Assignments

Contents

Lesson M1.L1: Primitive and Reference Types

Lesson Objectives:

Tradionally most of us as developers learn object-oriented programming by working with class based languages such as Java or C#. But learning object oriented JavaScript can be a different experience because JavaScript has no formal support for classes. Instead of defining classes from the beginning, with JavaScript we can just write code and create data structures as and when we need them.

Because it lacks classes, JavaScript also lacks class groupings such as packages. Whereas in languages like Java, package and class names define both the types of objects you use and the layout of files and folders in your project, programming in JavaScript is like starting with a blank slate: You can organize things any way you want.

To ease the transition from traditional object-oriented languages, JavaScript makes *objects* the central part of the language.

Almost all data in JavaScript is either an object or accessed through objects. In fact, even functions are represented as objects in JavaScript, which makes them first-class functions.

Working with and understanding objects is key to understanding JavaScript as a whole. We can create objects at any time and add or remove properties from them whenever you want. In addition, JavaScript Objects are extremely flexible and have capabilities that create unique and interesting patterns that are simply not possible in other languages.

This lesson focuses on how to identify and work with the **two primary JavaScript data types: primitive types and reference types.**

Though both are accessed through objects, they behave in different ways that are important to understand.

After completing this lesson, you will be able to:

- Understand What Are JavaScript Types?
- Understand Primitive Types and how to work with them
- Understand Reference Types and learn how to work work with them
- Understand the concept of Literal Forms and how to apply them effectively
- Understand and Use Primitive Wrapper Types

Lab 1 : Working with Primitive Types

All primitive types have literal representations of their values. Here are some examples of each type using its literal form:

```
// strings
var name = "Nicholas";
var selection = "a";

// numbers
var count = 25;
var cost = 1.51;

// boolean
var found = true;

// null
var object = null;

// undefined
var flag = undefined;
var ref; // assigned undefined automatically
```

Lab 2 : Identifying Primitive Types

The typeof operator works well with strings, numbers, Booleans, and undefined. Observe the result of execution in the comment.

```
1  console.log(typeof "Nicholas"); // "string"
2
3  console.log(typeof 10); // "number"
4
5  console.log(typeof 5.1); // "number"
6
7  console.log(typeof true); // "boolean"
8
9  console.log(typeof undefined); // "undefined"
```

As you might expect , typeof returns:

- "string" when the value is a string
- "number" when the value is a number (regardless of integer or floatingpoint values)
- "boolean" when the value is a Boolean;
- "undefined" when the value is undefined.

The Special Case of null Primitive:

The null primitive behaves differently.

Observe the following code and the result in the comment

```
1  console.log(typeof null); // "object"
```

The best way to determine if a value is null is to compare it against null directly, like this:

```
1  console.log(value === null); // true or false
```

Lab 3 : Working with Primitive Methods

Observe the following code with respect to the use of primitive methods. Also obseverve the comment section which describes what the method does:

```
var name = "Nicholas";
var lowercaseName = name.toLowerCase(); // convert to lowercase
var firstLetter = name.charAt(0); // get first character
var middleOfName = name.substring(2, 5); // get characters 2-4

var count = 10;
var fixedCount = count.toFixed(2); // convert to "10.00"
var hexCount = count.toString(16); // convert to "a"

var flag = true;
var stringFlag = flag.toString(); // convert to "true"
```

Despite the fact that they have methods, primitive values themselves are not objects. JavaScript makes them look like objects to provide a consistent experience in the Language.

Lesson M1.L2: Functions

Lesson Objectives:

As discussed in Lesson 1, functions are actually objects in JavaScript.

This lesson discusses the various ways that functions are defined and executed in JavaScript. Because functions are objects, they behave differently than functions in other languages, and this behavior is central to a good understanding of JavaScript.

After completing this lesson, you will be able to understand:

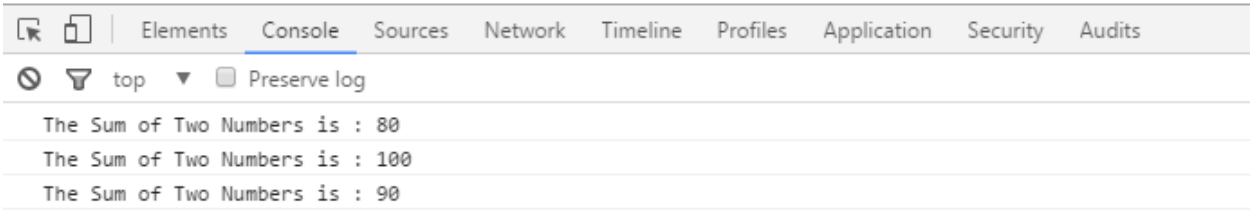
- What are Functions?
- Declarations vs. Expressions
- Functions as Values
- Parameters
- Overloading
- Object Methods
- The this Object
- Changing this

