, 22, 98, 99, 45 ];
for (var i = 0; i < numbers.length; i++) {
    console.log(numbers[i]);  //Print value to console
}
```

- **length** is a property of every array. In this case,numbers contains six values, so **numbers.length** resolves to 6, and the loop runs six times. If numberswere 10 positions long, the loop would run 10 times.

Functions

- Functions can take arguments or parameters as input, and then return values as output. Parentheses are used to ***call* (execute)** a function. If that function requires any **arguments (input values),** then they are *passed* to the function by including them in the parentheses.

```
var calculateTip = function(bill) {
    return bill * 0.2;
};
calculateTip(38);
```

- Beispiel einer anonymen Funktion aus der Open Data App:

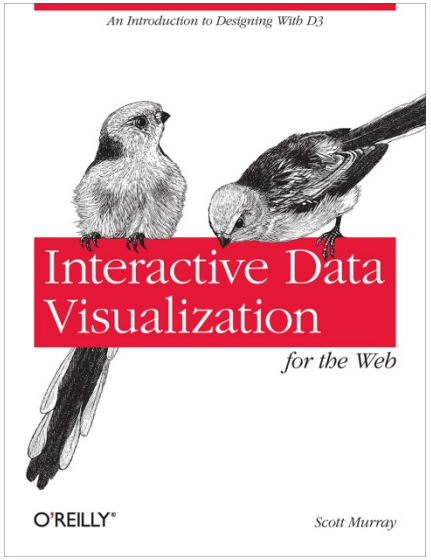
```
d3.json('cantons.json',function(err, data){
        var cantons = svg.selectAll('g').data(data);
```

Agenda

Chapter 3. Technology Fundamentals:

http://chimera.labs.oreilly.com/books/1230000000345/ch03.html

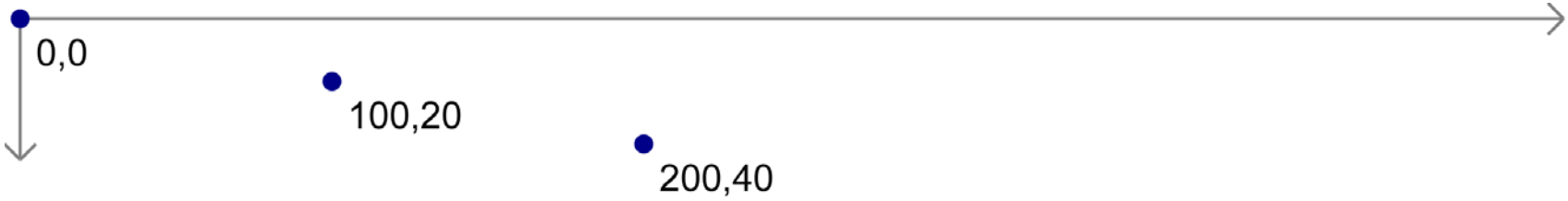
1. Web Servers
2. Hypertext Markup Language HTML
3. Cascading Style Sheets CSS
4. JavaScript
5. **Scalable Vector Graphics SVG**



The SVG Element

- Before you can draw anything, you must create an SVG element. Think of the SVG element as a canvas on which your visuals are rendered. (In that respect, SVG is conceptually similar to HTML's canvas element.)
- At a minimum, it's good to specify width and height values. If you don't specify these, the SVG will behave like a typically greedy, block-level HTML element and take up as much room as it can within its enclosing element:

```
<svg width="500" height="50">
</svg>
```



Simple Shapes

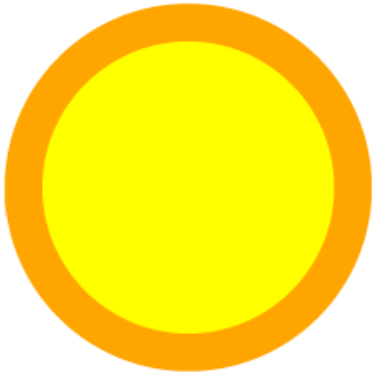
Einige Beispiele:

```
<rect x="0" y="0" width="500" height="50"/>

<circle cx="250" cy="25" r="25"/>

<ellipse cx="250" cy="25" rx="100" ry="25"/>

<line x1="0" y1="0" x2="500" y2="50"
stroke="black"/>

<text x="250" y="25" font-family="serif" font-
size="25" fill="gray">Easy-peasy</text>
```

Styling SVG Elements

fill	A color value. Just as with CSS, colors can be specified as named colors, hex values, or RGB or RGBA values
stroke	A color value
stroke-width	A numeric measurement (typically in pixels)
opacity	A numeric value between 0.0 (completely transparent) and 1.0 (completely opaque)

```
<circle cx="25" cy="25"
r="22"  fill="yellow"
stroke="orange" stroke-width="5"/>
```



SVG und CSS

Alternatively, we could strip the style attributes and assign the circle a class (just as if it were a normal HTML element):

```
<circle cx="25" cy="25" r="22" class="pumpkin"/>
```

and then put the fill, stroke, and stroke-width rules into a CSS style that targets the new class:

```
.pumpkin {
    fill: yellow;
    stroke: orange;
    stroke-width: 5;
 }
```

The CSS approach has a few obvious benefits:

- You can specify a style once and have it applied to multiple elements.
- CSS code is easier to read than inline attributes.
- For those reasons, the CSS approach might be more maintainable and make design changes faster to implement.