

OBJECT-ORIENTED PROGRAMMING (OOP)

Gary Dotzlaw
Dotzlaw Consulting

(9:20am – 10:10am)

About Me

- ▶ Doing software development for 25+ years
- ▶ Worked with 100+ clients around the world, all kinds of projects in all kinds of industries
- ▶ Extensive experience with large commercial projects
- ▶ Lots of Servoy tutorials at www.dotzlaw.com

Advantages of OOP

- ▶ Object oriented coding provides abstraction and encapsulation
- ▶ Build new objects from existing objects
- ▶ Modularized code; all your code is in one place
- ▶ Maintainable code; the code is easy to read and understand
- ▶ Extendable code; it is easy to add additional functionality

Primitive Types

- ▶ Boolean – True or False
- ▶ Number – Any integer or floating-point numeric value
- ▶ String – A character(s) delimited by single or double quotes
- ▶ Null – A primitive type that has only one value – null
- ▶ Undefined – A primitive type that only has one value – undefined (assigned to variables that are not initialized)

Built-in Objects (non-Primitives)

- ▶ Object – either literal/constructed form
- ▶ Function – either literal/constructed form
- ▶ Array – either literal/constructed form

- ▶ Date – only created using the constructed object form (`new Date()`)
- ▶ RegExp – only created using the constructed object form (`new RegExp()`)
- ▶ Error – only created using the constructed object form (`new Error()`)

What is an Object?

- ▶ An unordered list of properties (String) and a value
- ▶ If you need to support numbers or special characters in your property name, then wrap the property name in quotes
- ▶ The value of a property can be a primitive, or non-primitive, including another object, or a function (called a method)

Object	
name1	value1
name2	value2

Creating an Object

- ▶ Declarative (literal) syntax (preferred)

```
var oCar1 = {  
  tires      : 4,  
  passengers : 5,  
  color      : "Red"  
}
```

- ▶ Constructed form

```
var oCar2 = new Object();  
oCar2.tires = 4;  
oCar2.passengers = 5;  
oCar2.color = "Red";
```

- ▶ Both result in exactly the same sort of object, with key/value pairs

Dereferencing Objects

- ▶ Although JavaScript is a garbage-collected language, it is best to dereference objects you no longer need, so that the garbage collector can free up that memory
- ▶ The best way to dereference the object is to set it to null

```
var oCar2 = new Object();  
oCar2.tires = 4;  
oCar2.passengers = 5;  
oCar2.color = "Red";  
  
application.output("The car is " + oCar2.color); // Red  
  
oCar2 = null;  
application.output("The car is " + oCar2.color); // Error
```


Referencing Object Properties

- ▶ The “.” operator, referred to as “property access”
- ▶ The [] operator, referred to as “key access” (bracket syntax)
- ▶ JavaScript searches the object hierarchy (undefined if not found)

```
var oCar = {  
  tires      : 4,  
  passengers : 5,  
  color      : "Red"  
}  
  
// Property access  
application.output("The car is " + oCar.color); // Red
```

```
var oCar = {  
  tires      : 4,  
  passengers : 5,  
  color      : "Red",  
  model      : "Ford"  
}  
  
// Key access  
application.output("The car is " + oCar["color"]); // Red  
  
// Key access  
var sKey = "model"  
application.output("The car model is " + oCar[sKey]); // Ford
```

Adding and Removing Properties

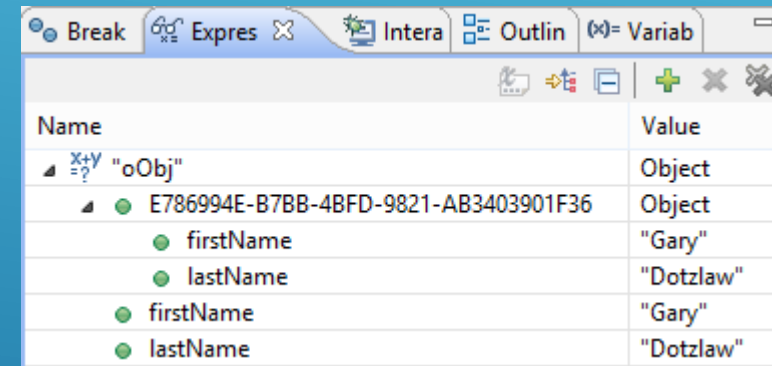
- ▶ You can add or remove properties at any time
- ▶ Property values can be primitives (string, number, boolean) or non-primitives (array, object, function)

```
var oObj = new Object();
oObj.firstName = "Gary";
oObj["lastName"] = "Dotzlaw"

application.output(oObj.firstName) // Gary
application.output(oObj.lastName) // Dotzlaw

var sUUID = application.getUUID();
oObj[sUUID] = new Object();
oObj[sUUID].firstName = "Gary";
oObj[sUUID].lastName = "Dotzlaw";

application.output(oObj[sUUID].firstName) // Gary
application.output(oObj[sUUID].lastName) // Dotzlaw
```

