```
/*!
 * Knockout JavaScript library v3.4.2
 * (c) The Knockout.js team - http://knockoutjs.com/
 * License: MIT (http://www.opensource.org/licenses/mit-license.php)
 */

(function(){
var DEBUG=true;
(function(undefined){
    // (0, eval)('this') is a robust way of getting a reference to the global object
    // For details, see http://stackoverflow.com/questions/14119988/return-this-0-evalthis/14120023#14120023
    var window = this || (0, eval)('this'),
        document = window['document'],
        navigator = window['navigator'],
        jQueryInstance = window["jQuery"],
        JSON = window["JSON"];
(function(factory) {
    // Support three module loading scenarios
    if (typeof define === 'function' && define['amd']) {
        // [1] AMD anonymous module
        define(['exports', 'require'], factory);
    } else if (typeof exports === 'object' && typeof module === 'object') {
        // [2] CommonJS/Node.js
        factory(module['exports'] || exports);  // module.exports is for Node.js
    } else {
        // [3] No module loader (plain <script> tag) - put directly in global namespace
        factory(window['ko'] = {});
    }
}(function(koExports, amdRequire){
// Internally, all KO objects are attached to koExports (even the non-exported ones whose names will be minified by the closure
compiler).
// In the future, the following "ko" variable may be made distinct from "koExports" so that private objects are not externally
reachable.
var ko = typeof koExports !== 'undefined' ? koExports : {};
// Google Closure Compiler helpers (used only to make the minified file smaller)
ko.exportSymbol = function(koPath, object) {
    var tokens = koPath.split(".");

    // In the future, "ko" may become distinct from "koExports" (so that non-exported objects are not reachable)
    // At that point, "target" would be set to: (typeof koExports !== "undefined" ? koExports : ko)
    var target = ko;

    for (var i = 0; i < tokens.length - 1; i++)
        target = target[tokens[i]];
    target[tokens[tokens.length - 1]] = object;
};
ko.exportProperty = function(owner, publicName, object) {
    owner[publicName] = object;
};
ko.version = "3.4.2";

ko.exportSymbol('version', ko.version);
// For any options that may affect various areas of Knockout and aren't directly associated with data binding.
ko.options = {
    'deferUpdates': false,
    'useOnlyNativeEvents': false
};

//ko.exportSymbol('options', ko.options);   // 'options' isn't minified
ko.utils = (function () {
    function objectForEach(obj, action) {
        for (var prop in obj) {
            if (obj.hasOwnProperty(prop)) {
                action(prop, obj[prop]);
            }
        }
    }

    function extend(target, source) {
        if (source) {
            for(var prop in source) {
                if(source.hasOwnProperty(prop)) {
                    target[prop] = source[prop];
                }
            }
        }
        return target;
    }

    function setPrototypeOf(obj, proto) {
        obj.__proto__ = proto;
        return obj;
    }

    var canSetPrototype = ({ __proto__: [] } instanceof Array);
    var canUseSymbols = !DEBUG && typeof Symbol === 'function';

    // Represent the known event types in a compact way, then at runtime transform it into a hash with event name as key (for fast
lookup)
    var knownEvents = {}, knownEventTypesByEventName = {};
    var keyEventTypeName = (navigator && /Firefox\/2/i.test(navigator.userAgent)) ? 'KeyboardEvent' : 'UIEvents';
    knownEvents[keyEventTypeName] = ['keyup', 'keydown', 'keypress'];
    knownEvents['MouseEvents'] = ['click', 'dblclick', 'mousedown', 'mouseup', 'mousemove', 'mouseover', 'mouseout', 'mouseenter',
'mouseleave'];
    objectForEach(knownEvents, function(eventType, knownEventsForType) {
        if (knownEventsForType.length) {
            for (var i = 0, j = knownEventsForType.length; i < j; i++)
                knownEventTypesByEventName[knownEventsForType[i]] = eventType;
        }
    });
    var eventsThatMustBeRegisteredUsingAttachEvent = { 'propertychange': true }; // Workaround for an IE9 issue -
https://github.com/SteveSanderson/knockout/issues/406
```

```
    // Detect IE versions for bug workarounds (uses IE conditionals, not UA string, for robustness)
    // Note that, since IE 10 does not support conditional comments, the following logic only detects IE < 10.
    // Currently this is by design, since IE 10+ behaves correctly when treated as a standard browser.
    // If there is a future need to detect specific versions of IE10+, we will amend this.
    var ieVersion = document && (function() {
        var version = 3, div = document.createElement('div'), iElems = div.getElementsByTagName('i');

        // Keep constructing conditional HTML blocks until we hit one that resolves to an empty fragment
        while (
            div.innerHTML = '<!--[if gt IE ' + (++version) + ']><i></i><![endif]-->',
            iElems[0]
        ) {}
        return version > 4 ? version : undefined;
    }());
    var isIe6 = ieVersion === 6,
        isIe7 = ieVersion === 7;

    function isClickOnCheckableElement(element, eventType) {
        if ((ko.utils.tagNameLower(element) !== "input") || !element.type) return false;
        if (eventType.toLowerCase() != "click") return false;
        var inputType = element.type;
        return (inputType == "checkbox") || (inputType == "radio");
    }

    // For details on the pattern for changing node classes
    // see: https://github.com/knockout/knockout/issues/1597
    var cssClassNameRegex = /\S+/g;

    function toggleDomNodeCssClass(node, classNames, shouldHaveClass) {
        var addOrRemoveFn;
        if (classNames) {
            if (typeof node.classList === 'object') {
                addOrRemoveFn = node.classList[shouldHaveClass ? 'add' : 'remove'];
                ko.utils.arrayForEach(classNames.match(cssClassNameRegex), function(className) {
                    addOrRemoveFn.call(node.classList, className);
                });
            } else if (typeof node.className['baseVal'] === 'string') {
                // SVG tag .classNames is an SVGAnimatedString instance
                toggleObjectClassPropertyString(node.className, 'baseVal', classNames, shouldHaveClass);
            } else {
                // node.className ought to be a string.
                toggleObjectClassPropertyString(node, 'className', classNames, shouldHaveClass);
            }
        }
    }

    function toggleObjectClassPropertyString(obj, prop, classNames, shouldHaveClass) {
        // obj/prop is either a node/'className' or a SVGAnimatedString/'baseVal'.
        var currentClassNames = obj[prop].match(cssClassNameRegex) || [];
        ko.utils.arrayForEach(classNames.match(cssClassNameRegex), function(className) {
            ko.utils.addOrRemoveItem(currentClassNames, className, shouldHaveClass);
        });
        obj[prop] = currentClassNames.join(" ");
    }

    return {
        fieldsIncludedWithJsonPost: ['authenticity_token', /^__RequestVerificationToken(_.*)?$/],

        arrayForEach: function (array, action) {
            for (var i = 0, j = array.length; i < j; i++)
                action(array[i], i);
        },

        arrayIndexOf: function (array, item) {
            if (typeof Array.prototype.indexOf == "function")
                return Array.prototype.indexOf.call(array, item);
            for (var i = 0, j = array.length; i < j; i++)
                if (array[i] === item)
                    return i;
            return -1;
        },

        arrayFirst: function (array, predicate, predicateOwner) {
            for (var i = 0, j = array.length; i < j; i++)
                if (predicate.call(predicateOwner, array[i], i))
                    return array[i];
            return null;
        },

        arrayRemoveItem: function (array, itemToRemove) {
            var index = ko.utils.arrayIndexOf(array, itemToRemove);
            if (index > 0) {
                array.splice(index, 1);
            }
            else if (index === 0) {
                array.shift();
            }
        },

        arrayGetDistinctValues: function (array) {
            array = array || [];
            var result = [];
            for (var i = 0, j = array.length; i < j; i++) {
                if (ko.utils.arrayIndexOf(result, array[i]) < 0)
                    result.push(array[i]);
            }
            return result;
        },

        arrayMap: function (array, mapping) {
            array = array || [];
            var result = [];
            for (var i = 0, j = array.length; i < j; i++)
```

```
            result.push(mapping(array[i], i));
        return result;
    },

    arrayFilter: function (array, predicate) {
        array = array || [];
        var result = [];
        for (var i = 0, j = array.length; i < j; i++)
            if (predicate(array[i], i))
                result.push(array[i]);
        return result;
    },

    arrayPushAll: function (array, valuesToPush) {
        if (valuesToPush instanceof Array)
            array.push.apply(array, valuesToPush);
        else
            for (var i = 0, j = valuesToPush.length; i < j; i++)
                array.push(valuesToPush[i]);
        return array;
    },

    addOrRemoveItem: function(array, value, included) {
        var existingEntryIndex = ko.utils.arrayIndexOf(ko.utils.peekObservable(array), value);
        if (existingEntryIndex < 0) {
            if (included)
                array.push(value);
        } else {
            if (!included)
                array.splice(existingEntryIndex, 1);
        }
    },

    canSetPrototype: canSetPrototype,

    extend: extend,

    setPrototypeOf: setPrototypeOf,

    setPrototypeOfOrExtend: canSetPrototype ? setPrototypeOf : extend,

    objectForEach: objectForEach,

    objectMap: function(source, mapping) {
        if (!source)
            return source;
        var target = {};
        for (var prop in source) {
            if (source.hasOwnProperty(prop)) {
                target[prop] = mapping(source[prop], prop, source);
            }
        }
        return target;
    },

    emptyDomNode: function (domNode) {
        while (domNode.firstChild) {
            ko.removeNode(domNode.firstChild);
        }
    },

    moveCleanedNodesToContainerElement: function(nodes) {
        // Ensure it's a real array, as we're about to reparent the nodes and
        // we don't want the underlying collection to change while we're doing that.
        var nodesArray = ko.utils.makeArray(nodes);
        var templateDocument = (nodesArray[0] && nodesArray[0].ownerDocument) || document;

        var container = templateDocument.createElement('div');
        for (var i = 0, j = nodesArray.length; i < j; i++) {
            container.appendChild(ko.cleanNode(nodesArray[i]));
        }
        return container;
    },

    cloneNodes: function (nodesArray, shouldCleanNodes) {
        for (var i = 0, j = nodesArray.length, newNodesArray = []; i < j; i++) {
            var clonedNode = nodesArray[i].cloneNode(true);
            newNodesArray.push(shouldCleanNodes ? ko.cleanNode(clonedNode) : clonedNode);
        }
        return newNodesArray;
    },

    setDomNodeChildren: function (domNode, childNodes) {
        ko.utils.emptyDomNode(domNode);
        if (childNodes) {
            for (var i = 0, j = childNodes.length; i < j; i++)
                domNode.appendChild(childNodes[i]);
        }
    },

    replaceDomNodes: function (nodeToReplaceOrNodeArray, newNodesArray) {
        var nodesToReplaceArray = nodeToReplaceOrNodeArray.nodeType ? [nodeToReplaceOrNodeArray] : nodeToReplaceOrNodeArray;
        if (nodesToReplaceArray.length > 0) {
            var insertionPoint = nodesToReplaceArray[0];
            var parent = insertionPoint.parentNode;
            for (var i = 0, j = newNodesArray.length; i < j; i++)
                parent.insertBefore(newNodesArray[i], insertionPoint);
            for (var i = 0, j = nodesToReplaceArray.length; i < j; i++) {
                ko.removeNode(nodesToReplaceArray[i]);
            }
        }
    },
```

```
        fixUpContinuousNodeArray: function(continuousNodeArray, parentNode) {
            // Before acting on a set of nodes that were previously outputted by a template function, we have to reconcile
            // them against what is in the DOM right now. It may be that some of the nodes have already been removed, or that
            // new nodes might have been inserted in the middle, for example by a binding. Also, there may previously have been
            // leading comment nodes (created by rewritten string-based templates) that have since been removed during binding.
            // So, this function translates the old "map" output array into its best guess of the set of current DOM nodes.
            //
            // Rules:
            //   [A] Any leading nodes that have been removed should be ignored
            //       These most likely correspond to memoization nodes that were already removed during binding
            //       See https://github.com/knockout/knockout/pull/440
            //   [B] Any trailing nodes that have been remove should be ignored
            //       This prevents the code here from adding unrelated nodes to the array while processing rule [C]
            //       See https://github.com/knockout/knockout/pull/1903
            //   [C] We want to output a continuous series of nodes. So, ignore any nodes that have already been removed,
            //       and include any nodes that have been inserted among the previous collection

            if (continuousNodeArray.length) {
                // The parent node can be a virtual element; so get the real parent node
                parentNode = (parentNode.nodeType === 8 && parentNode.parentNode) || parentNode;

                // Rule [A]
                while (continuousNodeArray.length && continuousNodeArray[0].parentNode !== parentNode)
                    continuousNodeArray.splice(0, 1);

                // Rule [B]
                while (continuousNodeArray.length > 1 && continuousNodeArray[continuousNodeArray.length - 1].parentNode !==
parentNode)
                    continuousNodeArray.length--;

                // Rule [C]
                if (continuousNodeArray.length > 1) {
                    var current = continuousNodeArray[0], last = continuousNodeArray[continuousNodeArray.length - 1];
                    // Replace with the actual new continuous node set
                    continuousNodeArray.length = 0;
                    while (current !== last) {
                        continuousNodeArray.push(current);
                        current = current.nextSibling;
                    }
                    continuousNodeArray.push(last);
                }
            }
            return continuousNodeArray;
        },

        setOptionNodeSelectionState: function (optionNode, isSelected) {
            // IE6 sometimes throws "unknown error" if you try to write to .selected directly, whereas Firefox struggles with
setAttribute. Pick one based on browser.
            if (ieVersion < 7)
                optionNode.setAttribute("selected", isSelected);
            else
                optionNode.selected = isSelected;
        },

        stringTrim: function (string) {
            return string === null || string === undefined ? '' :
                string.trim ?
                    string.trim() :
                    string.toString().replace(/^[\s\xa0]+|[\s\xa0]+$/g, '');
        },

        stringStartsWith: function (string, startsWith) {
            string = string || "";
            if (startsWith.length > string.length)
                return false;
            return string.substring(0, startsWith.length) === startsWith;
        },

        domNodeIsContainedBy: function (node, containedByNode) {
            if (node === containedByNode)
                return true;
            if (node.nodeType === 11)
                return false; // Fixes issue #1162 - can't use node.contains for document fragments on IE8
            if (containedByNode.contains)
                return containedByNode.contains(node.nodeType === 3 ? node.parentNode : node);
            if (containedByNode.compareDocumentPosition)
                return (containedByNode.compareDocumentPosition(node) & 16) == 16;
            while (node && node != containedByNode) {
                node = node.parentNode;
            }
            return !!node;
        },

        domNodeIsAttachedToDocument: function (node) {
            return ko.utils.domNodeIsContainedBy(node, node.ownerDocument.documentElement);
        },

        anyDomNodeIsAttachedToDocument: function(nodes) {
            return !!ko.utils.arrayFirst(nodes, ko.utils.domNodeIsAttachedToDocument);
        },

        tagNameLower: function(element) {
            // For HTML elements, tagName will always be upper case; for XHTML elements, it'll be lower case.
            // Possible future optimization: If we know it's an element from an XHTML document (not HTML),
            // we don't need to do the .toLowerCase() as it will always be lower case anyway.
            return element && element.tagName && element.tagName.toLowerCase();
        },

        catchFunctionErrors: function (delegate) {
            return ko['onError'] ? function () {
                try {
                    return delegate.apply(this, arguments);
                } catch (e) {
```

```
                ko['onError'] && ko['onError'](e);
                throw e;
            }
        } : delegate;
    },

    setTimeout: function (handler, timeout) {
        return setTimeout(ko.utils.catchFunctionErrors(handler), timeout);
    },

    deferError: function (error) {
        setTimeout(function () {
            ko['onError'] && ko['onError'](error);
            throw error;
        }, 0);
    },

    registerEventHandler: function (element, eventType, handler) {
        var wrappedHandler = ko.utils.catchFunctionErrors(handler);

        var mustUseAttachEvent = ieVersion && eventsThatMustBeRegisteredUsingAttachEvent[eventType];
        if (!ko.options['useOnlyNativeEvents'] && !mustUseAttachEvent && jQueryInstance) {
            jQueryInstance(element)['bind'](eventType, wrappedHandler);
        } else if (!mustUseAttachEvent && typeof element.addEventListener == "function")
            element.addEventListener(eventType, wrappedHandler, false);
        else if (typeof element.attachEvent != "undefined") {
            var attachEventHandler = function (event) { wrappedHandler.call(element, event); },
                attachEventName = "on" + eventType;
            element.attachEvent(attachEventName, attachEventHandler);

            // IE does not dispose attachEvent handlers automatically (unlike with addEventListener)
            // so to avoid leaks, we have to remove them manually. See bug #856
            ko.utils.domNodeDisposal.addDisposeCallback(element, function() {
                element.detachEvent(attachEventName, attachEventHandler);
            });
        } else
            throw new Error("Browser doesn't support addEventListener or attachEvent");
    },

    triggerEvent: function (element, eventType) {
        if (!(element && element.nodeType))
            throw new Error("element must be a DOM node when calling triggerEvent");

        // For click events on checkboxes and radio buttons, jQuery toggles the element checked state *after* the
        // event handler runs instead of *before*. (This was fixed in 1.9 for checkboxes but not for radio buttons.)
        // IE doesn't change the checked state when you trigger the click event using "fireEvent".
        // In both cases, we'll use the click method instead.
        var useClickWorkaround = isClickOnCheckableElement(element, eventType);

        if (!ko.options['useOnlyNativeEvents'] && jQueryInstance && !useClickWorkaround) {
            jQueryInstance(element)['trigger'](eventType);
        } else if (typeof document.createEvent == "function") {
            if (typeof element.dispatchEvent == "function") {
                var eventCategory = knownEventTypesByEventName[eventType] || "HTMLEvents";
                var event = document.createEvent(eventCategory);
                event.initEvent(eventType, true, true, window, 0, 0, 0, 0, 0, false, false, false, false, 0, element);
                element.dispatchEvent(event);
            }
            else
                throw new Error("The supplied element doesn't support dispatchEvent");
        } else if (useClickWorkaround && element.click) {
            element.click();
        } else if (typeof element.fireEvent != "undefined") {
            element.fireEvent("on" + eventType);
        } else {
            throw new Error("Browser doesn't support triggering events");
        }
    },

    unwrapObservable: function (value) {
        return ko.isObservable(value) ? value() : value;
    },

    peekObservable: function (value) {
        return ko.isObservable(value) ? value.peek() : value;
    },

    toggleDomNodeCssClass: toggleDomNodeCssClass,

    setTextContent: function(element, textContent) {
        var value = ko.utils.unwrapObservable(textContent);
        if ((value === null) || (value === undefined))
            value = "";

        // We need there to be exactly one child: a text node.
        // If there are no children, more than one, or if it's not a text node,
        // we'll clear everything and create a single text node.
        var innerTextNode = ko.virtualElements.firstChild(element);
        if (!innerTextNode || innerTextNode.nodeType != 3 || ko.virtualElements.nextSibling(innerTextNode)) {
            ko.virtualElements.setDomNodeChildren(element, [element.ownerDocument.createTextNode(value)]);
        } else {
            innerTextNode.data = value;
        }

        ko.utils.forceRefresh(element);
    },

    setElementName: function(element, name) {
        element.name = name;

        // Workaround IE 6/7 issue
        // - https://github.com/SteveSanderson/knockout/issues/197
        // - http://www.matts411.com/post/setting_the_name_attribute_in_ie_dom/
```

```
            if (ieVersion <= 7) {
                try {
                    element.mergeAttributes(document.createElement("<input name='" + element.name + "'/>"), false);
                }
                catch(e) {} // For IE9 with doc mode "IE9 Standards" and browser mode "IE9 Compatibility View"
            }
        },

        forceRefresh: function(node) {
            // Workaround for an IE9 rendering bug - https://github.com/SteveSanderson/knockout/issues/209
            if (ieVersion >= 9) {
                // For text nodes and comment nodes (most likely virtual elements), we will have to refresh the container
                var elem = node.nodeType == 1 ? node : node.parentNode;
                if (elem.style)
                    elem.style.zoom = elem.style.zoom;
            }
        },

        ensureSelectElementIsRenderedCorrectly: function(selectElement) {
            // Workaround for IE9 rendering bug - it doesn't reliably display all the text in dynamically-added select boxes unless
you force it to re-render by updating the width.
            // (See https://github.com/SteveSanderson/knockout/issues/312, http://stackoverflow.com/questions/5908494/select-only-
shows-first-char-of-selected-option)
            // Also fixes IE7 and IE8 bug that causes selects to be zero width if enclosed by 'if' or 'with'. (See issue #839)
            if (ieVersion) {
                var originalWidth = selectElement.style.width;
                selectElement.style.width = 0;
                selectElement.style.width = originalWidth;
            }
        },

        range: function (min, max) {
            min = ko.utils.unwrapObservable(min);
            max = ko.utils.unwrapObservable(max);
            var result = [];
            for (var i = min; i <= max; i++)
                result.push(i);
            return result;
        },

        makeArray: function(arrayLikeObject) {
            var result = [];
            for (var i = 0, j = arrayLikeObject.length; i < j; i++) {
                result.push(arrayLikeObject[i]);
            };
            return result;
        },

        createSymbolOrString: function(identifier) {
            return canUseSymbols ? Symbol(identifier) : identifier;
        },

        isIe6 : isIe6,
        isIe7 : isIe7,
        ieVersion : ieVersion,

        getFormFields: function(form, fieldName) {
            var fields =
ko.utils.makeArray(form.getElementsByTagName("input")).concat(ko.utils.makeArray(form.getElementsByTagName("textarea")));
            var isMatchingField = (typeof fieldName == 'string')
                ? function(field) { return field.name === fieldName }
                : function(field) { return fieldName.test(field.name) }; // Treat fieldName as regex or object containing predicate
            var matches = [];
            for (var i = fields.length - 1; i >= 0; i--) {
                if (isMatchingField(fields[i]))
                    matches.push(fields[i]);
            };
            return matches;
        },

        parseJson: function (jsonString) {
            if (typeof jsonString == "string") {
                jsonString = ko.utils.stringTrim(jsonString);
                if (jsonString) {
                    if (JSON && JSON.parse) // Use native parsing where available
                        return JSON.parse(jsonString);
                    return (new Function("return " + jsonString))(); // Fallback on less safe parsing for older browsers
                }
            }
            return null;
        },

        stringifyJson: function (data, replacer, space) {   // replacer and space are optional
            if (!JSON || !JSON.stringify)
                throw new Error("Cannot find JSON.stringify(). Some browsers (e.g., IE < 8) don't support it natively, but you can
overcome this by adding a script reference to json2.js, downloadable from http://www.json.org/json2.js");
            return JSON.stringify(ko.utils.unwrapObservable(data), replacer, space);
        },

        postJson: function (urlOrForm, data, options) {
            options = options || {};
            var params = options['params'] || {};
            var includeFields = options['includeFields'] || this.fieldsIncludedWithJsonPost;
            var url = urlOrForm;

            // If we were given a form, use its 'action' URL and pick out any requested field values
            if((typeof urlOrForm == 'object') && (ko.utils.tagNameLower(urlOrForm) === "form")) {
                var originalForm = urlOrForm;
                url = originalForm.action;
                for (var i = includeFields.length - 1; i >= 0; i--) {
                    var fields = ko.utils.getFormFields(originalForm, includeFields[i]);
                    for (var j = fields.length - 1; j >= 0; j--)
                        params[fields[j].name] = fields[j].value;
```

```
                }
            }

            data = ko.utils.unwrapObservable(data);
            var form = document.createElement("form");
            form.style.display = "none";
            form.action = url;
            form.method = "post";
            for (var key in data) {
                // Since 'data' this is a model object, we include all properties including those inherited from its prototype
                var input = document.createElement("input");
                input.type = "hidden";
                input.name = key;
                input.value = ko.utils.stringifyJson(ko.utils.unwrapObservable(data[key]));
                form.appendChild(input);
            }
            objectForEach(params, function(key, value) {
                var input = document.createElement("input");
                input.type = "hidden";
                input.name = key;
                input.value = value;
                form.appendChild(input);
            });
            document.body.appendChild(form);
            options['submitter'] ? options['submitter'](form) : form.submit();
            setTimeout(function () { form.parentNode.removeChild(form); }, 0);
        }
    }
}());

ko.exportSymbol('utils', ko.utils);
ko.exportSymbol('utils.arrayForEach', ko.utils.arrayForEach);
ko.exportSymbol('utils.arrayFirst', ko.utils.arrayFirst);
ko.exportSymbol('utils.arrayFilter', ko.utils.arrayFilter);
ko.exportSymbol('utils.arrayGetDistinctValues', ko.utils.arrayGetDistinctValues);
ko.exportSymbol('utils.arrayIndexOf', ko.utils.arrayIndexOf);
ko.exportSymbol('utils.arrayMap', ko.utils.arrayMap);
ko.exportSymbol('utils.arrayPushAll', ko.utils.arrayPushAll);
ko.exportSymbol('utils.arrayRemoveItem', ko.utils.arrayRemoveItem);
ko.exportSymbol('utils.extend', ko.utils.extend);
ko.exportSymbol('utils.fieldsIncludedWithJsonPost', ko.utils.fieldsIncludedWithJsonPost);
ko.exportSymbol('utils.getFormFields', ko.utils.getFormFields);
ko.exportSymbol('utils.peekObservable', ko.utils.peekObservable);
ko.exportSymbol('utils.postJson', ko.utils.postJson);
ko.exportSymbol('utils.parseJson', ko.utils.parseJson);
ko.exportSymbol('utils.registerEventHandler', ko.utils.registerEventHandler);
ko.exportSymbol('utils.stringifyJson', ko.utils.stringifyJson);
ko.exportSymbol('utils.range', ko.utils.range);
ko.exportSymbol('utils.toggleDomNodeCssClass', ko.utils.toggleDomNodeCssClass);
ko.exportSymbol('utils.triggerEvent', ko.utils.triggerEvent);
ko.exportSymbol('utils.unwrapObservable', ko.utils.unwrapObservable);
ko.exportSymbol('utils.objectForEach', ko.utils.objectForEach);
ko.exportSymbol('utils.addOrRemoveItem', ko.utils.addOrRemoveItem);
ko.exportSymbol('utils.setTextContent', ko.utils.setTextContent);
ko.exportSymbol('unwrap', ko.utils.unwrapObservable); // Convenient shorthand, because this is used so commonly

if (!Function.prototype['bind']) {
    // Function.prototype.bind is a standard part of ECMAScript 5th Edition (December 2009, http://www.ecma-
international.org/publications/files/ECMA-ST/ECMA-262.pdf)
    // In case the browser doesn't implement it natively, provide a JavaScript implementation. This implementation is based on the one
in prototype.js
    Function.prototype['bind'] = function (object) {
        var originalFunction = this;
        if (arguments.length === 1) {
            return function () {
                return originalFunction.apply(object, arguments);
            };
        } else {
            var partialArgs = Array.prototype.slice.call(arguments, 1);
            return function () {
                var args = partialArgs.slice(0);
                args.push.apply(args, arguments);
                return originalFunction.apply(object, args);
            };
        }
    };
}

ko.utils.domData = new (function () {
    var uniqueId = 0;
    var dataStoreKeyExpandoPropertyName = "__ko__" + (new Date).getTime();
    var dataStore = {};

    function getAll(node, createIfNotFound) {
        var dataStoreKey = node[dataStoreKeyExpandoPropertyName];
        var hasExistingDataStore = dataStoreKey && (dataStoreKey !== "null") && dataStore[dataStoreKey];
        if (!hasExistingDataStore) {
            if (!createIfNotFound)
                return undefined;
            dataStoreKey = node[dataStoreKeyExpandoPropertyName] = "ko" + uniqueId++;
            dataStore[dataStoreKey] = {};
        }
        return dataStore[dataStoreKey];
    }

    return {
        get: function (node, key) {
            var allDataForNode = getAll(node, false);
            return allDataForNode === undefined ? undefined : allDataForNode[key];
        },
        set: function (node, key, value) {
            if (value === undefined) {
                // Make sure we don't actually create a new domData key if we are actually deleting a value
```

```
                if (getAll(node, false) === undefined)
                    return;
            }
            var allDataForNode = getAll(node, true);
            allDataForNode[key] = value;
        },
        clear: function (node) {
            var dataStoreKey = node[dataStoreKeyExpandoPropertyName];
            if (dataStoreKey) {
                delete dataStore[dataStoreKey];
                node[dataStoreKeyExpandoPropertyName] = null;
                return true; // Exposing "did clean" flag purely so specs can infer whether things have been cleaned up as intended
            }
            return false;
        },

        nextKey: function () {
            return (uniqueId++) + dataStoreKeyExpandoPropertyName;
        }
    };
})();

ko.exportSymbol('utils.domData', ko.utils.domData);
ko.exportSymbol('utils.domData.clear', ko.utils.domData.clear); // Exporting only so specs can clear up after themselves fully

ko.utils.domNodeDisposal = new (function () {
    var domDataKey = ko.utils.domData.nextKey();
    var cleanableNodeTypes = { 1: true, 8: true, 9: true };       // Element, Comment, Document
    var cleanableNodeTypesWithDescendants = { 1: true, 9: true }; // Element, Document

    function getDisposeCallbacksCollection(node, createIfNotFound) {
        var allDisposeCallbacks = ko.utils.domData.get(node, domDataKey);
        if ((allDisposeCallbacks === undefined) && createIfNotFound) {
            allDisposeCallbacks = [];
            ko.utils.domData.set(node, domDataKey, allDisposeCallbacks);
        }
        return allDisposeCallbacks;
    }
    function destroyCallbacksCollection(node) {
        ko.utils.domData.set(node, domDataKey, undefined);
    }

    function cleanSingleNode(node) {
        // Run all the dispose callbacks
        var callbacks = getDisposeCallbacksCollection(node, false);
        if (callbacks) {
            callbacks = callbacks.slice(0); // Clone, as the array may be modified during iteration (typically, callbacks will remove
themselves)
            for (var i = 0; i < callbacks.length; i++)
                callbacks[i](node);
        }

        // Erase the DOM data
        ko.utils.domData.clear(node);

        // Perform cleanup needed by external libraries (currently only jQuery, but can be extended)
        ko.utils.domNodeDisposal["cleanExternalData"](node);

        // Clear any immediate-child comment nodes, as these wouldn't have been found by
        // node.getElementsByTagName("*") in cleanNode() (comment nodes aren't elements)
        if (cleanableNodeTypesWithDescendants[node.nodeType])
            cleanImmediateCommentTypeChildren(node);
    }

    function cleanImmediateCommentTypeChildren(nodeWithChildren) {
        var child, nextChild = nodeWithChildren.firstChild;
        while (child = nextChild) {
            nextChild = child.nextSibling;
            if (child.nodeType === 8)
                cleanSingleNode(child);
        }
    }

    return {
        addDisposeCallback : function(node, callback) {
            if (typeof callback != "function")
                throw new Error("Callback must be a function");
            getDisposeCallbacksCollection(node, true).push(callback);
        },

        removeDisposeCallback : function(node, callback) {
            var callbacksCollection = getDisposeCallbacksCollection(node, false);
            if (callbacksCollection) {
                ko.utils.arrayRemoveItem(callbacksCollection, callback);
                if (callbacksCollection.length == 0)
                    destroyCallbacksCollection(node);
            }
        },

        cleanNode : function(node) {
            // First clean this node, where applicable
            if (cleanableNodeTypes[node.nodeType]) {
                cleanSingleNode(node);

                // ... then its descendants, where applicable
                if (cleanableNodeTypesWithDescendants[node.nodeType]) {
                    // Clone the descendants list in case it changes during iteration
                    var descendants = [];
                    ko.utils.arrayPushAll(descendants, node.getElementsByTagName("*"));
                    for (var i = 0, j = descendants.length; i < j; i++)
                        cleanSingleNode(descendants[i]);
                }
            }
```

```
            return node;
        },

        removeNode : function(node) {
            ko.cleanNode(node);
            if (node.parentNode)
                node.parentNode.removeChild(node);
        },

        "cleanExternalData" : function (node) {
            // Special support for jQuery here because it's so commonly used.