Lab 4 : Declaring and Calling Functions

Now Consider the code snippet , that calls the add function three times with different parameter values:

```
function add(num1, num2) {
return num1 + num2;
}

console.log("The Sum of Two Numbers is : " + add(30,50));
console.log("The Sum of Two Numbers is : " + add(50,50));
console.log("The Sum of Two Numbers is : " + add(40,50));
```

The output is :

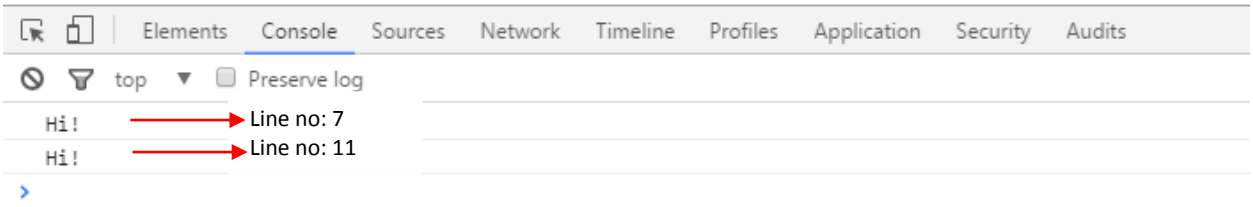


Lab 5 : Function as Values

Consider the following example:

```
1 // function declaration for sayHi
2
3 function sayHi() {
4 console.log("Hi!");
5 }
6
7 sayHi(); // outputs "Hi!"
8
9 // A variable named sayHi2 is created and assigned the value of sayHi
10
11 var sayHi2 = sayHi;
12
13 sayHi2(); // outputs "Hi!"
```

This generates the following output:



Lab 6 : Functions with Parameters

Here is a simple example using arguments and function arity; note that the number of arguments passed to the function has no effect on the reported arity:

```
function reflect(value) {
return value;
}

console.log(reflect("Hi!")); // "Hi!"
console.log(reflect("Hi!", 25)); // "Hi!"
console.log(reflect.length); // 1

reflect = function() {
return arguments[0];
};

console.log(reflect("Hi!")); // "Hi!"
console.log(reflect("Hi!", 25)); // "Hi!"
console.log(reflect.length); // 0
```

- This example first defines the reflect() function using a **single named parameter**, but there is **no error** when a second parameter is passed into the function.
- Also, the length property is 1 because there is a single named parameter. The reflect() function is then redefined with no named parameters; it returns arguments[0], which is the first argument that is passed in.
- This new version of the function works exactly the same as the previous version, but its length is 0.
- **The first implementation of reflect() is much easier to understand because it uses a named argument (as you would in other languages).**
- The version that uses the arguments object can be confusing because there are no named arguments, and you must read the body of the function to determine if arguments are used.

Sometimes, using arguments is actually more effective than naming parameters.

- For instance, suppose you want to create a function that accepts any number of parameters and returns their sum.
- You can't use named parameters because you don't know how many you will need, so in this case, using arguments is the best option.

```
function sum() {
var result = 0,
i = 0,
len = arguments.length;
while (i < len) {
result += arguments[i];
i++;
}
return result;
}
console.log(sum(1, 2)); // 3
console.log(sum(3, 4, 5, 6)); // 18
console.log(sum(50)); // 50
console.log(sum()); // 0
```

- The sum() function accepts any number of parameters and adds them together by iterating over the values in arguments with a while loop.
- This is exactly the same as if you had to add together an array of numbers.
- The function even works when no parameters are passed in, because result is initialized with a value of 0.

Lab 7 : Function Overloading

Look at what happens when you try to declare two functions with the same name:

```
function sayMessage(message) {
console.log(message);
}

function sayMessage() {
console.log("Default message");
}

sayMessage("Hello!"); // outputs "Default message"
```

- If this were another language, the output of sayMessage("Hello!") would likely be "Hello!".

" In JavaScript, however, when you define multiple functions with the same name, the one that appears last in your code wins. "

Mimicking Overloading

The fact that functions don't have signatures in JavaScript does not mean you can not mimic function overloading. You can retrieve the number of parameters that were passed in by using the arguments object, and you can use that information to determine what to do.