A screenshot of a debugger's variable window. The window has tabs for Break, Express, Intera, Outlin, and Variab. It shows a tree view of variables. The selected variable is "oObj", which is an Object. It has two properties: "firstName" and "lastName", both with string values "Gary" and "Dotzlaw" respectively. The table below shows the details of these properties.

Name	Value
oObj	Object
E786994E-B7BB-4BFD-9821-AB3403901F36	Object
firstName	"Gary"
lastName	"Dotzlaw"
firstName	"Gary"
lastName	"Dotzlaw"

Using Functions in Objects

- ▶ Add a method as a property of the Object

```
var oPerson = {  
  firstName : "Gary",  
  lastName  : "Dotzlaw",  
  fullName  : function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
  
application.output (oPerson.fullName()); // Gary Dotzlaw
```

Using Functions in Objects

- ▶ You can also pass in variables to the function, and chain functions together in your call
- ▶ Functions in objects can return different things

```
var oPerson = {  
  firstName : "Gary",  
  lastName  : "Dotzlaw",  
  /*  
   * @param {String} sFirstName  
   * @param {String} sLastName  
   */  
  setName   : function (sFirstName, sLastName){  
    this.firstName = sFirstName;  
    this.lastName  = sLastName;  
    return this;  
  },  
  fullName  : function(){  
    return this.firstName + " " + this.lastName;  
  }  
}  
  
application.output (oPerson.setName("John", "Dixon").fullName()); // John Dixon
```

Using Functions in Objects

- Functions in objects can call upon each other, so therefore you can have private functions (used internally by the object) and public functions (exposed in the public interface)

```
/**
 * @param {String} firstName
 * @param {String} lastName
 * @param {String} title
 * @public
 * @constructor
 */
function Person (firstName, lastName, title) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.title = title;
  this.fullName = function(){
    return this.firstName + " " + this.lastName;
  };
  this.fullNameTitle = function(){
    return this.fullName() + " " + this.title;
  }
}
```

The “this” keyword

- ▶ Represents the calling object
- ▶ Provides for “loose-coupling”
- ▶ You can reuse the function on different objects (call, apply, bind)
- ▶ The ability to use and manipulate the “this” value of functions is key to good object-oriented programming (more on this in a bit)

```
// Tight coupling (bad)
var oPerson = {
  firstName : "Gary",
  lastName  : "Dotzlaw",
  sayName   : function (){
    application.output(oPerson.firstName + " " + oPerson.lastName);
  }
}
oPerson.sayName(); // Gary Dotzlaw

// Loose coupling (good)
var oPerson1 = {
  firstName : "Gary",
  lastName  : "Dotzlaw",
  sayName   : function (){
    application.output(this.firstName + " " + this.lastName);
  }
}
oPerson1.sayName(); // Gary Dotzlaw
```

Delete Object['property']

- ▶ Delete a property from an object

```
/*
 * @param {String} [firstName]
 * @param {String} [lastName]
 * @public
 * @constructor
 */
function Person (firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function(){
        return this.firstName + " " + this.lastName;
    }
}

var oContact = new Person("Gary", "Dotzlaw");
application.output(oContact.fullName()); // Gary Dotzlaw
delete oContact.firstName;
application.output(oContact.fullName()); // Dotzlaw
```

