Module 7

Creating Objects and Methods by Using JavaScript

Module Overview

- Writing Well-Structured JavaScript Code
- Creating Custom Objects
- Extending Objects

Lesson 1: Writing Well-Structured JavaScript Code

- Scoping and Hoisting
- Managing the Global Namespace
- Singleton Objects and Global Functions in JavaScript

Scoping and Hoisting

- JavaScript variables have one of two scopes:
  - Global scope
  - Local scope within a function

- JavaScript does not support block scope
  - If you declare a variable inside a block, it is hoisted to function scope

```
var num = 7;

function demonstrateScopingAndHoisting() {
    if (true) {
        var num = 42;
    }
    alert("The value of num is " + num);      // Displays 42, not 7.
}
```

Managing the Global Namespace

- Global name clashes can be problematic in JavaScript
  - Your global variables might conflict with other global variables elsewhere in the web application
- JavaScript provides several mechanisms to avoid global name clashes
  - Immediate functions
  - Namespaces
  - Strict mode

Singleton Objects and Global Functions in JavaScript

- JavaScript defines several singleton objects, such as:
  - **Math**
  - **JSON**

- JavaScript also defines global functions, such as:
  - **parseInt()**
  - **parseFloat()**
  - **isNan()**

Lesson 2: Creating Custom Objects

- Creating Simple Objects
- Using Object Literal Notation
- Using Constructors
- Using Prototypes
- Using the Object.create Method

Creating Simple Objects

- There are several ways to create new objects in JavaScript:

```
var employee1 = new Object();
```

```
var employee2 = {};
```

- You can define properties and methods on an object:

```
var employee1 = {};
employee1.name = "John Smith";
employee1.age = 21;
employee1.salary = 10000;

employee1.payRise = function(amount) {
    // Inside a method, "this" means the current object.
    this.salary += amount;
    return this.salary;
}
```

Using Object Literal Notation

- Object literal notation provides a shorthand way to create new objects and assign properties and methods:

```
var employee2 = {
    name: "Mary Jones",
    age: 42,
    salary: 20000,

    payRise: function(amount) {
        this.salary += amount;
        return this.salary;
    },

    displayDetails: function() {
        alert(this.name + " is " + this.age + " and earns " + this.salary);
    }
};
```

Using Constructors

- Constructor functions define the shape of objects
  - They create and assign properties for the target object
  - The target object is referenced by the **this** keyword

```
var Account = function (id, name) {
    this.id = id;
    this.name = name;
    this.balance = 0;
    this.numTransactions = 0;
};
```

- Use the constructor function to create new objects with the specified properties:

```
var acc1 = new Account(1, "John");
var acc2 = new Account(2, "Mary");
```

Using Prototypes

- All objects created by using a constructor function have their own copy of the properties defined by the constructor
  - All JavaScript objects, including constructors, have a special property named **prototype**
  - Use the prototype to share function definitions between objects:

```
Account.prototype = {
    deposit: function(amount) {
        this.balance += amount;
        this.numTransactions++;
    },

    // Plus other methods…
};
```

Using the Object.create Method

- Use **Object.create()** to create an object based on existing prototype
  - Pass in a prototype object
  - Optionally pass in a properties object that specifies additional properties to add to the new object

```
var obj1 = Object.create(prototypeObject, propertiesObject);
```

- The new object has access to all the properties defined in the specified prototype
  - It forms the basis of the approach used by many JavaScript developers to implement inheritance.

Lesson 3: Extending Objects

- Implementing Encapsulation
- Implementing Inheritance by Chaining Prototypes
- Adding Functionality to Existing Objects
- Demonstration: Refining Code for Maintainability and Extensibility