            // Many jQuery plugins (including jquery.tmpl) store data using jQuery's equivalent of domData
            // so notify it to tear down any resources associated with the node & descendants here.
            if (jQueryInstance && (typeof jQueryInstance['cleanData'] == "function"))
                jQueryInstance['cleanData']([node]);
        }
    };
})();
ko.cleanNode = ko.utils.domNodeDisposal.cleanNode; // Shorthand name for convenience
ko.removeNode = ko.utils.domNodeDisposal.removeNode; // Shorthand name for convenience
ko.exportSymbol('cleanNode', ko.cleanNode);
ko.exportSymbol('removeNode', ko.removeNode);
ko.exportSymbol('utils.domNodeDisposal', ko.utils.domNodeDisposal);
ko.exportSymbol('utils.domNodeDisposal.addDisposeCallback', ko.utils.domNodeDisposal.addDisposeCallback);
ko.exportSymbol('utils.domNodeDisposal.removeDisposeCallback', ko.utils.domNodeDisposal.removeDisposeCallback);
(function () {
    var none = [0, "", ""],
        table = [1, "<table>", "</table>"],
        tbody = [2, "<table><tbody>", "</tbody></table>"],
        tr = [3, "<table><tbody><tr>", "</tr></tbody></table>"],
        select = [1, "<select multiple='multiple'>", "</select>"],
        lookup = {
            'thead': table,
            'tbody': table,
            'tfoot': table,
            'tr': tbody,
            'td': tr,
            'th': tr,
            'option': select,
            'optgroup': select
        },

        // This is needed for old IE if you're *not* using either jQuery or innerShiv. Doesn't affect other cases.
        mayRequireCreateElementHack = ko.utils.ieVersion <= 8;

    function getWrap(tags) {
        var m = tags.match(/^<([a-z]+)[ >]/);
        return (m && lookup[m[1]]) || none;
    }

    function simpleHtmlParse(html, documentContext) {
        documentContext || (documentContext = document);
        var windowContext = documentContext['parentWindow'] || documentContext['defaultView'] || window;

        // Based on jQuery's "clean" function, but only accounting for table-related elements.
        // If you have referenced jQuery, this won't be used anyway - KO will use jQuery's "clean" function directly

        // Note that there's still an issue in IE < 9 whereby it will discard comment nodes that are the first child of
        // a descendant node. For example: "<div><!-- mycomment -->abc</div>" will get parsed as "<div>abc</div>"
        // This won't affect anyone who has referenced jQuery, and there's always the workaround of inserting a dummy node
        // (possibly a text node) in front of the comment. So, KO does not attempt to workaround this IE issue automatically at
present.

        // Trim whitespace, otherwise indexOf won't work as expected
        var tags = ko.utils.stringTrim(html).toLowerCase(), div = documentContext.createElement("div"),
            wrap = getWrap(tags),
            depth = wrap[0];

        // Go to html and back, then peel off extra wrappers
        // Note that we always prefix with some dummy text, because otherwise, IE<9 will strip out leading comment nodes in
descendants. Total madness.
        var markup = "ignored<div>" + wrap[1] + html + wrap[2] + "</div>";
        if (typeof windowContext['innerShiv'] == "function") {
            // Note that innerShiv is deprecated in favour of html5shiv. We should consider adding
            // support for html5shiv (except if no explicit support is needed, e.g., if html5shiv
            // somehow shims the native APIs so it just works anyway)
            div.appendChild(windowContext['innerShiv'](markup));
        } else {
            if (mayRequireCreateElementHack) {
                // The document.createElement('my-element') trick to enable custom elements in IE6-8
                // only works if we assign innerHTML on an element associated with that document.
                documentContext.appendChild(div);
            }

            div.innerHTML = markup;

            if (mayRequireCreateElementHack) {
                div.parentNode.removeChild(div);
            }
        }

        // Move to the right depth
        while (depth--)
            div = div.lastChild;

        return ko.utils.makeArray(div.lastChild.childNodes);
    }

    function jQueryHtmlParse(html, documentContext) {
        // jQuery's "parseHTML" function was introduced in jQuery 1.8.0 and is a documented public API.
        if (jQueryInstance['parseHTML']) {
            return jQueryInstance['parseHTML'](html, documentContext) || []; // Ensure we always return an array and never null
        } else {
```

```
            // For jQuery < 1.8.0, we fall back on the undocumented internal "clean" function.
            var elems = jQueryInstance['clean']([html], documentContext);

            // As of jQuery 1.7.1, jQuery parses the HTML by appending it to some dummy parent nodes held in an in-memory document fragment.
            // Unfortunately, it never clears the dummy parent nodes from the document fragment, so it leaks memory over time.
            // Fix this by finding the top-most dummy parent element, and detaching it from its owner fragment.
            if (elems && elems[0]) {
                // Find the top-most parent element that's a direct child of a document fragment
                var elem = elems[0];
                while (elem.parentNode && elem.parentNode.nodeType !== 11 /* i.e., DocumentFragment */)
                    elem = elem.parentNode;
                // ... then detach it
                if (elem.parentNode)
                    elem.parentNode.removeChild(elem);
            }

            return elems;
        }
    }

    ko.utils.parseHtmlFragment = function(html, documentContext) {
        return jQueryInstance ?
            jQueryHtmlParse(html, documentContext) :   // As below, benefit from jQuery's optimisations where possible
            simpleHtmlParse(html, documentContext);  // ... otherwise, this simple logic will do in most common cases.
    };

    ko.utils.setHtml = function(node, html) {
        ko.utils.emptyDomNode(node);

        // There's no legitimate reason to display a stringified observable without unwrapping it, so we'll unwrap it
        html = ko.utils.unwrapObservable(html);

        if ((html !== null) && (html !== undefined)) {
            if (typeof html != 'string')
                html = html.toString();

            // jQuery contains a lot of sophisticated code to parse arbitrary HTML fragments,
            // for example <tr> elements which are not normally allowed to exist on their own.
            // If you've referenced jQuery we'll use that rather than duplicating its code.
            if (jQueryInstance) {
                jQueryInstance(node)['html'](html);
            } else {
                // ... otherwise, use KO's own parsing logic.
                var parsedNodes = ko.utils.parseHtmlFragment(html, node.ownerDocument);
                for (var i = 0; i < parsedNodes.length; i++)
                    node.appendChild(parsedNodes[i]);
            }
        }
    };
})();

ko.exportSymbol('utils.parseHtmlFragment', ko.utils.parseHtmlFragment);
ko.exportSymbol('utils.setHtml', ko.utils.setHtml);

ko.memoization = (function () {
    var memos = {};

    function randomMax8HexChars() {
        return (((1 + Math.random()) * 0x100000000) | 0).toString(16).substring(1);
    }
    function generateRandomId() {
        return randomMax8HexChars() + randomMax8HexChars();
    }
    function findMemoNodes(rootNode, appendToArray) {
        if (!rootNode)
            return;
        if (rootNode.nodeType == 8) {
            var memoId = ko.memoization.parseMemoText(rootNode.nodeValue);
            if (memoId != null)
                appendToArray.push({ domNode: rootNode, memoId: memoId });
        } else if (rootNode.nodeType == 1) {
            for (var i = 0, childNodes = rootNode.childNodes, j = childNodes.length; i < j; i++)
                findMemoNodes(childNodes[i], appendToArray);
        }
    }

    return {
        memoize: function (callback) {
            if (typeof callback != "function")
                throw new Error("You can only pass a function to ko.memoization.memoize()");
            var memoId = generateRandomId();
            memos[memoId] = callback;
            return "<!--[ko_memo:" + memoId + "]-->";
        },

        unmemoize: function (memoId, callbackParams) {
            var callback = memos[memoId];
            if (callback === undefined)
                throw new Error("Couldn't find any memo with ID " + memoId + ". Perhaps it's already been unmemoized.");
            try {
                callback.apply(null, callbackParams || []);
                return true;
            }
            finally { delete memos[memoId]; }
        },

        unmemoizeDomNodeAndDescendants: function (domNode, extraCallbackParamsArray) {
            var memos = [];
            findMemoNodes(domNode, memos);
            for (var i = 0, j = memos.length; i < j; i++) {
                var node = memos[i].domNode;
                var combinedParams = [node];
```

```
                if (extraCallbackParamsArray)
                    ko.utils.arrayPushAll(combinedParams, extraCallbackParamsArray);
                ko.memoization.unmemoize(memos[i].memoId, combinedParams);
                node.nodeValue = ""; // Neuter this node so we don't try to unmemoize it again
                if (node.parentNode)
                    node.parentNode.removeChild(node); // If possible, erase it totally (not always possible - someone else might just
hold a reference to it then call unmemoizeDomNodeAndDescendants again)
            }
        },

        parseMemoText: function (memoText) {
            var match = memoText.match(/^\[ko_memo\:(.*?)\]$/);
            return match ? match[1] : null;
        }
    };
})();

ko.exportSymbol('memoization', ko.memoization);
ko.exportSymbol('memoization.memoize', ko.memoization.memoize);
ko.exportSymbol('memoization.unmemoize', ko.memoization.unmemoize);
ko.exportSymbol('memoization.parseMemoText', ko.memoization.parseMemoText);
ko.exportSymbol('memoization.unmemoizeDomNodeAndDescendants', ko.memoization.unmemoizeDomNodeAndDescendants);
ko.tasks = (function () {
    var scheduler,
        taskQueue = [],
        taskQueueLength = 0,
        nextHandle = 1,
        nextIndexToProcess = 0;

    if (window['MutationObserver']) {
        // Chrome 27+, Firefox 14+, IE 11+, Opera 15+, Safari 6.1+
        // From https://github.com/petkaantonov/bluebird * Copyright (c) 2014 Petka Antonov * License: MIT
        scheduler = (function (callback) {
            var div = document.createElement("div");
            new MutationObserver(callback).observe(div, {attributes: true});
            return function () { div.classList.toggle("foo"); };
        })(scheduledProcess);
    } else if (document && "onreadystatechange" in document.createElement("script")) {
        // IE 6-10
        // From https://github.com/YuzuJS/setImmediate * Copyright (c) 2012 Barnesandnoble.com, llc, Donavon West, and Domenic
Denicola * License: MIT
        scheduler = function (callback) {
            var script = document.createElement("script");
            script.onreadystatechange = function () {
                script.onreadystatechange = null;
                document.documentElement.removeChild(script);
                script = null;
                callback();
            };
            document.documentElement.appendChild(script);
        };
    } else {
        scheduler = function (callback) {
            setTimeout(callback, 0);
        };
    }

    function processTasks() {
        if (taskQueueLength) {
            // Each mark represents the end of a logical group of tasks and the number of these groups is
            // limited to prevent unchecked recursion.
            var mark = taskQueueLength, countMarks = 0;

            // nextIndexToProcess keeps track of where we are in the queue; processTasks can be called recursively without issue
            for (var task; nextIndexToProcess < taskQueueLength; ) {
                if (task = taskQueue[nextIndexToProcess++]) {
                    if (nextIndexToProcess > mark) {
                        if (++countMarks >= 5000) {
                            nextIndexToProcess = taskQueueLength;   // skip all tasks remaining in the queue since any of them could
be causing the recursion
                            ko.utils.deferError(Error("'Too much recursion' after processing " + countMarks + " task groups."));
                            break;
                        }
                        mark = taskQueueLength;
                    }
                    try {
                        task();
                    } catch (ex) {
                        ko.utils.deferError(ex);
                    }
                }
            }
        }
    }

    function scheduledProcess() {
        processTasks();

        // Reset the queue
        nextIndexToProcess = taskQueueLength = taskQueue.length = 0;
    }

    function scheduleTaskProcessing() {
        ko.tasks['scheduler'](scheduledProcess);
    }

    var tasks = {
        'scheduler': scheduler,     // Allow overriding the scheduler

        schedule: function (func) {
            if (!taskQueueLength) {
                scheduleTaskProcessing();
            }
```

```
            taskQueue[taskQueueLength++] = func;
            return nextHandle++;
        },

        cancel: function (handle) {
            var index = handle - (nextHandle - taskQueueLength);
            if (index >= nextIndexToProcess && index < taskQueueLength) {
                taskQueue[index] = null;
            }
        },

        // For testing only: reset the queue and return the previous queue length
        'resetForTesting': function () {
            var length = taskQueueLength - nextIndexToProcess;
            nextIndexToProcess = taskQueueLength = taskQueue.length = 0;
            return length;
        },

        runEarly: processTasks
    };

    return tasks;
})();

ko.exportSymbol('tasks', ko.tasks);
ko.exportSymbol('tasks.schedule', ko.tasks.schedule);
//ko.exportSymbol('tasks.cancel', ko.tasks.cancel);  "cancel" isn't minified
ko.exportSymbol('tasks.runEarly', ko.tasks.runEarly);
ko.extenders = {
    'throttle': function(target, timeout) {
        // Throttling means two things:

        // (1) For dependent observables, we throttle *evaluations* so that, no matter how fast its dependencies
        //     notify updates, the target doesn't re-evaluate (and hence doesn't notify) faster than a certain rate
        target['throttleEvaluation'] = timeout;

        // (2) For writable targets (observables, or writable dependent observables), we throttle *writes*
        //     so the target cannot change value synchronously or faster than a certain rate
        var writeTimeoutInstance = null;
        return ko.dependentObservable({
            'read': target,
            'write': function(value) {
                clearTimeout(writeTimeoutInstance);
                writeTimeoutInstance = ko.utils.setTimeout(function() {
                    target(value);
                }, timeout);
            }
        });
    },

    'rateLimit': function(target, options) {
        var timeout, method, limitFunction;

        if (typeof options == 'number') {
            timeout = options;
        } else {
            timeout = options['timeout'];
            method = options['method'];
        }

        // rateLimit supersedes deferred updates
        target._deferUpdates = false;

        limitFunction = method == 'notifyWhenChangesStop' ?  debounce : throttle;
        target.limit(function(callback) {
            return limitFunction(callback, timeout);
        });
    },

    'deferred': function(target, options) {
        if (options !== true) {
            throw new Error('The \'deferred\' extender only accepts the value \'true\', because it is not supported to turn deferral
off once enabled.')
        }

        if (!target._deferUpdates) {
            target._deferUpdates = true;
            target.limit(function (callback) {
                var handle,
                    ignoreUpdates = false;
                return function () {
                    if (!ignoreUpdates) {
                        ko.tasks.cancel(handle);
                        handle = ko.tasks.schedule(callback);

                        try {
                            ignoreUpdates = true;
                            target['notifySubscribers'](undefined, 'dirty');
                        } finally {
                            ignoreUpdates = false;
                        }
                    }
                };
            });
        }
    },

    'notify': function(target, notifyWhen) {
        target["equalityComparer"] = notifyWhen == "always" ?
            null :  // null equalityComparer means to always notify
            valuesArePrimitiveAndEqual;
    }
```

```
};

var primitiveTypes = { 'undefined':1, 'boolean':1, 'number':1, 'string':1 };
function valuesArePrimitiveAndEqual(a, b) {
    var oldValueIsPrimitive = (a === null) || (typeof(a) in primitiveTypes);
    return oldValueIsPrimitive ? (a === b) : false;
}

function throttle(callback, timeout) {
    var timeoutInstance;
    return function () {
        if (!timeoutInstance) {
            timeoutInstance = ko.utils.setTimeout(function () {
                timeoutInstance = undefined;
                callback();
            }, timeout);
        }
    };
}

function debounce(callback, timeout) {
    var timeoutInstance;
    return function () {
        clearTimeout(timeoutInstance);
        timeoutInstance = ko.utils.setTimeout(callback, timeout);
    };
}

function applyExtenders(requestedExtenders) {
    var target = this;
    if (requestedExtenders) {
        ko.utils.objectForEach(requestedExtenders, function(key, value) {
            var extenderHandler = ko.extenders[key];
            if (typeof extenderHandler == 'function') {
                target = extenderHandler(target, value) || target;
            }
        });
    }
    return target;
}

ko.exportSymbol('extenders', ko.extenders);

ko.subscription = function (target, callback, disposeCallback) {
    this._target = target;
    this.callback = callback;
    this.disposeCallback = disposeCallback;
    this.isDisposed = false;
    ko.exportProperty(this, 'dispose', this.dispose);
};
ko.subscription.prototype.dispose = function () {
    this.isDisposed = true;
    this.disposeCallback();
};

ko.subscribable = function () {
    ko.utils.setPrototypeOfOrExtend(this, ko_subscribable_fn);
    ko_subscribable_fn.init(this);
}

var defaultEvent = "change";

// Moved out of "limit" to avoid the extra closure
function limitNotifySubscribers(value, event) {
    if (!event || event === defaultEvent) {
        this._limitChange(value);
    } else if (event === 'beforeChange') {
        this._limitBeforeChange(value);
    } else {
        this._origNotifySubscribers(value, event);
    }
}

var ko_subscribable_fn = {
    init: function(instance) {
        instance._subscriptions = { "change": [] };
        instance._versionNumber = 1;
    },

    subscribe: function (callback, callbackTarget, event) {
        var self = this;

        event = event || defaultEvent;
        var boundCallback = callbackTarget ? callback.bind(callbackTarget) : callback;

        var subscription = new ko.subscription(self, boundCallback, function () {
            ko.utils.arrayRemoveItem(self._subscriptions[event], subscription);
            if (self.afterSubscriptionRemove)
                self.afterSubscriptionRemove(event);
        });

        if (self.beforeSubscriptionAdd)
            self.beforeSubscriptionAdd(event);

        if (!self._subscriptions[event])
            self._subscriptions[event] = [];
        self._subscriptions[event].push(subscription);

        return subscription;
    },

    "notifySubscribers": function (valueToNotify, event) {
        event = event || defaultEvent;
```

```
        if (event === defaultEvent) {
            this.updateVersion();
        }
        if (this.hasSubscriptionsForEvent(event)) {
            var subs = event === defaultEvent && this._changeSubscriptions || this._subscriptions[event].slice(0);
            try {
                ko.dependencyDetection.begin(); // Begin suppressing dependency detection (by setting the top frame to undefined)
                for (var i = 0, subscription; subscription = subs[i]; ++i) {
                    // In case a subscription was disposed during the arrayForEach cycle, check
                    // for isDisposed on each subscription before invoking its callback
                    if (!subscription.isDisposed)
                        subscription.callback(valueToNotify);
                }
            } finally {
                ko.dependencyDetection.end(); // End suppressing dependency detection
            }
        }
    },

    getVersion: function () {
        return this._versionNumber;
    },

    hasChanged: function (versionToCheck) {
        return this.getVersion() !== versionToCheck;
    },

    updateVersion: function () {
        ++this._versionNumber;
    },

    limit: function(limitFunction) {
        var self = this, selfIsObservable = ko.isObservable(self),
            ignoreBeforeChange, notifyNextChange, previousValue, pendingValue, beforeChange = 'beforeChange';

        if (!self._origNotifySubscribers) {
            self._origNotifySubscribers = self["notifySubscribers"];
            self["notifySubscribers"] = limitNotifySubscribers;
        }

        var finish = limitFunction(function() {
            self._notificationIsPending = false;

            // If an observable provided a reference to itself, access it to get the latest value.
            // This allows computed observables to delay calculating their value until needed.
            if (selfIsObservable && pendingValue === self) {
                pendingValue = self._evalIfChanged ? self._evalIfChanged() : self();
            }
            var shouldNotify = notifyNextChange || self.isDifferent(previousValue, pendingValue);

            notifyNextChange = ignoreBeforeChange = false;

            if (shouldNotify) {
                self._origNotifySubscribers(previousValue = pendingValue);
            }
        });

        self._limitChange = function(value) {
            self._changeSubscriptions = self._subscriptions[defaultEvent].slice(0);
            self._notificationIsPending = ignoreBeforeChange = true;
            pendingValue = value;
            finish();
        };
        self._limitBeforeChange = function(value) {
            if (!ignoreBeforeChange) {
                previousValue = value;
                self._origNotifySubscribers(value, beforeChange);
            }
        };
        self._notifyNextChangeIfValueIsDifferent = function() {
            if (self.isDifferent(previousValue, self.peek(true /*evaluate*/))) {
                notifyNextChange = true;
            }
        };
    },

    hasSubscriptionsForEvent: function(event) {
        return this._subscriptions[event] && this._subscriptions[event].length;
    },

    getSubscriptionsCount: function (event) {
        if (event) {
            return this._subscriptions[event] && this._subscriptions[event].length || 0;
        } else {
            var total = 0;
            ko.utils.objectForEach(this._subscriptions, function(eventName, subscriptions) {
                if (eventName !== 'dirty')
                    total += subscriptions.length;
            });
            return total;
        }
    },

    isDifferent: function(oldValue, newValue) {
        return !this['equalityComparer'] || !this['equalityComparer'](oldValue, newValue);
    },

    extend: applyExtenders
};

ko.exportProperty(ko_subscribable_fn, 'subscribe', ko_subscribable_fn.subscribe);
ko.exportProperty(ko_subscribable_fn, 'extend', ko_subscribable_fn.extend);
ko.exportProperty(ko_subscribable_fn, 'getSubscriptionsCount', ko_subscribable_fn.getSubscriptionsCount);
```

```
// For browsers that support proto assignment, we overwrite the prototype of each
// observable instance. Since observables are functions, we need Function.prototype
// to still be in the prototype chain.
if (ko.utils.canSetPrototype) {
    ko.utils.setPrototypeOf(ko_subscribable_fn, Function.prototype);
}

ko.subscribable['fn'] = ko_subscribable_fn;


ko.isSubscribable = function (instance) {
    return instance != null && typeof instance.subscribe == "function" && typeof instance["notifySubscribers"] == "function";
};

ko.exportSymbol('subscribable', ko.subscribable);
ko.exportSymbol('isSubscribable', ko.isSubscribable);

ko.computedContext = ko.dependencyDetection = (function () {
    var outerFrames = [],
        currentFrame,
        lastId = 0;

    // Return a unique ID that can be assigned to an observable for dependency tracking.
    // Theoretically, you could eventually overflow the number storage size, resulting
    // in duplicate IDs. But in JavaScript, the largest exact integral value is 2^53
    // or 9,007,199,254,740,992. If you created 1,000,000 IDs per second, it would
    // take over 285 years to reach that number.
    // Reference http://blog.vjeux.com/2010/javascript/javascript-max_int-number-limits.html
    function getId() {
        return ++lastId;
    }

    function begin(options) {
        outerFrames.push(currentFrame);
        currentFrame = options;
    }

    function end() {
        currentFrame = outerFrames.pop();
    }

    return {
        begin: begin,

        end: end,

        registerDependency: function (subscribable) {
            if (currentFrame) {
                if (!ko.isSubscribable(subscribable))
                    throw new Error("Only subscribable things can act as dependencies");
                currentFrame.callback.call(currentFrame.callbackTarget, subscribable, subscribable._id || (subscribable._id =
getId()));
            }
        },

        ignore: function (callback, callbackTarget, callbackArgs) {
            try {
                begin();
                return callback.apply(callbackTarget, callbackArgs || []);
            } finally {
                end();
            }
        },

        getDependenciesCount: function () {
            if (currentFrame)
                return currentFrame.computed.getDependenciesCount();
        },

        isInitial: function() {
            if (currentFrame)
                return currentFrame.isInitial;
        }
    };
})();

ko.exportSymbol('computedContext', ko.computedContext);
ko.exportSymbol('computedContext.getDependenciesCount', ko.computedContext.getDependenciesCount);
ko.exportSymbol('computedContext.isInitial', ko.computedContext.isInitial);

ko.exportSymbol('ignoreDependencies', ko.ignoreDependencies = ko.dependencyDetection.ignore);
var observableLatestValue = ko.utils.createSymbolOrString('_latestValue');

ko.observable = function (initialValue) {
    function observable() {
        if (arguments.length > 0) {
            // Write

            // Ignore writes if the value hasn't changed
            if (observable.isDifferent(observable[observableLatestValue], arguments[0])) {
                observable.valueWillMutate();
                observable[observableLatestValue] = arguments[0];
                observable.valueHasMutated();
            }
            return this; // Permits chained assignments
        }
        else {
            // Read
            ko.dependencyDetection.registerDependency(observable); // The caller only needs to be notified of changes if they did a
"read" operation
            return observable[observableLatestValue];
        }
```

```
    }

    observable[observableLatestValue] = initialValue;

    // Inherit from 'subscribable'
    if (!ko.utils.canSetPrototype) {
        // 'subscribable' won't be on the prototype chain unless we put it there directly
        ko.utils.extend(observable, ko.subscribable['fn']);
    }
    ko.subscribable['fn'].init(observable);

    // Inherit from 'observable'
    ko.utils.setPrototypeOfOrExtend(observable, observableFn);

    if (ko.options['deferUpdates']) {
        ko.extenders['deferred'](observable, true);
    }

    return observable;
}

// Define prototype for observables
var observableFn = {
    'equalityComparer': valuesArePrimitiveAndEqual,
    peek: function() { return this[observableLatestValue]; },
    valueHasMutated: function () { this['notifySubscribers'](this[observableLatestValue]); },
    valueWillMutate: function () { this['notifySubscribers'](this[observableLatestValue], 'beforeChange'); }
};

// Note that for browsers that don't support proto assignment, the
// inheritance chain is created manually in the ko.observable constructor
if (ko.utils.canSetPrototype) {
    ko.utils.setPrototypeOf(observableFn, ko.subscribable['fn']);
}

var protoProperty = ko.observable.protoProperty = '__ko_proto__';
observableFn[protoProperty] = ko.observable;

ko.hasPrototype = function(instance, prototype) {
    if ((instance === null) || (instance === undefined) || (instance[protoProperty] === undefined)) return false;
    if (instance[protoProperty] === prototype) return true;
    return ko.hasPrototype(instance[protoProperty], prototype); // Walk the prototype chain
};

ko.isObservable = function (instance) {
    return ko.hasPrototype(instance, ko.observable);
}
ko.isWriteableObservable = function (instance) {
    // Observable
    if ((typeof instance == 'function') && instance[protoProperty] === ko.observable)
        return true;
    // Writeable dependent observable
    if ((typeof instance == 'function') && (instance[protoProperty] === ko.dependentObservable) && (instance.hasWriteFunction))
        return true;
    // Anything else
    return false;
}

ko.exportSymbol('observable', ko.observable);
ko.exportSymbol('isObservable', ko.isObservable);
ko.exportSymbol('isWriteableObservable', ko.isWriteableObservable);
ko.exportSymbol('isWritableObservable', ko.isWriteableObservable);
ko.exportSymbol('observable.fn', observableFn);
ko.exportProperty(observableFn, 'peek', observableFn.peek);
ko.exportProperty(observableFn, 'valueHasMutated', observableFn.valueHasMutated);
ko.exportProperty(observableFn, 'valueWillMutate', observableFn.valueWillMutate);
ko.observableArray = function (initialValues) {
    initialValues = initialValues || [];

    if (typeof initialValues != 'object' || !('length' in initialValues))
        throw new Error("The argument passed when initializing an observable array must be an array, or null, or undefined.");

    var result = ko.observable(initialValues);
    ko.utils.setPrototypeOfOrExtend(result, ko.observableArray['fn']);
    return result.extend({'trackArrayChanges':true});
};

ko.observableArray['fn'] = {
    'remove': function (valueOrPredicate) {
        var underlyingArray = this.peek();
        var removedValues = [];
        var predicate = typeof valueOrPredicate == "function" && !ko.isObservable(valueOrPredicate) ? valueOrPredicate : function
(value) { return value === valueOrPredicate; };
        for (var i = 0; i < underlyingArray.length; i++) {
            var value = underlyingArray[i];
            if (predicate(value)) {
                if (removedValues.length === 0) {
                    this.valueWillMutate();
                }
                removedValues.push(value);
                underlyingArray.splice(i, 1);
                i--;
            }
        }
        if (removedValues.length) {
            this.valueHasMutated();
        }
        return removedValues;
    },

    'removeAll': function (arrayOfValues) {
        // If you passed zero args, we remove everything
        if (arrayOfValues === undefined) {
```

```
            var underlyingArray = this.peek();
            var allValues = underlyingArray.slice(0);
            this.valueWillMutate();
            underlyingArray.splice(0, underlyingArray.length);
            this.valueHasMutated();
            return allValues;
        }
        // If you passed an arg, we interpret it as an array of entries to remove
        if (!arrayOfValues)
            return [];
        return this['remove'](function (value) {
            return ko.utils.arrayIndexOf(arrayOfValues, value) >= 0;
        });
    },

    'destroy': function (valueOrPredicate) {
        var underlyingArray = this.peek();
        var predicate = typeof valueOrPredicate == "function" && !ko.isObservable(valueOrPredicate) ? valueOrPredicate : function
 (value) { return value === valueOrPredicate; };
        this.valueWillMutate();
        for (var i = underlyingArray.length - 1; i >= 0; i--) {
            var value = underlyingArray[i];
            if (predicate(value))
                underlyingArray[i]["_destroy"] = true;
        }
        this.valueHasMutated();
    },

    'destroyAll': function (arrayOfValues) {
        // If you passed zero args, we destroy everything
        if (arrayOfValues === undefined)
            return this['destroy'](function() { return true });

        // If you passed an arg, we interpret it as an array of entries to destroy
        if (!arrayOfValues)
            return [];
        return this['destroy'](function (value) {
            return ko.utils.arrayIndexOf(arrayOfValues, value) >= 0;
        });
    },

    'indexOf': function (item) {
        var underlyingArray = this();
        return ko.utils.arrayIndexOf(underlyingArray, item);
    },

    'replace': function(oldItem, newItem) {
        var index = this['indexOf'](oldItem);
        if (index >= 0) {
            this.valueWillMutate();
            this.peek()[index] = newItem;
            this.valueHasMutated();
        }
    }
};

// Note that for browsers that don't support proto assignment, the
// inheritance chain is created manually in the ko.observableArray constructor
if (ko.utils.canSetPrototype) {
    ko.utils.setPrototypeOf(ko.observableArray['fn'], ko.observable['fn']);
}

// Populate ko.observableArray.fn with read/write functions from native arrays
// Important: Do not add any additional functions here that may reasonably be used to *read* data from the array
// because we'll eval them without causing subscriptions, so ko.computed output could end up getting stale
ko.utils.arrayForEach(["pop", "push", "reverse", "shift", "sort", "splice", "unshift"], function (methodName) {
    ko.observableArray['fn'][methodName] = function () {
        // Use "peek" to avoid creating a subscription in any computed that we're executing in the context of
        // (for consistency with mutating regular observables)
        var underlyingArray = this.peek();
        this.valueWillMutate();
        this.cacheDiffForKnownOperation(underlyingArray, methodName, arguments);
        var methodCallResult = underlyingArray[methodName].apply(underlyingArray, arguments);
        this.valueHasMutated();
        // The native sort and reverse methods return a reference to the array, but it makes more sense to return the observable array
 instead.