Protecting Properties

- ▶ Write your own Set and Get functions, and protect the properties with the JSDoc tag “@protected”
- ▶ Will not show-up in code completion, and will produce a build marker

```
/**
 * @constructor
 */
function Person () {
    /**
     * @protected
     */
    this.firstName = null;

    this.setFirstName = function(sFirstName){
        this.firstName = sFirstName;
    };
    this.getFirstName = function(){
        return this.firstName;
    }
}

var oPerson = new Person();
oPerson.setFirstName("Gary"); // full code completion
application.output(oPerson.getFirstName()); // Gary

oPerson.firstName = "Bob"; // no code completion, build marker
application.output(oPerson.getFirstName()); // Bob
```

```
/**
 * @constructor
 */
function Person2 () {
    /**
     * @type {String}
     * @protected
     */
    var _firstName = null;

    this.setFirstName = function(sFirstName){
        _firstName = sFirstName;
    };
    this.getFirstName = function(){
        return _firstName;
    }
}

var oPerson2 = new Person2();
oPerson2.setFirstName("Bill"); // full code completion
application.output(oPerson2.getFirstName()); // Bill

oPerson2._firstName = "Frank"; // is ignored
application.output(oPerson2.getFirstName()); // Bill
```


Property Descriptors

- ▶ Get information about the property

```
var oPerson = {  
    firstName : "Gary",  
    lastName  : "Dotzlaw"  
}  
  
application.output (Object.getOwnPropertyDescriptor(oPerson, "firstName"));  
// {value:Gary,writable:true,enumerable:true,configurable:true}
```

- ▶ You can define new properties using the descriptor

```
Object.defineProperty(oPerson, "Gender", {  
    value      : "Male",  
    writable   : true,  
    enumerable : true,  
    configurable: true  
})
```

- ▶ You can change existing properties using the descriptor

```
Object.defineProperty(oPerson, "firstName", {writable : false})  
  
application.output (Object.getOwnPropertyDescriptor(oPerson, "firstName"));  
// {value:Gary,writable:false,enumerable:true,configurable:true}
```

Preventing Object Modification

- ▶ You can prevent an object from being modified with "Object.preventExtensions(Object)"
- ▶ You can check to see if properties can be added to an object using "Object.isExtensible(Object)"

```
var oPerson = {  
  firstName : "Gary",  
  lastName  : "Dotzlaw",  
  sayName   : function () {  
    application.output(this.firstName + " " + this.lastName);  
  }  
}  
application.output(Object.isExtensible(oPerson)); // true  
  
// Lock the object  
Object.preventExtensions(oPerson);  
application.output(Object.isExtensible(oPerson)); // false
```

Object.seal(Object)

- ▶ Takes an existing object and prevents it from being extended or configured
- ▶ You cannot:
 - ▶ Add any new properties
 - ▶ Reconfigure or delete existing properties
- ▶ You can:
 - ▶ Modify existing property values
 - ▶ Enumerate the properties (for-in loop)
- ▶ You can check to see if an object is sealed using `Object.isSealed(Object)`

```
Object.seal(oPerson);  
application.output (Object.getOwnPropertyDescriptor(oPerson,"firstName"));  
// {value:Gary,writable:true,enumerable:true,configurable:false}
```

Object.freeze(Object)

- ▶ Object cannot be extended, configured, or have its values modified
- ▶ You cannot:
 - ▶ Add any new properties
 - ▶ Reconfigure or delete existing properties
 - ▶ Change existing property values
- ▶ You can:
 - ▶ Enumerate the properties (for-in loop)
- ▶ You can check to see if an object is frozen using `Object.isFrozen(Object)`

```
var myPerson = Object.freeze(oPerson);  
application.output (Object.getOwnPropertyDescriptor(myPerson, "firstName"));  
// {value:Gary,writable:false,enumerable:true,configurable:false}
```

Checking for Property Existence

- ▶ The “in” operator will check for property existence in the Object, or up the prototype chain
- ▶ The “hasOwnProperty()” checks to see if the Object has the property or not, and will not consult the prototype chain

```
var oPerson = {  
    firstName : "Gary",  
    lastName  : "Dotzlaw"  
}  
  
var myPerson = Object.create(oPerson);  
myPerson.gender = "Male";  
  
application.output(("firstName" in myPerson)); // true  
application.output(myPerson.hasOwnProperty("firstName")); // false  
application.output(myPerson.hasOwnProperty("gender")); // true
```