For example:

```
function sayMessage(message) {

if (arguments.length === 0) {
message = "Default message";
}

console.log(message);
}

sayMessage("Hello!"); // outputs "Hello!"
```

In this example:

- The sayMessage() function behaves differently based on the number of parameters that were passed in.
- If no parameters are passed in (arguments.length === 0), then a default message is used.
- Otherwise, the first parameter is used as the message.
- This is a little more involved than function overloading in other languages, but the end result is the same.

- If you really want to check for different data types, you can use typeof and instanceof.

“ In practice, checking the named parameter against undefined is more common than relying on arguments.length “

Lab 8 : The this object

```
function sayNameForAll() {
  console.log(this.name);
}
var person1 = {
  name: "Nicholas",
  sayName: sayNameForAll
};
var person2 = {
  name: "Greg",
  sayName: sayNameForAll
};
var name = "Michael";
person1.sayName(); // outputs "Nicholas"
person2.sayName(); // outputs "Greg"
sayNameForAll(); // outputs "Michael"
```

In this example :

- a function called **sayName** is defined first.
- Then, two object literals are created that assign **sayName** to be equal to the **sayNameForAll** function. Functions are just reference values, so you can assign them as property values on any number of objects.
- When sayName() is called on person1, it outputs "Nicholas"; when called on person2, it outputs "Greg". That's because this is set when the function is called, so this.name is accurate.
- The last part of this example defines a global variable called name. When sayNameForAll() is called directly, it outputs "Michael" because the global variable is considered a property of the global object.

Changing this

The ability to use and manipulate the **this value** of functions is key to good object-oriented programming in JavaScript.

Functions can be used in many different contexts, and they need to be able to work in each situation.

Even though this is typically assigned automatically, you can change its value to achieve different goals.

There are three function methods that allow you to change the value of this:

- The call() Method
- The apply() Method
- The bind() Method

The call() Method

The first function method for manipulating this is call(), which executes the function with a particular this value and with specific parameters.

- **The first parameter of call() is the value to which this should be equal when the function is executed.**
- All subsequent parameters are the parameters that should be passed into the function.
- For example:

Lab 8 : The call() Method

```
function sayNameForAll(label) {
  console.log(label + ":" + this.name);
}

var person1 = {
  name: "Nicholas"
};

var person2 = {
  name: "Greg"
};

var name = "Michael";

// where this = global scope , label = global
sayNameForAll.call(this, "global"); // outputs "global:Michael"

// where this = person1 scope , label = person1
sayNameForAll.call(person1, "person1"); // outputs "person1:Nicholas"

// where this = person1 scope , label = person2
sayNameForAll.call(person2, "person2"); // outputs "person2:Greg"
```

Important to Note:

- The First Parameter to the call method is the object to which this should be pointing to
- The Second Paramter is the value to the formal paramter label.
- The call method ends up calling/executing the sayNameForAll method with the appropriate binding of this object

The apply() Method

The second function method you can use to manipulate this is apply().

The apply() method works exactly the same as call() except that it accepts only two parameters:

- the value for this
- an array or array-like object of parameters to pass to the function

So, instead of individually naming each parameter using call(), you can easily pass arrays to apply() as the second argument.

Otherwise, call() and apply() behave identically.

Lab 9 : The apply() Method

This example shows the apply() method in action:

```
function sayNameForAll(label) {
console.log(label + ":" + this.name);
}

var person1 = {
name: "Nicholas"
};

var person2 = {
name: "Greg"
};

var name = "Michael";

sayNameForAll.apply(this, ["global"]); // outputs "global:Michael"
sayNameForAll.apply(person1, ["person1"]); // outputs "person1:Nicholas"
sayNameForAll.apply(person2, ["person2"]); // outputs "person2:Greg"
```

- This code takes the previous example and replaces call() with apply(); the result is exactly the same.
- The method you use typically depends on the type of data you have.
- If you already have an array of data, use apply(); if you just have individual variables, use call().
- **Think of apply() as a means to use an arguments object as the second parameter**

The bind() Method

The third function method for changing this is bind().

Lab 10 : The bind() Method

This method was added in ECMAScript 5, and it behaves quite differently than the other two.

The first argument to bind() is the this value for the new function. All other arguments represent named parameters that should be permanently set in the new function. You can still pass in any parameters that aren't permanently set later.

```
function sayNameForAll(label) {
console.log(label + ":" + this.name);
}

var person1 = {
name: "Nicholas"
};

var person2 = {
name: "Greg"
};

// create a function just for person1
var sayNameForPerson1 = sayNameForAll.bind(person1);
sayNameForPerson1("person1"); // outputs "person1:Nicholas"

// create a function just for person2
var sayNameForPerson2 = sayNameForAll.bind(person2, "person2");
sayNameForPerson2(); // outputs "person2:Greg"

// attaching a method to an object doesn't change 'this'
person2.sayName = sayNameForPerson1;
person2.sayName("person2"); // outputs "person2:Nicholas"
```

Important Points:

- Bind does not call the method where as call and apply all the methods
- Bind simply associates a generic method with a specific objects. So the bind activity should be followed by a an explict call to method

Lesson M1.L3: Understanding JavaScript Objects

Lesson Objectives:

Even though there are a number of built-in reference types in JavaScript, you will most likely create your own objects fairly frequently.

