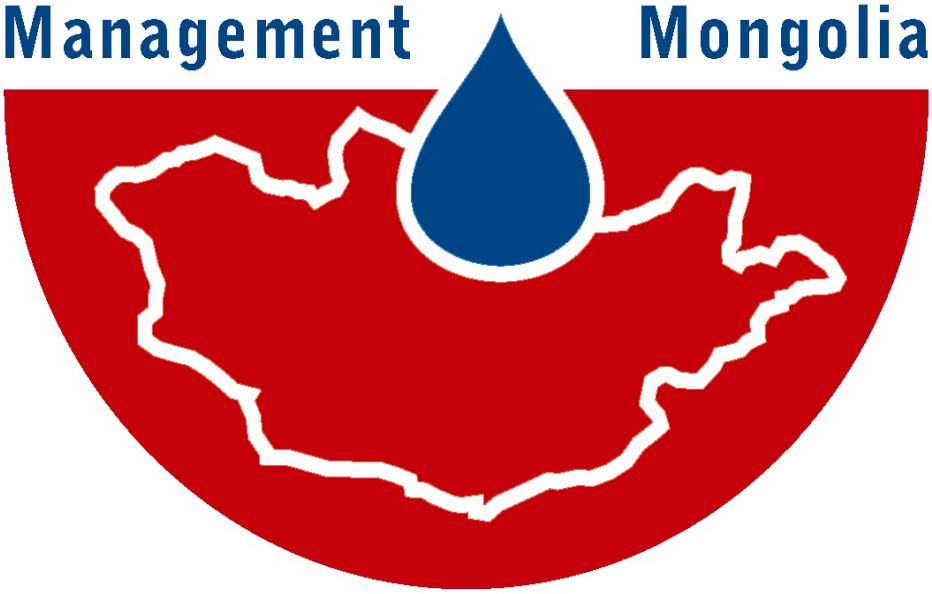
MoMo

Integrated Water Resources Management Mongolia



MoMo3 Workshop

2016-02-22 – 2016-02-26

Table of Contents

Welcome to the workshop

Introduction to core technologies behind the MoMo geoportal

Sources

- Workshop URL
- Download workshop (ZIP)
- Download workshop (PDF)
- Download workshop (EPUB)

Authors

- Marc Jansen (jansen@terrestris.de)
- Daniel Koch (koch@terrestris.de)

Overview

GeoServer

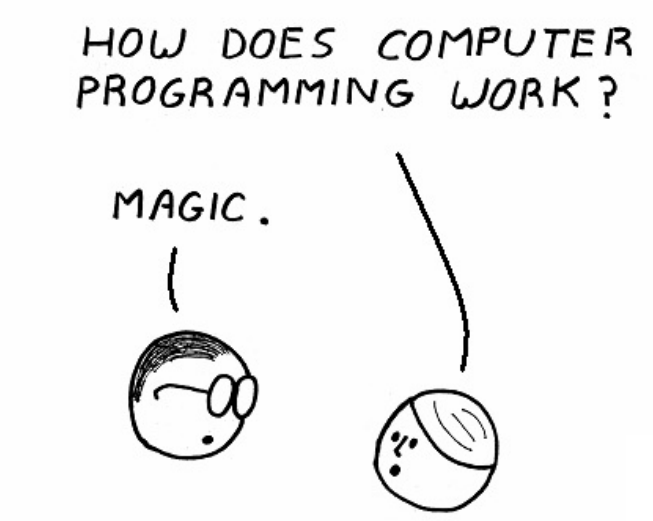
JavaScript

- Beginners

The beginners section is based on the work done in this repository. Thank you very much!

Learn Javascript

This book will teach you the basics of programming and Javascript. Whether you are an experienced programmer or not, this book is intended for everyone who wishes to learn the JavaScript programming language.



Screen

JavaScript (*JS for short*) is the programming language that enables web pages to respond to user interaction beyond the basic level. It was created in 1995, and is today one of the most famous and used programming languages.

Basics about Programming

In this first chapter, we'll learn the basics of programming and the Javascript language.

Programming means writing code. A book is made up of chapters, paragraphs, sentences, phrases, words and finally punctuation and letters, likewise a program can be broken down into smaller and smaller components. For now, the most important is a statement. A statement is analogous to a sentence in a book. On its own, it has structure and purpose, but without the context of the other statements around it, it isn't that meaningful.

A statement is more casually (and commonly) known as a *line of code*. That's because statements tend to be written on individual lines. As such, programs are read from top to bottom, left to right. You might be wondering what code (also called source code) is. That happens to be a broad term which can refer to the whole of the program or the smallest part. Therefore, a line of code is simply a line of your program.

Here is a simple example:

```
var hello = "Hello";
var world = "World";

// Message equals "Hello World"
var message = hello + " " + world;
```

This code can be executed by another program called an *interpreter* that will read the code, and execute all the statements in the right order.

Comments

Comments are statements that will not be executed by the interpreter, comments are used to mark annotations for other programmers or small descriptions of what your code does, thus making it easier for others to understand what your code does.

In Javascript, comments can be written in 2 different ways:

- Line starting with `//` :

```
// This is a comment, it will be ignored by the interpreter
var a = "this is a variable defined in a statement";
```

- Section of code starting with `/*` and ending with `*/` , this method is used for multi-line comments:

```
/*
This is a multi-line comment,
it will be ignored by the interpreter
*/
var a = "this is a variable defined in a statement";
```

Exercise

Mark the editor's contents as a comment

```
Mark me as a comment
or I'll throw an error
```

Variables

The first step towards really understanding programming is looking back at algebra. If you remember it from school, algebra starts with writing terms such as the following.

```
3 + 5 = 8
```

You start performing calculations when you introduce an unknown, for example x below:

```
3 + x = 8
```

Shifting those around you can determine x:

```
x = 8 - 3
-> x = 5
```

When you introduce more than one you make your terms more flexible - you are using variables:

```
x + y = 8
```

You can change the values of x and y and the formula can still be true:

```
x = 4
y = 4
```

or

```
x = 3
y = 5
```

The same is true for programming languages. In programming, variables are containers for values that change. Variables can hold all kind of values and also the results of computations. Variables have a name and a value separated by an equals sign (=). Variable names can be any letter or word, but bear in mind that there are restrictions from language to language of what you can use, as some words are reserved for other functionality.

Let's check out how it works in Javascript, The following code defines two variables, computes the result of adding the two and defines this result as a value of a third variable.

```
var x = 5;
var y = 6;
var result = x + y;
```

Variable types

Computers are sophisticated and can make use of more complex variables than just numbers. This is where variable types come in. Variables come in several types and different languages support different types.

The most common types are:

- **Numbers**
  - **Float**: a number, like 1.21323, 4, -33.5, 100004 or 0.123
  - **Integer**: a number like 1, 12, -33, 140 but not 1.233
- **String**: a line of text like "boat", "elephant" or "damn, you are tall!"
- **Boolean**: either true or false, but nothing else
- **Arrays**: a collection of values like: 1,2,3,4,'I am bored now'
- **Objects**: a representation of a more complex object
- **null**: a variable that contains null contains no valid Number, String, Boolean, Array, or Object
- **undefined**: the undefined value is obtained when you use an object property that does not exist, or a variable that has been declared, but has no value assigned to it.

JavaScript is a *"loosely typed"* language, which means that you don't have to explicitly declare what type of data the variables are. You just need to use the `var` keyword to indicate that you are declaring a variable, and the interpreter will work out what data type you are using from the context, and use of quotes.

Exercise

Create a variable named `a` using the keyword `var`.

Equality

Programmers frequently need to determine the equality of variables in relation to other variables. This is done using an equality operator.

The most basic equality operator is the `==` operator. This operator does everything it can to determine if two variables are equal, even if they are not of the same type.

For example, assume:

```
var foo = 42;
var bar = 42;
var baz = "42";
var qux = "life";
```

`foo == bar` will evaluate to `true` and `baz == qux` will evaluate to `false`, as one would expect. However, `foo == baz` will *also* evaluate to `true` despite `foo` and `baz` being different types. Behind the scenes the `==` equality operator attempts to force its operands to the same type before determining their equality. This is in contrast to the `===` equality operator.

The `===` equality operator determines that two variables are equal if they are of the same type *and* have the same value. With the same assumptions as before, this means that `foo === bar` will still evaluate to `true`, but `foo === baz` will now evaluate to `false`. `baz === qux` will still evaluate to `false`.

Numbers

JavaScript has **only one type of numbers** – 64-bit float point. It's the same as Java's `double`. Unlike most other programming languages, there is no separate integer type, so 1 and 1.0 are the same value.

In this chapter, we'll learn how to create numbers and perform operations on them (like additions and subtractions).

Creation

Creating a number is easy, it can be done just like for any other variable type using the `var` keyword.

Numbers can be created from a constant value:

```
// This is a float:
var a = 1.2;

// This is an integer:
var b = 10;
```

Or from the value of another variable:

```
var a = 2;
var b = a;
```

Exercise

Create a variable `x` which equals `10` and create a variable `y` which equals `a`.

```
var a = 11;
```

Operators

You can apply mathematic operations to numbers using some basic operators like:

- **Addition**: `c = a + b`
- **Subtraction**: `c = a - b`
- **Multiplication**: `c = a * b`
- **Division**: `c = a / b`

You can use parentheses just like in math to separate and group expressions: `c = (a / b) + d`

Exercise

Create a variable `x` equal to the sum of `a` and `b` divided by `c` and finally multiplied by `d`.

```
var a = 2034547;
var b = 1.567;
var c = 6758.768;
var d = 45084;

var x =
```

Advanced Operators

Some advanced operators can be used, such as:

- **Modulus (division remainder)**: `x = y % 2`
- **Increment**: Given a = 5
  - `c = a++` , Results: c = 5 and a = 6
  - `c = ++a` , Results: c = 6 and a = 6
- **Decrement**: Given a = 5
  - `c = a--` , Results: c = 5 and a = 4
  - `c = --a` , Results: c = 4 and a = 4

Exercise

Define a variable `c` as the modulus of the decremented value of `x` by 3.

```
var x = 10;

var c =
```

Strings

JavaScript strings share many similarities with string implementations from other high-level languages. They represent text based messages and data.

In this course we will cover the basics. How to create new strings and perform common operations on them.

Here is an example of a string:

```
"Hello World"
```

Creation

You can define strings in JavaScript by enclosing the text in single quotes or double quotes:

```
// Single quotes can be used
var str = 'Our lovely string';

// Double quotes as well
var otherStr = "Another nice string";
```

In Javascript, Strings can contain UTF-8 characters:

```
"▫▫ español English ▫▫▫▫▫▫ العربية português ▫▫▫▫▫ русский ▫▫▫ ▫▫▫▫▫▫ ▫▫▫";
```

Note: Strings can not be subtracted, multiplied or divided.

Exercise

Create a variable named `str` set to the value `'"abc"'`.