Enumerating Object Properties

- ▶ You can enumerate through the Object properties using a for-in loop
- ▶ Only properties that are set to “enumerable : true” will be processed

```
var Person = {  
    firstName : "John",  
    lastName  : "Dixon",  
    fullName  : function(){  
        return this.firstName + " " + this.lastName;  
    }  
},  
properties = [],  
prop;  
  
// Only enumerable properties  
for (prop in Person) {  
    properties.push(prop);  
}  
application.output(properties); // [firstName, lastName, fullName]  
  
Object.defineProperty(Person, "fullName", {enumerable : false});  
  
// Only enumerable properties  
properties = [];  
for (prop in Person) {  
    properties.push(prop);  
}  
application.output(properties); // [firstName, lastName]
```

Retrieving Property Names

- ▶ Use `Object.keys(Object)` to eliminate having to enumerate through an object to get the property names
 - ▶ Retrieves only the names of properties that are declared directly on the object (does not get inherited properties)
 - ▶ Only retrieves enumerable properties
- ▶ Use `Object.getOwnPropertyNames(Object)` will give you all property names, including the non-enumerable properties

```
var Person = {
    firstName : "John",
    lastName  : "Dixon",
    fullName  : function(){
        return this.firstName + " " + this.lastName;
    }
};

Object.defineProperty(Person, "fullName", {enumerable : false});

// Exclude non-enumerable properties
application.output(Object.keys(Person)); // [firstName, lastName]

// Include non-enumerable properties
application.output(Object.getOwnPropertyNames(Person)); // [firstName, lastName, fullName]
```

Object Creation using a Constructor

- ▶ A constructor is simply a function that is used with keyword “new” to create an object
- ▶ Use when you are creating multiple objects that are all similar, as they will all have the same properties and methods
- ▶ Constructor names should begin with a capital letter
- ▶ Use the JSDoc tag “@constructor”

```
/*
 * @param {String} firstName
 * @param {String} lastName
 * @public
 * @constructor
 */
function Person (firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function(){
        return this.firstName + " " + this.lastName;
    }
}

var oContact1 = new Person("Gary", "Dotzlaw");
var oContact2 = new Person("Frank", "Smith");

application.output(oContact1.fullName()); // Gary Dotzlaw
application.output(oContact2.fullName()); // Frank Smith
```


Object Constructor (7.4)

