Chapter 5 Prototype

1. The prototype property: available to you as soon as you define the function.

```
typeof foo.prototype;
> "object"                    // Its initial value is an "empty" object.
```

2. Adding methods and properties using the prototype

```
function Gadget(name, color) {
      this.name = name;
       this.color = color;
       this.whatAreYou = function () {
            return 'I am a ' + this.color + ' ' + this.name;
       };
}                                    // An exemplary object
Gadget.prototype.rating = 3;     // Use prototype to add method
Gadget.prototype.getInfo = function () {
        return 'Rating: ' + this.rating +', price: ' + this.price;
};
// Noted that once the prototype function is declared, all objects can access to it
// It's a live change
```

3. Own properties versus prototype properties
   a) What does the JavaScript engine do when we want to access to one property?
      i. The JavaScript engine looks through all of the properties of the object.
      ii. The script engine identifies the prototype of the constructor function used to create this object

```
newtoy.constructor === Gadget;
> true
newtoy.constructor.prototype.rating;
> 3
```

   b) Object() is the common parent of all objects, so toString() can be inherited from it.

4. Overwriting a prototype's property with an own property
   a) the own property takes precedence over the prototype's

```
Example:
function Gadget(name) {
      this.name = name;
}
Gadget.prototype.name = 'mirror';
var toy = new Gadget('camera');     // Creating a new object and accessing its name
toy.name;                           // property gives you the object's own name property
> "camera"
toy.hasOwnProperty('name');// hasOwnProperty() tells where the property was defined.
> true
```

```
delete toy.name;
toy.name;
> "mirror"                    //Now the prototype variable shines through
Object.hasOwnProperty('toString');
> false
Object.prototype.hasOwnProperty('toString');
True                          // toString() is defined in Object.prototype
```

b) instance.constructor.prototype = object.prototype

remember that only constructor can be used to change prototype

5. Enumerating properties

a) for is better suited for arrays and for-in is for objects

Example:

```
var params = {
    productid: 666,
    section: 'products'
};
query = [];
for (var i in params) {                    // iterating the object using the key
    query.push(i + '=' + params[i]);       // remember to use var
}
```

b) Note:

i. **Only enumerable properties show up in a for-in loop. For example, the length (for arrays) and constructor properties don't show up.**

ii. **Enumerable prototypes that come through the prototype chain also show up.**

iii. **propertyIsEnumerable() returns false for all of the prototype's properties, even those that are enumerable and show up in the for-in loop.**

```
newtoy.propertyIsEnumerable('price');
> false
newtoy.constructor.prototype.propertyIsEnumerable('price');
> true          // Remember to use the object/constructor to access the prototype property
```

6. isPrototypeOf()

```
function Human(name) {
    this.name = name;
}
Human.prototype = monkey;        //monkey must be an object, not an constructor
var george = new Human('George');
monkey.isPrototypeOf(george);    // This method tells you whether that specific object
> true                           // is used as a prototype of another object.
Object.getPrototypeOf(george)    // you can't in all browsers, but you can in most of them
> monkey;
```

7. The secret __proto__ link
   a) The secret link is exposed in most modern JavaScript environments as the __proto__ property.

```
var monkey = {
    feeds: 'bananas',
    breathes: 'air'
};
function Human() {}
Human.prototype = monkey;
developer.__proto__ === monkey;      // don't use it in your real scripts because
                                     //it does not exist in all browsers
```

   b) __proto__ is not the same as prototype, since __proto__ is **a property of the instances (objects)**, whereas prototype is **a property of the constructor functions used to create those objects.**

8. Augmenting built-in objects: add personal functionalities
   a) Example:

```
String.prototype.reverse = function () {
    return Array.prototype.reverse.apply(this.split('')).join('');
};                          // Add a reverse method to the string object
```

   b) Once you know JavaScript, you're expecting it to work the same way, no matter which third-party library or widget you're using. **Modifying core objects could confuse the users and maintainers of your code and create unexpected errors.**
   c) The most common and acceptable use case for augmenting built-in prototypes is to add support for new features **(ones that are already standardized by the ECMAScript committee and implemented in new browsers)** to old browsers.
   d)

```
if (typeof String.prototype.trim !== 'function') {
    String.prototype.trim = function () {     // You should first check if the function has
        return this.replace(/^\s+|\s+$/g,'');  // been defined
    };
}
```

9. Prototype gotchas
   a) The prototype chain is live except when you completely replace the prototype object
   b) prototype.constructor is not reliable

```
Dog.prototype = {              // this should be avoided in practical use.
    paws: 4,                   // It turns out that the old objects do not get access to the
    hair: true                 // new prototype's properties
};
lucy.constructor;                      // constructor property of the new
function Object() { [native code] } /  /object no longer reports correctly
```

   c) best practice: **When you overwrite the prototype, remember to reset the constructor property.**

```
Dog.prototype.constructor = Dog;  //remember it's object.prototype.constructor
```

d)
```
var shape={                          // This must be an object
    property:"shape",
    gettype:function(){return this.property;}
}
function Triangle(a,b,c){
    this.property = "triangle",
    this.a=a,this.b=b, this.c=c
}
Triangle.prototype = shape;                    //reset the prototype
Triangle.prototype.constructor = shape;   //reset the constructor of the prototype
Triangle.prototype.getPerimeter = function(){return this.a + this.b + this.c}
```

Chapter 6 Inheritance

1. Prototype chaining: default way of implementing inheritance
   a) This allows methods and properties of the prototype object to be used as if they belonged to the newly-created object.
   b) Prototype also has links to their prototypes, prototype chain, ending with Object.prototype.
   c) An object can access any property found somewhere down the inheritance chain.

2. Prototype chaining example
   a)
```
function Shape(){
    this.name = 'Shape';
    this.toString = function () {
        return this.name;
    };
}
function TwoDShape(){
    this.name = '2D shape';
}                          // remember that JavaScript works with objects, not classes
TwoDShape.prototype = new Shape();        //This is the key of inheritance
TwoDShape.prototype.constructor = TwoDShape;
    // Also reset the constructor property to avoid side effect
```

3. Moving shared properties to the prototype: own properties are not efficient
   a)
```
function Shape(){                //This is not efficient
    this.name = 'Shape';         //name is copied for every object.
}
function Shape() {}
Shape.prototype.name = 'Shape';
//only use it for properties that don't change from one instance to another.
TwoDShape.prototype = new Shape();                    // inherit from Shape
TwoDShape.prototype.constructor = TwoDShape;          // augment prototype
TwoDShape.prototype.name = '2D shape';
```

//Inheritance first before augmenting the prototype. Otherwise all augmented members will be wiped out.