        return methodCallResult === underlyingArray ? this : methodCallResult;
    };
});

// Populate ko.observableArray.fn with read-only functions from native arrays
ko.utils.arrayForEach(["slice"], function (methodName) {
    ko.observableArray['fn'][methodName] = function () {
        var underlyingArray = this();
        return underlyingArray[methodName].apply(underlyingArray, arguments);
    };
});

ko.exportSymbol('observableArray', ko.observableArray);
var arrayChangeEventName = 'arrayChange';
ko.extenders['trackArrayChanges'] = function(target, options) {
    // Use the provided options--each call to trackArrayChanges overwrites the previously set options
    target.compareArrayOptions = {};
    if (options && typeof options == "object") {
        ko.utils.extend(target.compareArrayOptions, options);
    }
    target.compareArrayOptions['sparse'] = true;

    // Only modify the target observable once
    if (target.cacheDiffForKnownOperation) {
        return;
    }
    var trackingChanges = false,
```

```
        cachedDiff = null,
        arrayChangeSubscription,
        pendingNotifications = 0,
        underlyingNotifySubscribersFunction,
        underlyingBeforeSubscriptionAddFunction = target.beforeSubscriptionAdd,
        underlyingAfterSubscriptionRemoveFunction = target.afterSubscriptionRemove;

    // Watch "subscribe" calls, and for array change events, ensure change tracking is enabled
    target.beforeSubscriptionAdd = function (event) {
        if (underlyingBeforeSubscriptionAddFunction)
            underlyingBeforeSubscriptionAddFunction.call(target, event);
        if (event === arrayChangeEventName) {
            trackChanges();
        }
    };
    // Watch "dispose" calls, and for array change events, ensure change tracking is disabled when all are disposed
    target.afterSubscriptionRemove = function (event) {
        if (underlyingAfterSubscriptionRemoveFunction)
            underlyingAfterSubscriptionRemoveFunction.call(target, event);
        if (event === arrayChangeEventName && !target.hasSubscriptionsForEvent(arrayChangeEventName)) {
            if (underlyingNotifySubscribersFunction) {
                target['notifySubscribers'] = underlyingNotifySubscribersFunction;
                underlyingNotifySubscribersFunction = undefined;
            }
            arrayChangeSubscription.dispose();
            trackingChanges = false;
        }
    };

    function trackChanges() {
        // Calling 'trackChanges' multiple times is the same as calling it once
        if (trackingChanges) {
            return;
        }

        trackingChanges = true;

        // Intercept "notifySubscribers" to track how many times it was called.
        underlyingNotifySubscribersFunction = target['notifySubscribers'];
        target['notifySubscribers'] = function(valueToNotify, event) {
            if (!event || event === defaultEvent) {
                ++pendingNotifications;
            }
            return underlyingNotifySubscribersFunction.apply(this, arguments);
        };

        // Each time the array changes value, capture a clone so that on the next
        // change it's possible to produce a diff
        var previousContents = [].concat(target.peek() || []);
        cachedDiff = null;
        arrayChangeSubscription = target.subscribe(function(currentContents) {
            // Make a copy of the current contents and ensure it's an array
            currentContents = [].concat(currentContents || []);

            // Compute the diff and issue notifications, but only if someone is listening
            if (target.hasSubscriptionsForEvent(arrayChangeEventName)) {
                var changes = getChanges(previousContents, currentContents);
            }

            // Eliminate references to the old, removed items, so they can be GCed
            previousContents = currentContents;
            cachedDiff = null;
            pendingNotifications = 0;

            if (changes && changes.length) {
                target['notifySubscribers'](changes, arrayChangeEventName);
            }
        });
    }

    function getChanges(previousContents, currentContents) {
        // We try to re-use cached diffs.
        // The scenarios where pendingNotifications > 1 are when using rate-limiting or the Deferred Updates
        // plugin, which without this check would not be compatible with arrayChange notifications. Normally,
        // notifications are issued immediately so we wouldn't be queueing up more than one.
        if (!cachedDiff || pendingNotifications > 1) {
            cachedDiff = ko.utils.compareArrays(previousContents, currentContents, target.compareArrayOptions);
        }

        return cachedDiff;
    }

    target.cacheDiffForKnownOperation = function(rawArray, operationName, args) {
        // Only run if we're currently tracking changes for this observable array
        // and there aren't any pending deferred notifications.
        if (!trackingChanges || pendingNotifications) {
            return;
        }
        var diff = [],
            arrayLength = rawArray.length,
            argsLength = args.length,
            offset = 0;

        function pushDiff(status, value, index) {
            return diff[diff.length] = { 'status': status, 'value': value, 'index': index };
        }
        switch (operationName) {
            case 'push':
                offset = arrayLength;
            case 'unshift':
                for (var index = 0; index < argsLength; index++) {
                    pushDiff('added', args[index], offset + index);
                }
```

```
                break;

            case 'pop':
                offset = arrayLength - 1;
            case 'shift':
                if (arrayLength) {
                    pushDiff('deleted', rawArray[offset], offset);
                }
                break;

            case 'splice':
                // Negative start index means 'from end of array'. After that we clamp to [0...arrayLength].
                // See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice
                var startIndex = Math.min(Math.max(0, args[0] < 0 ? arrayLength + args[0] : args[0]), arrayLength),
                    endDeleteIndex = argsLength === 1 ? arrayLength : Math.min(startIndex + (args[1] || 0), arrayLength),
                    endAddIndex = startIndex + argsLength - 2,
                    endIndex = Math.max(endDeleteIndex, endAddIndex),
                    additions = [], deletions = [];
                for (var index = startIndex, argsIndex = 2; index < endIndex; ++index, ++argsIndex) {
                    if (index < endDeleteIndex)
                        deletions.push(pushDiff('deleted', rawArray[index], index));
                    if (index < endAddIndex)
                        additions.push(pushDiff('added', args[argsIndex], index));
                }
                ko.utils.findMovesInArrayComparison(deletions, additions);
                break;

            default:
                return;
        }
        cachedDiff = diff;
    };
};
var computedState = ko.utils.createSymbolOrString('_state');

ko.computed = ko.dependentObservable = function (evaluatorFunctionOrOptions, evaluatorFunctionTarget, options) {
    if (typeof evaluatorFunctionOrOptions === "object") {
        // Single-parameter syntax - everything is on this "options" param
        options = evaluatorFunctionOrOptions;
    } else {
        // Multi-parameter syntax - construct the options according to the params passed
        options = options || {};
        if (evaluatorFunctionOrOptions) {
            options["read"] = evaluatorFunctionOrOptions;
        }
    }
    if (typeof options["read"] != "function")
        throw Error("Pass a function that returns the value of the ko.computed");

    var writeFunction = options["write"];
    var state = {
        latestValue: undefined,
        isStale: true,
        isDirty: true,
        isBeingEvaluated: false,
        suppressDisposalUntilDisposeWhenReturnsFalse: false,
        isDisposed: false,
        pure: false,
        isSleeping: false,
        readFunction: options["read"],
        evaluatorFunctionTarget: evaluatorFunctionTarget || options["owner"],
        disposeWhenNodeIsRemoved: options["disposeWhenNodeIsRemoved"] || options.disposeWhenNodeIsRemoved || null,
        disposeWhen: options["disposeWhen"] || options.disposeWhen,
        domNodeDisposalCallback: null,
        dependencyTracking: {},
        dependenciesCount: 0,
        evaluationTimeoutInstance: null
    };

    function computedObservable() {
        if (arguments.length > 0) {
            if (typeof writeFunction === "function") {
                // Writing a value
                writeFunction.apply(state.evaluatorFunctionTarget, arguments);
            } else {
                throw new Error("Cannot write a value to a ko.computed unless you specify a 'write' option. If you wish to read the
current value, don't pass any parameters.");
            }
            return this; // Permits chained assignments
        } else {
            // Reading the value
            ko.dependencyDetection.registerDependency(computedObservable);
            if (state.isDirty || (state.isSleeping && computedObservable.haveDependenciesChanged())) {
                computedObservable.evaluateImmediate();
            }
            return state.latestValue;
        }
    }

    computedObservable[computedState] = state;
    computedObservable.hasWriteFunction = typeof writeFunction === "function";

    // Inherit from 'subscribable'
    if (!ko.utils.canSetPrototype) {
        // 'subscribable' won't be on the prototype chain unless we put it there directly
        ko.utils.extend(computedObservable, ko.subscribable['fn']);
    }
    ko.subscribable['fn'].init(computedObservable);

    // Inherit from 'computed'
    ko.utils.setPrototypeOfOrExtend(computedObservable, computedFn);

    if (options['pure']) {
```

```
        state.pure = true;
        state.isSleeping = true;     // Starts off sleeping; will awake on the first subscription
        ko.utils.extend(computedObservable, pureComputedOverrides);
    } else if (options['deferEvaluation']) {
        ko.utils.extend(computedObservable, deferEvaluationOverrides);
    }

    if (ko.options['deferUpdates']) {
        ko.extenders['deferred'](computedObservable, true);
    }

    if (DEBUG) {
        // #1731 - Aid debugging by exposing the computed's options
        computedObservable["_options"] = options;
    }

    if (state.disposeWhenNodeIsRemoved) {
        // Since this computed is associated with a DOM node, and we don't want to dispose the computed
        // until the DOM node is *removed* from the document (as opposed to never having been in the document),
        // we'll prevent disposal until "disposeWhen" first returns false.
        state.suppressDisposalUntilDisposeWhenReturnsFalse = true;

        // disposeWhenNodeIsRemoved: true can be used to opt into the "only dispose after first false result"
        // behaviour even if there's no specific node to watch. In that case, clear the option so we don't try
        // to watch for a non-node's disposal. This technique is intended for KO's internal use only and shouldn't
        // be documented or used by application code, as it's likely to change in a future version of KO.
        if (!state.disposeWhenNodeIsRemoved.nodeType) {
            state.disposeWhenNodeIsRemoved = null;
        }
    }

    // Evaluate, unless sleeping or deferEvaluation is true
    if (!state.isSleeping && !options['deferEvaluation']) {
        computedObservable.evaluateImmediate();
    }

    // Attach a DOM node disposal callback so that the computed will be proactively disposed as soon as the node is
    // removed using ko.removeNode. But skip if isActive is false (there will never be any dependencies to dispose).
    if (state.disposeWhenNodeIsRemoved && computedObservable.isActive()) {
        ko.utils.domNodeDisposal.addDisposeCallback(state.disposeWhenNodeIsRemoved, state.domNodeDisposalCallback = function () {
            computedObservable.dispose();
        });
    }

    return computedObservable;
};

// Utility function that disposes a given dependencyTracking entry
function computedDisposeDependencyCallback(id, entryToDispose) {
    if (entryToDispose !== null && entryToDispose.dispose) {
        entryToDispose.dispose();
    }
}

// This function gets called each time a dependency is detected while evaluating a computed.
// It's factored out as a shared function to avoid creating unnecessary function instances during evaluation.
function computedBeginDependencyDetectionCallback(subscribable, id) {
    var computedObservable = this.computedObservable,
        state = computedObservable[computedState];
    if (!state.isDisposed) {
        if (this.disposalCount && this.disposalCandidates[id]) {
            // Don't want to dispose this subscription, as it's still being used
            computedObservable.addDependencyTracking(id, subscribable, this.disposalCandidates[id]);
            this.disposalCandidates[id] = null; // No need to actually delete the property - disposalCandidates is a transient object
anyway
            --this.disposalCount;
        } else if (!state.dependencyTracking[id]) {
            // Brand new subscription - add it
            computedObservable.addDependencyTracking(id, subscribable, state.isSleeping ? { _target: subscribable } :
computedObservable.subscribeToDependency(subscribable));
        }
        // If the observable we've accessed has a pending notification, ensure we get notified of the actual final value (bypass
 equality checks)
        if (subscribable._notificationIsPending) {
            subscribable._notifyNextChangeIfValueIsDifferent();
        }
    }
}

var computedFn = {
    "equalityComparer": valuesArePrimitiveAndEqual,
    getDependenciesCount: function () {
        return this[computedState].dependenciesCount;
    },
    addDependencyTracking: function (id, target, trackingObj) {
        if (this[computedState].pure && target === this) {
            throw Error("A 'pure' computed must not be called recursively");
        }

        this[computedState].dependencyTracking[id] = trackingObj;
        trackingObj._order = this[computedState].dependenciesCount++;
        trackingObj._version = target.getVersion();
    },
    haveDependenciesChanged: function () {
        var id, dependency, dependencyTracking = this[computedState].dependencyTracking;
        for (id in dependencyTracking) {
            if (dependencyTracking.hasOwnProperty(id)) {
                dependency = dependencyTracking[id];
                if ((this._evalDelayed && dependency._target._notificationIsPending) ||
dependency._target.hasChanged(dependency._version)) {
                    return true;
                }
            }
```

```
        }
    },
    markDirty: function () {
        // Process "dirty" events if we can handle delayed notifications
        if (this._evalDelayed && !this[computedState].isBeingEvaluated) {
            this._evalDelayed(false /*isChange*/);
        }
    },
    isActive: function () {
        var state = this[computedState];
        return state.isDirty || state.dependenciesCount > 0;
    },
    respondToChange: function () {
        // Ignore "change" events if we've already scheduled a delayed notification
        if (!this._notificationIsPending) {
            this.evaluatePossiblyAsync();
        } else if (this[computedState].isDirty) {
            this[computedState].isStale = true;
        }
    },
    subscribeToDependency: function (target) {
        if (target._deferUpdates && !this[computedState].disposeWhenNodeIsRemoved) {
            var dirtySub = target.subscribe(this.markDirty, this, 'dirty'),
                changeSub = target.subscribe(this.respondToChange, this);
            return {
                _target: target,
                dispose: function () {
                    dirtySub.dispose();
                    changeSub.dispose();
                }
            };
        } else {
            return target.subscribe(this.evaluatePossiblyAsync, this);
        }
    },
    evaluatePossiblyAsync: function () {
        var computedObservable = this,
            throttleEvaluationTimeout = computedObservable['throttleEvaluation'];
        if (throttleEvaluationTimeout && throttleEvaluationTimeout >= 0) {
            clearTimeout(this[computedState].evaluationTimeoutInstance);
            this[computedState].evaluationTimeoutInstance = ko.utils.setTimeout(function () {
                computedObservable.evaluateImmediate(true /*notifyChange*/);
            }, throttleEvaluationTimeout);
        } else if (computedObservable._evalDelayed) {
            computedObservable._evalDelayed(true /*isChange*/);
        } else {
            computedObservable.evaluateImmediate(true /*notifyChange*/);
        }
    },
    evaluateImmediate: function (notifyChange) {
        var computedObservable = this,
            state = computedObservable[computedState],
            disposeWhen = state.disposeWhen,
            changed = false;

        if (state.isBeingEvaluated) {
            // If the evaluation of a ko.computed causes side effects, it's possible that it will trigger its own re-evaluation.
            // This is not desirable (it's hard for a developer to realise a chain of dependencies might cause this, and they almost
            // certainly didn't intend infinite re-evaluations). So, for predictability, we simply prevent ko.computeds from causing
            // their own re-evaluation. Further discussion at https://github.com/SteveSanderson/knockout/pull/387
            return;
        }

        // Do not evaluate (and possibly capture new dependencies) if disposed
        if (state.isDisposed) {
            return;
        }

        if (state.disposeWhenNodeIsRemoved && !ko.utils.domNodeIsAttachedToDocument(state.disposeWhenNodeIsRemoved) || disposeWhen &&
disposeWhen()) {
            // See comment above about suppressDisposalUntilDisposeWhenReturnsFalse
            if (!state.suppressDisposalUntilDisposeWhenReturnsFalse) {
                computedObservable.dispose();
                return;
            }
        } else {
            // It just did return false, so we can stop suppressing now
            state.suppressDisposalUntilDisposeWhenReturnsFalse = false;
        }

        state.isBeingEvaluated = true;
        try {
            changed = this.evaluateImmediate_CallReadWithDependencyDetection(notifyChange);
        } finally {
            state.isBeingEvaluated = false;
        }

        if (!state.dependenciesCount) {
            computedObservable.dispose();
        }

        return changed;
    },
    evaluateImmediate_CallReadWithDependencyDetection: function (notifyChange) {
        // This function is really just part of the evaluateImmediate logic. You would never call it from anywhere else.
        // Factoring it out into a separate function means it can be independent of the try/catch block in evaluateImmediate,
        // which contributes to saving about 40% off the CPU overhead of computed evaluation (on V8 at least).

        var computedObservable = this,
            state = computedObservable[computedState],
            changed = false;

        // Initially, we assume that none of the subscriptions are still being used (i.e., all are candidates for disposal).
```

```
        // Then, during evaluation, we cross off any that are in fact still being used.
        var isInitial = state.pure ? undefined : !state.dependenciesCount,   // If we're evaluating when there are no previous
 dependencies, it must be the first time
            dependencyDetectionContext = {
                computedObservable: computedObservable,
                disposalCandidates: state.dependencyTracking,
                disposalCount: state.dependenciesCount
            };

        ko.dependencyDetection.begin({
            callbackTarget: dependencyDetectionContext,
            callback: computedBeginDependencyDetectionCallback,
            computed: computedObservable,
            isInitial: isInitial
        });

        state.dependencyTracking = {};
        state.dependenciesCount = 0;

        var newValue = this.evaluateImmediate_CallReadThenEndDependencyDetection(state, dependencyDetectionContext);

        if (computedObservable.isDifferent(state.latestValue, newValue)) {
            if (!state.isSleeping) {
                computedObservable["notifySubscribers"](state.latestValue, "beforeChange");
            }

            state.latestValue = newValue;
            if (DEBUG) computedObservable._latestValue = newValue;

            if (state.isSleeping) {
                computedObservable.updateVersion();
            } else if (notifyChange) {
                computedObservable["notifySubscribers"](state.latestValue);
            }
            changed = true;
        }

        if (isInitial) {
            computedObservable["notifySubscribers"](state.latestValue, "awake");
        }

        return changed;
    },
    evaluateImmediate_CallReadThenEndDependencyDetection: function (state, dependencyDetectionContext) {
        // This function is really part of the evaluateImmediate_CallReadWithDependencyDetection logic.
        // You'd never call it from anywhere else. Factoring it out means that evaluateImmediate_CallReadWithDependencyDetection
        // can be independent of try/finally blocks, which contributes to saving about 40% off the CPU
        // overhead of computed evaluation (on V8 at least).

        try {
            var readFunction = state.readFunction;
            return state.evaluatorFunctionTarget ? readFunction.call(state.evaluatorFunctionTarget) : readFunction();
        } finally {
            ko.dependencyDetection.end();

            // For each subscription no longer being used, remove it from the active subscriptions list and dispose it
            if (dependencyDetectionContext.disposalCount && !state.isSleeping) {
                ko.utils.objectForEach(dependencyDetectionContext.disposalCandidates, computedDisposeDependencyCallback);
            }

            state.isStale = state.isDirty = false;
        }
    },
    peek: function (evaluate) {
        // By default, peek won't re-evaluate, except while the computed is sleeping or to get the initial value when
"deferEvaluation" is set.
        // Pass in true to evaluate if needed.
        var state = this[computedState];
        if ((state.isDirty && (evaluate || !state.dependenciesCount)) || (state.isSleeping && this.haveDependenciesChanged())) {
            this.evaluateImmediate();
        }
        return state.latestValue;
    },
    limit: function (limitFunction) {
        // Override the limit function with one that delays evaluation as well
        ko.subscribable['fn'].limit.call(this, limitFunction);
        this._evalIfChanged = function () {
            if (this[computedState].isStale) {
                this.evaluateImmediate();
            } else {
                this[computedState].isDirty = false;
            }
            return this[computedState].latestValue;
        };
        this._evalDelayed = function (isChange) {
            this._limitBeforeChange(this[computedState].latestValue);

            // Mark as dirty
            this[computedState].isDirty = true;
            if (isChange) {
                this[computedState].isStale = true;
            }

            // Pass the observable to the "limit" code, which will evaluate it when
            // it's time to do the notification.
            this._limitChange(this);
        };
    },
    dispose: function () {
        var state = this[computedState];
        if (!state.isSleeping && state.dependencyTracking) {
            ko.utils.objectForEach(state.dependencyTracking, function (id, dependency) {
```

```
                if (dependency.dispose)
                    dependency.dispose();
            });
        }
        if (state.disposeWhenNodeIsRemoved && state.domNodeDisposalCallback) {
            ko.utils.domNodeDisposal.removeDisposeCallback(state.disposeWhenNodeIsRemoved, state.domNodeDisposalCallback);
        }
        state.dependencyTracking = null;
        state.dependenciesCount = 0;
        state.isDisposed = true;
        state.isStale = false;
        state.isDirty = false;
        state.isSleeping = false;
        state.disposeWhenNodeIsRemoved = null;
    }
};

var pureComputedOverrides = {
    beforeSubscriptionAdd: function (event) {
        // If asleep, wake up the computed by subscribing to any dependencies.
        var computedObservable = this,
            state = computedObservable[computedState];
        if (!state.isDisposed && state.isSleeping && event == 'change') {
            state.isSleeping = false;
            if (state.isStale || computedObservable.haveDependenciesChanged()) {
                state.dependencyTracking = null;
                state.dependenciesCount = 0;
                if (computedObservable.evaluateImmediate()) {
                    computedObservable.updateVersion();
                }
            } else {
                // First put the dependencies in order
                var dependeciesOrder = [];
                ko.utils.objectForEach(state.dependencyTracking, function (id, dependency) {
                    dependeciesOrder[dependency._order] = id;
                });
                // Next, subscribe to each one
                ko.utils.arrayForEach(dependeciesOrder, function (id, order) {
                    var dependency = state.dependencyTracking[id],
                        subscription = computedObservable.subscribeToDependency(dependency._target);
                    subscription._order = order;
                    subscription._version = dependency._version;
                    state.dependencyTracking[id] = subscription;
                });
            }
            if (!state.isDisposed) {     // test since evaluating could trigger disposal
                computedObservable["notifySubscribers"](state.latestValue, "awake");
            }
        }
    },
    afterSubscriptionRemove: function (event) {
        var state = this[computedState];
        if (!state.isDisposed && event == 'change' && !this.hasSubscriptionsForEvent('change')) {
            ko.utils.objectForEach(state.dependencyTracking, function (id, dependency) {
                if (dependency.dispose) {
                    state.dependencyTracking[id] = {
                        _target: dependency._target,
                        _order: dependency._order,
                        _version: dependency._version
                    };
                    dependency.dispose();
                }
            });
            state.isSleeping = true;
            this["notifySubscribers"](undefined, "asleep");
        }
    },
    getVersion: function () {
        // Because a pure computed is not automatically updated while it is sleeping, we can't
        // simply return the version number. Instead, we check if any of the dependencies have
        // changed and conditionally re-evaluate the computed observable.
        var state = this[computedState];
        if (state.isSleeping && (state.isStale || this.haveDependenciesChanged())) {
            this.evaluateImmediate();
        }
        return ko.subscribable['fn'].getVersion.call(this);
    }
};

var deferEvaluationOverrides = {
    beforeSubscriptionAdd: function (event) {
        // This will force a computed with deferEvaluation to evaluate when the first subscription is registered.
        if (event == 'change' || event == 'beforeChange') {
            this.peek();
        }
    }
};

// Note that for browsers that don't support proto assignment, the
// inheritance chain is created manually in the ko.computed constructor
if (ko.utils.canSetPrototype) {
    ko.utils.setPrototypeOf(computedFn, ko.subscribable['fn']);
}

// Set the proto chain values for ko.hasPrototype
var protoProp = ko.observable.protoProperty; // == "__ko_proto__"
ko.computed[protoProp] = ko.observable;
computedFn[protoProp] = ko.computed;

ko.isComputed = function (instance) {
    return ko.hasPrototype(instance, ko.computed);
};
```

```
ko.isPureComputed = function (instance) {
    return ko.hasPrototype(instance, ko.computed)
        && instance[computedState] && instance[computedState].pure;
};

ko.exportSymbol('computed', ko.computed);
ko.exportSymbol('dependentObservable', ko.computed);    // export ko.dependentObservable for backwards compatibility (1.x)
ko.exportSymbol('isComputed', ko.isComputed);
ko.exportSymbol('isPureComputed', ko.isPureComputed);
ko.exportSymbol('computed.fn', computedFn);
ko.exportProperty(computedFn, 'peek', computedFn.peek);
ko.exportProperty(computedFn, 'dispose', computedFn.dispose);
ko.exportProperty(computedFn, 'isActive', computedFn.isActive);
ko.exportProperty(computedFn, 'getDependenciesCount', computedFn.getDependenciesCount);

ko.pureComputed = function (evaluatorFunctionOrOptions, evaluatorFunctionTarget) {
    if (typeof evaluatorFunctionOrOptions === 'function') {
        return ko.computed(evaluatorFunctionOrOptions, evaluatorFunctionTarget, {'pure':true});
    } else {
        evaluatorFunctionOrOptions = ko.utils.extend({}, evaluatorFunctionOrOptions);   // make a copy of the parameter object
        evaluatorFunctionOrOptions['pure'] = true;
        return ko.computed(evaluatorFunctionOrOptions, evaluatorFunctionTarget);
    }
}
ko.exportSymbol('pureComputed', ko.pureComputed);

(function() {
    var maxNestedObservableDepth = 10; // Escape the (unlikely) pathalogical case where an observable's current value is itself (or
similar reference cycle)

    ko.toJS = function(rootObject) {
        if (arguments.length == 0)
            throw new Error("When calling ko.toJS, pass the object you want to convert.");
        // We just unwrap everything at every level in the object graph
        return mapJsObjectGraph(rootObject, function(valueToMap) {
            // Loop because an observable's value might in turn be another observable wrapper
            for (var i = 0; ko.isObservable(valueToMap) && (i < maxNestedObservableDepth); i++)
                valueToMap = valueToMap();
            return valueToMap;
        });
    };

    ko.toJSON = function(rootObject, replacer, space) {     // replacer and space are optional
        var plainJavaScriptObject = ko.toJS(rootObject);
        return ko.utils.stringifyJson(plainJavaScriptObject, replacer, space);
    };

    function mapJsObjectGraph(rootObject, mapInputCallback, visitedObjects) {
        visitedObjects = visitedObjects || new objectLookup();

        rootObject = mapInputCallback(rootObject);
        var canHaveProperties = (typeof rootObject == "object") && (rootObject !== null) && (rootObject !== undefined) && (!