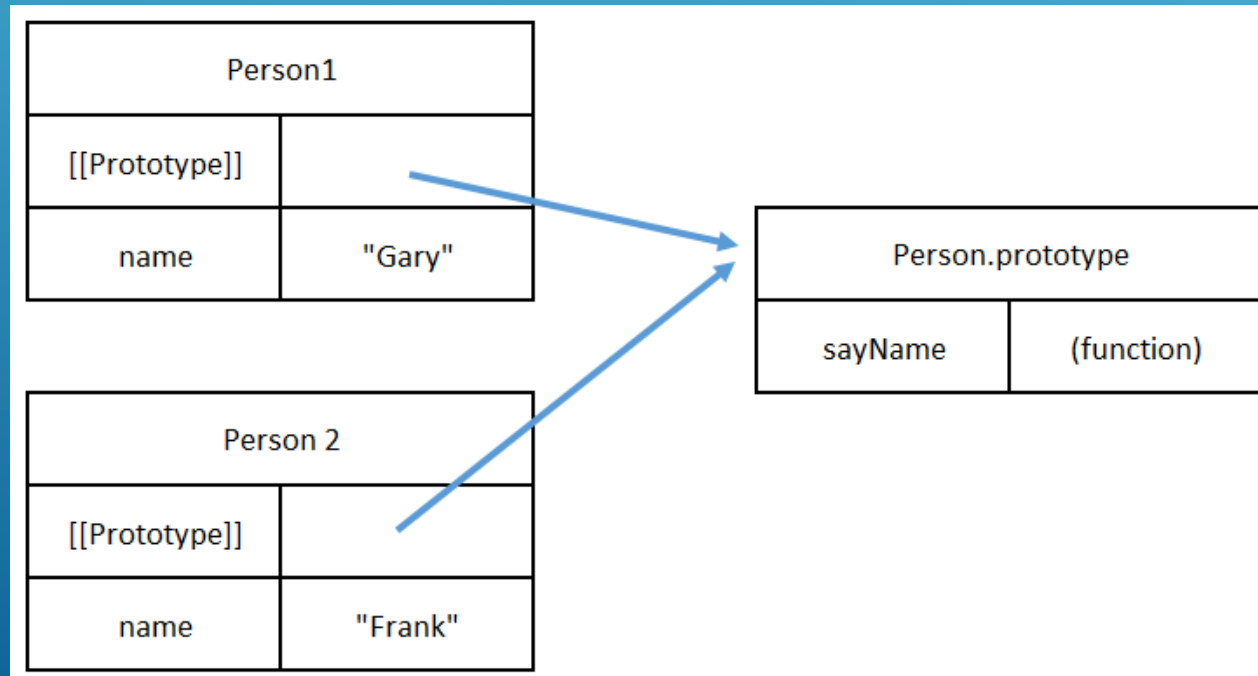
- Support for returning an instance of itself without warnings

```
/*
 * @param {String} firstName
 * @param {String} lastName
 * @param {String} title
 * @public
 * @constructor
 */
function Person (firstName, lastName, title){
    if (!(this instanceof Person)) { // New in Servoy 7.4; constructor is not called with the 'new' keyword
        return new Person(firstName, lastName, title);
    }
    this.firstName = firstName;
    this.lastName = lastName;
    this.title = title;
    this.fullName = function(){
        return this.firstName + " " + this.lastName;
    }
    this.fullNameTitle = function(){
        return this.fullName() + ", " + this.title;
    }
    return this;
}

var oContact = Person("Frank", "Smith", "President");
oContact.type = "Business Contact";
application.output(oContact.fullName()); // Frank Smith
application.output(oContact.fullNameTitle()); // Frank Smith, President
application.output(oContact.hasOwnProperty("firstName")); // true
```

Object Prototype

- ▶ Think of a prototype as a recipe for an object
- ▶ The prototype is shared among all of the object instances, and those instances can access properties of the prototype
- ▶ Put your common methods on the prototype, so that the one instance is shared by all the objects created with the constructor



Prototype – Step 1

- Create the basic constructor with the properties that need to be unique for each object occurrence

```
// Create the basic object
/*
 * @param {String} sName
 * @param {String} sColor
 * @param {Number} nRating
 * @param {Number} nCost
 * @public
 * @constructor
 */
function Fruit(sName, sColor, nRating, nCost) {
    this.name = sName;
    this.color = sColor;
    this.rating = nRating;
    this.cost = nCost;
}
```

Prototype – Step 2

- Use a self executing function (IIFE) to setup the prototype of the object when the scope in which the functions reside gets instantiated

```
//Adding to the prototype chain
/**
 * @private
 * @SuppressWarnings(unused)
 */
var initFruit = (function(){
    Fruit.prototype = {
        type: function () {
            return this.color + ' ' + this.name;
        },
        getInfo: function(){
            return 'Rating: ' + this.rating
                + ', Price: $' + this.getPrice();
        },
    };
    /**
     * @protected
     */
    getPrice: function(){
        return this.cost * 1.07;
    }
};
})();
```

Prototype - Step 3

- ▶ Create the new Object using the constructor, passing in the parameters
- ▶ We get the new Object with the properties set, and the prototype methods attached

```
/*
 * @constructor
 */
function Fruit(sName, sColor, nRating, nCost) {
    this.name = sName;
    this.color = sColor;
    this.rating = nRating;
    this.cost = nCost;
}

//Adding to the prototype chain
/**
 * @private
 * @SuppressWarnings(unused)
 */
var initFruit = (function(){
    Fruit.prototype = {
        type: function () {
            return this.color + ' ' + this.name;
        },
        getInfo: function(){
            return 'Rating: ' + this.rating
                + ', Price: $' + this.getPrice();
        },
        /**
         * @protected
         */
        getPrice: function(){
            return this.cost * 1.07;
        }
    };
})();

function testApple(){
    var oApple = new Fruit('Apple', 'Red', 5, 1.00);
    application.output("Type of Fruit: " + oApple.type()); // Type of Fruit: I am a Red Apple
    application.output("Type of Fruit: " + oApple.getInfo()); // Rating: 5, Price: $1.07
}
```