TwoDShape.prototype.isPrototypeOf(my);

> true // TwoDShape.prototype is an object

4. Inheriting the prototype only: more efficient that inheriting new Shape()
   a) new Shape() only gives you own shape properties that are not meant to be reused (otherwise they would be in the prototype). You gain a little more efficiency by:
      - Not creating a new object for the sake of inheritance alone
      - Having less lookups during runtime
   b) original one:

      TwoDShape.prototype = new Shape();

      TwoDShape.prototype = Shape.prototype; //This is more efficient,

      //side effect: because all the prototypes of the children and parents point to the same object, when a child modifies the prototype, the parents get the changes.

5. A temporary constructor – new F():use an intermediary to break the chain.
   a) **var F = function () {}; //An empty function**

      F.prototype = Shape.prototype;

      TwoDShape.prototype = new F();

      TwoDShape.prototype.constructor = TwoDShape;
   b) **Use this to supports the idea that only properties and methods added to the prototype should be inherited, and own properties should not while the chain is broken.**

6. Uber – access to the parent from a child object. Like the Java super keyword

```
function Shape() {}                          //Empty object
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
    Var const = this.constructor;      //const is the class now.
    return const.uber ? this.const.uber.toString() + ', ' + this.name : this.name;
};
TwoDShape.uber = Shape.prototype;      //Use [class-level] property to inherit method
my.toString();                         // The result is that when you call toString(), all toString()
> "Shape, 2D shape, Triangle"          // methods up the prototype chain are called
```

7. Isolating the inheritance part into a function

```
function extend(Child, Parent) {
    var F = function () {};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
    Child.uber = Parent.prototype;
}
```

```
function TwoDShape() {}                    // define -> inherit -> augment
extend(TwoDShape, Shape);
TwoDShape.prototype.name = '2D shape';
```

8. Copying properties
   a) A different approach: copies all of the properties from the parent's prototype to the child's prototype.

```
function extend2(Child, Parent) {
    var p = Parent.prototype;
    var c = Child.prototype;
    for (vari in p) {          // This method is a little inefficient compared to the child
        c[i] = p[i];           //previous method because properties of the prototype are
    }                          // being duplicated
    c.uber = p;                //here uber is defined in prototype
}
```

   b) Although it's not that efficient, but not so bad because only the primitive data types are duplicated. Additionally, this is beneficial during the prototype chain lookups as there are fewer chain links to follow before finding the property.

9. Heads-up when copying by reference, same as Java
   If you want to address the problem that objects are copied by reference, consider a deep copy, described further.

10. Objects inherit from objects: You can create objects just by using the object literal
    a) Original realization: *pseudo-classical inheritance pattern*

```
Child.prototype = new Parent();
```

    b) Using object copy(nothing with prototype and)

```
function extendCopy(p) {
    var c = {};                // An blank object
    for (vari in p) {
        c[i] = p[i];           //copy members(shallow copy)
    }
    c.uber = p;                //for calling methods of inherited object
    return c;                  //return a new object
}
vartwoDee = extendCopy(shape);
twoDee.name = '2D shape';      //Noted that here is purely object-level operation
twoDee.toString = function () {    // this method is somewhat verbose initialization
    return this.uber.toString() + ', ' + this.name;
};
```

    c) Solution for verbose initialization
    Having extendCopy() accept two parameters: **an object to inherit from** and **another object tliteral of properties to add to the copy** before it's returned, in other words just merge two objects.

11. Deep copy:
    a) Example:

```
function deepCopy(p, c) {
    c = c || {};
    for (vari in p) {                                   //loop the propery
        if (p.hasOwnProperty(i)) {                      //copy the property
            if (typeof p[i] === 'object') {             //if encountered on object
                c[i] = Array.isArray(p[i]) ? [] : {};   //see if it's an array
                deepCopy(p[i], c[i]);                   //dive into it
            } else {
                c[i] = p[i];
            }
        }
    }
    return c;
}
```

    b) Note: Two side notes about the deepCopy() function:
        i. **Filtering out non-own properties with hasOwnProperty() is always a good idea to make sure you don't carry over someone's additions to the core prototypes.**
        ii. Array.isArray() exists since ES5 because it's surprisingly hard otherwise to tell real arrays from objects.

```
For ES3, we have
if (Array.isArray !== "function") {
    Array.isArray = function (candidate) {
        Return Object.prototype.toString.call(candidate) ==='[object Array]';
    };
}
```

12. object()
    a) Example:

```
function object(o) {        // accepts an object and returns a new one that has the
    var n;                  // parent as a prototype
    function F() {}
    F.prototype = o;        //o is an object
    n = new F();
    n.uber = o;             // for super-like inheritance
    return n;
}
var triangle = object(twoDee);
triangle.name = 'Triangle';       //augmenting the object
```

    b) This pattern is also referred to as **prototypal inheritance**, because you use a parent object as the prototype of a child object. It's also adopted and built upon in ES5 and called **Object.create()(three properties)**

13. Using a mix of **prototypal inheritance** and **copying properties**
    a) Use prototypal inheritance to use an existing object as a prototype of a new one
    b) Copy all of the properties of another object into the newly created one
    c) Example:

```
function objectPlus(o, stuff) {          // o is the inherited parent
    var n;                               // stuff is the property object
    function F() {}
    F.prototype = o;
    n = new F();
    n.uber = o;
    for (vari in stuff) {
        n[i] = stuff[i];
    }
    return n;
}
vartwoDee = objectPlus(shape, {          //one-shot initialization
    name: '2D shape',
    toString: function () {
        return this.uber.toString() + ', ' + this.name;
    }
});
```

14. Multiple inheritance: just a loop of single inheritance
    a) Example:

```
function multi() {
    var n = {}, stuff, j = 0, len = arguments.length;
    for (j = 0; j <len; j++) {
        stuff = arguments[j];
        for (vari in stuff) {
            if (stuff.hasOwnProperty(i)) {
                n[i] = stuff[i];
            }
}
```

    b) multi() loops through the input objects in the order they appear and if it happens that two of them have the same property, **the last one wins.**

15. Mixin: By passing them all to multi() you get all their functionality without making them part of the inheritance tree.