Concatenation

Concatenation involves adding two or more strings together, creating a larger string containing the combined data of those original strings. This is done in JavaScript using the + operator.

```
var bigStr = 'Hi ' + 'JS strings are nice ' + 'and ' + 'easy to add';
```

Exercise

Add up the different names so that the `fullName` variable contains John's complete name.

```
var firstName = "John";
var lastName = "Smith";

var fullName =
```

Length

It's easy in Javascript to know how many characters are in string using the property `.length`.

```
// Just use the property .length
var size = 'Our lovely string'.length;
```

Note: Strings can not be substracted, multiplied or divided.

Exercise

Store in the variable named `size` the length of `str`.

```
var str = "Hello World";

var size =
```

Conditional Logic

A condition is a test for something. Conditions are very important for programming, in several ways:

First of all conditions can be used to ensure that your program works, regardless of what data you throw at it for processing. If you blindly trust data, you’ll get into trouble and your programs will fail. If you test that the thing you want to do is possible and has all the required information in the right format, that won’t happen, and your program will be a lot more stable. Taking such precautions is also known as programming defensively.

The other thing conditions can do for you is allow for branching. You might have encountered branching diagrams before, for example when filling out a form. Basically, this refers to executing different “branches” (parts) of code, depending on if the condition is met or not.

In this chapter, we'll learn the base of conditional logic in Javascript.

Condition If

The easiest condition is an if statement and its syntax is `if(condition){ do this … }` . The condition has to be true for the code inside the curly braces to be executed. You can for example test a string and set the value of another string dependent on its value:

```
var country = 'France';
var weather;
var food;
var currency;

if(country === 'England') {
    weather = 'horrible';
    food = 'filling';
    currency = 'pound sterling';
}

if(country === 'France') {
    weather = 'nice';
    food = 'stunning, but hardly ever vegetarian';
    currency = 'funny, small and colourful';
}

if(country === 'Germany') {
    weather = 'average';
    food = 'wurst thing ever';
    currency = 'funny, small and colourful';
}

var message = 'this is ' + country + ', the weather is ' +
            weather + ', the food is ' + food + ' and the ' +
            'currency is ' + currency;
```

Note: Conditions can also be nested.

Exercise

Fill up the value of `name` to validate the condition.

```
var name =

if (name === "John") {

}
```

Else

There is also an `else` clause that will be applied when the first condition isn't true. This is very powerful if you want to react to any value, but single out one in particular for special treatment:

```
var umbrellaMandatory;

if(country === 'England'){
    umbrellaMandatory = true;
} else {
    umbrellaMandatory = false;
}
```

The `else` clause can be joined with another `if`. Lets remake the example from the previous article:

```
if(country === 'England') {
    ...
} else if(country === 'France') {
    ...
} else if(country === 'Germany') {
    ...
}
```

Exercise

Fill up the value of `name` to validate the `else` condition.

```
var name =

if (name === "John") {

} else if (name === "Aaron") {
    // Valid this condition
}
```

Comparators

Lets now focus on the conditional part:

```
if (country === "France") {
    ...
}
```

The conditional part is the variable `country` followed by the three equal signs (`===`). Three equal signs tests if the variable `country` has both the correct value (`France`) and also the correct type (`String`). You can test conditions with double equal signs, too, however a conditional such as `if (x == 5)` would then return true for both `var x = 5;` and `var x = &quot;5&quot;;` . Depending on what your program is doing, this could make quite a difference. It is highly recommended as a best practice that you always compare equality with three equal signs (`===` and `!==`) instead of two (`==` and `!=`).

Other conditional test:

- `x &gt; a` : is x bigger than a?
- `x &lt; a` : is x less than a?
- `x &lt;= a` : is x less than or equal to a?
- `x &gt;=a` : is x greater than or equal to a?
- `x != a` : is x not a?
- `x` : does x exist?

Exercise

Add a condition to change the value of `a` to the number 10 if `x` is bigger than 5.

```
var x = 6;
var a = 0;
```

Logical Comparison

In order to avoid the if-else hassle, simple logical comparisons can be utilised.

```
var topper = (marks > 85) ? "YES" : "NO";
```

In the above example, `?` is a logical operator. The code says that if the value of marks is greater than 85 i.e. `marks &gt; 85` , then `topper = YES` ; otherwise `topper = NO` . Basically, if the comparison condition proves true, the first argument is accessed and if the comparison condition is false , the second argument is accessed.

Concatenate conditions

Furthermore you can concatenate different conditions with "or" or “and” statements, to test whether either statement is true, or both are true, respectively.

In JavaScript “or” is written as `||` and “and” is written as `&&`.

Say you want to test if the value of x is between 10 and 20—you could do that with a condition stating:

```
if(x > 10 && x < 20) {
    ...
}
```

If you want to make sure that country is either “England” or “Germany” you use:

```
if(country === 'England' || country === 'Germany') {
    ...
}
```

Note: Just like operations on numbers, Condtions can be grouped using parenthesis, ex: `if ( (name === &quot;John&quot; || name === &quot;Jennifer&quot;) && country === &quot;France&quot;)`.

Exercise

Fill up the 2 conditions so that `primaryCategory` equals `"E/J"` only if name equals `"John"` and country is `"England"`, and so that `secondaryCategory` equals `"E|J"` only if name equals `"John"` or country is `"England"`

```
var name = "John";
var country = "England";
var primaryCategory, secondaryCategory;

if ( /* Fill here */ ) {
    primaryCategory = "E/J";
}
if ( /* Fill here */ ) {
    secondaryCategory = "E|J";
}
```

Arrays

Arrays are a fundamental part of programming. An array is a list of data. We can store a lot of data in one variable, which makes our code more readable and easier to understand. It also makes it much easier to perform functions on related data.

The data in arrays are called **elements**.

Here is a simple array:

```
// 1, 1, 2, 3, 5, and 8 are the elements in this array
var numbers = [1, 1, 2, 3, 5, 8];
```

Indices

So you have your array of data elements, but what if you want to access a specific element? That is where indices come in. An **index** refers to a spot in the array. indices logically progress one by one, but it should be noted that the first index in an array is 0, as it is in most languages. Brackets [] are used to signify you are referring to an index of an array.

```
// This is an array of strings
var fruits = ["apple", "banana", "pineapple", "strawberry"];

// We set the variable banana to the value of the second element of
// the fruits array. Remember that indices start at 0, so 1 is the
// second element. Result: banana = "banana"
var banana = fruits[1];
```

Exercise

Define the variables using the indices of the array

```
var cars = ["Mazda", "Honda", "Chevy", "Ford"]
var honda =
var ford =
var chevy =
var mazda =
```

Length

Arrays have a property called length, and it's pretty much exactly as it sounds, it's the length of the array.

```
var array = [1 , 2, 3];

// Result: l = 3
var l = array.length;
```

Exercise

Define the variable a to be the number value of the length of the array

```
var array = [1, 1, 2, 3, 5, 8];
var l = array.length;
var a =
```

Loops

Loops are repetitive conditions where one variable in the loop changes. Loops are handy, if you want to run the same code over and over again, each time with a different value.

Instead of writing:

```
doThing(cars[0]);
doThing(cars[1]);
doThing(cars[2]);
doThing(cars[3]);
doThing(cars[4]);
```

You can write:

```
for (var i=0; i < cars.length; i++) {
    doThing(cars[i]);
}
```

For Loop

The easiest form of a loop is the for statement. This one has a syntax that is similar to an if statement, but with more options:

```
for(condition; end condition; change){
    // do it, do it now
}
```

Lets for example see how to execute the same code ten-times using a `for` loop:

```
for(var i = 0; i < 10; i = i + 1){
    // do this code ten-times
}
```

> ***Note***: `i = i + 1` can be written `i++`.

Exercise

Using a for-loop, create a variable named `message` that equals the concatenation of integers (0, 1, 2, ...) from 0 to 99.

```
var message = "";
```

While Loop

While Loops repetitively execute a block of code as long as a specified condition is true.

```
while(condition){
    // do it as long as condition is true
}
```

For example, the loop in this example will repetitively execute its block of code as long as the variable i is less than 5:

```
var i = 0, x = "";
while (i < 5) {
    x = x + "The number is " + i;
    i++;
}
```

The Do/While Loop is a variant of the while loop. This loop will execute the code block once before checking if the condition is true. It then repeats the loop as long as the condition is true:

```
do {
    // code block to be executed
} while (condition);
```

Note: Be careful to avoid infinite looping if the condition is always true!

Exercise

Using a while-loop, create a variable named `message` that equals the concatenation of integers (0, 1, 2, ...) as long as its length (`message.length`) is less than 100.

```
var message = "";
```

Do...While Loop

The do...while statement creates a loop that executes a specified statement until the test condition evaluates to be false. The condition is evaluated after executing the statement. Syntax for do... while is

```
do{
    // statement
}
while(expression) ;
```

Lets for example see how to print numbers less than 10 using `do...while` loop:

```
var i = 0;
do {
    document.write(i + " ");
    i++; // incrementing i by 1
} while (i < 10);
```

> ***Note***: `i = i + 1` can be written `i++`.

Exercise

Using a do...while-loop, print numbers between less than 5.

```
var i = 0;
```

Functions

Functions, are one of the most powerful and essential notions in programming.

Functions like mathematical functions perform transformations, they take input values called **arguments** and **return** an output value.

Declaring Functions

Functions, like variables, must be declared. Let's declare a function `double` that accepts an **argument** called `x` and **returns** the double of x :

```
function double(x) {
    return 2 * x;
}
```

> *Note:* the function above **may** be referenced before it has been defined.

Functions are also values in JavaScript; they can be stored in variables (just like numbers, strings, etc ...) and given to other functions as arguments :

```
var double = function(x) {
    return 2 * x;
};
```

> *Note:* the function above **may not** be referenced before it is defined, just like any other variable.

Exercise

Declare a function named `triple` that takes an argument and returns its triple.

Higher Order Functions

Higher order functions are functions that manipulate other functions. For example, a function can take other functions as arguments and/or produce a function as its return value. Such *fancy* functional techniques are powerful constructs available to you in JavaScript and other high-level languages like python, lisp, etc.

We will now create two simple functions, `add_2` and `double`, and a higher order function called `map`. `map` will accept two arguments, `func` and `list` (its declaration will therefore begin `map(func,list)`), and return an array. `func` (the first argument) will be a function that will be applied to each of the elements in the array `list` (the second argument).

```
// Define two simple functions
var add_2 = function(x) {
    return x + 2;
};
var double = function(x) {
    return 2 * x;
};

// map is cool function that accepts 2 arguments:
//  func    the function to call
//  list    a array of values to call func on
var map = function(func, list) {
    var output=[];              // output list
    for(idx in list) {
        output.push( func(list[idx]) );
    }
    return output;
}


// We use map to apply a function to an entire list
// of inputs to "map" them to a list of corresponding outputs
map(add_2, [5,6,7]) // => [7, 8, 9]
map(double, [5,6,7]) // => [10, 12, 14]
```

The functions in the above example are simple. However, when passed as arguments to other functions, they can be composed in unforeseen ways to build more complex functions.