Prototypal Inheritance with Object.create(Object,[prop])

- Creates a new object that inherits from the specified prototype object, and can be enhanced with optional properties

```
/*
 * @param {String} firstName
 * @param {String} lastName
 * @param {String} title
 * @public
 * @constructor
 */
function Person (firstName, lastName, title){
    if (!(this instanceof Person)){ // New in Servoy 7.4; constructor is not called with the 'new' keyword
        return new Person(firstName, lastName, title);
    }
    this.firstName = firstName;
    this.lastName = lastName;
    this.title = title;
    this.fullName = function(){
        return this.firstName + " " + this.lastName;
    }
    return this;
}

var oContact = Object.create(new Person("Frank", "Smith", "President"), {contactType: {value: "Business Contact", enumerable: true}});

application.output(oContact.fullName()); // Frank Smith
application.output(oContact.contactType); // Business Contact
```

```

/*
 * @constructor
 */
function Fruit(sName, sColor, nRating, nCost) {
    var myVar = "Something";
    this.name = sName;
    this.color = sColor;
    this.rating = nRating;
    this.cost = nCost;
}

//Adding to the prototype chain
/**
 * @private
 * @SuppressWarnings(unused)
 */
var initFruit = (function(){
    Fruit.prototype = {
        type: function () {
            return this.color + ' ' + this.name;
        },
        getInfo: function(){
            return 'Rating: ' + this.rating
                + ', Price: $' + this.getPrice();
        },
        /**
         * @protected
         */
        getPrice: function(){
            return this.cost * 1.07;
        },
        init: function (sName, sColor, nRating, nCost){
            this.name = sName;
            this.color = sColor;
            this.rating = nRating;
            this.cost = nCost;
            return this;
        }
    };
})();

```

```

/*
 * @constructor
 */
function Fruit(sName, sColor, nRating, nCost) {
    var myVar = "Something";
}

```

```

var oBanana = Object.create(Fruit.prototype, {vendor: {value: "Frank's", enumerable: true}});
oBanana.country = "Chile";
oBanana.init("Banana", "Yellow", 4, .85);
application.output("Type of Fruit: " + oBanana.type() + ", " + oBanana.getInfo() + ", from: " + oBanana.vendor + " in " + oBanana.country);
// Type of Fruit: Yellow Banana, Rating: 4, Price: $0.9095, from: Frank's in Chile

```

```

// Add to the prototype
Fruit.prototype.stock = null;
oBanana.stock = 10;
application.output(oBanana.name + " stock: " + oBanana.stock); // build marker, no code completion on name (use setter/getter)
// Banana stock: 10

```

```

var oPear = Object.create(Fruit.prototype, {});
oPear.init("Pear", "Yellow", 3, 1.00);
application.output("Type of Fruit: " + oPear.type() + ", " + oPear.getInfo() + ", from: " + oPear.vendor + " in " + oPear.country);
// Type of Fruit: Yellow Pear, Rating: 3, Price: $1.07, from: undefined in undefined

```

```

var oMango3 = Object.create(oBanana, {});
oMango3.country = "Brazil";
oMango3.vendor = "Murphy's";
oMango3.init("Mango", "Red", 2, 3.00);
application.output("Type of Fruit: " + oMango3.type() + ", " + oMango3.getInfo() + ", from: " + oMango3.vendor + " in " + oMango3.country);
// Type of Fruit: Red Mango, Rating: 2, Price: $3.21, from: Frank's in Brazil

```

```

application.output(Object.getOwnPropertyDescriptor(oBanana, "vendor"));
// {value:Frank's,writable:false,enumerable:true,configurable:false}
application.output(Object.getOwnPropertyDescriptor(oBanana, "country"));
// {value:Chile,writable:true,enumerable:true,configurable:true}

```

```

    this.cost = nCost;
    this.rating = nRating;
    this.cost = nCost;
    return this;
}
};
})();

```

Prototypal Inheritance

- ▶ Avoid public properties with non-primitive values (array, object) in the prototype, declare or initialize these in the constructor
- ▶ The use of variables on the constructor are limited in use to private local properties, as they cannot be accessed from the prototype, or other objects that inherit from it

Function.prototype.call(thisArg, [arg1, arg2, ...])