16. Parasitic inheritance
    a) A function that creates objects by taking all of the functionality from another object into a new one, augmenting the new object, and returning it, "pretending that it has done all the work".
    b) function triangle(s, h) { //**this is only a normal function, not a constructor**

```
    var that = object(twoD);        //that is only a name of an variable
    that.name ='Triangle';
    that.getArea = function () {
        return this.side * this.height / 2;
    };
    return that;
}
Var t = triangle();                 // no new operation
```

17. Borrowing a constructor
    a) The constructor of the child calls the constructor of the parent using either **call() or apply()** methods. This can be called **stealing a constructor**, or **inheritance by borrowing a constructor** if you want to be more subtle about it.
    b) The child constructor calls the parent'sconstructor and binds the child's newly-created this object as the parent's this. **The parent's own properties are recreated as the child's own properties.**
    c)
```
function Triangle() {                   //Shape is a function
    Shape.apply(this, arguments);       //will inherit everything in the original object,
}                                       //but not those added by prototype
Triangle.prototype = new Shape();       //inherit the prototype
Triangle.prototype.name = 'Triangle';   // own properties of the parent are inherited
                                        //twice.
```

18. Borrow a constructor and copy its prototype
    a) Example:
    b)
```
function Triangle() {
    Shape.apply(this, arguments);
}
extend2(Triangle, Shape);               //avoid double copy
Triangle.prototype.name = 'Triangle';
```

19. summary

#	Name	Example	Classification	Notes
1	Prototype chaining (pseudo-classical)	`Child.prototype = new Parent();`	• Works with constructors • Uses the prototype chain	• The default mechanism. • Tip: move all properties/ methods that are meant to be reused to the prototype, add the non-reusable as own properties.

2	Inherit only the prototype	`Child.prototype = Parent.prototype;`	• Works with constructors • Copies the prototype (no prototype chain, all share the same prototype object)	• More efficient, no new instances are created just for the sake of inheritance. • Prototype chain lookup during runtime- is fast, since there's no chain. • Drawback: children can modify parents' functionality.
3	Temporary constructor	`function extend(Child, Parent) { var F = function(){}; F.prototype = Parent.prototype; Child.prototype = new F(); Child.prototype.constructor = Child; Child.uber = Parent.prototype; }`	• Works with constructors • Uses the prototype chain	• Unlike #1, it only inherits properties of the prototype. Own properties (created with this inside the constructor) are not inherited. • Provides convenient access to the parent (through uber).
4	Copying the prototype properties	`function extend2(Child, Parent) { var p = Parent.prototype; var c = Child.prototype; for (vari in p) { c[i] = p[i]; } c.uber = p; }`	• Works with constructors • Copies properties • Uses the prototype chain	• All properties of the parent prototype become properties of the child prototype • No need to create a new object only for inheritance purposes • Shorter prototype chains
8	Extend and augment	`function objectPlus(o, stuff) { var n; function F() {} F.prototype = o; n = new F(); n.uber = o; for (vari in stuff) { n[i] = stuff[i]; } return n; }`	• Works with objects • Uses the prototype chain • Copies properties	• Mix of prototypal inheritance (#7) and copying properties (#5) • One function call to inherit and extend at the same time

5	Copy all properties (shallow copy)	`function extendCopy(p) { var c = {}; for (vari in p) { c[i] = p[i]; } c.uber = p; return c; }`	• Works with objects • Copies properties	• Simple • Doesn't use prototypes
6	Deep copy	Same as above, but recurse into objects	• Works with objects • Copies properties	Same as #5 but clones objects and arrays
7	Prototypal inheritance	`function object(o) { function F() {} F.prototype = o; return new F(); }`	• Works with objects • Uses the prototype chain	• No pseudo-classes, objects inherit from objects • Leverages the benefits of the prototype
9	Multiple inheritance	`function multi() { var n = {}, stuff, j = 0, len = arguments.length; for (j = 0; j <len; j++) { stuff = arguments[j]; for (vari in stuff) { n[i] = stuff[i]; } } return n; }`	• Works with objects • Copies properties	• A mixin-style implementation • Copies all the properties of all the parent objects in the order of appearance
10	Parasitic inheritance	`function parasite(victim) { var that = object(victim); that.more = 1; return that; }`	• Works with objects • Uses the prototype chain	• Constructor-like function creates objects • Copies an object, augments and returns the copy
11	Borrowing constructors	`function Child() { Parent.apply(this, arguments); }`	Works with constructors	• Inherits only own properties • Can be combined with #1 to inherit the prototype too • Convenient way to deal with the issues when a child inherits a property that is an object (and therefore passed by reference)

12	Borrow a constructor and copy the prototype	`function Child() { Parent.apply(this, arguments); }`
		
		`extend2(Child, Parent);`
			• Works with constructors
			• Uses the prototype chain
			• Copies properties
				• Combination of #11 and #4
				• Allows you to inherit both own properties and prototype properties without calling the parent constructor twice

20. case study
    a) The common functionality in Shape would be:
        - A draw() method that can draw any shape given the points
        - A getParameter() method
        - A property that contains an array of points
        - Other methods and properties as needed
    b)
```
function Shape() {
    this.points = [];
    this.lines= [];
    this.init();
}
```
    c)
```
function Triangle(a, b, c) {
    this.points = [a, b, c];
    this.getArea = function () {
    var p = this.getPerimeter(),
    s = p / 2;
        return Math.sqrt(
            s
            * (s - this.lines[0].length)
            * (s - this.lines[1].length)
            * (s - this.lines[2].length));
        };

function Square(p, side){
    Rectangle.call(this, p, side, side);
} }
(function () {
    var s = new Shape();
    Triangle.prototype = s;
    Rectangle.prototype = s;
    Square.prototype = s;
})();
```

Chapter 7 The Browser Environment

1. BOM and DOM – an overview
    a) Core ECMAScript objects: All the objects mentioned in the previous chapters
    b) DOM: Objects that have to do with the currently loaded page (the page is also called the document)
    c) BOM: Objects that deal with everything outside the page (the browser window and the desktop screen)

2. BOM
    a) The window object revisited: the window object also serves a second purpose providing

information about the browser environment.