(rootObject instanceof RegExp)) && (!(rootObject instanceof Date)) && (!(rootObject instanceof String)) && (!(rootObject instanceof
Number)) && (!(rootObject instanceof Boolean));
        if (!canHaveProperties)
            return rootObject;

        var outputProperties = rootObject instanceof Array ? [] : {};
        visitedObjects.save(rootObject, outputProperties);

        visitPropertiesOrArrayEntries(rootObject, function(indexer) {
            var propertyValue = mapInputCallback(rootObject[indexer]);

            switch (typeof propertyValue) {
                case "boolean":
                case "number":
                case "string":
                case "function":
                    outputProperties[indexer] = propertyValue;
                    break;
                case "object":
                case "undefined":
                    var previouslyMappedValue = visitedObjects.get(propertyValue);
                    outputProperties[indexer] = (previouslyMappedValue !== undefined)
                        ? previouslyMappedValue
                        : mapJsObjectGraph(propertyValue, mapInputCallback, visitedObjects);
                    break;
            }
        });

        return outputProperties;
    }

    function visitPropertiesOrArrayEntries(rootObject, visitorCallback) {
        if (rootObject instanceof Array) {
            for (var i = 0; i < rootObject.length; i++)
                visitorCallback(i);

            // For arrays, also respect toJSON property for custom mappings (fixes #278)
            if (typeof rootObject['toJSON'] == 'function')
                visitorCallback('toJSON');
        } else {
            for (var propertyName in rootObject) {
                visitorCallback(propertyName);
            }
        }
    };

    function objectLookup() {
        this.keys = [];
        this.values = [];
    };
```

```
    objectLookup.prototype = {
        constructor: objectLookup,
        save: function(key, value) {
            var existingIndex = ko.utils.arrayIndexOf(this.keys, key);
            if (existingIndex >= 0)
                this.values[existingIndex] = value;
            else {
                this.keys.push(key);
                this.values.push(value);
            }
        },
        get: function(key) {
            var existingIndex = ko.utils.arrayIndexOf(this.keys, key);
            return (existingIndex >= 0) ? this.values[existingIndex] : undefined;
        }
    };
})();

ko.exportSymbol('toJS', ko.toJS);
ko.exportSymbol('toJSON', ko.toJSON);
(function () {
    var hasDomDataExpandoProperty = '__ko__hasDomDataOptionValue__';

    // Normally, SELECT elements and their OPTIONs can only take value of type 'string' (because the values
    // are stored on DOM attributes). ko.selectExtensions provides a way for SELECTs/OPTIONs to have values
    // that are arbitrary objects. This is very convenient when implementing things like cascading dropdowns.
    ko.selectExtensions = {
        readValue : function(element) {
            switch (ko.utils.tagNameLower(element)) {
                case 'option':
                    if (element[hasDomDataExpandoProperty] === true)
                        return ko.utils.domData.get(element, ko.bindingHandlers.options.optionValueDomDataKey);
                    return ko.utils.ieVersion <= 7
                        ? (element.getAttributeNode('value') && element.getAttributeNode('value').specified ? element.value :
element.text)
                        : element.value;
                case 'select':
                    return element.selectedIndex >= 0 ? ko.selectExtensions.readValue(element.options[element.selectedIndex]) :
undefined;
                default:
                    return element.value;
            }
        },

        writeValue: function(element, value, allowUnset) {
            switch (ko.utils.tagNameLower(element)) {
                case 'option':
                    switch(typeof value) {
                        case "string":
                            ko.utils.domData.set(element, ko.bindingHandlers.options.optionValueDomDataKey, undefined);
                            if (hasDomDataExpandoProperty in element) { // IE <= 8 throws errors if you delete non-existent properties
from a DOM node
                                delete element[hasDomDataExpandoProperty];
                            }
                            element.value = value;
                            break;
                        default:
                            // Store arbitrary object using DomData
                            ko.utils.domData.set(element, ko.bindingHandlers.options.optionValueDomDataKey, value);
                            element[hasDomDataExpandoProperty] = true;

                            // Special treatment of numbers is just for backward compatibility. KO 1.2.1 wrote numerical values to
element.value.
                            element.value = typeof value === "number" ? value : "";
                            break;
                    }
                    break;
                case 'select':
                    if (value === "" || value === null)       // A blank string or null value will select the caption
                        value = undefined;
                    var selection = -1;
                    for (var i = 0, n = element.options.length, optionValue; i < n; ++i) {
                        optionValue = ko.selectExtensions.readValue(element.options[i]);
                        // Include special check to handle selecting a caption with a blank string value
                        if (optionValue == value || (optionValue == "" && value === undefined)) {
                            selection = i;
                            break;
                        }
                    }
                    if (allowUnset || selection >= 0 || (value === undefined && element.size > 1)) {
                        element.selectedIndex = selection;
                    }
                    break;
                default:
                    if ((value === null) || (value === undefined))
                        value = "";
                    element.value = value;
                    break;
            }
        }
    };
})();

ko.exportSymbol('selectExtensions', ko.selectExtensions);
ko.exportSymbol('selectExtensions.readValue', ko.selectExtensions.readValue);
ko.exportSymbol('selectExtensions.writeValue', ko.selectExtensions.writeValue);
ko.expressionRewriting = (function () {
    var javaScriptReservedWords = ["true", "false", "null", "undefined"];

    // Matches something that can be assigned to--either an isolated identifier or something ending with a property accessor
    // This is designed to be simple and avoid false negatives, but could produce false positives (e.g., a+b.c).
    // This also will not properly handle nested brackets (e.g., obj1[obj2['prop']]; see #911).
```

```
    var javaScriptAssignmentTarget = /^(?:[$_a-z][$\w]*|(.+)(\.\s*[$_a-z][$\w]*|\[.+\]))$/i;

    function getWriteableValue(expression) {
        if (ko.utils.arrayIndexOf(javaScriptReservedWords, expression) >= 0)
            return false;
        var match = expression.match(javaScriptAssignmentTarget);
        return match === null ? false : match[1] ? ('Object(' + match[1] + ')' + match[2]) : expression;
    }

    // The following regular expressions will be used to split an object-literal string into tokens

        // These two match strings, either with double quotes or single quotes
    var stringDouble = '"(?:[^"\\\\]|\\\\.)*"',
        stringSingle = "'(?:[^'\\\\]|\\\\.)*'",
        // Matches a regular expression (text enclosed by slashes), but will also match sets of divisions
        // as a regular expression (this is handled by the parsing loop below).
        stringRegexp = '/(?:[^/\\\\]|\\\\.)*/\w*',
        // These characters have special meaning to the parser and must not appear in the middle of a
        // token, except as part of a string.
        specials = ',"\'{}()/:[\\]',
        // Match text (at least two characters) that does not contain any of the above special characters,
        // although some of the special characters are allowed to start it (all but the colon and comma).
        // The text can contain spaces, but leading or trailing spaces are skipped.
        everyThingElse = '[^\\s:,/][^' + specials + ']*[^\\s' + specials + ']',
        // Match any non-space character not matched already. This will match colons and commas, since they're
        // not matched by "everyThingElse", but will also match any other single character that wasn't already
        // matched (for example: in "a: 1, b: 2", each of the non-space characters will be matched by oneNotSpace).
        oneNotSpace = '[^\\s]',

        // Create the actual regular expression by or-ing the above strings. The order is important.
        bindingToken = RegExp(stringDouble + '|' + stringSingle + '|' + stringRegexp + '|' + everyThingElse + '|' + oneNotSpace, 'g'),

        // Match end of previous token to determine whether a slash is a division or regex.
        divisionLookBehind = /[\])"'A-Za-z0-9_$]+$/,
        keywordRegexLookBehind = {'in':1,'return':1,'typeof':1};

    function parseObjectLiteral(objectLiteralString) {
        // Trim leading and trailing spaces from the string
        var str = ko.utils.stringTrim(objectLiteralString);

        // Trim braces '{' surrounding the whole object literal
        if (str.charCodeAt(0) === 123) str = str.slice(1, -1);

        // Split into tokens
        var result = [], toks = str.match(bindingToken), key, values = [], depth = 0;

        if (toks) {
            // Append a comma so that we don't need a separate code block to deal with the last item
            toks.push(',');

            for (var i = 0, tok; tok = toks[i]; ++i) {
                var c = tok.charCodeAt(0);
                // A comma signals the end of a key/value pair if depth is zero
                if (c === 44) { // ","
                    if (depth <= 0) {
                        result.push((key && values.length) ? {key: key, value: values.join('')} : {'unknown': key ||
values.join('')});
                        key = depth = 0;
                        values = [];
                        continue;
                    }
                // Simply skip the colon that separates the name and value
                } else if (c === 58) { // ":"
                    if (!depth && !key && values.length === 1) {
                        key = values.pop();
                        continue;
                    }
                // A set of slashes is initially matched as a regular expression, but could be division
                } else if (c === 47 && i && tok.length > 1) {  // "/"
                    // Look at the end of the previous token to determine if the slash is actually division
                    var match = toks[i-1].match(divisionLookBehind);
                    if (match && !keywordRegexLookBehind[match[0]]) {
                        // The slash is actually a division punctuator; re-parse the remainder of the string (not including the slash)
                        str = str.substr(str.indexOf(tok) + 1);
                        toks = str.match(bindingToken);
                        toks.push(',');
                        i = -1;
                        // Continue with just the slash
                        tok = '/';
                    }
                // Increment depth for parentheses, braces, and brackets so that interior commas are ignored
                } else if (c === 40 || c === 123 || c === 91) { // '(', '{', '['
                    ++depth;
                } else if (c === 41 || c === 125 || c === 93) { // ')', '}', ']'
                    --depth;
                // The key will be the first token; if it's a string, trim the quotes
                } else if (!key && !values.length && (c === 34 || c === 39)) { // '"', "'"
                    tok = tok.slice(1, -1);
                }
                values.push(tok);
            }
        }
        return result;
    }

    // Two-way bindings include a write function that allow the handler to update the value even if it's not an observable.
    var twoWayBindings = {};

    function preProcessBindings(bindingsStringOrKeyValueArray, bindingOptions) {
        bindingOptions = bindingOptions || {};

        function processKeyValue(key, val) {
            var writableVal;
```

```
        function callPreprocessHook(obj) {
            return (obj && obj['preprocess']) ? (val = obj['preprocess'](val, key, processKeyValue)) : true;
        }
        if (!bindingParams) {
            if (!callPreprocessHook(ko['getBindingHandler'](key)))
                return;

            if (twoWayBindings[key] && (writableVal = getWriteableValue(val))) {
                // For two-way bindings, provide a write method in case the value
                // isn't a writable observable.
                propertyAccessorResultStrings.push("'" + key + "':function(_z){" + writableVal + "=_z}");
            }
        }
        // Values are wrapped in a function so that each value can be accessed independently
        if (makeValueAccessors) {
            val = 'function(){return ' + val + ' }';
        }
        resultStrings.push("'" + key + "':" + val);
    }

    var resultStrings = [],
        propertyAccessorResultStrings = [],
        makeValueAccessors = bindingOptions['valueAccessors'],
        bindingParams = bindingOptions['bindingParams'],
        keyValueArray = typeof bindingsStringOrKeyValueArray === "string" ?
            parseObjectLiteral(bindingsStringOrKeyValueArray) : bindingsStringOrKeyValueArray;

    ko.utils.arrayForEach(keyValueArray, function(keyValue) {
        processKeyValue(keyValue.key || keyValue['unknown'], keyValue.value);
    });

    if (propertyAccessorResultStrings.length)
        processKeyValue('_ko_property_writers', "{" + propertyAccessorResultStrings.join(",") + " }");

    return resultStrings.join(",");
}

return {
    bindingRewriteValidators: [],

    twoWayBindings: twoWayBindings,

    parseObjectLiteral: parseObjectLiteral,

    preProcessBindings: preProcessBindings,

    keyValueArrayContainsKey: function(keyValueArray, key) {
        for (var i = 0; i < keyValueArray.length; i++)
            if (keyValueArray[i]['key'] == key)
                return true;
        return false;
    },

    // Internal, private KO utility for updating model properties from within bindings
    // property:            If the property being updated is (or might be) an observable, pass it here
    //                      If it turns out to be a writable observable, it will be written to directly
    // allBindings:         An object with a get method to retrieve bindings in the current execution context.
    //                      This will be searched for a '_ko_property_writers' property in case you're writing to a non-observable
    // key:                 The key identifying the property to be written. Example: for { hasFocus: myValue }, write to 'myValue'
by specifying the key 'hasFocus'
    // value:               The value to be written
    // checkIfDifferent:    If true, and if the property being written is a writable observable, the value will only be written if
    //                      it is !== existing value on that writable observable
    writeValueToProperty: function(property, allBindings, key, value, checkIfDifferent) {
        if (!property || !ko.isObservable(property)) {
            var propWriters = allBindings.get('_ko_property_writers');
            if (propWriters && propWriters[key])
                propWriters[key](value);
        } else if (ko.isWriteableObservable(property) && (!checkIfDifferent || property.peek() !== value)) {
            property(value);
        }
    }
};
})();

ko.exportSymbol('expressionRewriting', ko.expressionRewriting);
ko.exportSymbol('expressionRewriting.bindingRewriteValidators', ko.expressionRewriting.bindingRewriteValidators);
ko.exportSymbol('expressionRewriting.parseObjectLiteral', ko.expressionRewriting.parseObjectLiteral);
ko.exportSymbol('expressionRewriting.preProcessBindings', ko.expressionRewriting.preProcessBindings);

// Making bindings explicitly declare themselves as "two way" isn't ideal in the long term (it would be better if
// all bindings could use an official 'property writer' API without needing to declare that they might). However,
// since this is not, and has never been, a public API (_ko_property_writers was never documented), it's acceptable
// as an internal implementation detail in the short term.
// For those developers who rely on _ko_property_writers in their custom bindings, we expose _twoWayBindings as an
// undocumented feature that makes it relatively easy to upgrade to KO 3.0. However, this is still not an official
// public API, and we reserve the right to remove it at any time if we create a real public property writers API.
ko.exportSymbol('expressionRewriting._twoWayBindings', ko.expressionRewriting.twoWayBindings);

// For backward compatibility, define the following aliases. (Previously, these function names were misleading because
// they referred to JSON specifically, even though they actually work with arbitrary JavaScript object literal expressions.)
ko.exportSymbol('jsonExpressionRewriting', ko.expressionRewriting);
ko.exportSymbol('jsonExpressionRewriting.insertPropertyAccessorsIntoJson', ko.expressionRewriting.preProcessBindings);
(function() {
    // "Virtual elements" is an abstraction on top of the usual DOM API which understands the notion that comment nodes
    // may be used to represent hierarchy (in addition to the DOM's natural hierarchy).
    // If you call the DOM-manipulating functions on ko.virtualElements, you will be able to read and write the state
    // of that virtual hierarchy
    //
    // The point of all this is to support containerless templates (e.g., <!-- ko foreach:someCollection -->blah<!-- /ko -->)
    // without having to scatter special cases all over the binding and templating code.

    // IE 9 cannot reliably read the "nodeValue" property of a comment node (see
```

```
https://github.com/SteveSanderson/knockout/issues/186)
    // but it does give them a nonstandard alternative property called "text" that it can read reliably. Other browsers don't have
that property.
    // So, use node.text where available, and node.nodeValue elsewhere
    var commentNodesHaveTextProperty = document && document.createComment("test").text === "<!--test-->";

    var startCommentRegex = commentNodesHaveTextProperty ? /^<!--\s*ko(?:\s+([\s\S]+))?\s*-->$/ : /^\s*ko(?:\s+([\s\S]+))?\s*$/;
    var endCommentRegex =   commentNodesHaveTextProperty ? /^<!--\s*\/ko\s*-->$/ : /^\s*\/ko\s*$/;
    var htmlTagsWithOptionallyClosingChildren = { 'ul': true, 'ol': true };

    function isStartComment(node) {
        return (node.nodeType == 8) && startCommentRegex.test(commentNodesHaveTextProperty ? node.text : node.nodeValue);
    }

    function isEndComment(node) {
        return (node.nodeType == 8) && endCommentRegex.test(commentNodesHaveTextProperty ? node.text : node.nodeValue);
    }

    function getVirtualChildren(startComment, allowUnbalanced) {
        var currentNode = startComment;
        var depth = 1;
        var children = [];
        while (currentNode = currentNode.nextSibling) {
            if (isEndComment(currentNode)) {
                depth--;
                if (depth === 0)
                    return children;
            }

            children.push(currentNode);

            if (isStartComment(currentNode))
                depth++;
        }
        if (!allowUnbalanced)
            throw new Error("Cannot find closing comment tag to match: " + startComment.nodeValue);
        return null;
    }

    function getMatchingEndComment(startComment, allowUnbalanced) {
        var allVirtualChildren = getVirtualChildren(startComment, allowUnbalanced);
        if (allVirtualChildren) {
            if (allVirtualChildren.length > 0)
                return allVirtualChildren[allVirtualChildren.length - 1].nextSibling;
            return startComment.nextSibling;
        } else
            return null; // Must have no matching end comment, and allowUnbalanced is true
    }

    function getUnbalancedChildTags(node) {
        // e.g., from <div>OK</div><!-- ko blah --><span>Another</span>, returns: <!-- ko blah --><span>Another</span>
        //       from <div>OK</div><!-- /ko --><!-- /ko -->,             returns: <!-- /ko --><!-- /ko -->
        var childNode = node.firstChild, captureRemaining = null;
        if (childNode) {
            do {
                if (captureRemaining)                   // We already hit an unbalanced node and are now just scooping up all
subsequent nodes
                    captureRemaining.push(childNode);
                else if (isStartComment(childNode)) {
                    var matchingEndComment = getMatchingEndComment(childNode, /* allowUnbalanced: */ true);
                    if (matchingEndComment)             // It's a balanced tag, so skip immediately to the end of this virtual set
                        childNode = matchingEndComment;
                    else
                        captureRemaining = [childNode]; // It's unbalanced, so start capturing from this point
                } else if (isEndComment(childNode)) {
                    captureRemaining = [childNode];     // It's unbalanced (if it wasn't, we'd have skipped over it already), so start
capturing
                }
            } while (childNode = childNode.nextSibling);
        }
        return captureRemaining;
    }

    ko.virtualElements = {
        allowedBindings: {},

        childNodes: function(node) {
            return isStartComment(node) ? getVirtualChildren(node) : node.childNodes;
        },

        emptyNode: function(node) {
            if (!isStartComment(node))
                ko.utils.emptyDomNode(node);
            else {
                var virtualChildren = ko.virtualElements.childNodes(node);
                for (var i = 0, j = virtualChildren.length; i < j; i++)
                    ko.removeNode(virtualChildren[i]);
            }
        },

        setDomNodeChildren: function(node, childNodes) {
            if (!isStartComment(node))
                ko.utils.setDomNodeChildren(node, childNodes);
            else {
                ko.virtualElements.emptyNode(node);
                var endCommentNode = node.nextSibling; // Must be the next sibling, as we just emptied the children
                for (var i = 0, j = childNodes.length; i < j; i++)
                    endCommentNode.parentNode.insertBefore(childNodes[i], endCommentNode);
            }
        },

        prepend: function(containerNode, nodeToPrepend) {
            if (!isStartComment(containerNode)) {
```

```
            if (containerNode.firstChild)
                containerNode.insertBefore(nodeToPrepend, containerNode.firstChild);
            else
                containerNode.appendChild(nodeToPrepend);
        } else {
            // Start comments must always have a parent and at least one following sibling (the end comment)
            containerNode.parentNode.insertBefore(nodeToPrepend, containerNode.nextSibling);
        }
    },

    insertAfter: function(containerNode, nodeToInsert, insertAfterNode) {
        if (!insertAfterNode) {
            ko.virtualElements.prepend(containerNode, nodeToInsert);
        } else if (!isStartComment(containerNode)) {
            // Insert after insertion point
            if (insertAfterNode.nextSibling)
                containerNode.insertBefore(nodeToInsert, insertAfterNode.nextSibling);
            else
                containerNode.appendChild(nodeToInsert);
        } else {
            // Children of start comments must always have a parent and at least one following sibling (the end comment)
            containerNode.parentNode.insertBefore(nodeToInsert, insertAfterNode.nextSibling);
        }
    },

    firstChild: function(node) {
        if (!isStartComment(node))
            return node.firstChild;
        if (!node.nextSibling || isEndComment(node.nextSibling))
            return null;
        return node.nextSibling;
    },

    nextSibling: function(node) {
        if (isStartComment(node))
            node = getMatchingEndComment(node);
        if (node.nextSibling && isEndComment(node.nextSibling))
            return null;
        return node.nextSibling;
    },

    hasBindingValue: isStartComment,

    virtualNodeBindingValue: function(node) {
        var regexMatch = (commentNodesHaveTextProperty ? node.text : node.nodeValue).match(startCommentRegex);
        return regexMatch ? regexMatch[1] : null;
    },

    normaliseVirtualElementDomStructure: function(elementVerified) {
        // Workaround for https://github.com/SteveSanderson/knockout/issues/155
        // (IE <= 8 or IE 9 quirks mode parses your HTML weirdly, treating closing </li> tags as if they don't exist, thereby
moving comment nodes
        // that are direct descendants of <ul> into the preceding <li>)
        if (!htmlTagsWithOptionallyClosingChildren[ko.utils.tagNameLower(elementVerified)])
            return;

        // Scan immediate children to see if they contain unbalanced comment tags. If they do, those comment tags
        // must be intended to appear *after* that child, so move them there.
        var childNode = elementVerified.firstChild;
        if (childNode) {
            do {
                if (childNode.nodeType === 1) {
                    var unbalancedTags = getUnbalancedChildTags(childNode);
                    if (unbalancedTags) {
                        // Fix up the DOM by moving the unbalanced tags to where they most likely were intended to be placed -
*after* the child
                        var nodeToInsertBefore = childNode.nextSibling;
                        for (var i = 0; i < unbalancedTags.length; i++) {
                            if (nodeToInsertBefore)
                                elementVerified.insertBefore(unbalancedTags[i], nodeToInsertBefore);
                            else
                                elementVerified.appendChild(unbalancedTags[i]);
                        }
                    }
                }
            } while (childNode = childNode.nextSibling);
        }
    }
};
})();
ko.exportSymbol('virtualElements', ko.virtualElements);
ko.exportSymbol('virtualElements.allowedBindings', ko.virtualElements.allowedBindings);
ko.exportSymbol('virtualElements.emptyNode', ko.virtualElements.emptyNode);
//ko.exportSymbol('virtualElements.firstChild', ko.virtualElements.firstChild);     // firstChild is not minified
ko.exportSymbol('virtualElements.insertAfter', ko.virtualElements.insertAfter);
//ko.exportSymbol('virtualElements.nextSibling', ko.virtualElements.nextSibling);   // nextSibling is not minified
ko.exportSymbol('virtualElements.prepend', ko.virtualElements.prepend);
ko.exportSymbol('virtualElements.setDomNodeChildren', ko.virtualElements.setDomNodeChildren);
(function() {
    var defaultBindingAttributeName = "data-bind";

    ko.bindingProvider = function() {
        this.bindingCache = {};
    };

    ko.utils.extend(ko.bindingProvider.prototype, {
        'nodeHasBindings': function(node) {
            switch (node.nodeType) {
                case 1: // Element
                    return node.getAttribute(defaultBindingAttributeName) != null
                        || ko.components['getComponentNameForNode'](node);
                case 8: // Comment node
                    return ko.virtualElements.hasBindingValue(node);
```

```
                default: return false;
            }
        },

        'getBindings': function(node, bindingContext) {
            var bindingsString = this['getBindingsString'](node, bindingContext),
                parsedBindings = bindingsString ? this['parseBindingsString'](bindingsString, bindingContext, node) : null;
            return ko.components.addBindingsForCustomElement(parsedBindings, node, bindingContext, /* valueAccessors */ false);
        },

        'getBindingAccessors': function(node, bindingContext) {
            var bindingsString = this['getBindingsString'](node, bindingContext),
                parsedBindings = bindingsString ? this['parseBindingsString'](bindingsString, bindingContext, node, {
'valueAccessors': true }) : null;
            return ko.components.addBindingsForCustomElement(parsedBindings, node, bindingContext, /* valueAccessors */ true);
        },

        // The following function is only used internally by this default provider.
        // It's not part of the interface definition for a general binding provider.
        'getBindingsString': function(node, bindingContext) {
            switch (node.nodeType) {
                case 1: return node.getAttribute(defaultBindingAttributeName);   // Element
                case 8: return ko.virtualElements.virtualNodeBindingValue(node); // Comment node
                default: return null;
            }
        },

        // The following function is only used internally by this default provider.
        // It's not part of the interface definition for a general binding provider.
        'parseBindingsString': function(bindingsString, bindingContext, node, options) {
            try {
                var bindingFunction = createBindingsStringEvaluatorViaCache(bindingsString, this.bindingCache, options);
                return bindingFunction(bindingContext, node);
            } catch (ex) {
                ex.message = "Unable to parse bindings.\nBindings value: " + bindingsString + "\nMessage: " + ex.message;
                throw ex;
            }
        }
    });

    ko.bindingProvider['instance'] = new ko.bindingProvider();

    function createBindingsStringEvaluatorViaCache(bindingsString, cache, options) {
        var cacheKey = bindingsString + (options && options['valueAccessors'] || '');
        return cache[cacheKey]
            || (cache[cacheKey] = createBindingsStringEvaluator(bindingsString, options));
    }

    function createBindingsStringEvaluator(bindingsString, options) {
        // Build the source for a function that evaluates "expression"
        // For each scope variable, add an extra level of "with" nesting
        // Example result: with(sc1) { with(sc0) { return (expression) } }
        var rewrittenBindings = ko.expressionRewriting.preProcessBindings(bindingsString, options),
            functionBody = "with($context){with($data||{}){return{" + rewrittenBindings + "}}}";
        return new Function("$context", "$element", functionBody);
    }
})();

ko.exportSymbol('bindingProvider', ko.bindingProvider);
(function () {
    ko.bindingHandlers = {};

    // The following element types will not be recursed into during binding.
    var bindingDoesNotRecurseIntoElementTypes = {
        // Don't want bindings that operate on text nodes to mutate <script> and <textarea> contents,
        // because it's unexpected and a potential XSS issue.
        // Also bindings should not operate on <template> elements since this breaks in Internet Explorer
        // and because such elements' contents are always intended to be bound in a different context
        // from where they appear in the document.
        'script': true,
        'textarea': true,
        'template': true
};

// Use an overridable method for retrieving binding handlers so that a plugins may support dynamically created handlers
ko['getBindingHandler'] = function(bindingKey) {
    return ko.bindingHandlers[bindingKey];
};

// The ko.bindingContext constructor is only called directly to create the root context. For child
// contexts, use bindingContext.createChildContext or bindingContext.extend.
ko.bindingContext = function(dataItemOrAccessor, parentContext, dataItemAlias, extendCallback, options) {

    // The binding context object includes static properties for the current, parent, and root view models.
    // If a view model is actually stored in an observable, the corresponding binding context object, and
    // any child contexts, must be updated when the view model is changed.
    function updateContext() {
        // Most of the time, the context will directly get a view model object, but if a function is given,
        // we call the function to retrieve the view model. If the function accesses any observables or returns
        // an observable, the dependency is tracked, and those observables can later cause the binding
        // context to be updated.
        var dataItemOrObservable = isFunc ? dataItemOrAccessor() : dataItemOrAccessor,
            dataItem = ko.utils.unwrapObservable(dataItemOrObservable);

        if (parentContext) {
            // When a "parent" context is given, register a dependency on the parent context. Thus whenever the
            // parent context is updated, this context will also be updated.
            if (parentContext._subscribable)
                parentContext._subscribable();

            // Copy $root and any custom properties from the parent context
            ko.utils.extend(self, parentContext);
```

```
            // Because the above copy overwrites our own properties, we need to reset them.
            self._subscribable = subscribable;
        } else {
            self['$parents'] = [];
            self['$root'] = dataItem;

            // Export 'ko' in the binding context so it will be available in bindings and templates
            // even if 'ko' isn't exported as a global, such as when using an AMD loader.
            // See https://github.com/SteveSanderson/knockout/issues/490
            self['ko'] = ko;
        }
        self['$rawData'] = dataItemOrObservable;
        self['$data'] = dataItem;
        if (dataItemAlias)
            self[dataItemAlias] = dataItem;

        // The extendCallback function is provided when creating a child context or extending a context.
        // It handles the specific actions needed to finish setting up the binding context. Actions in this
        // function could also add dependencies to this binding context.
        if (extendCallback)
            extendCallback(self, parentContext, dataItem);

        return self['$data'];
    }
    function disposeWhen() {
        return nodes && !ko.utils.anyDomNodeIsAttachedToDocument(nodes);
    }

    var self = this,
        isFunc = typeof(dataItemOrAccessor) == "function" && !ko.isObservable(dataItemOrAccessor),
        nodes,
        subscribable;

    if (options && options['exportDependencies']) {
        // The "exportDependencies" option means that the calling code will track any dependencies and re-create
        // the binding context when they change.
        updateContext();
    } else {
        subscribable = ko.dependentObservable(updateContext, null, { disposeWhen: disposeWhen, disposeWhenNodeIsRemoved: true });

        // At this point, the binding context has been initialized, and the "subscribable" computed observable is
        // subscribed to any observables that were accessed in the process. If there is nothing to track, the
        // computed will be inactive, and we can safely throw it away. If it's active, the computed is stored in
        // the context object.
        if (subscribable.isActive()) {
            self._subscribable = subscribable;

            // Always notify because even if the model ($data) hasn't changed, other context properties might have changed
            subscribable['equalityComparer'] = null;

            // We need to be able to dispose of this computed observable when it's no longer needed. This would be
            // easy if we had a single node to watch, but binding contexts can be used by many different nodes, and
            // we cannot assume that those nodes have any relation to each other. So instead we track any node that
            // the context is attached to, and dispose the computed when all of those nodes have been cleaned.

            // Add properties to *subscribable* instead of *self* because any properties added to *self* may be overwritten on
updates
            nodes = [];
            subscribable._addNode = function(node) {
                nodes.push(node);
                ko.utils.domNodeDisposal.addDisposeCallback(node, function(node) {
                    ko.utils.arrayRemoveItem(nodes, node);
                    if (!nodes.length) {
                        subscribable.dispose();
                        self._subscribable = subscribable = undefined;
                    }
                });
            };
        }
    }

    // Extend the binding context hierarchy with a new view model object. If the parent context is watching
    // any observables, the new child context will automatically get a dependency on the parent context.
    // But this does not mean that the $data value of the child context will also get updated. If the child
    // view model also depends on the parent view model, you must provide a function that returns the correct
    // view model on each update.
    ko.bindingContext.prototype['createChildContext'] = function (dataItemOrAccessor, dataItemAlias, extendCallback, options) {
        return new ko.bindingContext(dataItemOrAccessor, this, dataItemAlias, function(self, parentContext) {
            // Extend the context hierarchy by setting the appropriate pointers
            self['$parentContext'] = parentContext;
            self['$parent'] = parentContext['$data'];
            self['$parents'] = (parentContext['$parents'] || []).slice(0);
            self['$parents'].unshift(self['$parent']);
            if (extendCallback)
                extendCallback(self);
        }, options);
    };

    // Extend the binding context with new custom properties. This doesn't change the context hierarchy.