For example, if we notice that we use the invocations `map(add_2, ...)` and `map(double, ...)` very often in our code, we could decide we want to create two special-purpose list processing functions that have the desired operation baked into them. Using function composition, we could do this as follows:

```
process_add_2 = function(list) {
    return map(add_2, list);
}
process_double = function(list) {
    return map(double, list);
}
process_add_2([5,6,7]) // => [7, 8, 9]
process_double([5,6,7]) // => [10, 12, 14]
```

Now let's create a function called `buildProcessor` that takes a function `func` as input and returns a `func`-processor, that is, a function that applies `func` to each input in list.

```
// a function that generates a list processor that performs
var buildProcessor = function(func) {
    var process_func = function(list) {
        return map(func, list);
    }
    return process_func;
}
// calling buildProcessor returns a function which is called with a list input


// using buildProcessor we could generate the add_2 and double list processors as follows:
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5,6,7]) // => [7, 8, 9]
process_double([5,6,7]) // => [10, 12, 14]
```

Let's look at another example. We'll create a function called `buildMultiplier` that takes a number `x` as input and returns a function that multiplies its argument by `x` :

```
var buildMultiplier = function(x) {
    return function(y) {
        return x * y;
    }
}

var double = buildMultiplier(2);
var triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

Exercise

Define a function named `negate` that takes `add1` as argument and returns a function, that returns the negation of the value returned by `add1`. (Things get a bit more complicated ;))

```
var add1 = function (x) {
    return x + 1;
};

var negate = function(func) {
    // TODO
};

// Should return -6
// Because (5+1) * -1 = -6
negate(add1)(5);
```

Objects

The primitive types of JavaScript are `true`, `false`, numbers, strings, `null` and `undefined`. **Every other value is an `object`.**

In JavaScript objects contain `propertyName`: `propertyValue` pairs.

Creation

There are two ways to create an `object` in JavaScript:

1. literal

```
var object = {};
  // Yes, simply a pair of curly braces!
```

> ***Note:*** this is the **recomended** way.

2. and object-oriented

```
var object = new Object();
```

> ***Note:*** it's almost like Java.

Properties

Object's property is a `propertyName` : `propertyValue` pair, where **property name can be only a string**. If it's not a string, it gets casted into a string. You can specify properties **when creating** an object **or later**. There may be zero or more properties separated by commas.

```
var language = {
    name: 'JavaScript',
    isSupportedByBrowsers: true,
    createdIn: 1995,
    author:{
        firstName: 'Brendan',
        lastName: 'Eich'
    },
 // Yes, objects can be nested!
    getAuthorFullName: function(){
        return this.author.firstName + " " + this.author.lastName;
    }
 // Yes, functions can be values too!
};
```

The following code demonstates how to **get** a property's value.

```
var variable = language.name;
 // variable now contains "JavaScript" string.
    variable = language['name'];
 // The lines above do the same thing. The difference is that the second one lets you use litteraly any string as a pro
    variable = language.newProperty;
 // variable is now undefined, because we have not assigned this property yet.
```

The following example shows how to **add** a new property **or change** an existing one.

```
language.newProperty = 'new value';
 // Now the object has a new property. If the property already exists, its value will be replaced.
language['newProperty'] = 'changed value';
 // Once again, you can access properties both ways. The first one (dot notation) is recomended.
```

Mutable

The difference between objects and primitive values is that **we can change objects**, whereas primitive values are immutable.

```
var myPrimitive = "first value";
    myPrimitive = "another value";
 // myPrimitive now points to another string.
var myObject = { key: "first value"};
    myObject.key = "another value";
 // myObject points to the same object.
```

Reference

Objects are **never copied**. They are passed around by reference.

```
// Imagine I had a pizza
var myPizza = {slices: 5};
 // And I shared it with You
var yourPizza = myPizza;
 // I eat another slice
myPizza.slices = myPizza.slices - 1;
var numberOfSlicesLeft = yourPizza.slices;
 // Now We have 4 slices because myPizza and yourPizza
 // reference to the same pizza object.
var a = {}, b = {}, c = {};
 // a, b, and c each refer to a
 // different empty object
a = b = c = {};
 // a, b, and c all refer to
 // the same empty object
```

Prototype

Every object is linked to a prototype object from which it inherits properties.

All objects created from object literals (`{}`) are automatically linked to Object.prototype, which is an object that comes standard with JavaScript.

When a JavaScript interpreter (a module in your browser) tries to find a property, which You want to retrieve, like in the following code:

```
var adult = {age: 26},
    retrievedProperty = adult.age;
 // The line above
```

First, the interpreter looks through every property the object itself has. For example, `adult` has only one own property — `age` . But besides that one it actually has a few more properties, which were inherited from Object.prototype.

```
var stringRepresentation = adult.toString();
 // the variable has value of '[object Object]'
```

`toString` is an Object.prototype's property, which was inherited. It has a value of a function, which returns a string representation of the object. If you want it to return a more meaningful representation, then you can override it. Simply add a new property to the adult object.

```
adult.toString = function(){
    return "I'm "+this.age;
}
```

If you call the `toString` function now, the interpreter will find the new property in the object itself and stop.

Thus the interpreter retrieves the first property it will find on the way from the object itself and further through its prototype.

To set your own object as a prototype instead of the default Object.prototype, you can invoke `Object.create` as follows:

```
var child = Object.create(adult);
 /* This way of creating objects lets us easily replace the default Object.prototype with the one we want. In this case,
child.age = 8;
 /* Previously, child didn't have its own age property, and the interpreter had to look further to the child's prototype
 Now, when we set the child's own age, the interpreter will not go further.
 Note: adult's age is still 26. */
var stringRepresentation = child.toString();
 // The value is "I'm 8".
 /* Note: we have not overridden the child's toString property, thus the adult's method will be invoked. If adult did no
```

`child` 's prototype is `adult` , whose prototype is `Object.prototype` . This sequence of prototypes is called **prototype chain**.

Delete

`delete` can be used to **remove a property** from an object. It will remove a property from the object if it has one. It will not look further in the prototype chain. Removing a property from an object may allow a property from the prototype chain to shine through:

```
var adult = {age:26},
    child = Object.create(adult);
    child.age = 8;

delete child.age;
 /* Remove age property from child, revealing the age of the prototype, because then it is not overriden. */
var prototypeAge = child.age;
 // 26, because child does not have its own age property.
```

Enumeration

The `for in` statement can loop over all of the property names in an object. The enumeration will include functions and prototype properties.

```
var fruit = {
    apple: 2,
    orange:5,
    pear:1
},
sentence = 'I have ',
quantity;
for (kind in fruit){
    quantity = fruit[kind];
    sentence += quantity+' '+kind+
                (quantity===1?'':'s')+
                ', ';
}
 // The following line removes the trailing coma.
sentence = sentence.substr(0,sentence.length-2)+'.';
 // I have 2 apples, 5 oranges, 1 pear.
```

Global footprint

If you are developing a module, which might be running on a web page, which also runs other modules, then you must beware the variable name overlapping.

Suppose we are developing a counter module:

```
var myCounter = {
    number : 0,
    plusPlus : function(){
        this.number : this.number + 1;
    },
    isGreaterThanTen : function(){
        return this.number > 10;
    }
}
```

> ***Note:*** this technique is often used with closures, to make the internal state immutable from the outside.

The module now takes only one variable name — `myCounter` . If any other module on the page makes use of such names like `number` or `isGreaterThanTen` then it's perfectly safe, because we will not override each others values;

OpenLayers Workshop

Welcome to the **OpenLayers 3 Workshop**. This workshop is designed to give you a comprehensive overview of OpenLayers as a web mapping solution.

Setup

These instructions assume that you are starting with an `openlayers-workshop.zip` archive from the latest workshop release. In addition, you'll need Node installed to run a development server for the OpenLayers library.

After extracting the zip, change into the `openlayers-workshop` directory and install some additional dependencies:

```
npm install
```

Now you're ready to start the workshop server. This serves up the workshop documentation in addition to providing a debug loader for the OpenLayers library.

```
npm start
```

This will start a development server where you can read the workshop documentation and work through the exercises: http://terrestris.github.io/momo3-ws/.

Overview

This workshop is presented as a set of modules. In each module you will perform a set of tasks designed to achieve a specific goal for that module. Each module builds upon lessons learned in previous modules and is designed to iteratively build up your knowledge base.

The following modules will be covered in this workshop:

- Basics - Learn how to add a map to a webpage with OpenLayers.
- Layers and Sources - Learn about layers and sources.
- Controls and Interactions - Learn about using map controls and interactions.
- Vector Topics - Explore vector layers in depth.
- Custom Builds - Create custom builds.

Basics

Creating a Map

In OpenLayers, a map is a collection of layers and various interactions and controls for dealing with user interaction. A map is generated with three basic ingredients: markup, style declarations, and initialization code.

Working Example

Let's take a look at a fully working example of an OpenLayers 3 map.

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <title>OpenLayers 3 example</title>
    <script src="/loader.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>
```

Tasks

1. Make sure you've completed the setup instructions to install dependencies and get the debug server running.
2. Copy the text above into a new file called `map.html`, and save it in the root of the workshop directory.
3. Open the working map in your web browser: http://terrestris.github.io/momo3-ws//map.html

A working map displaying imagery of the world

Having successfully created our first map, we'll continue by looking more closely at the parts.

Dissecting Your Map

As demonstrated in the previous section, a map is generated by bringing together markup, style declarations, and initialization code. We'll look at each of these parts in a bit more detail.

Map Markup

The markup for the map in the previous example generates a single document element:

```
<div id="map"></div>
```

This `<div>` element will serve as the container for our map viewport. Here we use a `<div>` element, but the container for the viewport can be any block-level element.

In this case, we give the container an `id` attribute so we can reference it as the target of our map.

Map Style

OpenLayers comes with a default stylesheet that specifies how map-related elements should be styled. We've explicitly included this stylesheet in the `map.html` page (`<link rel=&quot;stylesheet&quot; href=&quot;/ol.css&quot; type=&quot;text/css&quot;>`).

OpenLayers doesn't make any guesses about the size of your map. Because of this, following the default stylesheet, we need to include at least one custom style declaration to give the map some room on the page.

```
<link rel="stylesheet" href="/ol.css" type="text/css">
  <style>
    #map {
      height: 256px;
      width: 512px;
    }
  </style>
```

In this case, we're using the map container's `id` value as a selector, and we specify the width (`512px`) and the height (`256px`) for the map container.

The style declarations are directly included in the `<head>` of our document. In most cases, your map related style declarations will be a part of a larger website theme linked in external stylesheets.