1. There's a window object for every frame, iframe, pop up, or browser tab.

b) window.navigator: has some information about the browser and its capabilities.

1. Window.navigator.userAgent,: a long string of browser identification.
   window.navigator.userAgent; //under FF
   > "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/536.28.10 (KHTML, like Gecko) Version/6.0.3 Safari/536.28.10
   > "Mozilla/5.0 (compatible; **MSIE** 10.0; Windows NT 6.1; Trident/6.0)"
   Note: **It's better not to rely on the user agent string, but to use feature sniffing.** The reason is that **some browsers allow users to modify the string and pretend they are using a different browser.**

2. Feature sniffing

```
if (typeof window.addEventListener === 'function') {
    // feature is supported, let's use it
} else {
    // hmm, this feature is not supported, will have to
    // think of another way
}
```

3. window.location: contains information about the URL of the currently loaded page
   i. see what window.location contains

```
for (var i in location) {                    //location is an object
    if (typeof location[i] === "string") {
        console.log(i + ' = "' + location[i] + '"');
    }
}
href = "http://search.phpied.com:8080/search?q=java&what=script#results"
hash = "#results"
host = "search.phpied.com:8080"
hostname = "search.phpied.com"
pathname = "/search"
port = «8080»
protocol = «http:»
search = "?q=java&what=script"
```

   ii. navigating to another page

```
window.location.href = 'http://www.packtpub.com';
location.href = 'http://www.packtpub.com';
location = 'http://www.packtpub.com';
location.assign('http://www.packtpub.com');
location.replace('http://www.yahoo.com');    //not leaving a history record
```

iii. reload a page

location.reload();
location = location;

4. window.history: allows limited access to the previously visited pages in the same browser session

```
window.history.length;          // see how many pages the user has visited
> 5                             // You cannot see the actual URLs though
history.forward();              // navigate back and forth through the user's session
history.back();
history.go(-1);                 //go back one page
history.go(0);                  //reload current page

// HTML5 History API, which lets you change the URL without reloading the page.
//This can be used for browser to remember the current state

history.pushState({a: 1}, "", "hello");
history.pushState({b: 2}, "", "hello-you-too");
history.state;
```

5. window.frames: a collection of all of the frames in the current page including iframe

```
window.frames === window;       // window.frames always exists and points to
> true                          // window
frames.length                   //see how many frames in the current page
> 1

window.frames[0];               // get access to the iframe's window
window.frames[0].window;
window.frames[0].window.frames;
frames[0].window;
frames[0];

frames[0].window.location.reload();         //reload the frame

self === window;                // self is the same as window
> true
window.frames['myframe'] === window.frames[0];
> true                          // access the frame by name
```

6. window.screen: provides information about the environment **outside the browser**

```
window.screen.colorDepth;      //contains the color bit-depth (the color quality) of the
> 32                           // monitor.
screen.width;                  // check the available screen real estate
> 1440
screen.availWidth;
> 1440
screen.height;                 // height is the whole screen
> 900
screen.availHeight;            //availHeight subtracts any operating system menus
> 847                          // such as the Windows task bar
window.devicePixelRatio;
> 1                            // ratio between physical pixels and device pixels
```

7. window.open()/close():allows you to open new browser windows
   a) Generally you should be able to open a new window if it was initiated by the user. Otherwise, if you try to open a pop up as the page loads, it will most likely be blocked, because the user didn't initiate it explicitly.
   b) window.open() accepts the following parameters:
      - URL to load in the new window
      - Name of the new window, which can be used as the value of a form's target attribute
      - Comma-separated list of features. They are as follows:
        - resizable: Should the user be able to resize the new window
        - width, height: Width and height of the pop up
        - status: Should the status bar be visible

      It returns a reference to the window object of the newly created browser instance.

```
var win = window.open('http://www.packtpub.com', 'packt',
                      'width=300,height=300,resizable=yes');
                      //you can check win to see if this is blocked/
win.close()           //close the new window.
```

   c) Best practice: **stay away from opening new windows for accessibility and usability reasons.**

8. window.moveTo() and window.resizeTo():

```
window.moveTo(100, 100);      //moves the browser window to screen location
                              //x= 100 and y = 100
window.moveBy(10, -10)        // moves the window 10 pixels to the right and 10
                              pixels up from its current location
window.resizeTo(x, y) and window.resizeBy(x, y)
//accept the same parameters as the move methods but they resize the window as
//opposed to moving it
```

9. window.alert(), window.prompt(), and window.confirm()

   a) confirm() gives the user two options, OK and Cancel
   You'll notice the following things:
   - **JavaScript code execution freezes, waiting for the user's answer**
   - **Clicking on OK returns true, clicking on Cancel or closing the message using the X icon (or the ESC key) returns false**

   b) prompt() collects textual input
   The value of answer is one of the following:
   - null if you click on Cancel or the X icon, or press ESC
   - "" (empty string) if you click on OK or press Enter without typing anything
   - A text string if you type something and then click on OK (or press Enter)

10. window.setTimeout() and window.setInterval():allow for scheduling the execution of a piece of code
   a) setTimeout() attempts to execute the given code once **after a specified number of milliseconds.**

```
function boo() { alert('Boo!'); }
var id = setTimeout(boo, 2000);     // You can use this ID to cancel the timeout
clearTimeout(id);                   // using clearTimeout().
```

   b) setInterval() attempts to execute it **repeatedly** after a specified number of milliseconds has passed.

```
var id = setInterval(boo, 2000);    // You can use this ID to cancel the timeout
clearInterval(id);                  // using clearInterval(id);
```

   c) Both functions accept a pointer to a callback function as a first parameter.
   This alternative is preferred:

```
var id = setInterval(
      function () {
            alert('boo, boo');
      },
      2000
);
```

   d) Be aware that scheduling a function in some amount of milliseconds **is not a guarantee** that it will execute exactly at that time.
      1. most browsers don't have millisecond resolution time.
      2. browsers maintain a queue of what you request them to do. But if the queue is delayed by something slow happening, your function will have to wait.