As you do so, keep in mind that objects in JavaScript are dynamic, meaning that they can change at any point during code execution. Whereas class-based languages lock down objects based on a class definition, JavaScript objects have no such restrictions.

A large part of JavaScript programming is managing those objects, which is why understanding how objects work is key to understanding JavaScript as a whole. We will discuss all the important aspects of working with JavaScript Objects in this lesson.

After completing this lesson, you will be able to understand:

- What are JavaScript Objects?
- Defining Properties
- Detecting Properties
- Removing Properties
- Enumeration
- Types of Properties
- Property Attributes
- Preventing Object Modification

What are JavaScript Objects?

It helps to think of JavaScript objects as hash maps where properties are just key/value pairs.

You access object properties using either dot notation or bracket notation with a string identifier.

You can add a property at any time by assigning a value to it, and you can remove a property at any time with the delete operator.

You can always check whether a property exists by using the in operator on a property name and object.

If the property in question is an own property, you could also use hasOwnProperty(), which exists on every object.

All object properties are enumerable by default, which means that they will appear in a for-in loop or be retrieved by Object.keys().

Lab 11 : Defining Properties

For example:

```
1 var person1 = {
2 name: "Nicholas"
3 };
4
5 var person2 = new Object();
6 person2.name = "Nicholas";
7
8 person1.age = 55;
9 person2.age = 65;
10
11 person1.name = "Greg";
12 person2.name = "Michael";
```

- Both person1 and person2 are objects with a name property.
- Later: Line 8 , 9 : both objects are assigned an age property .
- Objects you create are always wide open for modification unless you specify otherwise. (We will learn how to achieve this later in this lesson).
- The last part of this example , Line 11 , 12 : changes the value of name on each object property values can be changed at any time as well

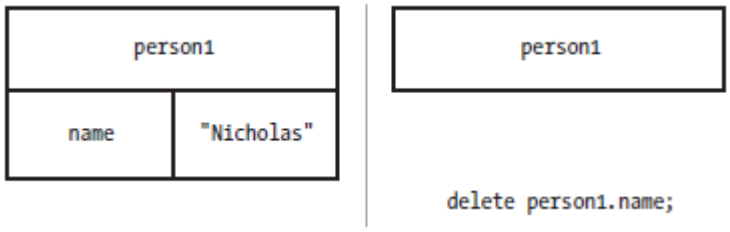
Lab 12 : Removing Properties

For example, the following listing shows the **delete operator** at work:

```
var person1 = {
name: "Nicholas"
};

console.log("name" in person1); // true

delete person1.name; // true - not output

console.log("name" in person1); // false

console.log(person1.name); // undefined
```

In this example, the name property is deleted from person1. The in operator returns false after the operation is complete. Also, note that attempting to access a property that does not exist will just return undefined.

The following Figure shows how delete affects an object.



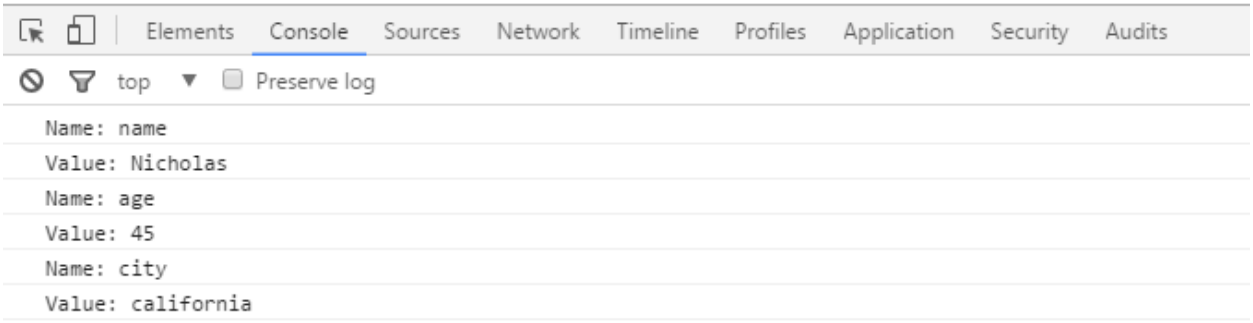
When you delete the name property, it completely disappears from person1.