Map Initialization

The next step in generating your map is to include some initialization code. In our case, we have included a `<script>` element at the bottom of our document `<body>` to do the work:

```
<script>
    var map = new ol.Map({
      target: 'map',
      layers: [
        new ol.layer.Tile({
          source: new ol.source.TileWMS({
            url: 'http://demo.opengeo.org/geoserver/wms',
            params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
          })
        })
      ],
      view: new ol.View({
        projection: 'EPSG:4326',
        center: [0, 0],
        zoom: 0,
        maxResolution: 0.703125
      })
    });
  </script>
```

The order of these steps is important. Before our custom script can be executed, the OpenLayers library must be loaded. In our example, the OpenLayers library is loaded in the `<head>` of our document with `<script src=&quot;/loader.js&quot;></script>`.

Similarly, our custom map initialization code (above) cannot run until the document element that serves as the viewport container, in this case `<div id=&quot;map&quot;></div>`, is ready. By including the initialization code at the end of the document `<body>`, we ensure that the library is loaded and the viewport container is ready before generating our map.

Let's look in more detail at what the map initialization script is doing. Our script creates a new `ol.Map` object with a few config options:

```
target: 'map'
```

We use the viewport container's `id` attribute value to tell the map constructor where to render the map. In this case, we pass the string value `&quot;map&quot;` as the target to the map constructor. This syntax is a shortcut for convenience. We could be more explicit and provide a direct reference to the element (e.g. `document.getElementById(&quot;map&quot;)`).

The layers config creates a layer to be displayed in our map:

```
layers: [
    new ol.layer.Tile({
      source: new ol.source.TileWMS({
        url: 'http://demo.opengeo.org/geoserver/wms',
        params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
      })
    })
  ],
```

Don't worry about the syntax here if this part is new to you. Layer creation will be covered in another module. The important part to understand is that our map view is a collection of layers. In order to see a map, we need to include at least one layer.

The final step is defining the view. We specify a projection, a center and a zoom level. We also specify a `maxResolution` to make sure we don't request bounding boxes that GeoWebCache cannot handle.

```
view: new ol.View({
     projection: 'EPSG:4326',
     center: [0, 0],
     zoom: 0,
     maxResolution: 0.703125
  })
```

You've successfully dissected your first map! Next let's learn more about developing with OpenLayers.

OpenLayers Resources

The OpenLayers library contains a wealth of functionality. Though the developers have worked hard to provide examples of that functionality and have organized the code in a way that allows other experienced developers to find their way around, many users find it a challenge to get started from scratch.

Learn by Example

New users will most likely find diving into the OpenLayer's example code and experimenting with the library's possible functionality the most useful way to begin.

- http://openlayers.org/en/master/examples/

Browse the Documentation

For further information on specific topics, browse the growing collection of OpenLayers documentation.

- http://openlayers.org/en/master/doc/quickstart.html
- http://openlayers.org/en/master/doc/tutorials

Find the API Reference

After understanding the basic components that make-up and control a map, search the API reference documentation for details on method signatures and object properties. If you only want to see the stable part of the API, make sure to check the `Stable Only` checkbox.

- http://openlayers.org/en/master/apidoc/

Join the Community

OpenLayers is supported and maintained by a community of developers and users like you. Whether you have questions to ask or code to contribute, you can get involved by using the `openlayers-3` tag on StackOverflow for usage questions or signing up for the developers mailing list.

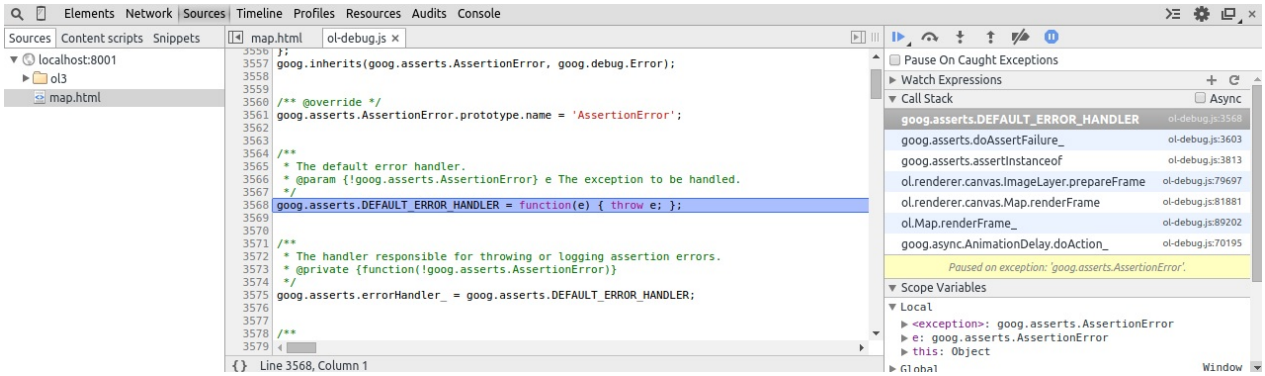
- http://stackoverflow.com/questions/tagged/openlayers-3
- https://groups.google.com/forum/#!forum/ol3-dev

Reporting issues

For reporting issues it is important to understand the several flavours in which the OpenLayers library is distributed:

- `ol.js` - the script which is built using the Closure Compiler in advanced mode (not human readable)
- `ol-debug.js` - human readable version to be used during development

When you encounter an issue, it is important to report the issue using `ol-debug.js`. Also include the full stack trace which you can find using Web Developer tools such as Chrome's Developer Tools. To test this out we are going to make a mistake in map.html by changing `ol.layer.Tile` into `ol.layer.Image`. The error you will see is: `Uncaught TypeError: undefined is not a function`. If you report this to the mailing list, nobody will know what it means. So first, we are going to change the script tag which points to `ol.js` to point to `ol-debug.js` instead. Reload the page. The debugger will now stop on the error, and we can see the full stack trace.



At a breakpoint in the debugger

Layers and Sources

- WMS sources
- Tiled sources
- Proprietary tile providers
- Vector data
- Image vector source

Web Map Service Layers

When you add a layer to your map, the layer's source is typically responsible for fetching the data to be displayed. The data requested can be either raster or vector data. You can think of raster data as information rendered as an image on the server side. Vector data is delivered as structured information from the server and may be rendered for display on the client (your browser).

There are many different types of services that provide raster map data. This section deals with providers that conform with the OGC (Open Geospatial Consortium, Inc.) Web Map Service (WMS) specification.

Creating a Layer

We'll start with a fully working map example and modify the layers to get an understanding of how they work.

Let's take a look at the following code:

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <script src="/loader.js" type="text/javascript"></script>
    <title>OpenLayers 3 example</title>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>
```

Tasks

1. If you haven't already done so, save the text above as `map.html` in the root of your workshop directory.
2. Open the page in your browser to confirm things work: http://terrestris.github.io/momo3-ws//map.html

The `ol.layer.Tile` Constructor

The `ol.layer.Tile` constructor gets an object literal of type `olx.layer.TileOptions` see: http://openlayers.org/en/master/apidoc/ol.layer.Tile.html In this case we are providing the source key of the options with an `ol.source.TileWMS`. A human-readable title for the layer can be provided with the title key, but basically any arbitrary name for the key can be used here. In OpenLayers 3 there is a separation between layers and sources, whereas in OpenLayers 2 this was all part of the layer.

`ol.layer.Tile` represents a regular grid of images, `ol.layer.Image` represents a single image. Depending on the layer type, you would use a different source (`ol.source.TileWMS` versus `ol.source.ImageWMS`) as well.

The ol.source.TileWMS Constructor

The `ol.source.TileWMS` constructor has a single argument which is defined by: http://openlayers.org/en/master/apidoc/ol.source.TileWMS.html. The url is the online resource of the WMS service, and params is an object literal with the parameter names and their values. Since the default WMS version is 1.3.0 now in OpenLayers, you might need to provide a lower version in the params if your WMS does not support WMS 1.3.0.

```
layers: [
    new ol.layer.Tile({
      title: 'Global Imagery',
      source: new ol.source.TileWMS({
        url: 'http://demo.opengeo.org/geoserver/wms',
        params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
      })
    })
  ]
```

Tasks

1. This same WMS offers a Natural Earth layer named `&#39;ne:NE1_HR_LC_SR_W_DR&#39;`. Change the value of the `LAYERS` parameter from `&#39;nasa:bluemarble&#39;` to `&#39;ne:NE1_HR_LC_SR_W_DR&#39;`. Your revised ol.layer.Tile Constructor should look like:

   ```
   new ol.layer.Tile({
      title: 'Global Imagery',
      source: new ol.source.TileWMS({
        url: 'http://demo.opengeo.org/geoserver/wms',
        params: {LAYERS: 'ne:NE1_HR_LC_SR_W_DR', VERSION: '1.1.1'}
      })
    })
   ```

2. Change your layer and source to have a single image instead of tiles. Look at the following API doc pages for hints: http://openlayers.org/en/master/apidoc/ol.layer.Image.html and http://openlayers.org/en/master/apidoc/ol.source.ImageWMS.html. Use the Network tab of your browser's developer tools to make sure a single image is requested and not 256x256 pixel tiles.

Having worked with dynamically rendered data from a Web Map Service, let's move on to learn about cached tile services.

Cached Tiles

By default, the Tile layer makes requests for 256 x 256 (pixel) images to fill your map viewport and beyond. As you pan and zoom around your map, more requests for images go out to fill the areas you haven't yet visited. While your browser will cache some requested images, a lot of processing work is typically required for the server to dynamically render images.

Since tiled layers make requests for images on a regular grid, it is possible for the server to cache these image requests and return the cached result next time you (or someone else) visits the same area - resulting in better performance all around.

`ol.source.XYZ`

The Web Map Service specification allows a lot of flexibility in terms of what a client can request. Without constraints, this makes caching difficult or impossible in practice.

At the opposite extreme, a service might offer tiles only at a fixed set of zoom levels and only for a regular grid. These can be generalized as tiled layers with an XYZ source - you can consider X and Y to indicate the column and row of the grid and Z to represent the zoom level.

`ol.source.OSM`

The OpenStreetMap (OSM) project is an effort to collect and make freely available map data for the world. OSM provides a few different renderings of their data as cached tile sets. These renderings conform to the basic XYZ grid arrangement and can be used in an OpenLayers map. The `ol.source.OSM` layer source accesses OpenStreetMap tiles.

Tasks

1. Open the `map.html` file from the previous section in a text editor and change the map initialization code to look like the following:

```
<script>
   var map = new ol.Map({
     target: 'map',
     layers: [
       new ol.layer.Tile({
         source: new ol.source.OSM()
       })
     ],
     view: new ol.View({
       center: ol.proj.fromLonLat([126.97, 37.56]),
       zoom: 9
     }),
     controls: ol.control.defaults({
       attributionOptions: {
         collapsible: false
       }
     })
   });
 </script>
```

2. In the `<head>` of the same document, add a few style declarations for the layer attribution.