11. requestAnimationFrame():tells the browser that you wish to perform an animation and requests that the browser call a specified function to update an animation before the next repaint.

    requestID = window.requestAnimationFrame(callback);

12. window.document: refers to the currently loaded document

3. DOM: represents an XML or an HTML document as a tree of nodes.
   a) there is a Core DOM specification that is applicable to all XML documents, and there is also an HTML DOM specification, which extends and builds upon the core DOM.

   b) Accessing DOM nodes

```
document.nodeType;
> 9
//There are 12 node types, represented by integers. As you can see, the document //node type is 9. The most commonly used are 1 (element), 2 (attribute), and 3 (text).
document.nodeName;
> "#document"
document.nodeValue;        // for text nodes the value is the actual text.
> null
```

   c) documentElement

```
document.documentElement.nodeType; //the root node
> 1                                  //element node
document.documentElement.nodeName;// For element nodes, both nodeName and
> "HTML"                           // tagName properties contain the name of the tag
document.documentElement.tagName;
> "HTML"
```

   d) Child nodes

```
document.documentElement.childNodes.length;
> 3
document.documentElement.childNodes[0];          // an empty HTML document
> <head>…</head>
document.documentElement.childNodes[1];
> #text
document.documentElement.childNodes[2];
> <body>…</body>
document.documentElement.childNodes[1].parentNode;
> <html>…</html>
```

   e) Attributes

```
bd.childNodes[1].attributes.length;        //check how many attributes
```

> 1
bd.childNodes[1].attributes[0].nodeName;
> "class"
bd.childNodes[1].attributes[0].nodeValue;
> "opener"
bd.childNodes[1].attributes['class'].nodeValue;
> "opener"
bd.childNodes[1].getAttribute('class');
> "opener" //You can access the attributes by index and by name

f) Accessing the content inside a tag

bd.childNodes[1].innerHTML;
> "first paragraph"
bd.childNodes[1].childNodes[0].nodeName;
> "#text"
bd.childNodes[1].childNodes[0].nodeValue;
> "first paragraph"

g) DOM access shortcuts: getElementsByTagName(), getElementsByName(), and getElementById().
1. getElementsByTagName() takes a tag name (the name of an element node) and returns an HTML collection (array-like object) of nodes with the matching tag name.

document.getElementsByTagName('p')[0];
> first paragraph
document.getElementsByTagName('p')[2].id;
> "closer"
document.getElementsByTagName('p')[0].className;
> "opener"
document.getElementsByTagName('*').length; //get all node
> 8 // To get all elements you can use IE's proprietary **document.all** //collection

2. **getElementByClassName():** This method finds elements using their class attribute
querySelector(): This method finds an element using a CSS selector string
querySelectorAll(): This method is the same as the previous one but returns all matching elements not just the first

h) Siblings, body, first, and last child
   a) nextSibling and previousSibling are two other convenient properties to navigate the DOM tree
   var para = document.getElementById('closer');
   para.nextSibling;
   > #text

```
para.previousSibling;
> #text
```

b) firstChild and lastChild are also convenient. firstChild is the same as childNodes[0] and lastChild is the same as childNodes[childNodes.length - 1]:

i) Walk the DOM

```
function walkDOM(n) {
    do {
        console.log(n);
        if (n.hasChildNodes()) {
            walkDOM(n.firstChild);
        }
    } while (n = n.nextSibling);
}
walkDOM(document.documentElement);
```

j) Modifying DOM nodes

```
my.innerHTML = 'final!!!';
> "final!!!"
```

k) Modifying styles

```
my.style.border = "1px solid red";
> "1px solid red"
my.style.cssText;
> "border: 1px solid red; font-weight: bold;
my.style.cssText += " border-style: dashed;"  // modifying styles is a string manipulation
> "border: 1px dashed red; font-weight: bold; border-style: dashed;"
```

l) Creating new nodes

```
var myp = document.createElement('p');
myp.innerHTML = 'yet another';
myp.style;              // the new element automatically gets all the default properties,
CSSStyleDeclaration
myp.style.border = '2px dotted blue';
> "2px dotted blue"
document.body.appendChild(myp);       //append to the DOM tree
```

m) DOM-only method

1. In pure DOM you need to perform the following steps:

1. Create a new text node containing yet another text
2. Create a new paragraph node
3. Append the text node as a child to the paragraph
4. Append the paragraph as a child to the body

2. cloneNode(): Create nodes is by copying (or cloning) existing ones. this method accepts a boolean parameter (true = deep copy with all the children, false = shallow copy, only this node).

```
document.body.appendChild(el.cloneNode(true));
```

3. insertBefore(): The same as appendChild(), but **accepts an extra parameter specifying where (before which element) to insert the new node.**

```
document.body.insertBefore(
        document.createTextNode('first boo!'),
        document.body.firstChild
);
```

4. Removing nodes: To remove nodes from the DOM tree, you can use the method **removeChild().**

```
var myp = document.getElementsByTagName('p')[1];
var removed = document.body.removeChild(myp);      // The method returns the
                                                   //removed node
```

5. replaceChild():removes a node and puts another one in its place.

```
var replaced = document.body.replaceChild(removed, p);
//Just like removeChild(), replaceChild() returns a reference to the node that is
//now out of the tree:

document.body.innerHTML = '';       //remove all children
> ""

function removeAll(n) {                 //small function to remove all child nodes
      while (n.firstChild) {
            n.removeChild(n.firstChild);
      }
}
removeAll(document.body);
```

n) HTML-only DOM objects: document.body is one example of a legacy object inherited from the prehistoric DOM Level 0 and moved to the HTML extension of the DOM

specification.

o) Primitive ways to access the document
- ✧ document.images: This is a collection of all of the images on the page. The Core DOM equivalent is document.getElementsByTagName('img')
- ✧ document.applets: This is the same as document. getElementsByTagName('applet')
- ✧ document.links
- ✧ document.anchors
- ✧ **document.forms** **// One of the most widely used collections**

p) document.forms: same as document.getElementsByTagName('forms')

```
document.forms[0].elements[0];    // access the first input of the first form
document.forms[0].elements[0].value = 'me@example.org';
                                  // change the text in the field
document.forms[0].elements[0].disabled = true;
                                  // disable the field dynamically
document.forms[0].elements['search'];
                          // access form elements by name attribute
```

q) document.write():insert HTML into the page while the page is being loaded.
Note, that you can only use document.write() while the page is being loaded. If you try it after page load, it will replace the content of the whole page.