    // Similarly to "child" contexts, provide a function here to make sure that the correct values are set
    // when an observable view model is updated.
    ko.bindingContext.prototype['extend'] = function(properties) {
        // If the parent context references an observable view model, "_subscribable" will always be the
        // latest view model object. If not, "_subscribable" isn't set, and we can use the static "$data" value.
        return new ko.bindingContext(this._subscribable || this['$data'], this, null, function(self, parentContext) {
            // This "child" context doesn't directly track a parent observable view model,
            // so we need to manually set the $rawData value to match the parent.
            self['$rawData'] = parentContext['$rawData'];
            ko.utils.extend(self, typeof(properties) == "function" ? properties() : properties);
        });
    };
```

```
    ko.bindingContext.prototype.createStaticChildContext = function (dataItemOrAccessor, dataItemAlias) {
        return this['createChildContext'](dataItemOrAccessor, dataItemAlias, null, { "exportDependencies": true });
    };

    // Returns the valueAccesor function for a binding value
    function makeValueAccessor(value) {
        return function() {
            return value;
        };
    }

    // Returns the value of a valueAccessor function
    function evaluateValueAccessor(valueAccessor) {
        return valueAccessor();
    }

    // Given a function that returns bindings, create and return a new object that contains
    // binding value-accessors functions. Each accessor function calls the original function
    // so that it always gets the latest value and all dependencies are captured. This is used
    // by ko.applyBindingsToNode and getBindingsAndMakeAccessors.
    function makeAccessorsFromFunction(callback) {
        return ko.utils.objectMap(ko.dependencyDetection.ignore(callback), function(value, key) {
            return function() {
                return callback()[key];
            };
        });
    }

    // Given a bindings function or object, create and return a new object that contains
    // binding value-accessors functions. This is used by ko.applyBindingsToNode.
    function makeBindingAccessors(bindings, context, node) {
        if (typeof bindings === 'function') {
            return makeAccessorsFromFunction(bindings.bind(null, context, node));
        } else {
            return ko.utils.objectMap(bindings, makeValueAccessor);
        }
    }

    // This function is used if the binding provider doesn't include a getBindingAccessors function.
    // It must be called with 'this' set to the provider instance.
    function getBindingsAndMakeAccessors(node, context) {
        return makeAccessorsFromFunction(this['getBindings'].bind(this, node, context));
    }

    function validateThatBindingIsAllowedForVirtualElements(bindingName) {
        var validator = ko.virtualElements.allowedBindings[bindingName];
        if (!validator)
            throw new Error("The binding '" + bindingName + "' cannot be used with virtual elements")
    }

    function applyBindingsToDescendantsInternal (bindingContext, elementOrVirtualElement,
bindingContextsMayDifferFromDomParentElement) {
        var currentChild,
            nextInQueue = ko.virtualElements.firstChild(elementOrVirtualElement),
            provider = ko.bindingProvider['instance'],
            preprocessNode = provider['preprocessNode'];

        // Preprocessing allows a binding provider to mutate a node before bindings are applied to it. For example it's
        // possible to insert new siblings after it, and/or replace the node with a different one. This can be used to
        // implement custom binding syntaxes, such as {{ value }} for string interpolation, or custom element types that
        // trigger insertion of <template> contents at that point in the document.
        if (preprocessNode) {
            while (currentChild = nextInQueue) {
                nextInQueue = ko.virtualElements.nextSibling(currentChild);
                preprocessNode.call(provider, currentChild);
            }
            // Reset nextInQueue for the next loop
            nextInQueue = ko.virtualElements.firstChild(elementOrVirtualElement);
        }

        while (currentChild = nextInQueue) {
            // Keep a record of the next child *before* applying bindings, in case the binding removes the current child from its
position
            nextInQueue = ko.virtualElements.nextSibling(currentChild);
            applyBindingsToNodeAndDescendantsInternal(bindingContext, currentChild, bindingContextsMayDifferFromDomParentElement);
        }
    }

    function applyBindingsToNodeAndDescendantsInternal (bindingContext, nodeVerified, bindingContextMayDifferFromDomParentElement) {
        var shouldBindDescendants = true;

        // Perf optimisation: Apply bindings only if...
        // (1) We need to store the binding context on this node (because it may differ from the DOM parent node's binding context)
        //     Note that we can't store binding contexts on non-elements (e.g., text nodes), as IE doesn't allow expando properties
for those
        // (2) It might have bindings (e.g., it has a data-bind attribute, or it's a marker for a containerless template)
        var isElement = (nodeVerified.nodeType === 1);
        if (isElement) // Workaround IE <= 8 HTML parsing weirdness
            ko.virtualElements.normaliseVirtualElementDomStructure(nodeVerified);

        var shouldApplyBindings = (isElement && bindingContextMayDifferFromDomParentElement)             // Case (1)
                                || ko.bindingProvider['instance']['nodeHasBindings'](nodeVerified);       // Case (2)
        if (shouldApplyBindings)
            shouldBindDescendants = applyBindingsToNodeInternal(nodeVerified, null, bindingContext,
bindingContextMayDifferFromDomParentElement)['shouldBindDescendants'];

        if (shouldBindDescendants && !bindingDoesNotRecurseIntoElementTypes[ko.utils.tagNameLower(nodeVerified)]) {
            // We're recursing automatically into (real or virtual) child nodes without changing binding contexts. So,
            //  * For children of a *real* element, the binding context is certainly the same as on their DOM .parentNode,
            //    hence bindingContextsMayDifferFromDomParentElement is false
            //  * For children of a *virtual* element, we can't be sure. Evaluating .parentNode on those children may
            //    skip over any number of intermediate virtual elements, any of which might define a custom binding context,
            //    hence bindingContextsMayDifferFromDomParentElement is true
```

```
            applyBindingsToDescendantsInternal(bindingContext, nodeVerified, /* bindingContextsMayDifferFromDomParentElement: */
!isElement);
        }
    }

    var boundElementDomDataKey = ko.utils.domData.nextKey();


    function topologicalSortBindings(bindings) {
        // Depth-first sort
        var result = [],                // The list of key/handler pairs that we will return
            bindingsConsidered = {},    // A temporary record of which bindings are already in 'result'
            cyclicDependencyStack = []; // Keeps track of a depth-search so that, if there's a cycle, we know which bindings caused it
        ko.utils.objectForEach(bindings, function pushBinding(bindingKey) {
            if (!bindingsConsidered[bindingKey]) {
                var binding = ko['getBindingHandler'](bindingKey);
                if (binding) {
                    // First add dependencies (if any) of the current binding
                    if (binding['after']) {
                        cyclicDependencyStack.push(bindingKey);
                        ko.utils.arrayForEach(binding['after'], function(bindingDependencyKey) {
                            if (bindings[bindingDependencyKey]) {
                                if (ko.utils.arrayIndexOf(cyclicDependencyStack, bindingDependencyKey) !== -1) {
                                    throw Error("Cannot combine the following bindings, because they have a cyclic dependency: " +
cyclicDependencyStack.join(", "));
                                } else {
                                    pushBinding(bindingDependencyKey);
                                }
                            }
                        });
                        cyclicDependencyStack.length--;
                    }
                    // Next add the current binding
                    result.push({ key: bindingKey, handler: binding });
                }
                bindingsConsidered[bindingKey] = true;
            }
        });

        return result;
    }

    function applyBindingsToNodeInternal(node, sourceBindings, bindingContext, bindingContextMayDifferFromDomParentElement) {
        // Prevent multiple applyBindings calls for the same node, except when a binding value is specified
        var alreadyBound = ko.utils.domData.get(node, boundElementDomDataKey);
        if (!sourceBindings) {
            if (alreadyBound) {
                throw Error("You cannot apply bindings multiple times to the same element.");
            }
            ko.utils.domData.set(node, boundElementDomDataKey, true);
        }

        // Optimization: Don't store the binding context on this node if it's definitely the same as on node.parentNode, because
        // we can easily recover it just by scanning up the node's ancestors in the DOM
        // (note: here, parent node means "real DOM parent" not "virtual parent", as there's no O(1) way to find the virtual parent)
        if (!alreadyBound && bindingContextMayDifferFromDomParentElement)
            ko.storedBindingContextForNode(node, bindingContext);

        // Use bindings if given, otherwise fall back on asking the bindings provider to give us some bindings
        var bindings;
        if (sourceBindings && typeof sourceBindings !== 'function') {
            bindings = sourceBindings;
        } else {
            var provider = ko.bindingProvider['instance'],
                getBindings = provider['getBindingAccessors'] || getBindingsAndMakeAccessors;

            // Get the binding from the provider within a computed observable so that we can update the bindings whenever
            // the binding context is updated or if the binding provider accesses observables.
            var bindingsUpdater = ko.dependentObservable(
                function() {
                    bindings = sourceBindings ? sourceBindings(bindingContext, node) : getBindings.call(provider, node,
bindingContext);
                    // Register a dependency on the binding context to support observable view models.
                    if (bindings && bindingContext._subscribable)
                        bindingContext._subscribable();
                    return bindings;
                },
                null, { disposeWhenNodeIsRemoved: node }
            );

            if (!bindings || !bindingsUpdater.isActive())
                bindingsUpdater = null;
        }

        var bindingHandlerThatControlsDescendantBindings;
        if (bindings) {
            // Return the value accessor for a given binding. When bindings are static (won't be updated because of a binding
            // context update), just return the value accessor from the binding. Otherwise, return a function that always gets
            // the latest binding value and registers a dependency on the binding updater.
            var getValueAccessor = bindingsUpdater
                ? function(bindingKey) {
                    return function() {
                        return evaluateValueAccessor(bindingsUpdater()[bindingKey]);
                    };
                } : function(bindingKey) {
                    return bindings[bindingKey];
                };

            // Use of allBindings as a function is maintained for backwards compatibility, but its use is deprecated
            function allBindings() {
                return ko.utils.objectMap(bindingsUpdater ? bindingsUpdater() : bindings, evaluateValueAccessor);
            }
            // The following is the 3.x allBindings API
```

```
            allBindings['get'] = function(key) {
                return bindings[key] && evaluateValueAccessor(getValueAccessor(key));
            };
            allBindings['has'] = function(key) {
                return key in bindings;
            };

            // First put the bindings into the right order
            var orderedBindings = topologicalSortBindings(bindings);

            // Go through the sorted bindings, calling init and update for each
            ko.utils.arrayForEach(orderedBindings, function(bindingKeyAndHandler) {
                // Note that topologicalSortBindings has already filtered out any nonexistent binding handlers,
                // so bindingKeyAndHandler.handler will always be nonnull.
                var handlerInitFn = bindingKeyAndHandler.handler["init"],
                    handlerUpdateFn = bindingKeyAndHandler.handler["update"],
                    bindingKey = bindingKeyAndHandler.key;

                if (node.nodeType === 8) {
                    validateThatBindingIsAllowedForVirtualElements(bindingKey);
                }

                try {
                    // Run init, ignoring any dependencies
                    if (typeof handlerInitFn == "function") {
                        ko.dependencyDetection.ignore(function() {
                            var initResult = handlerInitFn(node, getValueAccessor(bindingKey), allBindings, bindingContext['$data'],
bindingContext);

                            // If this binding handler claims to control descendant bindings, make a note of this
                            if (initResult && initResult['controlsDescendantBindings']) {
                                if (bindingHandlerThatControlsDescendantBindings !== undefined)
                                    throw new Error("Multiple bindings (" + bindingHandlerThatControlsDescendantBindings + " and " +
bindingKey + ") are trying to control descendant bindings of the same element. You cannot use these bindings together on the same
element.");
                                bindingHandlerThatControlsDescendantBindings = bindingKey;
                            }
                        });
                    }

                    // Run update in its own computed wrapper
                    if (typeof handlerUpdateFn == "function") {
                        ko.dependentObservable(
                            function() {
                                handlerUpdateFn(node, getValueAccessor(bindingKey), allBindings, bindingContext['$data'],
bindingContext);
                            },
                            null,
                            { disposeWhenNodeIsRemoved: node }
                        );
                    }
                } catch (ex) {
                    ex.message = "Unable to process binding \"" + bindingKey + ": " + bindings[bindingKey] + "\"\nMessage: " +
ex.message;
                    throw ex;
                }
            });
        }

        return {
            'shouldBindDescendants': bindingHandlerThatControlsDescendantBindings === undefined
        };
    };

    var storedBindingContextDomDataKey = ko.utils.domData.nextKey();
    ko.storedBindingContextForNode = function (node, bindingContext) {
        if (arguments.length == 2) {
            ko.utils.domData.set(node, storedBindingContextDomDataKey, bindingContext);
            if (bindingContext._subscribable)
                bindingContext._subscribable._addNode(node);
        } else {
            return ko.utils.domData.get(node, storedBindingContextDomDataKey);
        }
    }

    function getBindingContext(viewModelOrBindingContext) {
        return viewModelOrBindingContext && (viewModelOrBindingContext instanceof ko.bindingContext)
            ? viewModelOrBindingContext
            : new ko.bindingContext(viewModelOrBindingContext);
    }

    ko.applyBindingAccessorsToNode = function (node, bindings, viewModelOrBindingContext) {
        if (node.nodeType === 1) // If it's an element, workaround IE <= 8 HTML parsing weirdness
            ko.virtualElements.normaliseVirtualElementDomStructure(node);
        return applyBindingsToNodeInternal(node, bindings, getBindingContext(viewModelOrBindingContext), true);
    };

    ko.applyBindingsToNode = function (node, bindings, viewModelOrBindingContext) {
        var context = getBindingContext(viewModelOrBindingContext);
        return ko.applyBindingAccessorsToNode(node, makeBindingAccessors(bindings, context, node), context);
    };

    ko.applyBindingsToDescendants = function(viewModelOrBindingContext, rootNode) {
        if (rootNode.nodeType === 1 || rootNode.nodeType === 8)
            applyBindingsToDescendantsInternal(getBindingContext(viewModelOrBindingContext), rootNode, true);
    };

    ko.applyBindings = function (viewModelOrBindingContext, rootNode) {
        // If jQuery is loaded after Knockout, we won't initially have access to it. So save it here.
        if (!jQueryInstance && window['jQuery']) {
            jQueryInstance = window['jQuery'];
        }
```

```
        if (rootNode && (rootNode.nodeType !== 1) && (rootNode.nodeType !== 8))
            throw new Error("ko.applyBindings: first parameter should be your view model; second parameter should be a DOM node");
        rootNode = rootNode || window.document.body; // Make "rootNode" parameter optional

        applyBindingsToNodeAndDescendantsInternal(getBindingContext(viewModelOrBindingContext), rootNode, true);
    };

    // Retrieving binding context from arbitrary nodes
    ko.contextFor = function(node) {
        // We can only do something meaningful for elements and comment nodes (in particular, not text nodes, as IE can't store
 domdata for them)
        switch (node.nodeType) {
            case 1:
            case 8:
                var context = ko.storedBindingContextForNode(node);
                if (context) return context;
                if (node.parentNode) return ko.contextFor(node.parentNode);
                break;
        }
        return undefined;
    };
    ko.dataFor = function(node) {
        var context = ko.contextFor(node);
        return context ? context['$data'] : undefined;
    };

    ko.exportSymbol('bindingHandlers', ko.bindingHandlers);
    ko.exportSymbol('applyBindings', ko.applyBindings);
    ko.exportSymbol('applyBindingsToDescendants', ko.applyBindingsToDescendants);
    ko.exportSymbol('applyBindingAccessorsToNode', ko.applyBindingAccessorsToNode);
    ko.exportSymbol('applyBindingsToNode', ko.applyBindingsToNode);
    ko.exportSymbol('contextFor', ko.contextFor);
    ko.exportSymbol('dataFor', ko.dataFor);
})();
(function(undefined) {
    var loadingSubscribablesCache = {}, // Tracks component loads that are currently in flight
        loadedDefinitionsCache = {};    // Tracks component loads that have already completed

    ko.components = {
        get: function(componentName, callback) {
            var cachedDefinition = getObjectOwnProperty(loadedDefinitionsCache, componentName);
            if (cachedDefinition) {
                // It's already loaded and cached. Reuse the same definition object.
                // Note that for API consistency, even cache hits complete asynchronously by default.
                // You can bypass this by putting synchronous:true on your component config.
                if (cachedDefinition.isSynchronousComponent) {
                    ko.dependencyDetection.ignore(function() { // See comment in loaderRegistryBehaviors.js for reasoning
                        callback(cachedDefinition.definition);
                    });
                } else {
                    ko.tasks.schedule(function() { callback(cachedDefinition.definition); });
                }
            } else {
                // Join the loading process that is already underway, or start a new one.
                loadComponentAndNotify(componentName, callback);
            }
        },

        clearCachedDefinition: function(componentName) {
            delete loadedDefinitionsCache[componentName];
        },

        _getFirstResultFromLoaders: getFirstResultFromLoaders
    };

    function getObjectOwnProperty(obj, propName) {
        return obj.hasOwnProperty(propName) ? obj[propName] : undefined;
    }

    function loadComponentAndNotify(componentName, callback) {
        var subscribable = getObjectOwnProperty(loadingSubscribablesCache, componentName),
            completedAsync;
        if (!subscribable) {
            // It's not started loading yet. Start loading, and when it's done, move it to loadedDefinitionsCache.
            subscribable = loadingSubscribablesCache[componentName] = new ko.subscribable();
            subscribable.subscribe(callback);

            beginLoadingComponent(componentName, function(definition, config) {
                var isSynchronousComponent = !!(config && config['synchronous']);
                loadedDefinitionsCache[componentName] = { definition: definition, isSynchronousComponent: isSynchronousComponent };
                delete loadingSubscribablesCache[componentName];

                // For API consistency, all loads complete asynchronously. However we want to avoid
                // adding an extra task schedule if it's unnecessary (i.e., the completion is already
                // async).
                //
                // You can bypass the 'always asynchronous' feature by putting the synchronous:true
                // flag on your component configuration when you register it.
                if (completedAsync || isSynchronousComponent) {
                    // Note that notifySubscribers ignores any dependencies read within the callback.
                    // See comment in loaderRegistryBehaviors.js for reasoning
                    subscribable['notifySubscribers'](definition);
                } else {
                    ko.tasks.schedule(function() {
                        subscribable['notifySubscribers'](definition);
                    });
                }
            });
            completedAsync = true;
        } else {
            subscribable.subscribe(callback);
        }
    }
```

```
    function beginLoadingComponent(componentName, callback) {
        getFirstResultFromLoaders('getConfig', [componentName], function(config) {
            if (config) {
                // We have a config, so now load its definition
                getFirstResultFromLoaders('loadComponent', [componentName, config], function(definition) {
                    callback(definition, config);
                });
            } else {
                // The component has no config - it's unknown to all the loaders.
                // Note that this is not an error (e.g., a module loading error) - that would abort the
                // process and this callback would not run. For this callback to run, all loaders must
                // have confirmed they don't know about this component.
                callback(null, null);
            }
        });
    }

    function getFirstResultFromLoaders(methodName, argsExceptCallback, callback, candidateLoaders) {
        // On the first call in the stack, start with the full set of loaders
        if (!candidateLoaders) {
            candidateLoaders = ko.components['loaders'].slice(0); // Use a copy, because we'll be mutating this array
        }

        // Try the next candidate
        var currentCandidateLoader = candidateLoaders.shift();
        if (currentCandidateLoader) {
            var methodInstance = currentCandidateLoader[methodName];
            if (methodInstance) {
                var wasAborted = false,
                    synchronousReturnValue = methodInstance.apply(currentCandidateLoader, argsExceptCallback.concat(function(result) {
                        if (wasAborted) {
                            callback(null);
                        } else if (result !== null) {
                            // This candidate returned a value. Use it.
                            callback(result);
                        } else {
                            // Try the next candidate
                            getFirstResultFromLoaders(methodName, argsExceptCallback, callback, candidateLoaders);
                        }
                    }));

                // Currently, loaders may not return anything synchronously. This leaves open the possibility
                // that we'll extend the API to support synchronous return values in the future. It won't be
                // a breaking change, because currently no loader is allowed to return anything except undefined.
                if (synchronousReturnValue !== undefined) {
                    wasAborted = true;

                    // Method to suppress exceptions will remain undocumented. This is only to keep
                    // KO's specs running tidily, since we can observe the loading got aborted without
                    // having exceptions cluttering up the console too.
                    if (!currentCandidateLoader['suppressLoaderExceptions']) {
                        throw new Error('Component loaders must supply values by invoking the callback, not by returning values
synchronously.');
                    }
                }
            } else {
                // This candidate doesn't have the relevant handler. Synchronously move on to the next one.
                getFirstResultFromLoaders(methodName, argsExceptCallback, callback, candidateLoaders);
            }
        } else {
            // No candidates returned a value
            callback(null);
        }
    }

    // Reference the loaders via string name so it's possible for developers
    // to replace the whole array by assigning to ko.components.loaders
    ko.components['loaders'] = [];

    ko.exportSymbol('components', ko.components);
    ko.exportSymbol('components.get', ko.components.get);
    ko.exportSymbol('components.clearCachedDefinition', ko.components.clearCachedDefinition);
})();
(function(undefined) {

    // The default loader is responsible for two things:
    // 1. Maintaining the default in-memory registry of component configuration objects
    //    (i.e., the thing you're writing to when you call ko.components.register(someName, ...))
    // 2. Answering requests for components by fetching configuration objects
    //    from that default in-memory registry and resolving them into standard
    //    component definition objects (of the form { createViewModel: ..., template: ... })
    // Custom loaders may override either of these facilities, i.e.,
    // 1. To supply configuration objects from some other source (e.g., conventions)
    // 2. Or, to resolve configuration objects by loading viewmodels/templates via arbitrary logic.

    var defaultConfigRegistry = {};

    ko.components.register = function(componentName, config) {
        if (!config) {
            throw new Error('Invalid configuration for ' + componentName);
        }

        if (ko.components.isRegistered(componentName)) {
            throw new Error('Component ' + componentName + ' is already registered');
        }

        defaultConfigRegistry[componentName] = config;
    };

    ko.components.isRegistered = function(componentName) {
        return defaultConfigRegistry.hasOwnProperty(componentName);
    };
```

```
    ko.components.unregister = function(componentName) {
        delete defaultConfigRegistry[componentName];
        ko.components.clearCachedDefinition(componentName);
    };

    ko.components.defaultLoader = {
        'getConfig': function(componentName, callback) {
            var result = defaultConfigRegistry.hasOwnProperty(componentName)
                ? defaultConfigRegistry[componentName]
                : null;
            callback(result);
        },

        'loadComponent': function(componentName, config, callback) {
            var errorCallback = makeErrorCallback(componentName);
            possiblyGetConfigFromAmd(errorCallback, config, function(loadedConfig) {
                resolveConfig(componentName, errorCallback, loadedConfig, callback);
            });
        },

        'loadTemplate': function(componentName, templateConfig, callback) {
            resolveTemplate(makeErrorCallback(componentName), templateConfig, callback);
        },

        'loadViewModel': function(componentName, viewModelConfig, callback) {
            resolveViewModel(makeErrorCallback(componentName), viewModelConfig, callback);
        }
    };

    var createViewModelKey = 'createViewModel';

    // Takes a config object of the form { template: ..., viewModel: ... }, and asynchronously convert it
    // into the standard component definition format:
    //    { template: <ArrayOfDomNodes>, createViewModel: function(params, componentInfo) { ... } }.
    // Since both template and viewModel may need to be resolved asynchronously, both tasks are performed
    // in parallel, and the results joined when both are ready. We don't depend on any promises infrastructure,
    // so this is implemented manually below.
    function resolveConfig(componentName, errorCallback, config, callback) {
        var result = {},
            makeCallBackWhenZero = 2,
            tryIssueCallback = function() {
                if (--makeCallBackWhenZero === 0) {
                    callback(result);
                }
            },
            templateConfig = config['template'],
            viewModelConfig = config['viewModel'];

        if (templateConfig) {
            possiblyGetConfigFromAmd(errorCallback, templateConfig, function(loadedConfig) {
                ko.components._getFirstResultFromLoaders('loadTemplate', [componentName, loadedConfig], function(resolvedTemplate) {
                    result['template'] = resolvedTemplate;
                    tryIssueCallback();
                });
            });
        } else {
            tryIssueCallback();
        }

        if (viewModelConfig) {
            possiblyGetConfigFromAmd(errorCallback, viewModelConfig, function(loadedConfig) {
                ko.components._getFirstResultFromLoaders('loadViewModel', [componentName, loadedConfig], function(resolvedViewModel) {
                    result[createViewModelKey] = resolvedViewModel;
                    tryIssueCallback();
                });
            });
        } else {
            tryIssueCallback();
        }
    }

    function resolveTemplate(errorCallback, templateConfig, callback) {
        if (typeof templateConfig === 'string') {
            // Markup - parse it
            callback(ko.utils.parseHtmlFragment(templateConfig));
        } else if (templateConfig instanceof Array) {
            // Assume already an array of DOM nodes - pass through unchanged
            callback(templateConfig);
        } else if (isDocumentFragment(templateConfig)) {
            // Document fragment - use its child nodes
            callback(ko.utils.makeArray(templateConfig.childNodes));
        } else if (templateConfig['element']) {
            var element = templateConfig['element'];
            if (isDomElement(element)) {
                // Element instance - copy its child nodes
                callback(cloneNodesFromTemplateSourceElement(element));
            } else if (typeof element === 'string') {
                // Element ID - find it, then copy its child nodes
                var elemInstance = document.getElementById(element);
                if (elemInstance) {
                    callback(cloneNodesFromTemplateSourceElement(elemInstance));
                } else {
                    errorCallback('Cannot find element with ID ' + element);
                }
            } else {
                errorCallback('Unknown element type: ' + element);
            }
        } else {
            errorCallback('Unknown template value: ' + templateConfig);
        }
    }
```

```
    function resolveViewModel(errorCallback, viewModelConfig, callback) {
        if (typeof viewModelConfig === 'function') {
            // Constructor - convert to standard factory function format
            // By design, this does *not* supply componentInfo to the constructor, as the intent is that
            // componentInfo contains non-viewmodel data (e.g., the component's element) that should only
            // be used in factory functions, not viewmodel constructors.
            callback(function (params /*, componentInfo */) {
                return new viewModelConfig(params);
            });
        } else if (typeof viewModelConfig[createViewModelKey] === 'function') {
            // Already a factory function - use it as-is
            callback(viewModelConfig[createViewModelKey]);
        } else if ('instance' in viewModelConfig) {
            // Fixed object instance - promote to createViewModel format for API consistency
            var fixedInstance = viewModelConfig['instance'];
            callback(function (params, componentInfo) {
                return fixedInstance;
            });
        } else if ('viewModel' in viewModelConfig) {
            // Resolved AMD module whose value is of the form { viewModel: ... }
            resolveViewModel(errorCallback, viewModelConfig['viewModel'], callback);
        } else {
            errorCallback('Unknown viewModel value: ' + viewModelConfig);
        }
    }

    function cloneNodesFromTemplateSourceElement(elemInstance) {
        switch (ko.utils.tagNameLower(elemInstance)) {
            case 'script':
                return ko.utils.parseHtmlFragment(elemInstance.text);
            case 'textarea':
                return ko.utils.parseHtmlFragment(elemInstance.value);
            case 'template':
                // For browsers with proper <template> element support (i.e., where the .content property
                // gives a document fragment), use that document fragment.
                if (isDocumentFragment(elemInstance.content)) {
                    return ko.utils.cloneNodes(elemInstance.content.childNodes);
                }
        }

        // Regular elements such as <div>, and <template> elements on old browsers that don't really
        // understand <template> and just treat it as a regular container
        return ko.utils.cloneNodes(elemInstance.childNodes);
    }

    function isDomElement(obj) {
        if (window['HTMLElement']) {
            return obj instanceof HTMLElement;
        } else {
            return obj && obj.tagName && obj.nodeType === 1;
        }
    }

    function isDocumentFragment(obj) {
        if (window['DocumentFragment']) {
            return obj instanceof DocumentFragment;
        } else {
            return obj && obj.nodeType === 11;
        }
    }

    function possiblyGetConfigFromAmd(errorCallback, config, callback) {
        if (typeof config['require'] === 'string') {
            // The config is the value of an AMD module
            if (amdRequire || window['require']) {
                (amdRequire || window['require'])([config['require']], callback);
            } else {
                errorCallback('Uses require, but no AMD loader is present');
            }
        } else {
            callback(config);
        }
    }

    function makeErrorCallback(componentName) {
        return function (message) {
            throw new Error('Component \'' + componentName + '\': ' + message);
        };
    }

    ko.exportSymbol('components.register', ko.components.register);
    ko.exportSymbol('components.isRegistered', ko.components.isRegistered);
    ko.exportSymbol('components.unregister', ko.components.unregister);

    // Expose the default loader so that developers can directly ask it for configuration
    // or to resolve configuration
    ko.exportSymbol('components.defaultLoader', ko.components.defaultLoader);

    // By default, the default loader is the only registered component loader
    ko.components['loaders'].push(ko.components.defaultLoader);

    // Privately expose the underlying config registry for use in old-IE shim
    ko.components._allRegisteredComponents = defaultConfigRegistry;
})();
(function (undefined) {
    // Overridable API for determining which component name applies to a given node. By overriding this,
    // you can for example map specific tagNames to components that are not preregistered.
    ko.components['getComponentNameForNode'] = function(node) {
        var tagNameLower = ko.utils.tagNameLower(node);
        if (ko.components.isRegistered(tagNameLower)) {
            // Try to determine that this node can be considered a *custom* element; see
https://github.com/knockout/knockout/issues/1603
            if (tagNameLower.indexOf('-') != -1 || ('' + node) == "[object HTMLUnknownElement]" || (ko.utils.ieVersion <= 8 &&
```

```
 node.tagName === tagNameLower)) {
                return tagNameLower;
            }
        }
    };

    ko.components.addBindingsForCustomElement = function(allBindings, node, bindingContext, valueAccessors) {
        // Determine if it's really a custom element matching a component
        if (node.nodeType === 1) {
            var componentName = ko.components['getComponentNameForNode'](node);
            if (componentName) {
                // It does represent a component, so add a component binding for it
                allBindings = allBindings || {};

                if (allBindings['component']) {
                    // Avoid silently overwriting some other 'component' binding that may already be on the element
                    throw new Error('Cannot use the "component" binding on a custom element matching a component');
                }

                var componentBindingValue = { 'name': componentName, 'params': getComponentParamsFromCustomElement(node,
 bindingContext) };

                allBindings['component'] = valueAccessors
                    ? function() { return componentBindingValue; }
                    : componentBindingValue;
            }
        }

        return allBindings;
    }

    var nativeBindingProviderInstance = new ko.bindingProvider();

    function getComponentParamsFromCustomElement(elem, bindingContext) {
        var paramsAttribute = elem.getAttribute('params');

        if (paramsAttribute) {
            var params = nativeBindingProviderInstance['parseBindingsString'](paramsAttribute, bindingContext, elem, {
'valueAccessors': true, 'bindingParams': true }),
                rawParamComputedValues = ko.utils.objectMap(params, function(paramValue, paramName) {
                    return ko.computed(paramValue, null, { disposeWhenNodeIsRemoved: elem });
                }),
                result = ko.utils.objectMap(rawParamComputedValues, function(paramValueComputed, paramName) {
                    var paramValue = paramValueComputed.peek();
                    // Does the evaluation of the parameter value unwrap any observables?
                    if (!paramValueComputed.isActive()) {
                        // No it doesn't, so there's no need for any computed wrapper. Just pass through the supplied value directly.