Lab 13 : Working with Enumeration

For example, the following loop outputs the property names and values of an object:

```
var person1 = {
name: "Nicholas",
age:45,
city: "california"
};

var property;
for (property in person1) {
console.log("Name: " + property);
console.log("Value: " + person1[property]);
}
```

Observe the output:



Lab 14 : NonEnumerable and Non Configurable Object Properties

For example, suppose you want to make an object property nonenumerable and nonconfigurable:

```
var person1 = {
  name: "Nicholas"
};

Object.defineProperty(person1, "name", {
  enumerable: false
});

console.log("name" in person1); // true
console.log(person1.propertyIsEnumerable("name")); // false
var properties = Object.keys(person1);
console.log(properties.length); // 0

Object.defineProperty(person1, "name", {
  configurable: false
});

// try to delete the Property
delete person1.name;
console.log("name" in person1); // true
console.log(person1.name); // "Nicholas"
Object.defineProperty(person1, "name", { // error!!!
  configurable: true
});
```

- The name property is defined as usual , but it's then modified to set its [[Enumerable]] attribute to false.
- The propertyIsEnumerable() method now returns false because it references the new value of [[Enumerable]].
- After that, name is changed to be nonconfigurable.
- From now on, attempts to delete name fail because the property can't be changed, so name is still present on person1 .
- Calling Object.defineProperty() on name again would also result in no further changes to the property.
- Effectively, name is locked down as a property on person1.
- The last piece of the code tries to redefine name to be configurable once again .
- However, this throws an error because you can't make a nonconfigurable property configurable again.
- Attempting to change a data property into an accessor property or vice versa should also throw an error in this case.

Lab 15: Accessor Property Attributes

Consider this example :

```
var person1 = {
  _name: "Nicholas",
  get name() {
    console.log("Reading name");
    return this._name;
  },
  set name(value) {
    console.log("Setting name to %s", value);
    this._name = value;
  }
};
```

This code can also be written as follows:

```
var person1 = {
  _name: "Nicholas"
};

Object.defineProperty(person1, "name", {
  get: function() {
    console.log("Reading name");
    return this._name;
  },
  set: function(value) {
    console.log("Setting name to %s", value);
    this._name = value;
  },
  enumerable: true,
  configurable: true
});
```

Notice that the get and set keys on the object passed in to Object .defineProperty() are data properties that contain a function.

You can't use object literal accessor format here.

Setting the other attributes ([[Enumerable]] and [[Configurable]]) allows you to change how the accessor property works.

For example, you can create a nonconfigurable, nonenumerable, nonwritable property like this:

```
var person1 = {
  _name: "Nicholas"
};

Object.defineProperty(person1, "name", {
  get: function() {
    console.log("Reading name");
    return this._name;
  }
});

console.log("name" in person1); // true
console.log(person1.propertyIsEnumerable("name")); // false
delete person1.name;
console.log("name" in person1); // true
person1.name = "Greg";
console.log(person1.name); // "Nicholas"
```

In this code:

- The name property is an accessor property with only a getter.
- There is no setter or any other attributes to explicitly set to true, so the value can be read but not changed.

Lab 16 : Defining Multiple Properties

For example, the following code defines two properties:

```
var person1 = {};

Object.defineProperties(person1, {
  // data property to store data
  _name: {
    value: "Nicholas",
    enumerable: true,
    configurable: true,
    writable: true
  },

  // accessor property
  name: {
    get: function() {
      console.log("Reading name");
      return this._name;
    },
    set: function(value) {
      console.log("Setting name to %s", value);
      this._name = value;
    },
    enumerable: true,
    configurable: true
  }
});
```

This example:

- Defines _name as a data property to contain information and name as an accessor property .
- You can define any number of properties using Object.defineProperties(); you can even change existing properties and create new ones at the same time.
- The effect is the same as calling Object.defineProperty() multiple times.

Lab 17 : Sealing Objects

You can check to see whether an object is sealed using Object.isSealed() as follows:

```
var person1 = {
  name: "Nicholas"
};

console.log(Object.isExtensible(person1)); // true
console.log(Object.isSealed(person1)); // false
Object.seal(person1);
console.log(Object.isExtensible(person1)); // false
console.log(Object.isSealed(person1)); // true

person1.sayName = function() {
  console.log(this.name);
};

console.log("sayName" in person1); // false
person1.name = "Greg";
console.log(person1.name); // "Greg"
delete person1.name;
console.log("name" in person1); // true
console.log(person1.name); // "Greg"
var descriptor = Object.getOwnPropertyDescriptor(person1, "name");
console.log(descriptor.configurable); // false
```

- This code seals person1 so you can't add or remove properties.
- Since all sealed objects are nonextensible, Object.isExtensible() returns false when used on person1, and the attempt to add a method called sayName() fails silently.
- Also, though person1.name is successfully changed to a new value, the attempt to delete it y fails.