r) Cookies, title, referrer, domain

i. Unlike the previous ones, for these properties there are **no core DOM equivalents.**
ii. document.cookie:a property that contains a cookie
iii. document.title allows you to change the title of the page displayed in the browser window. **This doesn't change the value of the <title> element**.
iv. document.referrer tells you the URL of the previously-visited page
v. document.domain gives you access to the domain name of the currently loaded page.
This is commonly used when you need to perform so-called **domain relaxation.** Imagine your page is www.yahoo.com and inside it you have an iframe hosted on music.yahoo.com subdomain. These are two separate domains so the browser's security restrictions won't allow the page and the iframe to communicate. To resolve this you can set document.domain on both pages to yahoo.com and they'll be able to talk to each other
You can only set the domain to a less-specific one, for example, you can change www.yahoo.com to yahoo.com, but you cannot change yahoo.com to www.yahoo.com or any other non-yahoo domain.

s) Events

i. Browser broadcasts events and your code could be notified should it decide to tune in and listen to the events as they happen.

ii. Inline HTML attributes

```
<div onclick="alert('Ouch!')">click</div>
```

//least maintainable, don't use that

iii. Element Properties: have some code executed when a click event fires is to assign a function to the onclick property of a DOM node element.

```
<div id="my-div">click</div>
<script>
    var myelement = document.getElementById('my-div');
    myelement.onclick = function () {
        alert('Ouch!');
        alert('And double ouch!');
    };
</script>
```

// better because it helps you keep your clean of any JavaScript code.

// drawback that you can attach only one function to the event

t) DOM event listeners: best way to work with browser events

```
var mypara = document.getElementById('closer');
mypara.addEventListener('click', function () {  //false means that
    alert('Boo!');
    }, false);
mypara.addEventListener('click', console.log.bind(console), false);
mypara.removeEventListener('click', console.log.bind(console), false);
```

// **Note, that when you remove a listener, you have to pass a pointer to the same function you previously attached or the body is exactly the same.**

u) Capturing and bubbling

i. The process of propagating an event can be implemented in two ways:

1. Event capturing: The click happens on the document first, then it propagates down to the body, the list, the list item, and finally to the link.
2. Event bubbling: The click happens on the link and then bubbles up to the document

ii. Event propagates from the document to the link (target) and then bubbles back up to the document.

iii. Best practice'

1. The third parameter to addEventListener() specifies whether or not capturing should be used. **In order to have your code more portable across browsers, it's better to always set this parameter to false and use**

bubbling only.

2. You can stop the propagation of the event in your listeners so that it stops bubbling up and never reaches the document. To do this you can call the **stopPropagation()** method of the event object (there is an example in the next section).
3. **You can also use event delegation.** If you have ten buttons inside a , you can always attach ten event listeners, one for each button. But a smarter thing to do is to **attach only one listener to the wrapping and once the event happens, check which button was the target of the click.**

iv. Stop propagation

```
function paraHandler(e) {
    alert('clicked paragraph');
    e.stopPropagation();          //use the method of the event
}
```

v) Prevent default behavior: disable the default behavior

```
var all_links = document.getElementsByTagName('a');
for (var i = 0; i < all_links.length; i++) {                    // loop all links
    all_links[i].addEventListener(
        'click',                                                // event type
        function (e) {                                          // handler
            if (!confirm('Sure you want to follow this link?')) {
                e.preventDefault();
            }
        },
        false                                                   // don't use capturing
    );
}
```

w) Cross-browser event listeners

i. how IE is different

1. In IE there's no addEventListener() method, although since IE Version 5 there is an equivalent **attachEvent()**. For earlier versions, your only choice is accessing the property (such as onclick) directly.
2. click event becomes onclick when using attachEvent().
3. If you listen to events the old-fashioned way (for example, by setting a function value to the onclick property), when the callback function is invoked, **it doesn't get an event object passed as a parameter**. But, regardless of how you attach the listener in IE, there is always **a global object window.event** that points to the latest event.
4. In IE the event object **doesn't get a target attribute** telling you the element on

which the event fired, but it does have an equivalent property called **srcElement.**

5. As mentioned before, event capturing doesn't apply to all events, so only bubbling should be used.
6. There's no stopPropagation() method, but you can set the IE-only **cancelBubble property to true.**
7. There's no preventDefault() method, but you can set the IE-only **returnValue property to false**.
8. To stop listening to an event, instead of removeEventListener() in IE you'll need **detachEvent().**

Example:

```
function callback(evt) {
    evt = evt || window.event;
    var target = evt.target || evt.srcElement;
    console.log(target.nodeName);
}
if (document.addEventListener) {                    // Modern browsers
    document.addEventListener('click', callback, false);
} else if (document.attachEvent) {                  // old IE
    document.attachEvent('onclick', callback);
} else {
    document.onclick = callback;                    // ancient
}
```

x) Types of events

i. Mouse events

mouseup, mousedown, click (the sequence is mousedown-up-click), dblclick mouseover (mouse is over an element), mouseout (mouse was over an element but left it), mousemove

ii. Keyboard events

keydown, keypress, keyup (occur in this sequence)

iii. Loading/window events

1. load (an image or a page and all of its components are done loading), unload (user leaves the page), beforeunload (the script can provide the user with an option to stop the unload)
2. abort (user stops loading the page or an image in IE), error (a JavaScript error, also when an image cannot be loaded in IE)
3. resize (the browser window is resized), scroll (the page is scrolled), contextmenu (the right-click menu appears)

iv. Form events

1. focus (enter a form field), blur (leave the form field)
2. change (leave a field after the value has changed), select (select text in

a text field)

3. reset (wipe out all user input), submit (send the form)

Additionally, modern browsers provide **drag events (dragstart, dragend, drop, and others) and touch devices provide touchstart, touchmove, and touchend.**

y) XMLHttpRequest: allows you to send HTTP requests from JavaScript

i. AJAX stands for Asynchronous JavaScript and XML.