```
<style>
     #map {
         width: 512px;
         height: 256px;
     }
     .ol-attribution a {
         color: black;
     }
 </style>
```

3. Save your changes, and refresh the page in your browser: http://terrestris.github.io/momo3-ws//map.html

A Closer Look

Projections

Review the view definition of the map:

```
view: new ol.View({
    center: ol.proj.fromLonLat([126.97, 37.56]),
    zoom: 9
  })
```

Geospatial data can come in any number of coordinate reference systems. One data set might use geographic coordinates (longitude and latitude) in degrees, and another might have coordinates in a local projection with units in meters. A full discussion of coordinate reference systems is beyond the scope of this module, but it is important to understand the basic concept.

OpenLayers 3 needs to know the coordinate system for your data. Internally, this is represented with an `ol.proj.Projection` object but strings can also be supplied.

The OpenStreetMap tiles that we will be using are in a Mercator projection. Because of this, we need to set the initial center using Mercator coordinates. Since it is relatively easy to find out the coordinates for a place of interest in geographic coordinates, we use the `ol.proj.fromLonLat` method to turn geographic coordinates (`&#39;EPSG:4326&#39;`) into Mercator coordinates (`&#39;EPSG:3857&#39;`).

Alternative Projections

OpenLayers 3 includes transforms between Geographic (`&#39;EPSG:4326&#39;`) and Web Mercator (`&#39;EPSG:3857&#39;`) coordinate reference systems. Because of this, we can use the `ol.proj.fromLonLat` function above without any extra work. If you want to work with data in a different projection, you need to include some additional information before using the `ol.proj.*` functions.

For example, if you wanted to work with data in the `&#39;EPSG:21781&#39;` coordinate reference system, you would include the following two script tags in your page:

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/proj4js/2.3.6/proj4.js" type="text/javascript"></script>
  <script src="http://epsg.io/21781-1753.js" type="text/javascript"></script>
```

Then in your application code, you could register this projection and set its validity extent as follows:

```
// This creates a projection object for the EPSG:21781 projection
  // and sets a "validity extent" in that projection object.
  var projection = ol.proj.get('EPSG:21781');
  projection.setExtent([485869.5728, 76443.1884, 837076.5648, 299941.7864]);
```

The extent information can be looked up at http://epsg.io/, using the EPSG code.

Layer Creation

```
layers: [
    new ol.layer.Tile({
      source: new ol.source.OSM()
    })
  ],
```

As before, we create a layer and add it to the layers array of our map config object. This time, we accept all the default options for the source.

Style

```
.ol-attribution a {
    color: black;
  }
```

A treatment of map controls is also outside of the scope of this module, but these style declarations give you a sneak preview. By default, an `ol.control.Attribution` control is added to all maps. This lets layer sources display attribution information in the map viewport. The declarations above alter the style of this attribution for our map (notice the Copyright line at the bottom right of the map).

Attribution Control Configuration

By default the `ol.control.Attribution` adds an `i` (information) button that can be pressed to actually displays the attribution information. To comply to OpenStreetMap's Terms Of Use, and always display the OpenStreetMap attribution information, the following is used in the options object passed to the `ol.Map` constructor:

```
controls: ol.control.defaults({
    attributionOptions: {
      collapsible: false
    }
  })
```

This removes the `i` button, and makes the attribution information always visible.

Having mastered layers with publicly available cached tile sets, let's move on to working with proprietary raster layers.

Proprietary Raster Layers

In previous sections, we displayed layers based on a standards compliant WMS (OGC Web Map Service) and a custom tile cache. Online mapping (or at least the tiled map client) was largely popularized by the availability of proprietary map tile services. OpenLayers provides layer types that work with these proprietary services through their APIs.

In this section, we'll build on the example developed in the previous section by adding a layer using tiles from Bing.

Bing!

Let's add a Bing layer.

Tasks

1. In your `map.html` file, find where the OSM (OpenStreetMap) source is configured and change it into an `ol.source.BingMaps`

   ```
   source: new ol.source.BingMaps({
      imagerySet: 'Road',
      key: 'Ak-dzM4wZjSqTlzveKz5u0d4IQ4bRzVI309GxmkgSVr1ewS6iPSrOvOKhA-CJlm3'
    })
   ```

   Note - The Bing tiles API requires that you register for an API key to use with your mapping application. The example here uses an API key that you should not use in production. To use the Bing layer in production, register for an API key at https://www.bingmapsportal.com/.

2. Save your changes and reload `map.html` in your browser: http://terrestris.github.io/momo3-ws//map.html

Complete Working Example

Your revised `map.html` file should look something like this:

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
      .ol-attribution a {
        color: black;
      }
    </style>
    <script src="/loader.js" type="text/javascript"></script>
    <title>OpenLayers 3 example</title>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map" class="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            source: new ol.source.BingMaps({
              imagerySet: 'Road',
              key: 'Ak-dzM4wZjSqTlzveKz5u0d4IQ4bRzVI309GxmkgSVr1ewS6iPSrOvOKhA-CJlm3'
            })
          })
        ],
        view: new ol.View({
          center: ol.proj.fromLonLat([126.97, 37.56]),
          zoom: 9
        })
      });
    </script>
  </body>
</html>
```

Vector Layers

Vector Layers are represented by `ol.layer.Vector` and handle the client-side display of vector data. Currently OpenLayers 3 supports full vector rendering in the Canvas renderer, but only point geometries in the WebGL renderer.

Rendering Features Client-Side

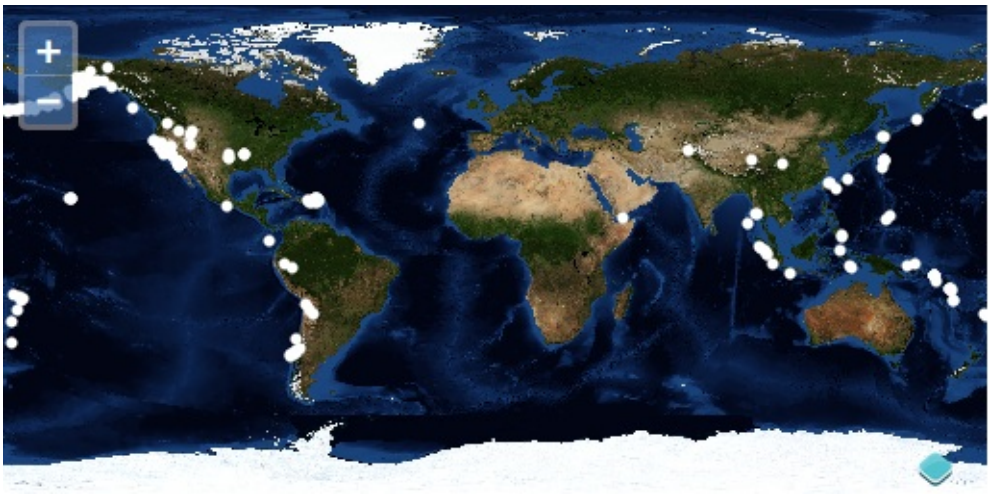
Let's go back to the WMS example to get a basic world map. We'll add some feature data on top of this in a vector layer.

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <title>OpenLayers 3 example</title>
    <script src="/loader.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>
```

Tasks

1. Open `map.html` in your text editor and copy in the contents of the initial WMS example. Save your changes and confirm that things look good in your browser: http://terrestris.github.io/momo3-ws//map.html
2. In your map initialization code add another layer after the Tile layer (paste the following). This adds a new vector layer to your map that requests a set of features stored in GeoJSON:

```
new ol.layer.Vector({
   title: 'Earthquakes',
   source: new ol.source.Vector({
     url: '/data/layers/7day-M2.5.json',
     format: new ol.format.GeoJSON()
   }),
   style: new ol.style.Style({
     image: new ol.style.Circle({
       radius: 3,
       fill: new ol.style.Fill({color: 'white'})
     })
   })
 })
```



A Closer Look

Let's examine that vector layer creation to get an idea of what is going on.

```
new ol.layer.Vector({
  title: 'Earthquakes',
  source: new ol.source.Vector({
    url: '/data/layers/7day-M2.5.json',
    format: new ol.format.GeoJSON()
  }),
  style: new ol.style.Style({
    image: new ol.style.Circle({
      radius: 3,
      fill: new ol.style.Fill({color: 'white'})
    })
  })
})
```

The layer is given the title `&#39;Earthquakes&#39;` and some custom options. In the options object, we've included a `source` of type `ol.source.Vector` which points to a url. We've given the source a `format` that will be used for parsing the data.

Note - In the case where you want to style the features based on an attribute, you would use a style function instead of an `ol.style.Style` for the `style` config option of `ol.layer.Vector`.

Bonus Tasks

1. The white circles on the map represent `ol.Feature` objects on your `ol.layer.Vector` layer. Each of these features has attribute data with `title` and `summary` properties. Register a `&#39;singleclick&#39;` listener on your map that calls `forEachFeatureAtPixel` on the map, and displays earthquake information below the map viewport.
2. The data for the vector layer comes from an earthquake feed published by the USGS (http://earthquake.usgs.gov/earthquakes/catalogs/). See if you can find additional data with spatial information in a format

supported by OpenLayers 3. If you save another document representing spatial data in your `data` directory, you should be able to view it in a vector layer on your map.

Solutions

As a solution to the first bonus task you can add an `info` div below the map:

```
<div id="info"></div>
```

and add the following JavaScript code to display the title of the clicked feature:

```
map.on('singleclick', function(e) {
  var feature = map.forEachFeatureAtPixel(e.pixel, function(feature) {
    return feature;
  });
  var infoElement = document.getElementById('info');
  infoElement.innerHTML = feature ? feature.get('title') : '';
});
```

Image Vector

In the previous example using an `ol.layer.Vector` you can see that the features are re-rendered continuously during animated zooming (the size of the point symbolizers remains fixed). With a vector layer, OpenLayers will re-render the source data with each animation frame. This provides consistent rendering of line strokes, point symbolizers, and labels with changes in the view resolution.

An alternative rendering strategy is to avoid re-rendering data during view transitions and instead reposition and scale the rendered output from the previous view state. This can be accomplished by using an `ol.layer.Image` with an `ol.source.ImageVector`. With this combination, "snapshots" of your data are rendered when the view is not animating, and these snapshots are reused during view transitions.

The example below uses an `ol.layer.Image` with an `ol.source.ImageVector`. Though this example only renders a small quantity of data, this combination would be appropriate for applications that render large quantities of relatively static data.

`ol.source.ImageVector`

Let's go back to the vector layer example to get earthquake data on top of a world map.