Lab 18 : Freezing Objects

For example:

```
var person1 = {
  name: "Nicholas"
};

console.log(Object.isExtensible(person1)); // true
console.log(Object.isSealed(person1)); // false
console.log(Object.isFrozen(person1)); // false
Object.freeze(person1);
console.log(Object.isExtensible(person1)); // false
console.log(Object.isSealed(person1)); // true
console.log(Object.isFrozen(person1)); // true

person1.sayName = function() {
  console.log(this.name);
};

console.log("sayName" in person1); // false
person1.name = "Greg";
console.log(person1.name); // "Nicholas"
delete person1.name;
console.log("name" in person1); // true
console.log(person1.name); // "Nicholas"
var descriptor = Object.getOwnPropertyDescriptor(person1, "name");
console.log(descriptor.configurable); // false
console.log(descriptor.writable); // false
```

In this example:

- person1 is frozen .
- Frozen objects are also considered nonextensible and sealed, so Object.isExtensible() returns false and Object.isSealed() returns true .
- The name property can't be changed, so even though it is assigned to "Greg", the operation fails, and subsequent checks of name will still return "Nicholas".

Lesson M1.L4: Constructors and Prototypes

Lesson Objectives:

Understanding constructors and prototypes constitute the crux of Object Oreinted JavaScript so without the deeper understanding you won't truly appreciate the language. Because JavaScript lacks classes, it turns to constructors and prototypes to bring a similar order to objects. But just because some of the patterns resemble classes doesn't mean they behave the same way.

In this lesson, you'll explore constructors and prototypes in detail to see how JavaScript uses them to create objects.

After completing this lesson, you will be able to understand:

- Constructors
- Prototypes
- The [[Prototype]] Property
- Using Prototypes with Constructors
- Changing Prototypes
- Built-in Object Prototypes

Lab 19 : Prototypes

- For example:

```
var book = {
  title: "The Future Shock"
};

console.log("title" in book); // true
console.log(book.hasOwnProperty("title")); // true
console.log("hasOwnProperty" in book); // true
console.log(book.hasOwnProperty("hasOwnProperty")); // false
console.log(Object.prototype.hasOwnProperty("hasOwnProperty")); // true
```

Even though there is no definition for hasOwnProperty() on book, that method can still be accessed as book.hasOwnProperty() because the definition does exist on Object.prototype.

Remember that the in operator returns true for both prototype properties and own properties.

The [[Prototype]] Property

An instance keeps track of its prototype through an internal property called [[Prototype]].

This property is a pointer back to the prototype object that the instance is using.

When you create a new object using new, the constructor's prototype property is assigned to the [[Prototype]] property of that new object.

Lab 20 : Using Prototypes with Constructors

For example, consider the following new Person constructor:

```
function Person(name) {
  this.name = name;
}
  Person.prototype.sayName = function() {
  console.log(this.name);
};
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
console.log(person1.name); // "Nicholas"
console.log(person2.name); // "Greg"
person1.sayName(); // outputs "Nicholas"
person2.sayName(); // outputs "Greg"
```

- In this version of the Person constructor, sayName() is defined on the prototype instead of in the constructor.
- The object instances work exactly the same, even though sayName() is now a prototype property instead of an own property.
- Because person1 and person2 are each base references for their calls to sayName(), the this value is assigned to person1 and person2, respectively.

You can also store other types of data on the prototype, but be careful when using reference values. Because these values are shared across instances, you might not expect one instance to be able to change values that another instance will access.

Lab 21 : Changing Prototypes

```
function Person(name) {
  this.name = name;
}
Person.prototype = {
  constructor: Person,
  sayName: function() {
    console.log(this.name);
  },
  toString: function() {
    return "[Person " + this.name + "]";
  }
};
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
console.log("sayHi" in person1); // false
console.log("sayHi" in person2); // false
// add a new method
Person.prototype.sayHi = function() {
  console.log("Hi");
};
person1.sayHi(); // outputs "Hi"
person2.sayHi(); // outputs "Hi"
```

- In this code, the Person type starts out with only two methods, sayName() and toString() .
- Two instances of Person are created , and then the sayHi() method is added to the prototype.
- After that point, both instances can now access sayHi() .
- The search for a named property happens each time that property is accessed, so the experience is seamless.