- Asynchronous because after sending an HTTP request your code doesn't need to wait for the response
- JavaScript because it's obvious that XHR objects are created with JavaScript.
- XML because initially developers were making HTTP requests for XML documents and were using the data contained in them to update the page. **This is no longer a common practice, though, as you can request data in plain text, in the much more convenient JSON format, or simply as HTML ready to be inserted into the page.**

ii.
```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = myCallback;
xhr.open('GET', 'somefile.txt', true);
xhr.send('');
```

iii. Processing the response

The possible values of the readyState property are as follows:

- 0-uninitialized
- 1-loading
- 2-loaded
- 3-interactive
- 4-complete

```
function myCallback() {
    if (xhr.readyState < 4) {
        return; // not ready yet
    }
    if (xhr.status !== 200) {
        alert('Error!'); // the HTTP status code is not OK
        return;
    }
    // all is fine, do the work
    alert(xhr.responseText);
}
```

iv. Creating XMLHttpRequest objects in IE prior to Version 7

a fully-cross-browser solution:

```
var ids = ['MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP',
                                                    'Microsoft.XMLHTTP'];
var xhr;
if (XMLHttpRequest) {
    xhr = new XMLHttpRequest();
} else {
    // IE: try to find an ActiveX object to use
    for (var i = 0; i < ids.length; i++) {
        try {
            xhr = new ActiveXObject(ids[i]);
            break;
        } catch (e) {}
      }
}
```

v. Distinguish between requests

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = (function (myxhr) {
        return function () {
            myCallback(myxhr);
        };                              //a closure, localize myxhr
    }(xhr));
xhr.open('GET', 'somefile.txt', true);      //even xhr is used for a second requests
xhr.send('');                               //it keeps pointing to the original object
```

chapter 8 Coding and Design Patterns: a pattern is a good solution to a common problem

1. Coding patterns

   a) Separating behavior
      i. Content
         1. The style attribute of HTML tags should not be used, if possible.
         2. Presentational HTML tags such as <font> should not be used at all.
         3. Tags should be used for their semantic meaning, not because of how browsers render them by default. For instance, developers sometimes use a tag where a <p> would be more appropriate. It's also favorable to use and instead of <b> and <i> as the latter describe the visual presentation rather than the meaning.

      ii. Presentation

         1. Keep presentation out of the content is to **reset, or nullify** all browser defaults. Reset.css

iii. Behavior

1. Behavior is usually added by using JavaScript that is isolated to <script> tags, and preferably contained in external files.
2. Best practice:
    - Minimize the number of <script> tags
    - **Avoid inline event handlers**
    - Do not use CSS expressions
    - Dynamically add markup that has no purpose if JavaScript is disabled by the user
    - Towards the end of your content when you are ready to close the <body> tag, insert a single external.js file. **The reason is that JavaScript blocks the DOM construction of the page and in some browsers even the downloads of the other components that follow**

    HTML5 ways: <script **defer** src="behaviors.js"></script>

b) Namespace: Global variables should be avoided in order to reduce the possibility of variable naming collisions

i. The idea is simple, you create only one global object and all your other variables and functions become properties of that object.

ii. Namespace constructors: Here is how you can have a DOM utility that has an Element constructor, which allows you to create DOM elements easier:

```
MYAPP.dom = {};
MYAPP.dom.Element = function (type, properties) {
    var tmp = document.createElement(type);
    for (var i in properties) {                    //property is an object
        if (properties.hasOwnProperty(i)) {
            tmp.setAttribute(i, properties[i]);
        }
    }
    return tmp;
};
var link = new MYAPP.dom.Element('a', {href: 'http://phpied.com', target: '_blank'});
```

iii. A namespace() method: use more convenient syntax to create a namespace

```
MYAPP.namespace = function (name) {
    var parts = name.split('.');
    var current = MYAPP;
    for (var i = 0; i < parts.length; i++) {
        if (!current[parts[i]]) {
            current[parts[i]] = {};          //recursively create a namespace
        }
        current = current[parts[i]];         // dive into the next layer
```

```
    }
};
MYAPP.namespace('dom.style');
> var MYAPP = {
    dom: {
        style: {}
      }
  };
```

c) Init-time branching

i. You need to branch your code depending on what's supported by the browser currently executing your script. But this branching can happen far too often and, as a result, may slow down the script execution

ii. Use **function definition** to do branching

```
if (window.addEventListener) {          //feature sniffing
    MYAPP.event.addListener = function (el, type, fn) {
        el.addEventListener(type, fn, false);
    };
    MYAPP.event.removeListener = function (el, type, fn) {
        el.removeEventListener(type, fn, false);
    };
    } else if (document.attachEvent) { // IE
        MYAPP.event.addListener = function (el, type, fn) {
            el.attachEvent('on' + type, fn);
    };
        MYAPP.event.removeListener = function (el, type, fn){
            el.detachEvent('on' + type, fn);
      };
      } else { // older browsers
        MYAPP.event.addListener = function (el, type, fn) {
            el['on' + type] = fn;
      };
      MYAPP.event.removeListener = function (el, type) {
        el['on' + type] = null;
      };
}
```

Note: watch out for when sniffing features is not to **assume too much after checking for one feature**. it's best to individually c**heck for features you intend to use** and don't generalize on what a certain browser supports.

d) Lazy definition: branching happens only **when the function is called for the first time**.

i. When the function is called, it **redefines itself with the best implementation.**

ii. The lazy definition also makes the initialization process lighter as there'**s no init-**

time branching work to be done.

iii. Example:

```
var MYAPP = {};
MYAPP.myevent = {
    addListener: function (el, type, fn) {
      if (el.addEventListener) {
          MYAPP.myevent.addListener = function (el, type, fn) {
              el.addEventListener(type, fn, false);
          };                              //redefine itself for modern browsers
      } else if (el.attachEvent) {
          MYAPP.myevent.addListener = function (el, type, fn) {
              el.attachEvent('on' + type, fn);
          };
      } else {
          MYAPP.myevent.addListener = function (el, type, fn) {
              el['on' + type] = fn;
          };
      }
    MYAPP.myevent.addListener(el, type, fn);      //process first binding
    }
};
```

e) Configuration object: convenient when you have a lot of options.

i. Instead of having many parameters, you can **use one parameter** and **make it an object**. The **properties of the object** are the actual parameters.

ii. Benefits

1. The order doesn't matter
2. You can easily skip parameters that you don't want to set
3. It's easy to add more optional configuration attributes.
4. It makes the code more readable because the configuration object's properties are present in the calling code along with their names

iii. Example:

```
MYAPP.dom.FancyButton = function (text, conf) {
    var type = conf.type || 'submit';
    var font = conf.font || 'Verdana';
};
var config = {
   font: 'Arial, Verdana, sans-serif',
   color: 'white'
};
```

iv. Drawback: trivial to abuse the technique. You will find it tempting to keep adding some that are not entirely optional or some that are dependent on other properties.

v. Best practice: **all these properties should be independent and optional**

f) Private properties and methods

i. Private methods

```
MYAPP.dom.FancyButton = function (text, conf) {
    var styles = {                          //a private property
        font: 'Verdana',
        border: '1px solid black',
        color: 'black',
        background: 'grey'
    };
    function setStyles(b) {                 //a private method
        var i;
        for (i in styles) {
            if (styles.hasOwnProperty(i)) {
                b.style[i] = conf[i] || styles[i];
            }
        }
    }
    conf = conf || {};
    var b = document.createElement('input');
    b.type = conf.type || 'submit';
    b.value = text;
    setStyles(b);
    return b;
};
```

ii. Privileged methods: normal public methods that can access private methods or properties.