Implementing Encapsulation

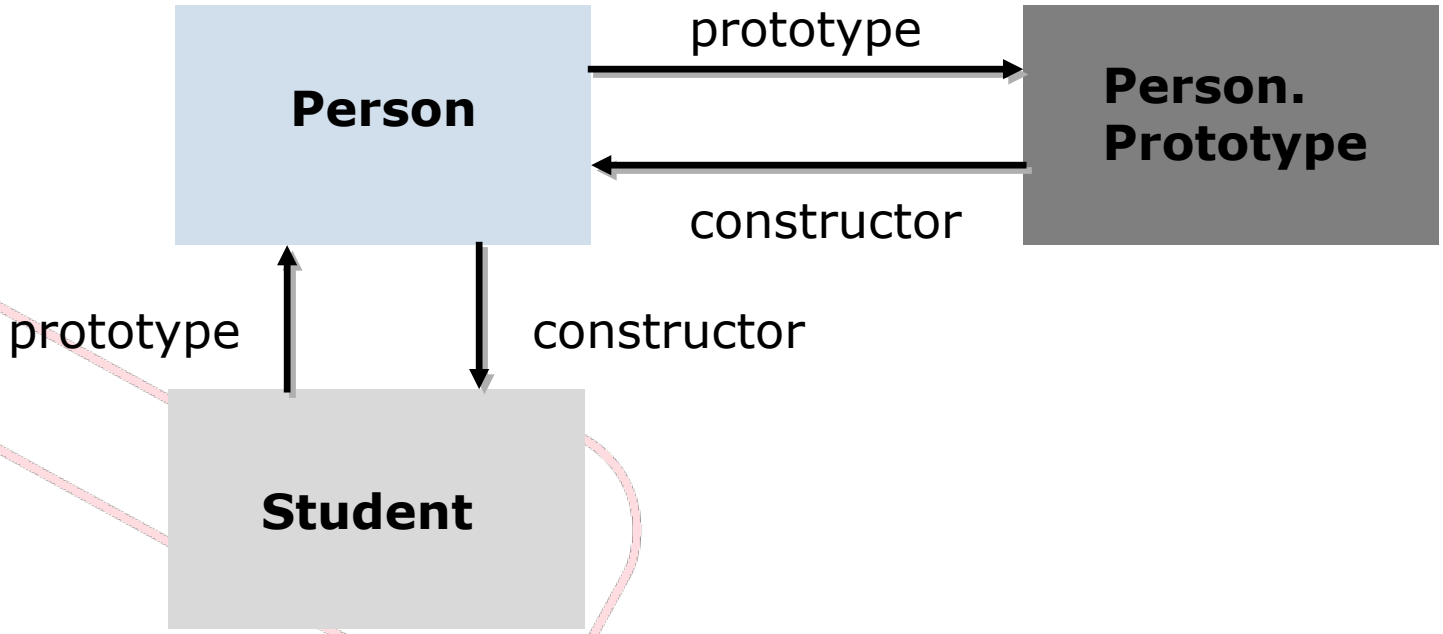
- To define private members for an object, declare variables in the constructor and omit the **this** keyword
- To define public accessor functions for an object, declare methods in the constructor and include the **this** keyword

```
var Person = function(name, age)
{
    // Private properties.
    var _name, _age;

    // Public accessor functions.
    this.getName = function()
    {
        return _name;
    }
    …
}
```

Implementing Inheritance by Chaining Prototypes

- Define the base constructor and prototype
- Define the derived constructor
- Set the **prototype** property of the derived constructor to an instance of the base object



Adding Functionality to Existing Objects

- Get the prototype for an object
- Assign a new property to the object

```
var Point = function(x, y) {
    this.x = x;
    this.y = y;
}

Point.prototype.moveBy = function(deltaX, deltaY) { … }
Point.prototype.moveTo = function(otherPoint) { … }

var p1= new Point(100, 200);
p1.moveBy(10, 20);
p1.moveTo(anotherPoint);
```

- Use the **apply** method to resolve references to **this** in generic functions

Demonstration: Refining Code for Maintainability and Extensibility

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Refining Code for Maintainability and Extensibility

- Exercise 1: Object Inheritance
- Exercise 2: Refactoring JavaScript Code to Use Objects

Logon Information

- Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
- User Name: **Student**
- Password: **Pa$$w0rd**

Estimated Time: 60 minutes

Lab Scenario

The existing JavaScript code for the ContosoConf website has been written without much high-level structure or organization. While this approach is fine for small pieces of code, it will not scale up for a larger project. An unstructured collection of functions and variables scattered throughout a JavaScript file can quickly become unmaintainable.

Before implementing more website features by using JavaScript, you decide to refactor the existing code to introduce better organizational practices. The resulting code will be more maintainable and provide a good pattern for implementing future website features.

Module Review and Takeaways

- Review Question(s)