Lesson M1.L5: Inheritance

Lesson Objectives:

Learning how to create objects is the first step to understanding object-oriented programming. The second step is to understand inheritance. In traditional object-oriented languages, classes inherit properties from other classes.

In JavaScript, however, inheritance can occur between objects with no classlike structure defining the relationship. The mechanism for this inheritance is one with which you are already familiar: prototypes.

After completing this lesson, you will be able to understand:

- Prototype Chaining and Object.prototype
- Methods Inherited from Object.prototype
- Modifying Object.prototype
- Object Inheritance
- Constructor Inheritance
- Accessing Supertype Methods

Lab 22 : Modifying Object.prototype

Take a look at what can happen:

```
Object.prototype.add = function(value) {
return this + value;
};

var book = {
title: "Future Shock"
};

console.log(book.add(5)); // "[object Object]5"
console.log("title".add("end")); // "titleend"
// in a web browser
console.log(document.add(true)); // "[object HTMLDocument]true"
console.log(window.add(5)); // "[object Window]true"
```

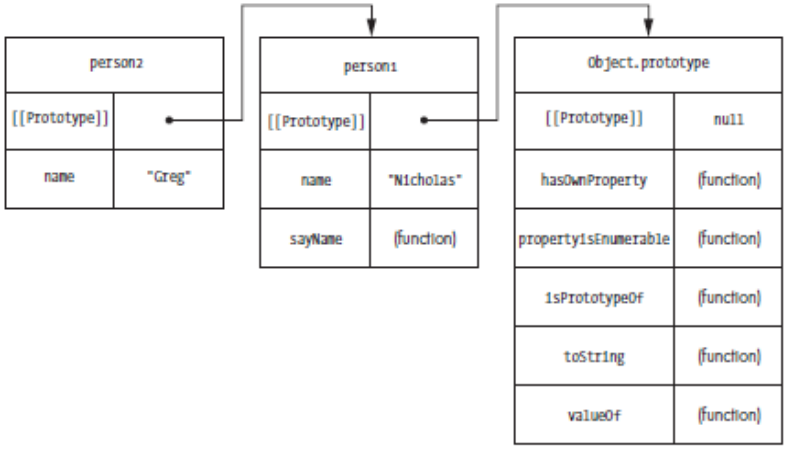
Adding Object.prototype.add() causes all objects to have an add() method, whether or not it actually makes sense.

Lab 23 : Inheriting From Other Objects

Inheriting from other objects is much more interesting:

```
var person1 = {
  name: "Nicholas",
  sayName: function() {
    console.log(this.name);
  }
};

var person2 = Object.create(person1, {
  name: {
    configurable: true,
    enumerable: true,
    value: "Greg",
    writable: true
  }
});

person1.sayName(); // outputs "Nicholas"
person2.sayName(); // outputs "Greg"
console.log(person1.hasOwnProperty("sayName")); // true
console.log(person1.isPrototypeOf(person2)); // true
console.log(person2.hasOwnProperty("sayName")); // false
```

- This code creates an object, person1, with a name property and a sayName() method.
- The person2 object inherits from person1, so it inherits both name and sayName().
- However, person2 is defined via Object.create(), which also defines an own name property for person2. This own property shadows the prototype property of the same name and is used in its place.
- So, person1.sayName() outputs "Nicholas", while person2.sayName() outputs "Greg".
- Keep in mind that sayName() still exists only on person1 and is being inherited by person2.
- The inheritance chain in this example is longer for person2 than it is for person1.
- The person2 object inherits from the person1 object, and the person1 object inherits from Object.prototype.



- When a property is accessed on an object, the JavaScript engine goes through a search process.
- If the property is found on the instance (that is, if it's an own property), that property value is used.
- If the property is not found on the instance, the search continues on [[Prototype]].
- If the property is still not found, the search continues to that object's [[Prototype]], and so on until the end of the chain is reached.
- That chain usually ends with Object.prototype, whose [[Prototype]] is set to null.

Lab 24: Constructor Inheritance

In effect, the JavaScript engine does the following for you:

```
// you write this
function YourConstructor() {
  // initialization
}
// JavaScript engine does this for you behind the scenes
YourConstructor.prototype = Object.create(Object.prototype, {
  constructor: {
    configurable: true,
    enumerable: true,
    value: YourConstructor
    writable: true
  }
});
```

So without doing anything extra, this code sets the constructor's prototype property to an object that inherits from Object.prototype, which means any instances of YourConstructor also inherit from Object .prototype.

YourConstructor is a subtype of Object, and Object is a supertype of YourConstructor.