                        // Example: "someVal: firstName, age: 123" (whether or not firstName is an observable/computed)
                        return paramValue;
                    } else {
                        // Yes it does. Supply a computed property that unwraps both the outer (binding expression)
                        // level of observability, and any inner (resulting model value) level of observability.
                        // This means the component doesn't have to worry about multiple unwrapping. If the value is a
                        // writable observable, the computed will also be writable and pass the value on to the observable.
                        return ko.computed({
                            'read': function() {
                                return ko.utils.unwrapObservable(paramValueComputed());
                            },
                            'write': ko.isWriteableObservable(paramValue) && function(value) {
                                paramValueComputed()(value);
                            },
                            disposeWhenNodeIsRemoved: elem
                        });
                    }
                });

            // Give access to the raw computeds, as long as that wouldn't overwrite any custom param also called '$raw'
            // This is in case the developer wants to react to outer (binding) observability separately from inner
            // (model value) observability, or in case the model value observable has subobservables.
            if (!result.hasOwnProperty('$raw')) {
                result['$raw'] = rawParamComputedValues;
            }

            return result;
        } else {
            // For consistency, absence of a "params" attribute is treated the same as the presence of
            // any empty one. Otherwise component viewmodels need special code to check whether or not
            // 'params' or 'params.$raw' is null/undefined before reading subproperties, which is annoying.
            return { '$raw': {} };
        }
    }

    // ---------------------------------------------------------------------------
    // Compatibility code for older (pre-HTML5) IE browsers

    if (ko.utils.ieVersion < 9) {
        // Whenever you preregister a component, enable it as a custom element in the current document
        ko.components['register'] = (function(originalFunction) {
            return function(componentName) {
                document.createElement(componentName); // Allows IE<9 to parse markup containing the custom element
                return originalFunction.apply(this, arguments);
            }
        })(ko.components['register']);

        // Whenever you create a document fragment, enable all preregistered component names as custom elements
        // This is needed to make innerShiv/jQuery HTML parsing correctly handle the custom elements
        document.createDocumentFragment = (function(originalFunction) {
            return function() {
                var newDocFrag = originalFunction(),
                    allComponents = ko.components._allRegisteredComponents;
                for (var componentName in allComponents) {
                    if (allComponents.hasOwnProperty(componentName)) {
                        newDocFrag.createElement(componentName);
```

```
                    }
                }
                return newDocFrag;
            };
        })(document.createDocumentFragment);
    }
})();(function(undefined) {

    var componentLoadingOperationUniqueId = 0;

    ko.bindingHandlers['component'] = {
        'init': function(element, valueAccessor, ignored1, ignored2, bindingContext) {
            var currentViewModel,
                currentLoadingOperationId,
                disposeAssociatedComponentViewModel = function () {
                    var currentViewModelDispose = currentViewModel && currentViewModel['dispose'];
                    if (typeof currentViewModelDispose === 'function') {
                        currentViewModelDispose.call(currentViewModel);
                    }
                    currentViewModel = null;
                    // Any in-flight loading operation is no longer relevant, so make sure we ignore its completion
                    currentLoadingOperationId = null;
                },
                originalChildNodes = ko.utils.makeArray(ko.virtualElements.childNodes(element));

            ko.utils.domNodeDisposal.addDisposeCallback(element, disposeAssociatedComponentViewModel);

            ko.computed(function () {
                var value = ko.utils.unwrapObservable(valueAccessor()),
                    componentName, componentParams;

                if (typeof value === 'string') {
                    componentName = value;
                } else {
                    componentName = ko.utils.unwrapObservable(value['name']);
                    componentParams = ko.utils.unwrapObservable(value['params']);
                }

                if (!componentName) {
                    throw new Error('No component name specified');
                }

                var loadingOperationId = currentLoadingOperationId = ++componentLoadingOperationUniqueId;
                ko.components.get(componentName, function(componentDefinition) {
                    // If this is not the current load operation for this element, ignore it.
                    if (currentLoadingOperationId !== loadingOperationId) {
                        return;
                    }

                    // Clean up previous state
                    disposeAssociatedComponentViewModel();

                    // Instantiate and bind new component. Implicitly this cleans any old DOM nodes.
                    if (!componentDefinition) {
                        throw new Error('Unknown component \'' + componentName + '\'');
                    }
                    cloneTemplateIntoElement(componentName, componentDefinition, element);
                    var componentViewModel = createViewModel(componentDefinition, element, originalChildNodes, componentParams),
                        childBindingContext = bindingContext['createChildContext'](componentViewModel, /* dataItemAlias */ undefined,
function(ctx) {
                            ctx['$component'] = componentViewModel;
                            ctx['$componentTemplateNodes'] = originalChildNodes;
                        });
                    currentViewModel = componentViewModel;
                    ko.applyBindingsToDescendants(childBindingContext, element);
                });
            }, null, { disposeWhenNodeIsRemoved: element });

            return { 'controlsDescendantBindings': true };
        }
    };

    ko.virtualElements.allowedBindings['component'] = true;

    function cloneTemplateIntoElement(componentName, componentDefinition, element) {
        var template = componentDefinition['template'];
        if (!template) {
            throw new Error('Component \'' + componentName + '\' has no template');
        }

        var clonedNodesArray = ko.utils.cloneNodes(template);
        ko.virtualElements.setDomNodeChildren(element, clonedNodesArray);
    }

    function createViewModel(componentDefinition, element, originalChildNodes, componentParams) {
        var componentViewModelFactory = componentDefinition['createViewModel'];
        return componentViewModelFactory
            ? componentViewModelFactory.call(componentDefinition, componentParams, { 'element': element, 'templateNodes':
originalChildNodes })
            : componentParams; // Template-only component
    }

})();
var attrHtmlToJavascriptMap = { 'class': 'className', 'for': 'htmlFor' };
ko.bindingHandlers['attr'] = {
    'update': function(element, valueAccessor, allBindings) {
        var value = ko.utils.unwrapObservable(valueAccessor()) || {};
        ko.utils.objectForEach(value, function(attrName, attrValue) {
            attrValue = ko.utils.unwrapObservable(attrValue);

            // To cover cases like "attr: { checked:someProp }", we want to remove the attribute entirely
            // when someProp is a "no value"-like value (strictly null, false, or undefined)
            // (because the absence of the "checked" attr is how to mark an element as not checked, etc.)
```

```
            var toRemove = (attrValue === false) || (attrValue === null) || (attrValue === undefined);
            if (toRemove)
                element.removeAttribute(attrName);

            // In IE <= 7 and IE8 Quirks Mode, you have to use the Javascript property name instead of the
            // HTML attribute name for certain attributes. IE8 Standards Mode supports the correct behavior,
            // but instead of figuring out the mode, we'll just set the attribute through the Javascript
            // property for IE <= 8.
            if (ko.utils.ieVersion <= 8 && attrName in attrHtmlToJavascriptMap) {
                attrName = attrHtmlToJavascriptMap[attrName];
                if (toRemove)
                    element.removeAttribute(attrName);
                else
                    element[attrName] = attrValue;
            } else if (!toRemove) {
                element.setAttribute(attrName, attrValue.toString());
            }

            // Treat "name" specially - although you can think of it as an attribute, it also needs
            // special handling on older versions of IE (https://github.com/SteveSanderson/knockout/pull/333)
            // Deliberately being case-sensitive here because XHTML would regard "Name" as a different thing
            // entirely, and there's no strong reason to allow for such casing in HTML.
            if (attrName === "name") {
                ko.utils.setElementName(element, toRemove ? "" : attrValue.toString());
            }
        });
    }
};
(function() {

ko.bindingHandlers['checked'] = {
    'after': ['value', 'attr'],
    'init': function (element, valueAccessor, allBindings) {
        var checkedValue = ko.pureComputed(function() {
            // Treat "value" like "checkedValue" when it is included with "checked" binding
            if (allBindings['has']('checkedValue')) {
                return ko.utils.unwrapObservable(allBindings.get('checkedValue'));
            } else if (allBindings['has']('value')) {
                return ko.utils.unwrapObservable(allBindings.get('value'));
            }

            return element.value;
        });

        function updateModel() {
            // This updates the model value from the view value.
            // It runs in response to DOM events (click) and changes in checkedValue.
            var isChecked = element.checked,
                elemValue = useCheckedValue ? checkedValue() : isChecked;

            // When we're first setting up this computed, don't change any model state.
            if (ko.computedContext.isInitial()) {
                return;
            }

            // We can ignore unchecked radio buttons, because some other radio
            // button will be getting checked, and that one can take care of updating state.
            if (isRadio && !isChecked) {
                return;
            }

            var modelValue = ko.dependencyDetection.ignore(valueAccessor);
            if (valueIsArray) {
                var writableValue = rawValueIsNonArrayObservable ? modelValue.peek() : modelValue;
                if (oldElemValue !== elemValue) {
                    // When we're responding to the checkedValue changing, and the element is
                    // currently checked, replace the old elem value with the new elem value
                    // in the model array.
                    if (isChecked) {
                        ko.utils.addOrRemoveItem(writableValue, elemValue, true);
                        ko.utils.addOrRemoveItem(writableValue, oldElemValue, false);
                    }

                    oldElemValue = elemValue;
                } else {
                    // When we're responding to the user having checked/unchecked a checkbox,
                    // add/remove the element value to the model array.
                    ko.utils.addOrRemoveItem(writableValue, elemValue, isChecked);
                }
                if (rawValueIsNonArrayObservable && ko.isWriteableObservable(modelValue)) {
                    modelValue(writableValue);
                }
            } else {
                ko.expressionRewriting.writeValueToProperty(modelValue, allBindings, 'checked', elemValue, true);
            }
        };

        function updateView() {
            // This updates the view value from the model value.
            // It runs in response to changes in the bound (checked) value.
            var modelValue = ko.utils.unwrapObservable(valueAccessor());

            if (valueIsArray) {
                // When a checkbox is bound to an array, being checked represents its value being present in that array
                element.checked = ko.utils.arrayIndexOf(modelValue, checkedValue()) >= 0;
            } else if (isCheckbox) {
                // When a checkbox is bound to any other value (not an array), being checked represents the value being trueish
                element.checked = modelValue;
            } else {
                // For radio buttons, being checked means that the radio button's value corresponds to the model value
                element.checked = (checkedValue() === modelValue);
            }
        };
```

```
        var isCheckbox = element.type == "checkbox",
            isRadio = element.type == "radio";

        // Only bind to check boxes and radio buttons
        if (!isCheckbox && !isRadio) {
            return;
        }

        var rawValue = valueAccessor(),
            valueIsArray = isCheckbox && (ko.utils.unwrapObservable(rawValue) instanceof Array),
            rawValueIsNonArrayObservable = !(valueIsArray && rawValue.push && rawValue.splice),
            oldElemValue = valueIsArray ? checkedValue() : undefined,
            useCheckedValue = isRadio || valueIsArray;

        // IE 6 won't allow radio buttons to be selected unless they have a name
        if (isRadio && !element.name)
            ko.bindingHandlers['uniqueName']['init'](element, function() { return true });

        // Set up two computeds to update the binding:

        // The first responds to changes in the checkedValue value and to element clicks
        ko.computed(updateModel, null, { disposeWhenNodeIsRemoved: element });
        ko.utils.registerEventHandler(element, "click", updateModel);

        // The second responds to changes in the model value (the one associated with the checked binding)
        ko.computed(updateView, null, { disposeWhenNodeIsRemoved: element });

        rawValue = undefined;
    }
};
ko.expressionRewriting.twoWayBindings['checked'] = true;

ko.bindingHandlers['checkedValue'] = {
    'update': function (element, valueAccessor) {
        element.value = ko.utils.unwrapObservable(valueAccessor());
    }
};

})();var classesWrittenByBindingKey = '__ko__cssValue';
ko.bindingHandlers['css'] = {
    'update': function (element, valueAccessor) {
        var value = ko.utils.unwrapObservable(valueAccessor());
        if (value !== null && typeof value == "object") {
            ko.utils.objectForEach(value, function(className, shouldHaveClass) {
                shouldHaveClass = ko.utils.unwrapObservable(shouldHaveClass);
                ko.utils.toggleDomNodeCssClass(element, className, shouldHaveClass);
            });
        } else {
            value = ko.utils.stringTrim(String(value || '')); // Make sure we don't try to store or set a non-string value
            ko.utils.toggleDomNodeCssClass(element, element[classesWrittenByBindingKey], false);
            element[classesWrittenByBindingKey] = value;
            ko.utils.toggleDomNodeCssClass(element, value, true);
        }
    }
};
ko.bindingHandlers['enable'] = {
    'update': function (element, valueAccessor) {
        var value = ko.utils.unwrapObservable(valueAccessor());
        if (value && element.disabled)
            element.removeAttribute("disabled");
        else if ((!value) && (!element.disabled))
            element.disabled = true;
    }
};

ko.bindingHandlers['disable'] = {
    'update': function (element, valueAccessor) {
        ko.bindingHandlers['enable']['update'](element, function() { return !ko.utils.unwrapObservable(valueAccessor()) });
    }
};
// For certain common events (currently just 'click'), allow a simplified data-binding syntax
// e.g. click:handler instead of the usual full-length event:{click:handler}
function makeEventHandlerShortcut(eventName) {
    ko.bindingHandlers[eventName] = {
        'init': function(element, valueAccessor, allBindings, viewModel, bindingContext) {
            var newValueAccessor = function () {
                var result = {};
                result[eventName] = valueAccessor();
                return result;
            };
            return ko.bindingHandlers['event']['init'].call(this, element, newValueAccessor, allBindings, viewModel, bindingContext);
        }
    }
}

ko.bindingHandlers['event'] = {
    'init' : function (element, valueAccessor, allBindings, viewModel, bindingContext) {
        var eventsToHandle = valueAccessor() || {};
        ko.utils.objectForEach(eventsToHandle, function(eventName) {
            if (typeof eventName == "string") {
                ko.utils.registerEventHandler(element, eventName, function (event) {
                    var handlerReturnValue;
                    var handlerFunction = valueAccessor()[eventName];
                    if (!handlerFunction)
                        return;

                    try {
                        // Take all the event args, and prefix with the viewmodel
                        var argsForHandler = ko.utils.makeArray(arguments);
                        viewModel = bindingContext['$data'];
                        argsForHandler.unshift(viewModel);
                        handlerReturnValue = handlerFunction.apply(viewModel, argsForHandler);
```

```
                    } finally {
                        if (handlerReturnValue !== true) { // Normally we want to prevent default action. Developer can override this
be explicitly returning true.
                            if (event.preventDefault)
                                event.preventDefault();
                            else
                                event.returnValue = false;
                        }
                    }

                    var bubble = allBindings.get(eventName + 'Bubble') !== false;
                    if (!bubble) {
                        event.cancelBubble = true;
                        if (event.stopPropagation)
                            event.stopPropagation();
                    }
                });
            }
        });
    }
};
// "foreach: someExpression" is equivalent to "template: { foreach: someExpression }"
// "foreach: { data: someExpression, afterAdd: myfn }" is equivalent to "template: { foreach: someExpression, afterAdd: myfn }"
ko.bindingHandlers['foreach'] = {
    makeTemplateValueAccessor: function(valueAccessor) {
        return function() {
            var modelValue = valueAccessor(),
                unwrappedValue = ko.utils.peekObservable(modelValue);    // Unwrap without setting a dependency here

            // If unwrappedValue is the array, pass in the wrapped value on its own
            // The value will be unwrapped and tracked within the template binding
            // (See https://github.com/SteveSanderson/knockout/issues/523)
            if ((!unwrappedValue) || typeof unwrappedValue.length == "number")
                return { 'foreach': modelValue, 'templateEngine': ko.nativeTemplateEngine.instance };

            // If unwrappedValue.data is the array, preserve all relevant options and unwrap again value so we get updates
            ko.utils.unwrapObservable(modelValue);
            return {
                'foreach': unwrappedValue['data'],
                'as': unwrappedValue['as'],
                'includeDestroyed': unwrappedValue['includeDestroyed'],
                'afterAdd': unwrappedValue['afterAdd'],
                'beforeRemove': unwrappedValue['beforeRemove'],
                'afterRender': unwrappedValue['afterRender'],
                'beforeMove': unwrappedValue['beforeMove'],
                'afterMove': unwrappedValue['afterMove'],
                'templateEngine': ko.nativeTemplateEngine.instance
            };
        };
    },
    'init': function(element, valueAccessor, allBindings, viewModel, bindingContext) {
        return ko.bindingHandlers['template']['init'](element,
ko.bindingHandlers['foreach'].makeTemplateValueAccessor(valueAccessor));
    },
    'update': function(element, valueAccessor, allBindings, viewModel, bindingContext) {
        return ko.bindingHandlers['template']['update'](element,
ko.bindingHandlers['foreach'].makeTemplateValueAccessor(valueAccessor), allBindings, viewModel, bindingContext);
    }
};
ko.expressionRewriting.bindingRewriteValidators['foreach'] = false; // Can't rewrite control flow bindings
ko.virtualElements.allowedBindings['foreach'] = true;
var hasfocusUpdatingProperty = '__ko_hasfocusUpdating';
var hasfocusLastValue = '__ko_hasfocusLastValue';
ko.bindingHandlers['hasfocus'] = {
    'init': function(element, valueAccessor, allBindings) {
        var handleElementFocusChange = function(isFocused) {
            // Where possible, ignore which event was raised and determine focus state using activeElement,
            // as this avoids phantom focus/blur events raised when changing tabs in modern browsers.
            // However, not all KO-targeted browsers (Firefox 2) support activeElement. For those browsers,
            // prevent a loss of focus when changing tabs/windows by setting a flag that prevents hasfocus
            // from calling 'blur()' on the element when it loses focus.
            // Discussion at https://github.com/SteveSanderson/knockout/pull/352
            element[hasfocusUpdatingProperty] = true;
            var ownerDoc = element.ownerDocument;
            if ("activeElement" in ownerDoc) {
                var active;
                try {
                    active = ownerDoc.activeElement;
                } catch(e) {
                    // IE9 throws if you access activeElement during page load (see issue #703)
                    active = ownerDoc.body;
                }
                isFocused = (active === element);
            }
            var modelValue = valueAccessor();
            ko.expressionRewriting.writeValueToProperty(modelValue, allBindings, 'hasfocus', isFocused, true);

            //cache the latest value, so we can avoid unnecessarily calling focus/blur in the update function
            element[hasfocusLastValue] = isFocused;
            element[hasfocusUpdatingProperty] = false;
        };
        var handleElementFocusIn = handleElementFocusChange.bind(null, true);
        var handleElementFocusOut = handleElementFocusChange.bind(null, false);

        ko.utils.registerEventHandler(element, "focus", handleElementFocusIn);
        ko.utils.registerEventHandler(element, "focusin", handleElementFocusIn); // For IE
        ko.utils.registerEventHandler(element, "blur",  handleElementFocusOut);
        ko.utils.registerEventHandler(element, "focusout",  handleElementFocusOut); // For IE
    },
    'update': function(element, valueAccessor) {
        var value = !!ko.utils.unwrapObservable(valueAccessor());

        if (!element[hasfocusUpdatingProperty] && element[hasfocusLastValue] !== value) {
```

```
            value ? element.focus() : element.blur();

            // In IE, the blur method doesn't always cause the element to lose focus (for example, if the window is not in focus).
            // Setting focus to the body element does seem to be reliable in IE, but should only be used if we know that the current
            // element was focused already.
            if (!value && element[hasfocusLastValue]) {
                element.ownerDocument.body.focus();
            }

            // For IE, which doesn't reliably fire "focus" or "blur" events synchronously
            ko.dependencyDetection.ignore(ko.utils.triggerEvent, null, [element, value ? "focusin" : "focusout"]);
        }
    }
};
ko.expressionRewriting.twoWayBindings['hasfocus'] = true;

ko.bindingHandlers['hasFocus'] = ko.bindingHandlers['hasfocus']; // Make "hasFocus" an alias
ko.expressionRewriting.twoWayBindings['hasFocus'] = true;
ko.bindingHandlers['html'] = {
    'init': function() {
        // Prevent binding on the dynamically-injected HTML (as developers are unlikely to expect that, and it has security
implications)
        return { 'controlsDescendantBindings': true };
    },
    'update': function (element, valueAccessor) {
        // setHtml will unwrap the value if needed
        ko.utils.setHtml(element, valueAccessor());
    }
};
// Makes a binding like with or if
function makeWithIfBinding(bindingKey, isWith, isNot, makeContextCallback) {
    ko.bindingHandlers[bindingKey] = {
        'init': function(element, valueAccessor, allBindings, viewModel, bindingContext) {
            var didDisplayOnLastUpdate,
                savedNodes;
            ko.computed(function() {
                var rawValue = valueAccessor(),
                    dataValue = ko.utils.unwrapObservable(rawValue),
                    shouldDisplay = !isNot !== !dataValue, // equivalent to isNot ? !dataValue : !!dataValue
                    isFirstRender = !savedNodes,
                    needsRefresh = isFirstRender || isWith || (shouldDisplay !== didDisplayOnLastUpdate);

                if (needsRefresh) {
                    // Save a copy of the inner nodes on the initial update, but only if we have dependencies.
                    if (isFirstRender && ko.computedContext.getDependenciesCount()) {
                        savedNodes = ko.utils.cloneNodes(ko.virtualElements.childNodes(element), true /* shouldCleanNodes */);
                    }

                    if (shouldDisplay) {
                        if (!isFirstRender) {
                            ko.virtualElements.setDomNodeChildren(element, ko.utils.cloneNodes(savedNodes));
                        }
                        ko.applyBindingsToDescendants(makeContextCallback ? makeContextCallback(bindingContext, rawValue) :
bindingContext, element);
                    } else {
                        ko.virtualElements.emptyNode(element);
                    }

                    didDisplayOnLastUpdate = shouldDisplay;
                }
            }, null, { disposeWhenNodeIsRemoved: element });
            return { 'controlsDescendantBindings': true };
        }
    };
    ko.expressionRewriting.bindingRewriteValidators[bindingKey] = false; // Can't rewrite control flow bindings
    ko.virtualElements.allowedBindings[bindingKey] = true;
}

// Construct the actual binding handlers
makeWithIfBinding('if');
makeWithIfBinding('ifnot', false /* isWith */, true /* isNot */);
makeWithIfBinding('with', true /* isWith */, false /* isNot */,
    function(bindingContext, dataValue) {
        return bindingContext.createStaticChildContext(dataValue);
    }
);
var captionPlaceholder = {};
ko.bindingHandlers['options'] = {
    'init': function(element) {
        if (ko.utils.tagNameLower(element) !== "select")
            throw new Error("options binding applies only to SELECT elements");

        // Remove all existing <option>s.
        while (element.length > 0) {
            element.remove(0);
        }

        // Ensures that the binding processor doesn't try to bind the options
        return { 'controlsDescendantBindings': true };
    },
    'update': function (element, valueAccessor, allBindings) {
        function selectedOptions() {
            return ko.utils.arrayFilter(element.options, function (node) { return node.selected; });
        }

        var selectWasPreviouslyEmpty = element.length == 0,
            multiple = element.multiple,
            previousScrollTop = (!selectWasPreviouslyEmpty && multiple) ? element.scrollTop : null,
            unwrappedArray = ko.utils.unwrapObservable(valueAccessor()),
            valueAllowUnset = allBindings.get('valueAllowUnset') && allBindings['has']('value'),
            includeDestroyed = allBindings.get('optionsIncludeDestroyed'),
            arrayToDomNodeChildrenOptions = {},
            captionValue,
```

```
            filteredArray,
            previousSelectedValues = [];

        if (!valueAllowUnset) {
            if (multiple) {
                previousSelectedValues = ko.utils.arrayMap(selectedOptions(), ko.selectExtensions.readValue);
            } else if (element.selectedIndex >= 0) {
                previousSelectedValues.push(ko.selectExtensions.readValue(element.options[element.selectedIndex]));
            }
        }

        if (unwrappedArray) {
            if (typeof unwrappedArray.length == "undefined") // Coerce single value into array
                unwrappedArray = [unwrappedArray];

            // Filter out any entries marked as destroyed
            filteredArray = ko.utils.arrayFilter(unwrappedArray, function(item) {
                return includeDestroyed || item === undefined || item === null || !ko.utils.unwrapObservable(item['_destroy']);
            });

            // If caption is included, add it to the array
            if (allBindings['has']('optionsCaption')) {
                captionValue = ko.utils.unwrapObservable(allBindings.get('optionsCaption'));
                // If caption value is null or undefined, don't show a caption
                if (captionValue !== null && captionValue !== undefined) {
                    filteredArray.unshift(captionPlaceholder);
                }
            }
        } else {
            // If a falsy value is provided (e.g. null), we'll simply empty the select element
        }

        function applyToObject(object, predicate, defaultValue) {
            var predicateType = typeof predicate;
            if (predicateType == "function")    // Given a function; run it against the data value
                return predicate(object);
            else if (predicateType == "string") // Given a string; treat it as a property name on the data value
                return object[predicate];
            else                                // Given no optionsText arg; use the data value itself
                return defaultValue;
        }

        // The following functions can run at two different times:
        // The first is when the whole array is being updated directly from this binding handler.
        // The second is when an observable value for a specific array entry is updated.
        // oldOptions will be empty in the first case, but will be filled with the previously generated option in the second.
        var itemUpdate = false;
        function optionForArrayItem(arrayEntry, index, oldOptions) {
            if (oldOptions.length) {
                previousSelectedValues = !valueAllowUnset && oldOptions[0].selected ? [ ko.selectExtensions.readValue(oldOptions[0]) ]
: [];
                itemUpdate = true;
            }
            var option = element.ownerDocument.createElement("option");
            if (arrayEntry === captionPlaceholder) {
                ko.utils.setTextContent(option, allBindings.get('optionsCaption'));
                ko.selectExtensions.writeValue(option, undefined);
            } else {
                // Apply a value to the option element
                var optionValue = applyToObject(arrayEntry, allBindings.get('optionsValue'), arrayEntry);
                ko.selectExtensions.writeValue(option, ko.utils.unwrapObservable(optionValue));

                // Apply some text to the option element
                var optionText = applyToObject(arrayEntry, allBindings.get('optionsText'), optionValue);
                ko.utils.setTextContent(option, optionText);
            }
            return [option];
        }

        // By using a beforeRemove callback, we delay the removal until after new items are added. This fixes a selection
        // problem in IE<=8 and Firefox. See https://github.com/knockout/knockout/issues/1208
        arrayToDomNodeChildrenOptions['beforeRemove'] =
            function (option) {
                element.removeChild(option);
            };

        function setSelectionCallback(arrayEntry, newOptions) {
            if (itemUpdate && valueAllowUnset) {
                // The model value is authoritative, so make sure its value is the one selected
                // There is no need to use dependencyDetection.ignore since setDomNodeChildrenFromArrayMapping does so already.
                ko.selectExtensions.writeValue(element, ko.utils.unwrapObservable(allBindings.get('value')), true /* allowUnset */);
            } else if (previousSelectedValues.length) {
                // IE6 doesn't like us to assign selection to OPTION nodes before they're added to the document.
                // That's why we first added them without selection. Now it's time to set the selection.
                var isSelected = ko.utils.arrayIndexOf(previousSelectedValues, ko.selectExtensions.readValue(newOptions[0])) >= 0;
                ko.utils.setOptionNodeSelectionState(newOptions[0], isSelected);

                // If this option was changed from being selected during a single-item update, notify the change
                if (itemUpdate && !isSelected) {
                    ko.dependencyDetection.ignore(ko.utils.triggerEvent, null, [element, "change"]);
                }
            }
        }

        var callback = setSelectionCallback;
        if (allBindings['has']('optionsAfterRender') && typeof allBindings.get('optionsAfterRender') == "function") {
            callback = function(arrayEntry, newOptions) {
                setSelectionCallback(arrayEntry, newOptions);
                ko.dependencyDetection.ignore(allBindings.get('optionsAfterRender'), null, [newOptions[0], arrayEntry !==
captionPlaceholder ? arrayEntry : undefined]);
            }
        }
```

```
        ko.utils.setDomNodeChildrenFromArrayMapping(element, filteredArray, optionForArrayItem, arrayToDomNodeChildrenOptions,
callback);

        ko.dependencyDetection.ignore(function () {
            if (valueAllowUnset) {
                // The model value is authoritative, so make sure its value is the one selected
                ko.selectExtensions.writeValue(element, ko.utils.unwrapObservable(allBindings.get('value')), true /* allowUnset */);
            } else {
                // Determine if the selection has changed as a result of updating the options list
                var selectionChanged;
                if (multiple) {
                    // For a multiple-select box, compare the new selection count to the previous one
                    // But if nothing was selected before, the selection can't have changed
                    selectionChanged = previousSelectedValues.length && selectedOptions().length < previousSelectedValues.length;
                } else {
                    // For a single-select box, compare the current value to the previous value
                    // But if nothing was selected before or nothing is selected now, just look for a change in selection
                    selectionChanged = (previousSelectedValues.length && element.selectedIndex >= 0)
                        ? (ko.selectExtensions.readValue(element.options[element.selectedIndex]) !== previousSelectedValues[0])
                        : (previousSelectedValues.length || element.selectedIndex >= 0);
                }

                // Ensure consistency between model value and selected option.