```
var MYAPP = {};
MYAPP.dom = (function () {
    var _setStyle = function (el, prop, value) {
        console.log('setStyle');
    };
    var _getStyle = function (el, prop) {
        console.log('getStyle');
    };
    return {                                //use closure
        setStyle: _setStyle,
        getStyle: _getStyle,
        yetAnother: _setStyle
    };
}());
```

iii. When you expose something private, keep in mind that objects (and functions and arrays are objects too) are passed **by reference** and, **therefore, can be modified from the outside.**

g) Immediate functions: especially suitable for on-off initialization tasks performed when the script loads.
   i. If the creation of these objects is more complicated and involves some initialization work, then you can do this in the first part of the self-executable function and return a single object

h) Modules: Combining several of the previous patterns, gives you a new pattern, commonly referred to as a module pattern.
   i. Note: JavaScript doesn't have a built-in concept of modules, although this is planned for the future via export and import declarations
   ii. The module pattern includes:
      - Namespaces to reduce naming conflicts among modules
      - An immediate function to provide a private scope and initialization
      - Private properties and methods
      - Returning an object that has the public API of the module
   iii. Example:

```
namespace('MYAPP.module.amazing');
MYAPP.module.amazing = (function () {
    var another = MYAPP.module.another;   // short names for dependencies
    var i, j;                             // local/private variables
    function hidden() {}                  // private functions
    return {                              // public API
        hi: function () {
            return "hello";
        }
    };
}());
```

i) Chaining: allows you to invoke multiple methods on one line as if the methods are the links in a chain.
   i. The functions should **return this;**

```
document.body.appendChild(
    new MYAPP.dom.Element('span')
    .setText('hello')
    .setStyle('color', 'red')
    .setStyle('font', 'Verdana')
);
```

   ii. Drawback: it makes it a little harder to debug when an error occurs somewhere in a long chain and you don't know which link is to blame because they are all on the same line.

j) JSON: it's trivial to work with JSON in JavaScript

```
var response = JSON.parse(xhr.responseText);
var str = JSON.stringify({hello: "you"})
```

2. Design patterns: class-like syntax and still implement the singleton pattern

a) Singleton

```
function Logger() {
    if (Logger.single_instance) {
        Logger.single_instance = this;
    }
    return Logger.single_instance;
}
```

the property of the Logger constructor is publicly visible, so it can be overwritten at any time. **You can use closure to make it private.**

b) Factory

```
var MYAPP = {};
MYAPP.dom = {};
MYAPP.dom.Text = function (url) {
    this.url = url;
    this.insert = function (where) {
        var txt = document.createTextNode(this.url);
        where.appendChild(txt);
    };
};
MYAPP.dom.Link = function (url) {
    this.url = url;
    this.insert = function (where) {
        var link = document.createElement('a');
        link.href = this.url;
        link.appendChild(document.createTextNode(this.url));
        where.appendChild(link);
    };
};
MYAPP.dom.Image = function (url) {
    this.url = url;
    this.insert = function (where) {
        var im = document.createElement('img');
        im.src = this.url;
        where.appendChild(im);
    };
};
MYAPP.dom.factory = function (type, url) {    //this is the factory method
    return new MYAPP.dom[type](url);          //you should add some validation
};
var image = MYAPP.dom.factory("Image", url);
```

c) Decorator: Instead of using inheritance where you extend in a linear way (parent-child-grandchild), you can have one base object and a pool of different decorator objects that provide extra functionality.

```
var tree = {};
tree.decorate = function () {
    alert('Make sure the tree won\'t fall');
};
tree.getDecorator = function (deco) {
    tree[deco].prototype = this;
    return new tree[deco];
};
tree.RedBalls = function () {
    this.decorate = function () {
        this.RedBalls.prototype.decorate();      //call the prototype first
        alert('Put on some red balls');          //then it's unique function.
    };
};
tree = tree.getDecorator('BlueBalls');
tree = tree.getDecorator('Angel');
tree = tree.getDecorator('RedBalls');
tree.decorate();
```

d) Observer: an example implementation of the push model

i. The observer object will have the following properties and methods:

1. An array of subscribers that are **just callback functions**
2. addSubscriber() and removeSubscriber() methods that add to, and remove from, the subscribers collection
3. A publish() method that takes data and calls all subscribers
4. A make() method that **takes any object and turns it into a publisher** by adding all of the methods mentioned previously to it

ii. Example:

```
var observer = {
    addSubscriber: function (callback) {
        if (typeof callback === "function") {
            this.subscribers[this.subscribers.length] = callback;
        }
    },
    removeSubscriber: function (callback) {
        for (var i = 0; i < this.subscribers.length; i++) {
            if (this.subscribers[i] === callback) {
                delete this.subscribers[i];
            }
```

```
            }
        },
        publish: function (what) {
            for (var i = 0; i < this.subscribers.length; i++) {
                if (typeof this.subscribers[i] === 'function') {
                    this.subscribers[i](what);
                }
            }
        },
        make: function (o) {                    // turns an object into a publisher
            for (var i in this) {
                if (this.hasOwnProperty(i)) {
                    o[i] = this[i];
                }
            }
            o.subscribers = [];
        }
};
var blogger = {
        writeBlogPost: function() {
            var content = 'Today is ' + new Date();
            this.publish(content);
        }
};
blogger.addSubscriber(jack.read);
blogger.addSubscriber(jill.gossip);
blogger.writeBlogPost();
```