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <title>OpenLayers 3 example</title>
    <script src="/loader.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          }),
          new ol.layer.Vector({
            title: 'Earthquakes',
            source: new ol.source.Vector({
              url: '/data/layers/7day-M2.5.json',
              format: new ol.format.GeoJSON()
            }),
            style: new ol.style.Style({
              image: new ol.style.Circle({
                radius: 3,
                fill: new ol.style.Fill({color: 'white'})
              })
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>
```

Tasks

1. Open `map.html` in your text editor and copy in the contents of the vector example from above. Save your changes and confirm that things look good in your browser: http://terrestris.github.io/momo3-ws//map.html
2. Change the vector layer into:

```
  new ol.layer.Image({
     title: 'Earthquakes',
     source: new ol.source.ImageVector({
       source: new ol.source.Vector({
         url: '/data/layers/7day-M2.5.json',
         format: new ol.format.GeoJSON()
       }),
       style: new ol.style.Style({
         image: new ol.style.Circle({
         radius: 3,
           fill: new ol.style.Fill({color: 'white'})
         })
       })
     })
   })
```

3. Reload http://terrestris.github.io/momo3-ws//map.html in the browser *Note* - You will see the same vector data but depicted as an image. This will still enable things like feature detection, but the vector data will be less sharp. So this is essentially a trade-off between performance and quality.

A Closer Look

Let's examine the layer creation to get an idea of what is going on.

```
new ol.layer.Image({
    title: 'Earthquakes',
    source: new ol.source.ImageVector({
      source: new ol.source.Vector({
        url: '/data/layers/7day-M2.5.json',
        format: new ol.format.GeoJSON()
      }),
      style: new ol.style.Style({
        image: new ol.style.Circle({
        radius: 3,
          fill: new ol.style.Fill({color: 'white'})
        })
      })
    })
  })
```

We are using an `ol.layer.Image` instead of an `ol.layer.Vector`. However, we can still use vector data here through `ol.source.ImageVector` that connects to our original `ol.source.Vector` vector source. The style is provided as config of `ol.source.ImageVector` and not on the layer.

Bonus Tasks

1. Verify that feature detection still works by registering a `&#39;singleclick&#39;` listener on your map that calls `forEachFeatureAtPixel` on the map, and displays earthquake information below the map viewport.

Controls and interactions

- Scale line control
- Select interaction
- Draw interaction
- Modify interaction

Displaying a Scale Line

Another typical widget to display on maps is a scale bar. OpenLayers 3 provides an `ol.control.ScaleLine` for just this.

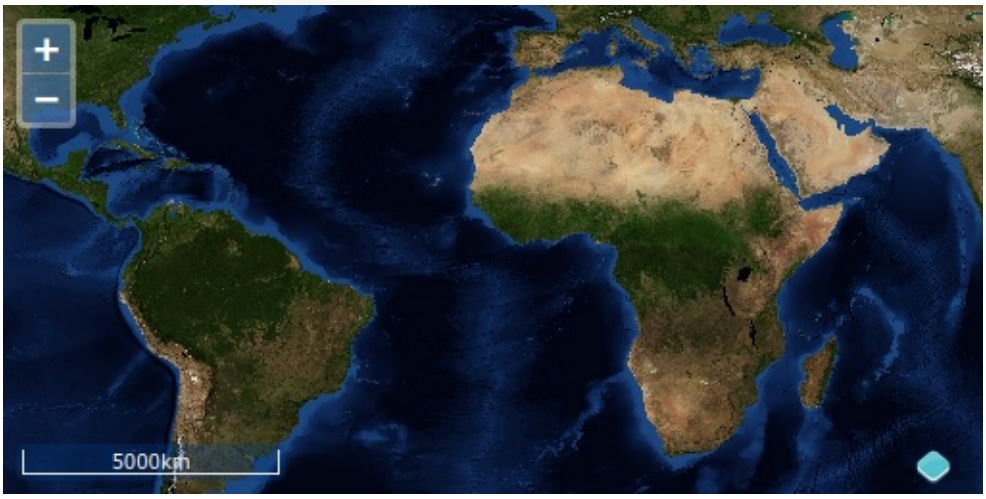
Creating a ScaleLine Control

Tasks

1. Open the `map.html` in your text editor.
2. Somewhere in the map config, add the following code to create a new scale line control for your map:

```
controls: ol.control.defaults().extend([
  new ol.control.ScaleLine()
]),
```

3. Save your changes and open `map.html` in your browser: http://terrestris.github.io/momo3-ws//map.html



Moving the ScaleLine Control

You may find the scale bar a bit hard to read over the imagery. There are a few approaches to take in order to improve scale visibility. If you want to keep the control inside the map viewport, you can add some style declarations within the CSS of your document. To test this out, you can include a background color and padding to the scale bar with something like the following:

```
.ol-scale-line {
    background: black;
    padding: 5px;
  }
```

However, for the sake of this exercise, let's say you think the map viewport is getting unbearably crowded. To avoid such over-crowding, you can display the scale in a different location. To accomplish this, we need to first create an additional element in our markup and then tell the scale control to render itself within this new element.

Tasks

1. Create a new block level element in the `<body>` of your page. To make this element easy to refer to, we'll give it an `id` attribute. Insert the following markup somewhere in the `<body>` of your `map.html` page. (Placing the scale element right after the map viewport element `<div id=&quot;map&quot;></div>` makes sense.):

```
<div id="scale-line" class="scale-line"></div>
```

2. Now modify the code creating the scale control so that it refers to the `scale-line` element:

```
controls: ol.control.defaults().extend([
  new ol.control.ScaleLine({className: 'ol-scale-line', target: document.getElementById('scale-line')})
]),
```

3. Save your changes and open `map.html` in your browser: http://terrestris.github.io/momo3-ws//map.html
4. "Fix" the position of the control with, for example, the following CSS rules:

```
.scale-line {
  position: absolute;
  top: 350px;
}
.ol-scale-line {
  position: relative;
  bottom: 0px;
  left: 0px;
}
```

5. Now save your changes and view `map.html` again in your browser: http://terrestris.github.io/momo3-ws//map.html

Note - To create a custom control you can inherit (by using `ol.inherits`) from `ol.control.Control`. To see an example of this check out: http://openlayers.org/en/master/examples/custom-controls.html.

Selecting Features

As we've seen in the layers module, we can pull features as vectors and draw them on top of a base map. One of the advantages of serving vector data is that users can interact with the data. In this example, we create a vector layer where users can select and view feature information.

The previous example demonstrated the use of an `ol.control.Control` on the map. Controls have a visual representation on the map or add DOM elements to the document. An `ol.interaction.Interaction` is responsible for handling user interaction, but typically has no visual representation. This example demonstrates the use of the `ol.interaction.Select` for interacting with features from vector layers.

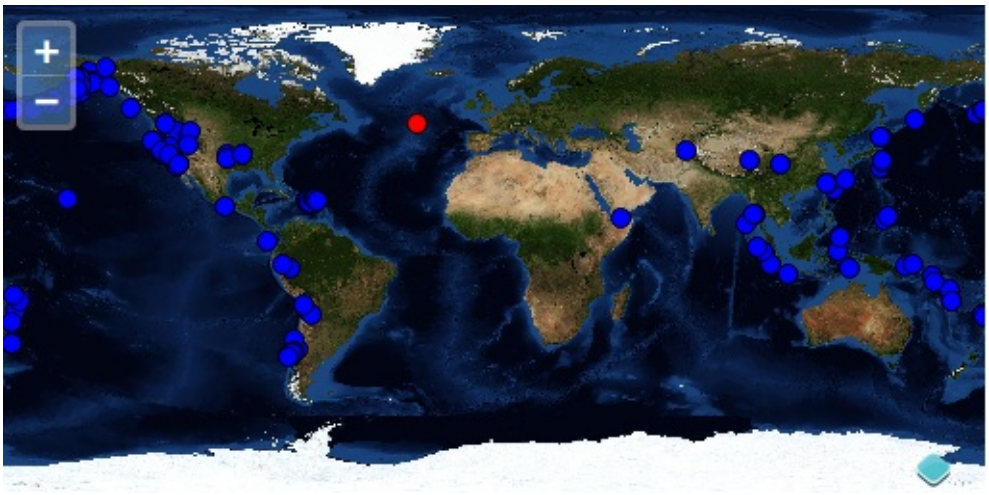
Create a Vector Layer and a Select Interaction

Tasks

1. Let's start with the vector layer example from a previous section. Open `map.html` in your text editor and make sure it looks something like the following:

```
<!doctype html>
<html lang="en">
 <head>
   <link rel="stylesheet" href="/ol.css" type="text/css">
   <style>
   #map {
     height: 256px;
     width: 512px;
   }
   </style>
   <script src="/loader.js" type="text/javascript"></script>
   <title>OpenLayers 3 example</title>
 </head>
 <body>
   <h1>My Map</h1>
   <div id="map"></div>
   <script type="text/javascript">
     var map = new ol.Map({
       interactions: ol.interaction.defaults().extend([
         new ol.interaction.Select({
           style: new ol.style.Style({
             image: new ol.style.Circle({
               radius: 5,
               fill: new ol.style.Fill({
                 color: '#FF0000'
               }),
               stroke: new ol.style.Stroke({
                 color: '#000000'
               })
             })
           })
         })
       ]),
       target: 'map',
       layers: [
         new ol.layer.Tile({
           title: 'Global Imagery',
           source: new ol.source.TileWMS({
             url: 'http://demo.opengeo.org/geoserver/wms',
             params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
           })
         }),
         new ol.layer.Vector({
           title: 'Earthquakes',
           source: new ol.source.Vector({
             url: '/data/layers/7day-M2.5.json',
             format: new ol.format.GeoJSON()
           }),
           style: new ol.style.Style({
             image: new ol.style.Circle({
               radius: 5,
               fill: new ol.style.Fill({
                 color: '#0000FF'
               }),
               stroke: new ol.style.Stroke({
                 color: '#000000'
               })
             })
           })
         })
       ],
       view: new ol.View({
         projection: 'EPSG:4326',
         center: [0, 0],
         zoom: 1
       })
     });
   </script>
 </body>
</html>
```

2. Save your changes to `map.html` and open the page in your browser: http://terrestris.github.io/momo3-ws//map.html. To see feature selection in action, use the mouse-click to select an earthquake:



Drawing Features

New features can be drawn by using an `ol.interaction.Draw`. A draw interaction is constructed with a vector source and a geometry type.

Create a Vector Layer and a Draw Interaction

Tasks

1. Let's start with the example below. Open `map.html` in your text editor and make sure it looks something like the following:

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
    #map {
      height: 256px;
      width: 512px;
    }
    </style>
    <script src="/loader.js" type="text/javascript"></script>
    <title>OpenLayers 3 example</title>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var source = new ol.source.Vector({
        url: '/data/layers/7day-M2.5.json',
        format: new ol.format.GeoJSON()
      });
      var draw = new ol.interaction.Draw({
        source: source,
        type: 'Point'
      });
      var map = new ol.Map({
        interactions: ol.interaction.defaults().extend([draw]),
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          }),
          new ol.layer.Vector({
            title: 'Earthquakes',
            source: source,
            style: new ol.style.Style({
              image: new ol.style.Circle({
                radius: 5,
                fill: new ol.style.Fill({
                  color: '#0000FF'
                }),
                stroke: new ol.style.Stroke({
                  color: '#000000'
                })
              })
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 1
        })
      });
    </script>
  </body>
</html>
```

2. Save your changes to `map.html` and open the page in your browser: http://terrestris.github.io/momo3-ws//map.html. To see drawing of point geometries in action, click in the map to add a new feature:

Bonus Tasks

1. Create a listener which gets the new feature's X and Y after it is drawn.

Modifying Features

Modifying features works by using an `ol.interaction.Select` in combination with an `ol.interaction.Modify`. They share a common collection (`ol.Collection`) of features. Features selected with the `ol.interaction.Select` become candidates for modifications with the `ol.interaction.Modify`.