                // If the dropdown was changed so that selection is no longer the same,
                // notify the value or selectedOptions binding.
                if (selectionChanged) {
                    ko.utils.triggerEvent(element, "change");
                }
            }
        });

        // Workaround for IE bug
        ko.utils.ensureSelectElementIsRenderedCorrectly(element);
        if (previousScrollTop && Math.abs(previousScrollTop - element.scrollTop) > 20)
            element.scrollTop = previousScrollTop;
    }
};
ko.bindingHandlers['options'].optionValueDomDataKey = ko.utils.domData.nextKey();
ko.bindingHandlers['selectedOptions'] = {
    'after': ['options', 'foreach'],
    'init': function (element, valueAccessor, allBindings) {
        ko.utils.registerEventHandler(element, "change", function () {
            var value = valueAccessor(), valueToWrite = [];
            ko.utils.arrayForEach(element.getElementsByTagName("option"), function(node) {
                if (node.selected)
                    valueToWrite.push(ko.selectExtensions.readValue(node));
            });
            ko.expressionRewriting.writeValueToProperty(value, allBindings, 'selectedOptions', valueToWrite);
        });
    },
    'update': function (element, valueAccessor) {
        if (ko.utils.tagNameLower(element) != "select")
            throw new Error("values binding applies only to SELECT elements");

        var newValue = ko.utils.unwrapObservable(valueAccessor()),
            previousScrollTop = element.scrollTop;

        if (newValue && typeof newValue.length == "number") {
            ko.utils.arrayForEach(element.getElementsByTagName("option"), function(node) {
                var isSelected = ko.utils.arrayIndexOf(newValue, ko.selectExtensions.readValue(node)) >= 0;
                if (node.selected != isSelected) {      // This check prevents flashing of the select element in IE
                    ko.utils.setOptionNodeSelectionState(node, isSelected);
                }
            });
        }

        element.scrollTop = previousScrollTop;
    }
};
ko.expressionRewriting.twoWayBindings['selectedOptions'] = true;
ko.bindingHandlers['style'] = {
    'update': function (element, valueAccessor) {
        var value = ko.utils.unwrapObservable(valueAccessor() || {});
        ko.utils.objectForEach(value, function(styleName, styleValue) {
            styleValue = ko.utils.unwrapObservable(styleValue);

            if (styleValue === null || styleValue === undefined || styleValue === false) {
                // Empty string removes the value, whereas null/undefined have no effect
                styleValue = "";
            }

            element.style[styleName] = styleValue;
        });
    }
};
ko.bindingHandlers['submit'] = {
    'init': function (element, valueAccessor, allBindings, viewModel, bindingContext) {
        if (typeof valueAccessor() != "function")
            throw new Error("The value for a submit binding must be a function");
        ko.utils.registerEventHandler(element, "submit", function (event) {
            var handlerReturnValue;
            var value = valueAccessor();
            try { handlerReturnValue = value.call(bindingContext['$data'], element); }
            finally {
                if (handlerReturnValue !== true) { // Normally we want to prevent default action. Developer can override this be
explicitly returning true.
                    if (event.preventDefault)
                        event.preventDefault();
                    else
                        event.returnValue = false;
                }
            }
```

```
        });
    }
};
ko.bindingHandlers['text'] = {
    'init': function() {
        // Prevent binding on the dynamically-injected text node (as developers are unlikely to expect that, and it has security
implications).
        // It should also make things faster, as we no longer have to consider whether the text node might be bindable.
        return { 'controlsDescendantBindings': true };
    },
    'update': function (element, valueAccessor) {
        ko.utils.setTextContent(element, valueAccessor());
    }
};
ko.virtualElements.allowedBindings['text'] = true;
(function () {

if (window && window.navigator) {
    var parseVersion = function (matches) {
        if (matches) {
            return parseFloat(matches[1]);
        }
    };

    // Detect various browser versions because some old versions don't fully support the 'input' event
    var operaVersion = window.opera && window.opera.version && parseInt(window.opera.version()),
        userAgent = window.navigator.userAgent,
        safariVersion = parseVersion(userAgent.match(/^(?:(?!chrome).)*version\/([^ ]*) safari/i)),
        firefoxVersion = parseVersion(userAgent.match(/Firefox\/([^ ]*)/));
}

// IE 8 and 9 have bugs that prevent the normal events from firing when the value changes.
// But it does fire the 'selectionchange' event on many of those, presumably because the
// cursor is moving and that counts as the selection changing. The 'selectionchange' event is
// fired at the document level only and doesn't directly indicate which element changed. We
// set up just one event handler for the document and use 'activeElement' to determine which
// element was changed.
if (ko.utils.ieVersion < 10) {
    var selectionChangeRegisteredName = ko.utils.domData.nextKey(),
        selectionChangeHandlerName = ko.utils.domData.nextKey();
    var selectionChangeHandler = function(event) {
        var target = this.activeElement,
            handler = target && ko.utils.domData.get(target, selectionChangeHandlerName);
        if (handler) {
            handler(event);
        }
    };
    var registerForSelectionChangeEvent = function (element, handler) {
        var ownerDoc = element.ownerDocument;
        if (!ko.utils.domData.get(ownerDoc, selectionChangeRegisteredName)) {
            ko.utils.domData.set(ownerDoc, selectionChangeRegisteredName, true);
            ko.utils.registerEventHandler(ownerDoc, 'selectionchange', selectionChangeHandler);
        }
        ko.utils.domData.set(element, selectionChangeHandlerName, handler);
    };
}

ko.bindingHandlers['textInput'] = {
    'init': function (element, valueAccessor, allBindings) {

        var previousElementValue = element.value,
            timeoutHandle,
            elementValueBeforeEvent;

        var updateModel = function (event) {
            clearTimeout(timeoutHandle);
            elementValueBeforeEvent = timeoutHandle = undefined;

            var elementValue = element.value;
            if (previousElementValue !== elementValue) {
                // Provide a way for tests to know exactly which event was processed
                if (DEBUG && event) element['_ko_textInputProcessedEvent'] = event.type;
                previousElementValue = elementValue;
                ko.expressionRewriting.writeValueToProperty(valueAccessor(), allBindings, 'textInput', elementValue);
            }
        };

        var deferUpdateModel = function (event) {
            if (!timeoutHandle) {
                // The elementValueBeforeEvent variable is set *only* during the brief gap between an
                // event firing and the updateModel function running. This allows us to ignore model
                // updates that are from the previous state of the element, usually due to techniques
                // such as rateLimit. Such updates, if not ignored, can cause keystrokes to be lost.
                elementValueBeforeEvent = element.value;
                var handler = DEBUG ? updateModel.bind(element, {type: event.type}) : updateModel;
                timeoutHandle = ko.utils.setTimeout(handler, 4);
            }
        };

        // IE9 will mess up the DOM if you handle events synchronously which results in DOM changes (such as other bindings);
        // so we'll make sure all updates are asynchronous
        var ieUpdateModel = ko.utils.ieVersion == 9 ? deferUpdateModel : updateModel;

        var updateView = function () {
            var modelValue = ko.utils.unwrapObservable(valueAccessor());

            if (modelValue === null || modelValue === undefined) {
                modelValue = '';
            }

            if (elementValueBeforeEvent !== undefined && modelValue === elementValueBeforeEvent) {
                ko.utils.setTimeout(updateView, 4);
                return;
```

```
        }

        // Update the element only if the element and model are different. On some browsers, updating the value
        // will move the cursor to the end of the input, which would be bad while the user is typing.
        if (element.value !== modelValue) {
            previousElementValue = modelValue;  // Make sure we ignore events (propertychange) that result from updating the value
            element.value = modelValue;
        }
    };

    var onEvent = function (event, handler) {
        ko.utils.registerEventHandler(element, event, handler);
    };

    if (DEBUG && ko.bindingHandlers['textInput']['_forceUpdateOn']) {
        // Provide a way for tests to specify exactly which events are bound
        ko.utils.arrayForEach(ko.bindingHandlers['textInput']['_forceUpdateOn'], function(eventName) {
            if (eventName.slice(0,5) == 'after') {
                onEvent(eventName.slice(5), deferUpdateModel);
            } else {
                onEvent(eventName, updateModel);
            }
        });
    } else {
        if (ko.utils.ieVersion < 10) {
            // Internet Explorer <= 8 doesn't support the 'input' event, but does include 'propertychange' that fires whenever
            // any property of an element changes. Unlike 'input', it also fires if a property is changed from JavaScript code,
            // but that's an acceptable compromise for this binding. IE 9 does support 'input', but since it doesn't fire it
            // when using autocomplete, we'll use 'propertychange' for it also.
            onEvent('propertychange', function(event) {
                if (event.propertyName === 'value') {
                    ieUpdateModel(event);
                }
            });
            if (ko.utils.ieVersion == 8) {
                // IE 8 has a bug where it fails to fire 'propertychange' on the first update following a value change from
                // JavaScript code. It also doesn't fire if you clear the entire value. To fix this, we bind to the following
                // events too.
                onEvent('keyup', updateModel);      // A single keystoke
                onEvent('keydown', updateModel);    // The first character when a key is held down
            }
            if (ko.utils.ieVersion >= 8) {
                // Internet Explorer 9 doesn't fire the 'input' event when deleting text, including using
                // the backspace, delete, or ctrl-x keys, clicking the 'x' to clear the input, dragging text
                // out of the field, and cutting or deleting text using the context menu. 'selectionchange'
                // can detect all of those except dragging text out of the field, for which we use 'dragend'.
                // These are also needed in IE8 because of the bug described above.
                registerForSelectionChangeEvent(element, ieUpdateModel);  // 'selectionchange' covers cut, paste, drop, delete,
etc.
                onEvent('dragend', deferUpdateModel);
            }
        } else {
            // All other supported browsers support the 'input' event, which fires whenever the content of the element is changed
            // through the user interface.
            onEvent('input', updateModel);

            if (safariVersion < 5 && ko.utils.tagNameLower(element) === "textarea") {
                // Safari <5 doesn't fire the 'input' event for <textarea> elements (it does fire 'textInput'
                // but only when typing). So we'll just catch as much as we can with keydown, cut, and paste.
                onEvent('keydown', deferUpdateModel);
                onEvent('paste', deferUpdateModel);
                onEvent('cut', deferUpdateModel);
            } else if (operaVersion < 11) {
                // Opera 10 doesn't always fire the 'input' event for cut, paste, undo & drop operations.
                // We can try to catch some of those using 'keydown'.
                onEvent('keydown', deferUpdateModel);
            } else if (firefoxVersion < 4.0) {
                // Firefox <= 3.6 doesn't fire the 'input' event when text is filled in through autocomplete
                onEvent('DOMAutoComplete', updateModel);
                // Firefox <=3.5 doesn't fire the 'input' event when text is dropped into the input.
                onEvent('dragdrop', updateModel);       // <3.5
                onEvent('drop', updateModel);           // 3.5
            }
        }

        // Bind to the change event so that we can catch programmatic updates of the value that fire this event.
        onEvent('change', updateModel);

        ko.computed(updateView, null, { disposeWhenNodeIsRemoved: element });
    }
};
ko.expressionRewriting.twoWayBindings['textInput'] = true;

// textinput is an alias for textInput
ko.bindingHandlers['textinput'] = {
    // preprocess is the only way to set up a full alias
    'preprocess': function (value, name, addBinding) {
        addBinding('textInput', value);
    }
};

})();ko.bindingHandlers['uniqueName'] = {
    'init': function (element, valueAccessor) {
        if (valueAccessor()) {
            var name = "ko_unique_" + (++ko.bindingHandlers['uniqueName'].currentIndex);
            ko.utils.setElementName(element, name);
        }
    }
};
ko.bindingHandlers['uniqueName'].currentIndex = 0;
```

```
ko.bindingHandlers['value'] = {
    'after': ['options', 'foreach'],
    'init': function (element, valueAccessor, allBindings) {
        // If the value binding is placed on a radio/checkbox, then just pass through to checkedValue and quit
        if (element.tagName.toLowerCase() == "input" && (element.type == "checkbox" || element.type == "radio")) {
            ko.applyBindingAccessorsToNode(element, { 'checkedValue': valueAccessor });
            return;
        }

        // Always catch "change" event; possibly other events too if asked
        var eventsToCatch = ["change"];
        var requestedEventsToCatch = allBindings.get("valueUpdate");
        var propertyChangedFired = false;
        var elementValueBeforeEvent = null;

        if (requestedEventsToCatch) {
            if (typeof requestedEventsToCatch == "string") // Allow both individual event names, and arrays of event names
                requestedEventsToCatch = [requestedEventsToCatch];
            ko.utils.arrayPushAll(eventsToCatch, requestedEventsToCatch);
            eventsToCatch = ko.utils.arrayGetDistinctValues(eventsToCatch);
        }

        var valueUpdateHandler = function() {
            elementValueBeforeEvent = null;
            propertyChangedFired = false;
            var modelValue = valueAccessor();
            var elementValue = ko.selectExtensions.readValue(element);
            ko.expressionRewriting.writeValueToProperty(modelValue, allBindings, 'value', elementValue);
        }

        // Workaround for https://github.com/SteveSanderson/knockout/issues/122
        // IE doesn't fire "change" events on textboxes if the user selects a value from its autocomplete list
        var ieAutoCompleteHackNeeded = ko.utils.ieVersion && element.tagName.toLowerCase() == "input" && element.type == "text"
                                       && element.autocomplete != "off" && (!element.form || element.form.autocomplete != "off");
        if (ieAutoCompleteHackNeeded && ko.utils.arrayIndexOf(eventsToCatch, "propertychange") == -1) {
            ko.utils.registerEventHandler(element, "propertychange", function () { propertyChangedFired = true });
            ko.utils.registerEventHandler(element, "focus", function () { propertyChangedFired = false });
            ko.utils.registerEventHandler(element, "blur", function() {
                if (propertyChangedFired) {
                    valueUpdateHandler();
                }
            });
        }

        ko.utils.arrayForEach(eventsToCatch, function(eventName) {
            // The syntax "after<eventname>" means "run the handler asynchronously after the event"
            // This is useful, for example, to catch "keydown" events after the browser has updated the control
            // (otherwise, ko.selectExtensions.readValue(this) will receive the control's value *before* the key event)
            var handler = valueUpdateHandler;
            if (ko.utils.stringStartsWith(eventName, "after")) {
                handler = function() {
                    // The elementValueBeforeEvent variable is non-null *only* during the brief gap between
                    // a keyX event firing and the valueUpdateHandler running, which is scheduled to happen
                    // at the earliest asynchronous opportunity. We store this temporary information so that
                    // if, between keyX and valueUpdateHandler, the underlying model value changes separately,
                    // we can overwrite that model value change with the value the user just typed. Otherwise,
                    // techniques like rateLimit can trigger model changes at critical moments that will
                    // override the user's inputs, causing keystrokes to be lost.
                    elementValueBeforeEvent = ko.selectExtensions.readValue(element);
                    ko.utils.setTimeout(valueUpdateHandler, 0);
                };
                eventName = eventName.substring("after".length);
            }
            ko.utils.registerEventHandler(element, eventName, handler);
        });

        var updateFromModel = function () {
            var newValue = ko.utils.unwrapObservable(valueAccessor());
            var elementValue = ko.selectExtensions.readValue(element);

            if (elementValueBeforeEvent !== null && newValue === elementValueBeforeEvent) {
                ko.utils.setTimeout(updateFromModel, 0);
                return;
            }

            var valueHasChanged = (newValue !== elementValue);

            if (valueHasChanged) {
                if (ko.utils.tagNameLower(element) === "select") {
                    var allowUnset = allBindings.get('valueAllowUnset');
                    var applyValueAction = function () {
                        ko.selectExtensions.writeValue(element, newValue, allowUnset);
                    };
                    applyValueAction();

                    if (!allowUnset && newValue !== ko.selectExtensions.readValue(element)) {
                        // If you try to set a model value that can't be represented in an already-populated dropdown, reject that
change,
                        // because you're not allowed to have a model value that disagrees with a visible UI selection.
                        ko.dependencyDetection.ignore(ko.utils.triggerEvent, null, [element, "change"]);
                    } else {
                        // Workaround for IE6 bug: It won't reliably apply values to SELECT nodes during the same execution thread
                        // right after you've changed the set of OPTION nodes on it. So for that node type, we'll schedule a second
thread
                        // to apply the value as well.
                        ko.utils.setTimeout(applyValueAction, 0);
                    }
                } else {
                    ko.selectExtensions.writeValue(element, newValue);
                }
            }
        };
```

```
            ko.computed(updateFromModel, null, { disposeWhenNodeIsRemoved: element });
        },
        'update': function() {} // Keep for backwards compatibility with code that may have wrapped value binding
    };
    ko.expressionRewriting.twoWayBindings['value'] = true;
    ko.bindingHandlers['visible'] = {
        'update': function (element, valueAccessor) {
            var value = ko.utils.unwrapObservable(valueAccessor());
            var isCurrentlyVisible = !(element.style.display == "none");
            if (value && !isCurrentlyVisible)
                element.style.display = "";
            else if ((!value) && isCurrentlyVisible)
                element.style.display = "none";
        }
    };
    // 'click' is just a shorthand for the usual full-length event:{click:handler}
    makeEventHandlerShortcut('click');
    // If you want to make a custom template engine,
    //
    // [1] Inherit from this class (like ko.nativeTemplateEngine does)
    // [2] Override 'renderTemplateSource', supplying a function with this signature:
    //
    //        function (templateSource, bindingContext, options) {
    //            // - templateSource.text() is the text of the template you should render
    //            // - bindingContext.$data is the data you should pass into the template
    //            //   - you might also want to make bindingContext.$parent, bindingContext.$parents,
    //            //     and bindingContext.$root available in the template too
    //            // - options gives you access to any other properties set on "data-bind: { template: options }"
    //            // - templateDocument is the document object of the template
    //            //
    //            // Return value: an array of DOM nodes
    //        }
    //
    // [3] Override 'createJavaScriptEvaluatorBlock', supplying a function with this signature:
    //
    //        function (script) {
    //            // Return value: Whatever syntax means "Evaluate the JavaScript statement 'script' and output the result"
    //            //               For example, the jquery.tmpl template engine converts 'someScript' to '${ someScript }'
    //        }
    //
    //     This is only necessary if you want to allow data-bind attributes to reference arbitrary template variables.
    //     If you don't want to allow that, you can set the property 'allowTemplateRewriting' to false (like ko.nativeTemplateEngine does)
    //     and then you don't need to override 'createJavaScriptEvaluatorBlock'.

    ko.templateEngine = function () { };

    ko.templateEngine.prototype['renderTemplateSource'] = function (templateSource, bindingContext, options, templateDocument) {
        throw new Error("Override renderTemplateSource");
    };

    ko.templateEngine.prototype['createJavaScriptEvaluatorBlock'] = function (script) {
        throw new Error("Override createJavaScriptEvaluatorBlock");
    };

    ko.templateEngine.prototype['makeTemplateSource'] = function(template, templateDocument) {
        // Named template
        if (typeof template == "string") {
            templateDocument = templateDocument || document;
            var elem = templateDocument.getElementById(template);
            if (!elem)
                throw new Error("Cannot find template with ID " + template);
            return new ko.templateSources.domElement(elem);
        } else if ((template.nodeType == 1) || (template.nodeType == 8)) {
            // Anonymous template
            return new ko.templateSources.anonymousTemplate(template);
        } else
            throw new Error("Unknown template type: " + template);
    };

    ko.templateEngine.prototype['renderTemplate'] = function (template, bindingContext, options, templateDocument) {
        var templateSource = this['makeTemplateSource'](template, templateDocument);
        return this['renderTemplateSource'](templateSource, bindingContext, options, templateDocument);
    };

    ko.templateEngine.prototype['isTemplateRewritten'] = function (template, templateDocument) {
        // Skip rewriting if requested
        if (this['allowTemplateRewriting'] === false)
            return true;
        return this['makeTemplateSource'](template, templateDocument)['data']("isRewritten");
    };

    ko.templateEngine.prototype['rewriteTemplate'] = function (template, rewriterCallback, templateDocument) {
        var templateSource = this['makeTemplateSource'](template, templateDocument);
        var rewritten = rewriterCallback(templateSource['text']());
        templateSource['text'](rewritten);
        templateSource['data']("isRewritten", true);
    };

    ko.exportSymbol('templateEngine', ko.templateEngine);

    ko.templateRewriting = (function () {
        var memoizeDataBindingAttributeSyntaxRegex = /(<([a-z]+\d*)(?:\s+(?!data-bind\s*=\s*)[a-z0-9\-]+(?:=(?:\"[^\"]*\"|\'[^\']*\'|
[^>]*))?)*\s+)data-bind\s*=\s*(["'])([\s\S]*?)\3/gi;
        var memoizeVirtualContainerBindingSyntaxRegex = /<!--\s*ko\b\s*([\s\S]*?)\s*-->/g;

        function validateDataBindValuesForRewriting(keyValueArray) {
            var allValidators = ko.expressionRewriting.bindingRewriteValidators;
            for (var i = 0; i < keyValueArray.length; i++) {
                var key = keyValueArray[i]['key'];
                if (allValidators.hasOwnProperty(key)) {
                    var validator = allValidators[key];

                    if (typeof validator === "function") {
```

```
                    var possibleErrorMessage = validator(keyValueArray[i]['value']);
                    if (possibleErrorMessage)
                        throw new Error(possibleErrorMessage);
                } else if (!validator) {
                    throw new Error("This template engine does not support the '" + key + "' binding within its templates");
                }
            }
        }
    }

    function constructMemoizedTagReplacement(dataBindAttributeValue, tagToRetain, nodeName, templateEngine) {
        var dataBindKeyValueArray = ko.expressionRewriting.parseObjectLiteral(dataBindAttributeValue);
        validateDataBindValuesForRewriting(dataBindKeyValueArray);
        var rewrittenDataBindAttributeValue = ko.expressionRewriting.preProcessBindings(dataBindKeyValueArray,
{'valueAccessors':true});

        // For no obvious reason, Opera fails to evaluate rewrittenDataBindAttributeValue unless it's wrapped in an additional
        // anonymous function, even though Opera's built-in debugger can evaluate it anyway. No other browser requires this
        // extra indirection.
        var applyBindingsToNextSiblingScript =
            "ko.__tr_ambtns(function($context,$element){return(function(){return{ " + rewrittenDataBindAttributeValue + " } })()},'" +
nodeName.toLowerCase() + "')";
        return templateEngine['createJavaScriptEvaluatorBlock'](applyBindingsToNextSiblingScript) + tagToRetain;
    }

    return {
        ensureTemplateIsRewritten: function (template, templateEngine, templateDocument) {
            if (!templateEngine['isTemplateRewritten'](template, templateDocument))
                templateEngine['rewriteTemplate'](template, function (htmlString) {
                    return ko.templateRewriting.memoizeBindingAttributeSyntax(htmlString, templateEngine);
                }, templateDocument);
        },

        memoizeBindingAttributeSyntax: function (htmlString, templateEngine) {
            return htmlString.replace(memoizeDataBindingAttributeSyntaxRegex, function () {
                return constructMemoizedTagReplacement(/* dataBindAttributeValue: */ arguments[4], /* tagToRetain: */ arguments[1], /*
nodeName: */ arguments[2], templateEngine);
            }).replace(memoizeVirtualContainerBindingSyntaxRegex, function() {
                return constructMemoizedTagReplacement(/* dataBindAttributeValue: */ arguments[1], /* tagToRetain: */ "<!-- ko -->",
/* nodeName: */ "#comment", templateEngine);
            });
        },

        applyMemoizedBindingsToNextSibling: function (bindings, nodeName) {
            return ko.memoization.memoize(function (domNode, bindingContext) {
                var nodeToBind = domNode.nextSibling;
                if (nodeToBind && nodeToBind.nodeName.toLowerCase() === nodeName) {
                    ko.applyBindingAccessorsToNode(nodeToBind, bindings, bindingContext);
                }
            });
        }
    }
})();


// Exported only because it has to be referenced by string lookup from within rewritten template
ko.exportSymbol('__tr_ambtns', ko.templateRewriting.applyMemoizedBindingsToNextSibling);
(function() {
    // A template source represents a read/write way of accessing a template. This is to eliminate the need for template
loading/saving
    // logic to be duplicated in every template engine (and means they can all work with anonymous templates, etc.)
    //
    // Two are provided by default:
    //  1. ko.templateSources.domElement       - reads/writes the text content of an arbitrary DOM element
    //  2. ko.templateSources.anonymousElement - uses ko.utils.domData to read/write text *associated* with the DOM element, but
    //                                           without reading/writing the actual element text content, since it will be overwritten
    //                                           with the rendered template output.
    // You can implement your own template source if you want to fetch/store templates somewhere other than in DOM elements.
    // Template sources need to have the following functions:
    //   text()                         - returns the template text from your storage location
    //   text(value)            - writes the supplied template text to your storage location
    //   data(key)                      - reads values stored using data(key, value) - see below
    //   data(key, value)       - associates "value" with this template and the key "key". Is used to store information like
"isRewritten".
    //
    // Optionally, template sources can also have the following functions:
    //   nodes()            - returns a DOM element containing the nodes of this template, where available
    //   nodes(value)       - writes the given DOM element to your storage location
    // If a DOM element is available for a given template source, template engines are encouraged to use it in preference over text()
    // for improved speed. However, all templateSources must supply text() even if they don't supply nodes().
    //
    // Once you've implemented a templateSource, make your template engine use it by subclassing whatever template engine you were
    // using and overriding "makeTemplateSource" to return an instance of your custom template source.

    ko.templateSources = {};

    // ---- ko.templateSources.domElement -----

    // template types
    var templateScript = 1,
        templateTextArea = 2,
        templateTemplate = 3,
        templateElement = 4;

    ko.templateSources.domElement = function(element) {
        this.domElement = element;

        if (element) {
            var tagNameLower = ko.utils.tagNameLower(element);
            this.templateType =
                tagNameLower === "script" ? templateScript :
                tagNameLower === "textarea" ? templateTextArea :
                    // For browsers with proper <template> element support, where the .content property gives a document fragment
```

```
                tagNameLower == "template" && element.content && element.content.nodeType === 11 ? templateTemplate :
                templateElement;
        }
    }

    ko.templateSources.domElement.prototype['text'] = function(/* valueToWrite */) {
        var elemContentsProperty = this.templateType === templateScript ? "text"
                                 : this.templateType === templateTextArea ? "value"
                                 : "innerHTML";

        if (arguments.length == 0) {
            return this.domElement[elemContentsProperty];
        } else {
            var valueToWrite = arguments[0];
            if (elemContentsProperty === "innerHTML")
                ko.utils.setHtml(this.domElement, valueToWrite);
            else
                this.domElement[elemContentsProperty] = valueToWrite;
        }
    };

    var dataDomDataPrefix = ko.utils.domData.nextKey() + "_";
    ko.templateSources.domElement.prototype['data'] = function(key /*, valueToWrite */) {
        if (arguments.length === 1) {
            return ko.utils.domData.get(this.domElement, dataDomDataPrefix + key);
        } else {
            ko.utils.domData.set(this.domElement, dataDomDataPrefix + key, arguments[1]);
        }
    };

    var templatesDomDataKey = ko.utils.domData.nextKey();
    function getTemplateDomData(element) {
        return ko.utils.domData.get(element, templatesDomDataKey) || {};
    }
    function setTemplateDomData(element, data) {
        ko.utils.domData.set(element, templatesDomDataKey, data);
    }

    ko.templateSources.domElement.prototype['nodes'] = function(/* valueToWrite */) {
        var element = this.domElement;
        if (arguments.length == 0) {
            var templateData = getTemplateDomData(element),
                containerData = templateData.containerData;
            return containerData || (
                this.templateType === templateTemplate ? element.content :
                this.templateType === templateElement ? element :
                undefined);
        } else {
            var valueToWrite = arguments[0];
            setTemplateDomData(element, {containerData: valueToWrite});
        }
    };

    // ---- ko.templateSources.anonymousTemplate -----
    // Anonymous templates are normally saved/retrieved as DOM nodes through "nodes".
    // For compatibility, you can also read "text"; it will be serialized from the nodes on demand.
    // Writing to "text" is still supported, but then the template data will not be available as DOM nodes.

    ko.templateSources.anonymousTemplate = function(element) {
        this.domElement = element;
    }
    ko.templateSources.anonymousTemplate.prototype = new ko.templateSources.domElement();
    ko.templateSources.anonymousTemplate.prototype.constructor = ko.templateSources.anonymousTemplate;
    ko.templateSources.anonymousTemplate.prototype['text'] = function(/* valueToWrite */) {
        if (arguments.length == 0) {
            var templateData = getTemplateDomData(this.domElement);
            if (templateData.textData === undefined && templateData.containerData)
                templateData.textData = templateData.containerData.innerHTML;
            return templateData.textData;
        } else {
            var valueToWrite = arguments[0];
            setTemplateDomData(this.domElement, {textData: valueToWrite});
        }
    };

    ko.exportSymbol('templateSources', ko.templateSources);
    ko.exportSymbol('templateSources.domElement', ko.templateSources.domElement);
    ko.exportSymbol('templateSources.anonymousTemplate', ko.templateSources.anonymousTemplate);
})();
(function () {
    var _templateEngine;
    ko.setTemplateEngine = function (templateEngine) {
        if ((templateEngine != undefined) && !(templateEngine instanceof ko.templateEngine))
            throw new Error("templateEngine must inherit from ko.templateEngine");
        _templateEngine = templateEngine;
    }

    function invokeForEachNodeInContinuousRange(firstNode, lastNode, action) {
        var node, nextInQueue = firstNode, firstOutOfRangeNode = ko.virtualElements.nextSibling(lastNode);
        while (nextInQueue && ((node = nextInQueue) !== firstOutOfRangeNode)) {
            nextInQueue = ko.virtualElements.nextSibling(node);
            action(node, nextInQueue);
        }
    }

    function activateBindingsOnContinuousNodeArray(continuousNodeArray, bindingContext) {
        // To be used on any nodes that have been rendered by a template and have been inserted into some parent element
        // Walks through continuousNodeArray (which *must* be continuous, i.e., an uninterrupted sequence of sibling nodes, because
        // the algorithm for walking them relies on this), and for each top-level item in the virtual-element sense,
        // (1) Does a regular "applyBindings" to associate bindingContext with this node and to activate any non-memoized bindings
        // (2) Unmemoizes any memos in the DOM subtree (e.g., to activate bindings that had been memoized during template rewriting)

        if (continuousNodeArray.length) {
```

```
            var firstNode = continuousNodeArray[0],
                lastNode = continuousNodeArray[continuousNodeArray.length - 1],
                parentNode = firstNode.parentNode,
                provider = ko.bindingProvider['instance'],
                preprocessNode = provider['preprocessNode'];

            if (preprocessNode) {
                invokeForEachNodeInContinuousRange(firstNode, lastNode, function(node, nextNodeInRange) {
                    var nodePreviousSibling = node.previousSibling;
                    var newNodes = preprocessNode.call(provider, node);
                    if (newNodes) {
                        if (node === firstNode)
                            firstNode = newNodes[0] || nextNodeInRange;
                        if (node === lastNode)
                            lastNode = newNodes[newNodes.length - 1] || nodePreviousSibling;
                    }
                });

                // Because preprocessNode can change the nodes, including the first and last nodes, update continuousNodeArray to
match.