- The call() method calls a constructor function with a given “this” value and arguments provided individually

```
/**
 * @constructor
 */
var Person = function (sFirstName, sLastName){
    this.firstName = sFirstName;
    this.lastName = sLastName;
    this.fullName = function(){
        return this.firstName + " " + this.lastName;
    }
}

/**
 * @constructor
 * @extends {Person}
 */
function BusinessContact (firstName, lastName){
    Person.call(this, firstName, lastName);
    this.type = "Business Contact";
}

var oBusinessContact = new BusinessContact("Gary", "Dotzlaw");
application.output(oBusinessContact.fullName()); // Gary Dotzlaw
```

Function.prototype.apply(thisArg, [argsArray])

- The apply() method calls a constructor function with a given “this” value and arguments provided as an array

```
/**
 * @constructor
 */
var Person = function (sFirstName, sLastName){
    this.firstName = sFirstName;
    this.lastName = sLastName;
    this.fullName = function(){
        return this.firstName + " " + this.lastName;
    }
}

/**
 * @constructor
 * @extends {Person}
 */
function BusinessContact (firstName, lastName){
    Person.apply(this, [firstName, lastName]);
    this.type = "Business Contact";
}

var oBusinessContact = new BusinessContact("Gary", "Dotzlaw");
application.output(oBusinessContact.fullName()); // Gary Dotzlaw
```

Function.prototype.bind(thisArg, [argsArray])

- ▶ The bind() method creates a new function that, when called, has its “this” keyword set to the provided value. You can also pass an array of arguments. Useful for borrowing a method from another object.

```
var User = {
  firstName : "Gary",
  lastName  : "Dotzlaw"
}

var Person = {
  firstName : "John",
  lastName  : "Dixon",
  fullName  : function() {
    return this.firstName + " " + this.lastName;
  }
}

User.fullName = Person.fullName.bind(User);
application.output(User.fullName()); // Gary Dotzlaw
```

Prototypal Inheritance

- ▶ Putting it all together in Servoy; building new objects from existing objects, and extending the prototype chain

```
/**
 * @constructor
 */
var Person = function(firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName
}
```

```
/**
 * Self-executing IIFE
 * @private
 * @SuppressWarnings(unused)
 */
var initPerson = (function(){
    Person.prototype = {
        fullName: function(){
            return this.firstName + " " + this.lastName;
        }
    }
    // extending the prototype
    Person.prototype.lastFirstName = function() {
        return this.lastName + ", " + this.firstName;
    }
})();
```

```
/**
 * @extends {Person}
 * @constructor
 */
var SalesPerson = function(firstName, lastName, rate) {
    // call the base constructor object
    Person.call(this, firstName, lastName);
    this.commissionRate = rate;
}
```

```
/**
 * Self-executing IIFE
 * @private
 * @SuppressWarnings(unused)
 */
var initSalesPerson = (function(){
    SalesPerson.prototype = Object.create(Person.prototype, {});
    // Properly set the constructor
    SalesPerson.prototype.constructor = SalesPerson;
    SalesPerson.prototype.getCommission = function(price) {
        return "Commission: $" + (price * this.commissionRate / 100);
    }
})();
```

```
function test(){
    var oSalesman = new SalesPerson("Gary", "Dotzlaw", 10);
    application.output(oSalesman.getCommission(100)); // Commission: $10
    application.output(oSalesman.fullName()); // Gary Dotzlaw
    application.output(oSalesman.lastFirstName()); // Dotzlaw, Gary
}
```

Working with Objects

▶ Typical Use Cases:

- ▶ Global Cache
- ▶ System Preferences
- ▶ Event Routing
- ▶ Dialogs
- ▶ PopUp Info
- ▶ PickList
- ▶ Navigation
- ▶ Security
- ▶ Dashboards & Widgets
- ▶ Calendar & Scheduling
- ▶ Document Management
- ▶ Reporting