Create a Vector Layer and a Modify Interaction

Tasks

1. Let's start with the working example. Open `map.html` in your text editor and make sure it looks something like the following:

```
<!doctype html>
<html lang="en">
 <head>
   <link rel="stylesheet" href="/ol.css" type="text/css">
   <style>
   #map {
     height: 256px;
     width: 512px;
   }
   </style>
   <script src="/loader.js" type="text/javascript"></script>
   <title>OpenLayers 3 example</title>
 </head>
 <body>
   <h1>My Map</h1>
   <div id="map"></div>
   <script type="text/javascript">
     var source = new ol.source.Vector({
       url: '/data/layers/7day-M2.5.json',
       format: new ol.format.GeoJSON()
     });
     var style = new ol.style.Style({
       image: new ol.style.Circle({
         radius: 7,
           fill: new ol.style.Fill({
           color: [0, 153, 255, 1]
         }),
         stroke: new ol.style.Stroke({
           color: [255, 255, 255, 0.75],
           width: 1.5
         })
       }),
       zIndex: 100000
     });
     var select = new ol.interaction.Select({style: style});
     var modify = new ol.interaction.Modify({
       features: select.getFeatures()
     });
     var map = new ol.Map({
       interactions: ol.interaction.defaults().extend([select, modify]),
       target: 'map',
       layers: [
         new ol.layer.Tile({
           title: 'Global Imagery',
           source: new ol.source.TileWMS({
             url: 'http://demo.opengeo.org/geoserver/wms',
             params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
           })
         }),
         new ol.layer.Vector({
           title: 'Earthquakes',
           source: source,
           style: new ol.style.Style({
```

```
              image: new ol.style.Circle({
                radius: 5,
                fill: new ol.style.Fill({
                  color: '#0000FF'
                }),
                stroke: new ol.style.Stroke({
                  color: '#000000'
                })
              })
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 1
        })
      });
    </script>
  </body>
</html>
```

2. Save your changes to `map.html` and open the page in your browser: http://terrestris.github.io/momo3-ws//map.html. To see feature modification in action, use the mouse-click to select an earth quake and then drag to move the point.

A Closer Look

Let's examine how modifying features works.

```
var style = new ol.style.Style({
    image: new ol.style.Circle({
      radius: 7,
        fill: new ol.style.Fill({
        color: [0, 153, 255, 1]
      }),
      stroke: new ol.style.Stroke({
        color: [255, 255, 255, 0.75],
        width: 1.5
      })
    }),
    zIndex: 100000
  });
  var select = new ol.interaction.Select({style: style});
  var modify = new ol.interaction.Modify({
    features: select.getFeatures()
  });
```

We create 2 interactions, an `ol.interaction.Select` to select the features before modifying them, and an `ol.interaction.Modify` to actually modify the geometries. They share the same `ol.Collection` of features. Features selected using `ol.interaction.Modify` become candidates for modification with the `ol.interaction.Modify` . As previously, the `ol.interaction.Select` is configured with a style object, which effectively defines the style used for drawing selected features. When the user clicks in the map again, the feature will be drawn using the layer's style.

Vector Topics

Working with Vector Formats

The base `ol.layer.Vector` constructor provides a fairly flexible layer type. By default, when you create a new vector layer, no assumptions are made about where the features for the layer will come from, since this is the domain of `ol.source.Vector`. Before getting into styling vector features, this section introduces the basics of vector formats.

`ol.format`

The `ol.format` classes in OpenLayers 3 are responsible for parsing data from the server representing vector features. The format turns raw feature data into `ol.Feature` objects.

Consider the two blocks of data below. Both represent the same `ol.Feature` object (a point in Barcelona, Spain). The first is serialized as GeoJSON (using the `ol.format.GeoJSON` parser). The second is serialized as KML (OGC Keyhole Markup Language) (using the `ol.format.KML` parser).

GeoJSON Example

```
{
  "type": "Feature",
  "id": "OpenLayers.Feature.Vector_107",
  "properties": {},
  "geometry": {
    "type": "Point",
    "coordinates": [-104.98, 39.76]
  }
}
```

KML Example

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
  <Placemark>
    <Point>
      <coordinates>-104.98,39.76</coordinates>
    </Point>
  </Placemark>
</kml>
```

Understanding Style

When styling HTML elements, you might use CSS like the following:

```
.someClass {
    background-color: blue;
    border-width: 1px;
    border-color: olive;
  }
```

The `.someClass` text is a selector (in this case it selects all elements that include the class name `&#39;someClass&#39;`) and the block that follows is a group of named properties and values, otherwise known as style declarations.

Layer style

A vector layer can have styles. More specifically, a vector layer can be configured with an `ol.style.Style` object, an array of `ol.style.Style` objects, or a function that takes an `ol.Feature` instance and a resolution and returns an array of `ol.style.Style` objects.

Here's an example of a vector layer configured with a static style:

```
var layer = new ol.layer.Vector({
    source: new ol.source.Vector(),
    style: new ol.style.Style({
      // ...
    })
  });
```

And here's an example of a vector layer configured with a style function that applies a style to all features that have an attribute named `class` with a value of `&#39;someClass&#39;` :

```
var layer = new ol.layer.Vector({
    source: new ol.source.Vector(),
    style: function(feature, resolution) {
      if (feature.get('class') === 'someClass') {
        // create styles...
        return styles;
      }
    },
  });
```

Symbolizers

The equivalent of a declaration block in CSS is a `symbolizer` in OpenLayers 3 (these are typically instances of `ol.style` classes). To paint polygon features with a blue background and a 1 pixel wide olive stroke, you would use two symbolizers like the following:

```
new ol.style.Style({
    fill: new ol.style.Fill({
      color: 'blue'
    }),
    stroke: new ol.style.Stroke({
      color: 'olive',
      width: 1
    })
  });
```

Depending on the geometry type, different symbolizers can be applied. Lines work like polygons, but they cannot have a fill. Points can be styled with `ol.style.Circle` or `ol.style.Icon`. The former is used to render circle shapes, and the latter uses graphics from file (e.g. png images). Here is an example for a style with a circle:

```
new ol.style.Circle({
    radius: 20,
    fill: new ol.style.Fill({
      color: '#ff9900',
      opacity: 0.6
    }),
    stroke: new ol.style.Stroke({
      color: '#ffcc00',
      opacity: 0.4
    })
  });
```

`ol.style.Style`

An `ol.style.Style` object has 4 keys: `fill`, `image`, `stroke` and `text`. It also has an optional `zIndex` property. The style function will return an array of `ol.style.Style` objects.

If you want all features to be colored red except for those that have a `class` attribute with the value of `&quot;someClass&quot;` (and you want those features colored blue with an 1px wide olive stroke), you would create a style function that looked like the following (by the way, it is important to create objects outside of the style function so they can be reused, but for simplicity reasons the objects are created inline in the example below):

```
var primaryStyles = [
    new ol.style.Style({
      fill: new ol.style.Fill({
        color: 'blue'
      }),
      stroke: new ol.style.Stroke({
        color: 'olive',
        width: 1
      })
    })];
    var otherStyle = [new ol.style.Style({
      fill: new ol.style.Fill({
        color: 'red'
      })
    })
  ];
  var otherStyles = [
    // define other styles here
  ]

  layer.setStyle(function(feature, resolution) {
    if (feature.get('class') === 'someClass') {
      return primaryStyles;
    } else {
      return otherStyles;
    }
  });
```

Note - It is important to create the style arrays outside of the actual style function. The style function is called many times during rendering, and you'll get smoother animation if your style functions don't create a lot of garbage.

A feature also has a style config option that can take a function having only resolution as argument. This makes it possible to style individual features (based on resolution).

Pseudo-classes

CSS allows for pseudo-classes on selectors. These basically limit the application of style declarations based on contexts that are not easily represented in the selector, such as mouse position, neighboring elements, or browser history. In OpenLayers 3, a somewhat similar concept is having a style config option on an `ol.interaction.Select`.

An example is:

```
var select = new ol.interaction.Select({
    style: new ol.style.Style({
      fill: new ol.style.Fill({
        color: 'rgba(255,255,255,0.5)'
      })
    })
  });
```

With the basics of styling under your belt, it's time to move on to styling vector layers.

Styling Vector Layers

1. We'll start with a working example that displays building footprints in a vector layer. Open your text editor and save the following as `map.html` in the root of your workshop directory:

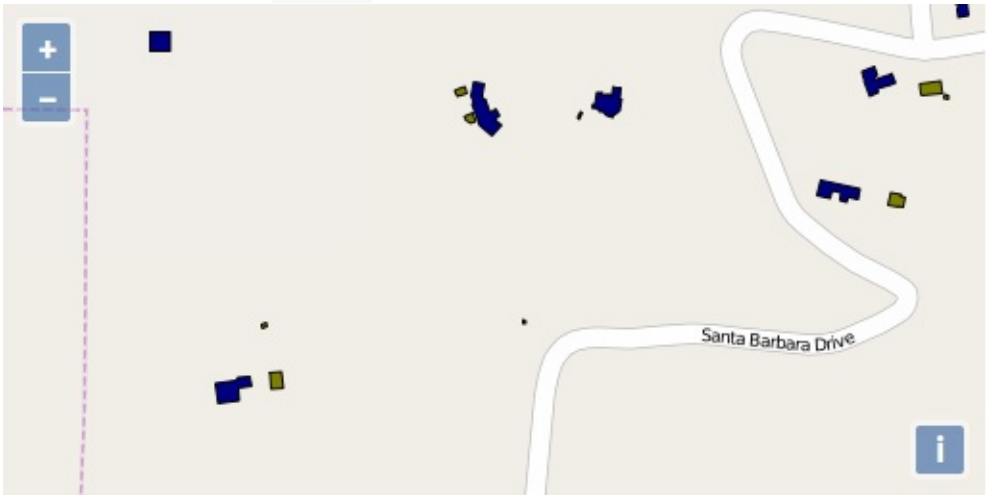
```
<!doctype html>
<html lang="en">
<head>
  <link rel="stylesheet" href="/ol.css" type="text/css">
  <style>
  #map {
    height: 256px;
    width: 512px;
  }
  </style>
  <title>OpenLayers 3 example</title>
  <script src="/loader.js" type="text/javascript"></script>
</head>
<body>
  <h1>My Map</h1>
  <div id="map"></div>
  <script type="text/javascript">
    var map = new ol.Map({
      target: 'map',
      layers: [
        new ol.layer.Tile({
          source: new ol.source.OSM()
        }),
        new ol.layer.Vector({
          title: 'Buildings',
          source: new ol.source.Vector({
            url: '/data/layers/buildings.kml',
            format: new ol.format.KML({
              extractStyles: false
            })
          }),
          style: new ol.style.Style({
            stroke: new ol.style.Stroke({color: 'red', width: 2})
          })
        })
      ],
      view: new ol.View({
        center: ol.proj.fromLonLat([-122.79264450073244, 42.30975194250527]),
        zoom: 16
      })
    });
  </script>
</body>
</html>
```

2. Open this `map.html` file in your browser to see buildings with a red outline: http://terrestris.github.io/momo3-ws//map.html
3. With a basic understanding of styling in OpenLayers, we can create a style function that displays buildings in different colors based on the size of their footprint. In your map initialization code, add the following two styles arrays and replace the `style` option for the `&#39;Buildings&#39;` layer with the style function below:

```
var defaultStyles = [
   new ol.style.Style({
     fill: new ol.style.Fill({color: 'navy'}),
     stroke: new ol.style.Stroke({color: 'black', width: 1})
   })
 ];
 var smallStyles = [
   new ol.style.Style({
     fill: new ol.style.Fill({color: 'olive'}),
     stroke: new ol.style.Stroke({color: 'black', width: 1})
   })
 ];

 function style(feature, resolution) {
   if (feature.get('shape_area') < 3000) {
     return smallStyles;
   } else {
     return defaultStyles;
   }
 }
```

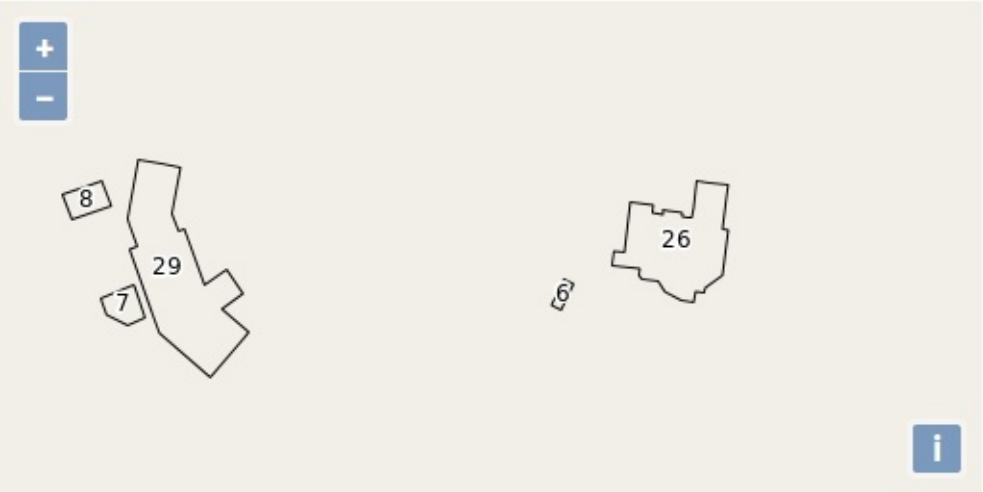
4. Save your changes and open `map.html` in your browser: http://terrestris.github.io/momo3-ws//map.html



5. Now as a final step, let's add a label to the buildings. For simplicity we're only using a label and a black outline as the style.

```
style: (function() {
   var stroke = new ol.style.Stroke({
     color: 'black'
   });
   var textStroke = new ol.style.Stroke({
     color: '#fff',
     width: 3
   });
   var textFill = new ol.style.Fill({
     color: '#000'
   });
   return function(feature, resolution) {
     return [new ol.style.Style({
       stroke: stroke,
       text: new ol.style.Text({
         font: '12px Calibri,sans-serif',
         text: feature.get('key'),
         fill: textFill,
         stroke: textStroke
       })
     })];
   };
 })()
```

6. Save your changes and open `map.html` in your browser: http://terrestris.github.io/momo3-ws//map.html


8
29
7
26
6

Custom Builds

Understanding custom builds

OpenLayers 3 is a big library providing a lot of functionality. So it is unlikely that an application will need and use all the functionality OpenLayers 3 provides. Custom builds (a.k.a. application-specific builds) are OpenLayers 3 builds with just the functionality your application needs. Custom builds are often much smaller that the full build, so creating custom builds is often a very good idea.

Requirements

OpenLayers 3 builds are created by using the Closure Compiler. The goal of the Closure Compiler is to compile JavaScript to better JavaScript, that takes less time to downoad and run faster.

The Closure Compiler is a Java program, so running the Compiler requires a Java Virtual Machine. So before jumping to the next section, and creating a custom build, make sure Java is installed on your machine.

You just need the Java Runtime Environment, which you can download from the Oracle Java site. For example, for Windows, you would download and install `jre-8u60-windows-i586.exe`.

Build configuration file

Creating a custom build requires writing a build configuration file. The format of build configuration files is JSON. Here is a simple example of a build configuration file:

```
{
  "exports": [
    "ol.Map",
    "ol.View",
    "ol.layer.Tile",
    "ol.source.OSM"
  ],
  "jvm": [],
  "umd": true,
  "compile": {
    "externs": [
      "externs/bingmaps.js",
      "externs/closure-compiler.js",
      "externs/esrijson.js",
      "externs/geojson.js",
      "externs/oli.js",
      "externs/olx.js",
      "externs/proj4js.js",
      "externs/tilejson.js",
      "externs/topojson.js"
    ],
    "define": [
      "goog.array.ASSUME_NATIVE_FUNCTIONS=true",
      "goog.dom.ASSUME_STANDARDS_MODE=true",
      "goog.json.USE_NATIVE_JSON=true"
    ],
    "jscomp_error": [
      "*"
    ],
    "jscomp_off": [
      "useOfGoogBase",
      "unnecessaryCasts",
      "lintChecks"
    ],
    "extra_annotation_name": [
      "api", "observable"
    ],
    "compilation_level": "ADVANCED",
    "warning_level": "VERBOSE",
    "use_types_for_optimization": true,
    "manage_closure_dependencies": true
  }
}
```

The most relevant part of this configuration object is the `exports` array. This array declares the functions/constructors you use in your JavaScript code. For example, the above configuration file is what you'd use for the following JavaScript code:

```
var map = new ol.Map({
  target: 'map',
  layers: [
    new ol.layer.Tile({
      source: new ol.source.OSM()
    })
  ],
  view: new ol.View({
    center: [0, 0],
    zoom: 4
  })
});
```

Creating custom builds

In this section we're going to create a custom build for the map you created at the last chapter.

1. Start with the `map.html` file you created previously:

```
<!doctype html>
 <html lang="en">
 <head>
   <link rel="stylesheet" href="/ol.css" type="text/css">
   <style>
   #map {
     height: 256px;
     width: 512px;
   }
   </style>
   <title>OpenLayers 3 example</title>
   <script src="/loader.js" type="text/javascript"></script>
 </head>
 <body>
   <h1>My Map</h1>
   <div id="map"></div>
   <script type="text/javascript">
     var style = (function() {
       var stroke = new ol.style.Stroke({
         color: 'black'
       });
       var textStroke = new ol.style.Stroke({
         color: '#fff',
         width: 3
       });
       var textFill = new ol.style.Fill({
         color: '#000'
       });
       return function(feature, resolution) {
         return [new ol.style.Style({
           stroke: stroke,
           text: new ol.style.Text({
             font: '12px Calibri,sans-serif',
             text: feature.get('key'),
             fill: textFill,
             stroke: textStroke
           })
         })];
       };
     })();
     var map = new ol.Map({
       target: 'map',
       layers: [
         new ol.layer.Tile({
           source: new ol.source.OSM()
         }),
         new ol.layer.Vector({
           title: 'Buildings',
           source: new ol.source.Vector({
             url: '/data/layers/buildings.kml',
             format: new ol.format.KML({
               extractStyles: false
             })
           }),
           style: style
         })
       ],
       view: new ol.View({
         center: ol.proj.fromLonLat([-122.79264450073244, 42.30975194250527]),
         zoom: 16
       })
     });
   </script>
 </body>
 </html>
```

2. Create a build configuration file for that map:

```
{
   "exports": [
       "ol.Map",
       "ol.View",
       "ol.format.KML",
       "ol.layer.Tile",
       "ol.layer.Vector",
       "ol.proj.fromLonLat",
       "ol.source.OSM",
       "ol.source.Vector",
       "ol.style.Fill",
       "ol.style.Stroke",
       "ol.style.Style",
       "ol.style.Text"
   ],
   "jvm": [],
   "umd": true,
   "compile": {
     "externs": [
       "externs/bingmaps.js",
       "externs/closure-compiler.js",
       "externs/esrijson.js",
       "externs/geojson.js",
       "externs/oli.js",
       "externs/olx.js",
       "externs/proj4js.js",
       "externs/tilejson.js",
       "externs/topojson.js"
     ],
     "define": [
       "goog.array.ASSUME_NATIVE_FUNCTIONS=true",
       "goog.dom.ASSUME_STANDARDS_MODE=true",
       "goog.json.USE_NATIVE_JSON=true",
       "ol.ENABLE_DOM=false",
       "ol.ENABLE_WEBGL=false",
       "ol.ENABLE_PROJ4JS=false",
       "ol.ENABLE_IMAGE=false",
       "goog.DEBUG=false"
     ],
     "jscomp_error": [
       "*"
     ],
     "jscomp_off": [
       "useOfGoogBase",
       "unnecessaryCasts",
       "lintChecks"
     ],
     "extra_annotation_name": [
       "api", "observable"
     ],
     "compilation_level": "ADVANCED",
     "warning_level": "VERBOSE",
     "use_types_for_optimization": true,
     "manage_closure_dependencies": true
   }
 }
```

3. Create the custom build using `OpenLayers` 's `build.js` Node script:

```
$ node node_modules/openlayers/tasks/build.js ol-custom.json ol-custom.js
```

   This will generate the `ol-custom.js` custom build at the root of the the project.

4. Now change `map.html` to use the custom build (`ol-custom.js`) rather than the development loader.

   So change

```
<script src="/loader.js" type="text/javascript"></script>
```

to

```
<script src="/ol-custom.js" type="text/javascript"></script>
```

The page should now load much faster than before!

ExtJS

GeoExt

Wrap-Up

Glossary

term

Definition for this term

3.1.1. Basics 3.1.1. Basics