                // We need the full set, including inner nodes, because the unmemoize step might remove the first node (and so the
real
                // first node needs to be in the array).
                continuousNodeArray.length = 0;
                if (!firstNode) { // preprocessNode might have removed all the nodes, in which case there's nothing left to do
                    return;
                }
                if (firstNode === lastNode) {
                    continuousNodeArray.push(firstNode);
                } else {
                    continuousNodeArray.push(firstNode, lastNode);
                    ko.utils.fixUpContinuousNodeArray(continuousNodeArray, parentNode);
                }
            }

            // Need to applyBindings *before* unmemoziation, because unmemoization might introduce extra nodes (that we don't want to
re-bind)
            // whereas a regular applyBindings won't introduce new memoized nodes
            invokeForEachNodeInContinuousRange(firstNode, lastNode, function(node) {
                if (node.nodeType === 1 || node.nodeType === 8)
                    ko.applyBindings(bindingContext, node);
            });
            invokeForEachNodeInContinuousRange(firstNode, lastNode, function(node) {
                if (node.nodeType === 1 || node.nodeType === 8)
                    ko.memoization.unmemoizeDomNodeAndDescendants(node, [bindingContext]);
            });

            // Make sure any changes done by applyBindings or unmemoize are reflected in the array
            ko.utils.fixUpContinuousNodeArray(continuousNodeArray, parentNode);
        }
    }

    function getFirstNodeFromPossibleArray(nodeOrNodeArray) {
        return nodeOrNodeArray.nodeType ? nodeOrNodeArray
                                        : nodeOrNodeArray.length > 0 ? nodeOrNodeArray[0]
                                        : null;
    }

    function executeTemplate(targetNodeOrNodeArray, renderMode, template, bindingContext, options) {
        options = options || {};
        var firstTargetNode = targetNodeOrNodeArray && getFirstNodeFromPossibleArray(targetNodeOrNodeArray);
        var templateDocument = (firstTargetNode || template || {}).ownerDocument;
        var templateEngineToUse = (options['templateEngine'] || _templateEngine);
        ko.templateRewriting.ensureTemplateIsRewritten(template, templateEngineToUse, templateDocument);
        var renderedNodesArray = templateEngineToUse['renderTemplate'](template, bindingContext, options, templateDocument);

        // Loosely check result is an array of DOM nodes
        if ((typeof renderedNodesArray.length != "number") || (renderedNodesArray.length > 0 && typeof renderedNodesArray[0].nodeType
!= "number"))
            throw new Error("Template engine must return an array of DOM nodes");

        var haveAddedNodesToParent = false;
        switch (renderMode) {
            case "replaceChildren":
                ko.virtualElements.setDomNodeChildren(targetNodeOrNodeArray, renderedNodesArray);
                haveAddedNodesToParent = true;
                break;
            case "replaceNode":
                ko.utils.replaceDomNodes(targetNodeOrNodeArray, renderedNodesArray);
                haveAddedNodesToParent = true;
                break;
            case "ignoreTargetNode": break;
            default:
                throw new Error("Unknown renderMode: " + renderMode);
        }

        if (haveAddedNodesToParent) {
            activateBindingsOnContinuousNodeArray(renderedNodesArray, bindingContext);
            if (options['afterRender'])
                ko.dependencyDetection.ignore(options['afterRender'], null, [renderedNodesArray, bindingContext['$data']]);
        }

        return renderedNodesArray;
    }

    function resolveTemplateName(template, data, context) {
        // The template can be specified as:
        if (ko.isObservable(template)) {
            // 1. An observable, with string value
            return template();
        } else if (typeof template === 'function') {
            // 2. A function of (data, context) returning a string
            return template(data, context);
```

```
        } else {
            // 3. A string
            return template;
        }
    }

    ko.renderTemplate = function (template, dataOrBindingContext, options, targetNodeOrNodeArray, renderMode) {
        options = options || {};
        if ((options['templateEngine'] || _templateEngine) == undefined)
            throw new Error("Set a template engine before calling renderTemplate");
        renderMode = renderMode || "replaceChildren";

        if (targetNodeOrNodeArray) {
            var firstTargetNode = getFirstNodeFromPossibleArray(targetNodeOrNodeArray);

            var whenToDispose = function () { return (!firstTargetNode) || !ko.utils.domNodeIsAttachedToDocument(firstTargetNode); };
// Passive disposal (on next evaluation)
            var activelyDisposeWhenNodeIsRemoved = (firstTargetNode && renderMode == "replaceNode") ? firstTargetNode.parentNode :
firstTargetNode;

            return ko.dependentObservable( // So the DOM is automatically updated when any dependency changes
                function () {
                    // Ensure we've got a proper binding context to work with
                    var bindingContext = (dataOrBindingContext && (dataOrBindingContext instanceof ko.bindingContext))
                        ? dataOrBindingContext
                        : new ko.bindingContext(dataOrBindingContext, null, null, null, { "exportDependencies": true });

                    var templateName = resolveTemplateName(template, bindingContext['$data'], bindingContext),
                        renderedNodesArray = executeTemplate(targetNodeOrNodeArray, renderMode, templateName, bindingContext,
options);

                    if (renderMode == "replaceNode") {
                        targetNodeOrNodeArray = renderedNodesArray;
                        firstTargetNode = getFirstNodeFromPossibleArray(targetNodeOrNodeArray);
                    }
                },
                null,
                { disposeWhen: whenToDispose, disposeWhenNodeIsRemoved: activelyDisposeWhenNodeIsRemoved }
            );
        } else {
            // We don't yet have a DOM node to evaluate, so use a memo and render the template later when there is a DOM node
            return ko.memoization.memoize(function (domNode) {
                ko.renderTemplate(template, dataOrBindingContext, options, domNode, "replaceNode");
            });
        }
    };

    ko.renderTemplateForEach = function (template, arrayOrObservableArray, options, targetNode, parentBindingContext) {
        // Since setDomNodeChildrenFromArrayMapping always calls executeTemplateForArrayItem and then
        // activateBindingsCallback for added items, we can store the binding context in the former to use in the latter.
        var arrayItemContext;

        // This will be called by setDomNodeChildrenFromArrayMapping to get the nodes to add to targetNode
        var executeTemplateForArrayItem = function (arrayValue, index) {
            // Support selecting template as a function of the data being rendered
            arrayItemContext = parentBindingContext['createChildContext'](arrayValue, options['as'], function(context) {
                context['$index'] = index;
            });

            var templateName = resolveTemplateName(template, arrayValue, arrayItemContext);
            return executeTemplate(null, "ignoreTargetNode", templateName, arrayItemContext, options);
        }

        // This will be called whenever setDomNodeChildrenFromArrayMapping has added nodes to targetNode
        var activateBindingsCallback = function(arrayValue, addedNodesArray, index) {
            activateBindingsOnContinuousNodeArray(addedNodesArray, arrayItemContext);
            if (options['afterRender'])
                options['afterRender'](addedNodesArray, arrayValue);

            // release the "cache" variable, so that it can be collected by
            // the GC when its value isn't used from within the bindings anymore.
            arrayItemContext = null;
        };

        return ko.dependentObservable(function () {
            var unwrappedArray = ko.utils.unwrapObservable(arrayOrObservableArray) || [];
            if (typeof unwrappedArray.length == "undefined") // Coerce single value into array
                unwrappedArray = [unwrappedArray];

            // Filter out any entries marked as destroyed
            var filteredArray = ko.utils.arrayFilter(unwrappedArray, function(item) {
                return options['includeDestroyed'] || item === undefined || item === null ||
!ko.utils.unwrapObservable(item['_destroy']);
            });

            // Call setDomNodeChildrenFromArrayMapping, ignoring any observables unwrapped within (most likely from a callback
function).
            // If the array items are observables, though, they will be unwrapped in executeTemplateForArrayItem and managed within
setDomNodeChildrenFromArrayMapping.
            ko.dependencyDetection.ignore(ko.utils.setDomNodeChildrenFromArrayMapping, null, [targetNode, filteredArray,
executeTemplateForArrayItem, options, activateBindingsCallback]);

        }, null, { disposeWhenNodeIsRemoved: targetNode });
    };

    var templateComputedDomDataKey = ko.utils.domData.nextKey();
    function disposeOldComputedAndStoreNewOne(element, newComputed) {
        var oldComputed = ko.utils.domData.get(element, templateComputedDomDataKey);
        if (oldComputed && (typeof(oldComputed.dispose) == 'function'))
            oldComputed.dispose();
        ko.utils.domData.set(element, templateComputedDomDataKey, (newComputed && newComputed.isActive()) ? newComputed : undefined);
    }
```

```
    ko.bindingHandlers['template'] = {
        'init': function(element, valueAccessor) {
            // Support anonymous templates
            var bindingValue = ko.utils.unwrapObservable(valueAccessor());
            if (typeof bindingValue == "string" || bindingValue['name']) {
                // It's a named template - clear the element
                ko.virtualElements.emptyNode(element);
            } else if ('nodes' in bindingValue) {
                // We've been given an array of DOM nodes. Save them as the template source.
                // There is no known use case for the node array being an observable array (if the output
                // varies, put that behavior *into* your template - that's what templates are for), and
                // the implementation would be a mess, so assert that it's not observable.
                var nodes = bindingValue['nodes'] || [];
                if (ko.isObservable(nodes)) {
                    throw new Error('The "nodes" option must be a plain, non-observable array.');
                }
                var container = ko.utils.moveCleanedNodesToContainerElement(nodes); // This also removes the nodes from their current
parent
                new ko.templateSources.anonymousTemplate(element)['nodes'](container);
            } else {
                // It's an anonymous template - store the element contents, then clear the element
                var templateNodes = ko.virtualElements.childNodes(element),
                    container = ko.utils.moveCleanedNodesToContainerElement(templateNodes); // This also removes the nodes from their
current parent
                new ko.templateSources.anonymousTemplate(element)['nodes'](container);
            }
            return { 'controlsDescendantBindings': true };
        },
        'update': function (element, valueAccessor, allBindings, viewModel, bindingContext) {
            var value = valueAccessor(),
                options = ko.utils.unwrapObservable(value),
                shouldDisplay = true,
                templateComputed = null,
                templateName;

            if (typeof options == "string") {
                templateName = value;
                options = {};
            } else {
                templateName = options['name'];

                // Support "if"/"ifnot" conditions
                if ('if' in options)
                    shouldDisplay = ko.utils.unwrapObservable(options['if']);
                if (shouldDisplay && 'ifnot' in options)
                    shouldDisplay = !ko.utils.unwrapObservable(options['ifnot']);
            }

            if ('foreach' in options) {
                // Render once for each data point (treating data set as empty if shouldDisplay==false)
                var dataArray = (shouldDisplay && options['foreach']) || [];
                templateComputed = ko.renderTemplateForEach(templateName || element, dataArray, options, element, bindingContext);
            } else if (!shouldDisplay) {
                ko.virtualElements.emptyNode(element);
            } else {
                // Render once for this single data point (or use the viewModel if no data was provided)
                var innerBindingContext = ('data' in options) ?
                    bindingContext.createStaticChildContext(options['data'], options['as']) :  // Given an explitit 'data' value, we
create a child binding context for it
                    bindingContext;                                                           // Given no explicit 'data' value, we
retain the same binding context
                templateComputed = ko.renderTemplate(templateName || element, innerBindingContext, options, element);
            }

            // It only makes sense to have a single template computed per element (otherwise which one should have its output
displayed?)
            disposeOldComputedAndStoreNewOne(element, templateComputed);
        }
    };

    // Anonymous templates can't be rewritten. Give a nice error message if you try to do it.
    ko.expressionRewriting.bindingRewriteValidators['template'] = function(bindingValue) {
        var parsedBindingValue = ko.expressionRewriting.parseObjectLiteral(bindingValue);

        if ((parsedBindingValue.length == 1) && parsedBindingValue[0]['unknown'])
            return null; // It looks like a string literal, not an object literal, so treat it as a named template (which is allowed
for rewriting)

        if (ko.expressionRewriting.keyValueArrayContainsKey(parsedBindingValue, "name"))
            return null; // Named templates can be rewritten, so return "no error"
        return "This template engine does not support anonymous templates nested within its templates";
    };

    ko.virtualElements.allowedBindings['template'] = true;
})();

ko.exportSymbol('setTemplateEngine', ko.setTemplateEngine);
ko.exportSymbol('renderTemplate', ko.renderTemplate);
// Go through the items that have been added and deleted and try to find matches between them.
ko.utils.findMovesInArrayComparison = function (left, right, limitFailedCompares) {
    if (left.length && right.length) {
        var failedCompares, l, r, leftItem, rightItem;
        for (failedCompares = l = 0; (!limitFailedCompares || failedCompares < limitFailedCompares) && (leftItem = left[l]); ++l) {
            for (r = 0; rightItem = right[r]; ++r) {
                if (leftItem['value'] === rightItem['value']) {
                    leftItem['moved'] = rightItem['index'];
                    rightItem['moved'] = leftItem['index'];
                    right.splice(r, 1);         // This item is marked as moved; so remove it from right list
                    failedCompares = r = 0;     // Reset failed compares count because we're checking for consecutive failures
                    break;
                }
            }
            failedCompares += r;
```

```
        }
    }
};

ko.utils.compareArrays = (function () {
    var statusNotInOld = 'added', statusNotInNew = 'deleted';

    // Simple calculation based on Levenshtein distance.
    function compareArrays(oldArray, newArray, options) {
        // For backward compatibility, if the third arg is actually a bool, interpret
        // it as the old parameter 'dontLimitMoves'. Newer code should use { dontLimitMoves: true }.
        options = (typeof options === 'boolean') ? { 'dontLimitMoves': options } : (options || {});
        oldArray = oldArray || [];
        newArray = newArray || [];

        if (oldArray.length < newArray.length)
            return compareSmallArrayToBigArray(oldArray, newArray, statusNotInOld, statusNotInNew, options);
        else
            return compareSmallArrayToBigArray(newArray, oldArray, statusNotInNew, statusNotInOld, options);
    }

    function compareSmallArrayToBigArray(smlArray, bigArray, statusNotInSml, statusNotInBig, options) {
        var myMin = Math.min,
            myMax = Math.max,
            editDistanceMatrix = [],
            smlIndex, smlIndexMax = smlArray.length,
            bigIndex, bigIndexMax = bigArray.length,
            compareRange = (bigIndexMax - smlIndexMax) || 1,
            maxDistance = smlIndexMax + bigIndexMax + 1,
            thisRow, lastRow,
            bigIndexMaxForRow, bigIndexMinForRow;

        for (smlIndex = 0; smlIndex <= smlIndexMax; smlIndex++) {
            lastRow = thisRow;
            editDistanceMatrix.push(thisRow = []);
            bigIndexMaxForRow = myMin(bigIndexMax, smlIndex + compareRange);
            bigIndexMinForRow = myMax(0, smlIndex - 1);
            for (bigIndex = bigIndexMinForRow; bigIndex <= bigIndexMaxForRow; bigIndex++) {
                if (!bigIndex)
                    thisRow[bigIndex] = smlIndex + 1;
                else if (!smlIndex)  // Top row - transform empty array into new array via additions
                    thisRow[bigIndex] = bigIndex + 1;
                else if (smlArray[smlIndex - 1] === bigArray[bigIndex - 1])
                    thisRow[bigIndex] = lastRow[bigIndex - 1];                  // copy value (no edit)
                else {
                    var northDistance = lastRow[bigIndex] || maxDistance;       // not in big (deletion)
                    var westDistance = thisRow[bigIndex - 1] || maxDistance;    // not in small (addition)
                    thisRow[bigIndex] = myMin(northDistance, westDistance) + 1;
                }
            }
        }

        var editScript = [], meMinusOne, notInSml = [], notInBig = [];
        for (smlIndex = smlIndexMax, bigIndex = bigIndexMax; smlIndex || bigIndex;) {
            meMinusOne = editDistanceMatrix[smlIndex][bigIndex] - 1;
            if (bigIndex && meMinusOne === editDistanceMatrix[smlIndex][bigIndex-1]) {
                notInSml.push(editScript[editScript.length] = {     // added
                    'status': statusNotInSml,
                    'value': bigArray[--bigIndex],
                    'index': bigIndex });
            } else if (smlIndex && meMinusOne === editDistanceMatrix[smlIndex - 1][bigIndex]) {
                notInBig.push(editScript[editScript.length] = {     // deleted
                    'status': statusNotInBig,
                    'value': smlArray[--smlIndex],
                    'index': smlIndex });
            } else {
                --bigIndex;
                --smlIndex;
                if (!options['sparse']) {
                    editScript.push({
                        'status': "retained",
                        'value': bigArray[bigIndex] });
                }
            }
        }

        // Set a limit on the number of consecutive non-matching comparisons; having it a multiple of
        // smlIndexMax keeps the time complexity of this algorithm linear.
        ko.utils.findMovesInArrayComparison(notInBig, notInSml, !options['dontLimitMoves'] && smlIndexMax * 10);

        return editScript.reverse();
    }

    return compareArrays;
})();

ko.exportSymbol('utils.compareArrays', ko.utils.compareArrays);
(function () {
    // Objective:
    // * Given an input array, a container DOM node, and a function from array elements to arrays of DOM nodes,
    //   map the array elements to arrays of DOM nodes, concatenate together all these arrays, and use them to populate the container
DOM node
    // * Next time we're given the same combination of things (with the array possibly having mutated), update the container DOM node
    //   so that its children is again the concatenation of the mappings of the array elements, but don't re-map any array elements
that we
    //   previously mapped - retain those nodes, and just insert/delete other ones

    // "callbackAfterAddingNodes" will be invoked after any "mapping"-generated nodes are inserted into the container node
    // You can use this, for example, to activate bindings on those nodes.

    function mapNodeAndRefreshWhenChanged(containerNode, mapping, valueToMap, callbackAfterAddingNodes, index) {
        // Map this array value inside a dependentObservable so we re-map when any dependency changes
        var mappedNodes = [];
```

```
        var dependentObservable = ko.dependentObservable(function() {
            var newMappedNodes = mapping(valueToMap, index, ko.utils.fixUpContinuousNodeArray(mappedNodes, containerNode)) || [];

            // On subsequent evaluations, just replace the previously-inserted DOM nodes
            if (mappedNodes.length > 0) {
                ko.utils.replaceDomNodes(mappedNodes, newMappedNodes);
                if (callbackAfterAddingNodes)
                    ko.dependencyDetection.ignore(callbackAfterAddingNodes, null, [valueToMap, newMappedNodes, index]);
            }

            // Replace the contents of the mappedNodes array, thereby updating the record
            // of which nodes would be deleted if valueToMap was itself later removed
            mappedNodes.length = 0;
            ko.utils.arrayPushAll(mappedNodes, newMappedNodes);
        }, null, { disposeWhenNodeIsRemoved: containerNode, disposeWhen: function() { return
!ko.utils.anyDomNodeIsAttachedToDocument(mappedNodes); } });
        return { mappedNodes : mappedNodes, dependentObservable : (dependentObservable.isActive() ? dependentObservable : undefined)
};
    }

    var lastMappingResultDomDataKey = ko.utils.domData.nextKey(),
        deletedItemDummyValue = ko.utils.domData.nextKey();

    ko.utils.setDomNodeChildrenFromArrayMapping = function (domNode, array, mapping, options, callbackAfterAddingNodes) {
        // Compare the provided array against the previous one
        array = array || [];
        options = options || {};
        var isFirstExecution = ko.utils.domData.get(domNode, lastMappingResultDomDataKey) === undefined;
        var lastMappingResult = ko.utils.domData.get(domNode, lastMappingResultDomDataKey) || [];
        var lastArray = ko.utils.arrayMap(lastMappingResult, function (x) { return x.arrayEntry; });
        var editScript = ko.utils.compareArrays(lastArray, array, options['dontLimitMoves']);

        // Build the new mapping result
        var newMappingResult = [];
        var lastMappingResultIndex = 0;
        var newMappingResultIndex = 0;

        var nodesToDelete = [];
        var itemsToProcess = [];
        var itemsForBeforeRemoveCallbacks = [];
        var itemsForMoveCallbacks = [];
        var itemsForAfterAddCallbacks = [];
        var mapData;

        function itemMovedOrRetained(editScriptIndex, oldPosition) {
            mapData = lastMappingResult[oldPosition];
            if (newMappingResultIndex !== oldPosition)
                itemsForMoveCallbacks[editScriptIndex] = mapData;
            // Since updating the index might change the nodes, do so before calling fixUpContinuousNodeArray
            mapData.indexObservable(newMappingResultIndex++);
            ko.utils.fixUpContinuousNodeArray(mapData.mappedNodes, domNode);
            newMappingResult.push(mapData);
            itemsToProcess.push(mapData);
        }

        function callCallback(callback, items) {
            if (callback) {
                for (var i = 0, n = items.length; i < n; i++) {
                    if (items[i]) {
                        ko.utils.arrayForEach(items[i].mappedNodes, function(node) {
                            callback(node, i, items[i].arrayEntry);
                        });
                    }
                }
            }
        }

        for (var i = 0, editScriptItem, movedIndex; editScriptItem = editScript[i]; i++) {
            movedIndex = editScriptItem['moved'];
            switch (editScriptItem['status']) {
                case "deleted":
                    if (movedIndex === undefined) {
                        mapData = lastMappingResult[lastMappingResultIndex];

                        // Stop tracking changes to the mapping for these nodes
                        if (mapData.dependentObservable) {
                            mapData.dependentObservable.dispose();
                            mapData.dependentObservable = undefined;
                        }

                        // Queue these nodes for later removal
                        if (ko.utils.fixUpContinuousNodeArray(mapData.mappedNodes, domNode).length) {
                            if (options['beforeRemove']) {
                                newMappingResult.push(mapData);
                                itemsToProcess.push(mapData);
                                if (mapData.arrayEntry === deletedItemDummyValue) {
                                    mapData = null;
                                } else {
                                    itemsForBeforeRemoveCallbacks[i] = mapData;
                                }
                            }
                            if (mapData) {
                                nodesToDelete.push.apply(nodesToDelete, mapData.mappedNodes);
                            }
                        }
                    }
                    lastMappingResultIndex++;
                    break;

                case "retained":
                    itemMovedOrRetained(i, lastMappingResultIndex++);
                    break;
```

```
                case "added":
                    if (movedIndex !== undefined) {
                        itemMovedOrRetained(i, movedIndex);
                    } else {
                        mapData = { arrayEntry: editScriptItem['value'], indexObservable: ko.observable(newMappingResultIndex++) };
                        newMappingResult.push(mapData);
                        itemsToProcess.push(mapData);
                        if (!isFirstExecution)
                            itemsForAfterAddCallbacks[i] = mapData;
                    }
                    break;
            }
        }

        // Store a copy of the array items we just considered so we can difference it next time
        ko.utils.domData.set(domNode, lastMappingResultDomDataKey, newMappingResult);

        // Call beforeMove first before any changes have been made to the DOM
        callCallback(options['beforeMove'], itemsForMoveCallbacks);

        // Next remove nodes for deleted items (or just clean if there's a beforeRemove callback)
        ko.utils.arrayForEach(nodesToDelete, options['beforeRemove'] ? ko.cleanNode : ko.removeNode);

        // Next add/reorder the remaining items (will include deleted items if there's a beforeRemove callback)
        for (var i = 0, nextNode = ko.virtualElements.firstChild(domNode), lastNode, node; mapData = itemsToProcess[i]; i++) {
            // Get nodes for newly added items
            if (!mapData.mappedNodes)
                ko.utils.extend(mapData, mapNodeAndRefreshWhenChanged(domNode, mapping, mapData.arrayEntry, callbackAfterAddingNodes,
 mapData.indexObservable));

            // Put nodes in the right place if they aren't there already
            for (var j = 0; node = mapData.mappedNodes[j]; nextNode = node.nextSibling, lastNode = node, j++) {
                if (node !== nextNode)
                    ko.virtualElements.insertAfter(domNode, node, lastNode);
            }

            // Run the callbacks for newly added nodes (for example, to apply bindings, etc.)
            if (!mapData.initialized && callbackAfterAddingNodes) {
                callbackAfterAddingNodes(mapData.arrayEntry, mapData.mappedNodes, mapData.indexObservable);
                mapData.initialized = true;
            }
        }

        // If there's a beforeRemove callback, call it after reordering.
        // Note that we assume that the beforeRemove callback will usually be used to remove the nodes using
        // some sort of animation, which is why we first reorder the nodes that will be removed. If the
        // callback instead removes the nodes right away, it would be more efficient to skip reordering them.
        // Perhaps we'll make that change in the future if this scenario becomes more common.
        callCallback(options['beforeRemove'], itemsForBeforeRemoveCallbacks);

        // Replace the stored values of deleted items with a dummy value. This provides two benefits: it marks this item
        // as already "removed" so we won't call beforeRemove for it again, and it ensures that the item won't match up
        // with an actual item in the array and appear as "retained" or "moved".
        for (i = 0; i < itemsForBeforeRemoveCallbacks.length; ++i) {
            if (itemsForBeforeRemoveCallbacks[i]) {
                itemsForBeforeRemoveCallbacks[i].arrayEntry = deletedItemDummyValue;
            }
        }

        // Finally call afterMove and afterAdd callbacks
        callCallback(options['afterMove'], itemsForMoveCallbacks);
        callCallback(options['afterAdd'], itemsForAfterAddCallbacks);
    }
})();

ko.exportSymbol('utils.setDomNodeChildrenFromArrayMapping', ko.utils.setDomNodeChildrenFromArrayMapping);
ko.nativeTemplateEngine = function () {
    this['allowTemplateRewriting'] = false;
}

ko.nativeTemplateEngine.prototype = new ko.templateEngine();
ko.nativeTemplateEngine.prototype.constructor = ko.nativeTemplateEngine;
ko.nativeTemplateEngine.prototype['renderTemplateSource'] = function (templateSource, bindingContext, options, templateDocument) {
    var useNodesIfAvailable = !(ko.utils.ieVersion < 9), // IE<9 cloneNode doesn't work properly
        templateNodesFunc = useNodesIfAvailable ? templateSource['nodes'] : null,
        templateNodes = templateNodesFunc ? templateSource['nodes']() : null;

    if (templateNodes) {
        return ko.utils.makeArray(templateNodes.cloneNode(true).childNodes);
    } else {
        var templateText = templateSource['text']();
        return ko.utils.parseHtmlFragment(templateText, templateDocument);
    }
};

ko.nativeTemplateEngine.instance = new ko.nativeTemplateEngine();
ko.setTemplateEngine(ko.nativeTemplateEngine.instance);

ko.exportSymbol('nativeTemplateEngine', ko.nativeTemplateEngine);
(function() {
    ko.jqueryTmplTemplateEngine = function () {
        // Detect which version of jquery-tmpl you're using. Unfortunately jquery-tmpl
        // doesn't expose a version number, so we have to infer it.
        // Note that as of Knockout 1.3, we only support jQuery.tmpl 1.0.0pre and later,
        // which KO internally refers to as version "2", so older versions are no longer detected.
        var jQueryTmplVersion = this.jQueryTmplVersion = (function() {
            if (!jQueryInstance || !(jQueryInstance['tmpl']))
                return 0;
            // Since it exposes no official version number, we use our own numbering system. To be updated as jquery-tmpl evolves.
            try {
                if (jQueryInstance['tmpl']['tag']['tmpl']['open'].toString().indexOf('__') >= 0) {
                    // Since 1.0.0pre, custom tags should append markup to an array called "__"
                    return 2; // Final version of jquery.tmpl
```

```
            }
        } catch(ex) { /* Apparently not the version we were looking for */ }

        return 1; // Any older version that we don't support
    })();

    function ensureHasReferencedJQueryTemplates() {
        if (jQueryTmplVersion < 2)
            throw new Error("Your version of jQuery.tmpl is too old. Please upgrade to jQuery.tmpl 1.0.0pre or later.");
    }

    function executeTemplate(compiledTemplate, data, jQueryTemplateOptions) {
        return jQueryInstance['tmpl'](compiledTemplate, data, jQueryTemplateOptions);
    }

    this['renderTemplateSource'] = function(templateSource, bindingContext, options, templateDocument) {
        templateDocument = templateDocument || document;
        options = options || {};
        ensureHasReferencedJQueryTemplates();

        // Ensure we have stored a precompiled version of this template (don't want to reparse on every render)
        var precompiled = templateSource['data']('precompiled');
        if (!precompiled) {
            var templateText = templateSource['text']() || "";
            // Wrap in "with($whatever.koBindingContext) { ... }"
            templateText = "{{ko_with $item.koBindingContext}}" + templateText + "{{/ko_with}}";

            precompiled = jQueryInstance['template'](null, templateText);
            templateSource['data']('precompiled', precompiled);
        }

        var data = [bindingContext['$data']]; // Prewrap the data in an array to stop jquery.tmpl from trying to unwrap any arrays
        var jQueryTemplateOptions = jQueryInstance['extend']({ 'koBindingContext': bindingContext }, options['templateOptions']);
            var resultNodes = executeTemplate(precompiled, data, jQueryTemplateOptions);
            resultNodes['appendTo'](templateDocument.createElement("div")); // Using "appendTo" forces jQuery/jQuery.tmpl to perform
necessary cleanup work

            jQueryInstance['fragments'] = {}; // Clear jQuery's fragment cache to avoid a memory leak after a large number of template
renders
            return resultNodes;
        };

        this['createJavaScriptEvaluatorBlock'] = function(script) {
            return "{{ko_code ((function() { return " + script + " })()) }}";
        };

        this['addTemplate'] = function(templateName, templateMarkup) {
            document.write("<script type='text/html' id='" + templateName + "'>" + templateMarkup + "<" + "/script>");
        };

        if (jQueryTmplVersion > 0) {
            jQueryInstance['tmpl']['tag']['ko_code'] = {
                open: "__.push($1 || '');"
            };
            jQueryInstance['tmpl']['tag']['ko_with'] = {
                open: "with($1) {",
                close: "} "
            };
        }
    };

    ko.jqueryTmplTemplateEngine.prototype = new ko.templateEngine();
    ko.jqueryTmplTemplateEngine.prototype.constructor = ko.jqueryTmplTemplateEngine;

    // Use this one by default *only if jquery.tmpl is referenced*
    var jqueryTmplTemplateEngineInstance = new ko.jqueryTmplTemplateEngine();
    if (jqueryTmplTemplateEngineInstance.jQueryTmplVersion > 0)
        ko.setTemplateEngine(jqueryTmplTemplateEngineInstance);

    ko.exportSymbol('jqueryTmplTemplateEngine', ko.jqueryTmplTemplateEngine);
})();
}));
}());
})();
```