Working with Objects

- Pass an Object as a parameter to a function or even a constructor

```
/**
 * Start price chart lookup
 *
 * @param {{
 * cust_id :   UUID,
 * item_id  :   UUID,
 * qty      :   Number
 * }} oPrice - the price chart object
 *
 * @returns {Number} The price for customer and item
 */
function getListPrice(oPrice){
    var nPrice = 0;

    // lots of code here

    return nPrice;
}
```

```
/**
 * Start price chart lookup
 *
 * @param {{cust_id : UUID, item_id : UUID, qty : Number}} oPrice - the price chart object
 *
 * @returns {Number} The price for customer and item
 */
function getListPrice(oPrice){
    var nPrice = 0;

    // lots of code here

    return nPrice;
}
```

Working with Objects

- Return an Object from a function

```
/**
 * Start price chart lookup
 *
 * @param {{cust_id : UUID, item_id : UUID, qty : Number}} oPrice - the price chart object
 *
 * @returns {{cost:Number, mrkUp:Number, price:Number, tax:Number, priceTax:Number}} The price object
 */
function getListPrice(oPrice){
    var nPrice = 0,
        nCost = 0,
        nMrkUp = 0,
        nTax = 0,
        oObj = {};

    //
    // lots of code here
    //
    oObj.cost = nCost;
    oObj.mrkUp = nMrkUp;
    oObj.price = nPrice;
    oObj.tax = nTax;
    oObj.priceTax = nPrice + nTax;

    return oObj;
}
```

```
/**
 * Start price chart lookup
 *
 * @param {{
 *   cust_id : UUID,
 *   item_id : UUID,
 *   qty : Number
 * }} oPrice - the price chart object
 *
 * @returns {{
 *   cost :Number,
 *   mrkUp :Number,
 *   price :Number,
 *   tax :Number,
 *   priceTax :Number
 * }} The price object
 */
function getListPrice(oPrice){
    var nPrice = 0,
        nCost = 0,
        nMrkUp = 0,
        nTax = 0,
        oObj = {};

    //
    // lots of code here
    //
    oObj.cost = nCost;
    oObj.mrkUp = nMrkUp;
    oObj.price = nPrice;
    oObj.tax = nTax;
    oObj.priceTax = nPrice + nTax;

    return oObj;
}
```

Working with Objects

- ▶ Using a global cache
 - ▶ Create a variable for the cache (in a scope)
 - ▶ Initialize before calling your method, and destroy it after
 - ▶ In your method, check to see if you did it before, and if so, return it the cached value
 - ▶ If this is the first time, then run the method and store the result in the cache for the next time

```
// in a scope define an object variable
var Items = {};

// somewhere in your code, initialize the object
scopes.cache.Items = {};
/**
 * run your code to look up item pricing
 */
// clear the cache when done
scopes.cache.Items = null;

// at the start of your item pricing function
if (scopes.cache.Items && scopes.cache.Items[rRec.item_id]){
    return scopes.cache.Items[rRec.item_id];
}
/**
 * first time we have seen this item, so run the complex price rules
 */
// before returning the object, store it into the cache
if (scopes.cache.Items && !scopes.cache.Items[rRec.item_id]){
    scopes.cache.Items[rRec.item_id] = {};
    scopes.cache.Items[rRec.item_id].cost = nCost;
    scopes.cache.Items[rRec.item_id].mrkUp = nMrkUp;
    scopes.cache.Items[rRec.item_id].price = nPrice;
    scopes.cache.Items[rRec.item_id].tax = nTax;
    scopes.cache.Items[rRec.item_id].priceTax = nPrice + nTax;
}
return scopes.cache.Items[rRec.item_id];
// exit the item pricing function
```


Working with Objects

- ▶ What does a real object in Servoy look?
 - ▶ Table View Persistence

Summary

- ▶ Learn how to work with objects in Servoy; it's easy and very powerful
- ▶ Implement OOP in your solutions; make your code modularized, maintainable, and extendable
- ▶ Objects and OOP can help you elegantly solve some very complicated problems
- ▶ Servoy support for OOP continues to grow
- ▶ Remember to ask for a raise, now that you are the OOP guru!

Next

► Advanced Servoy Mobile: Custom UI and SQL Lite

Paolo Aronne

(10:30am – 11:20am)