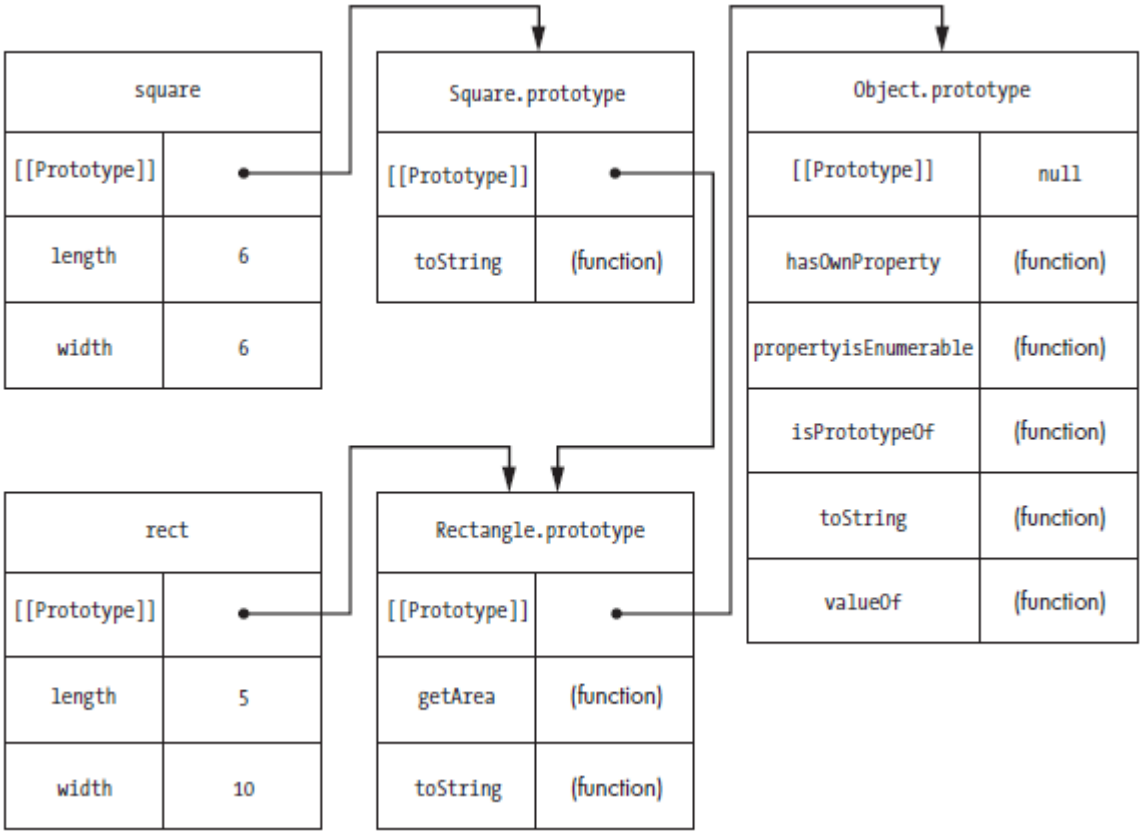
Because the prototype property is writable, you can change the prototype chain by overwriting it.

Lab 25 : Overwriting Prototype Chain

Consider the following example:

```
function Rectangle(length, width) {
  this.length = length;
  this.width = width;
}
Rectangle.prototype.getArea = function() {
  return this.length * this.width;
};
Rectangle.prototype.toString = function() {
  return "[Rectangle " + this.length + "x" + this.width + "]";
};
// inherits from Rectangle

function Square(size) {
  this.length = size;
  this.width = size;
}
Square.prototype = new Rectangle();
Square.prototype.constructor = Square;
Square.prototype.toString = function() {
  return "[Square " + this.length + "x" + this.width + "]";
};
var rect = new Rectangle(5, 10);
var square = new Square(6);
console.log(rect.getArea()); // 50
console.log(square.getArea()); // 36
console.log(rect.toString()); // "[Rectangle 5x10]"
console.log(square.toString()); // "[Square 6x6]"
console.log(rect instanceof Rectangle); // true
console.log(rect instanceof Object); // true
console.log(square instanceof Square); // true
console.log(square instanceof Rectangle); // true
console.log(square instanceof Object); // true
```

- In this code, there are two constructors: Rectangle and Square .
- The Square constructor has its prototype property overwritten with an instance of Rectangle.
- No arguments are passed into Rectangle at this point because they don't need to be used, and if they were, all instances of Square would share the same dimensions.
- To change the prototype chain this way, you always need to make sure that the constructor won't throw an error if the arguments aren't supplied and that the constructor isn't altering any sort of global state, such as keeping track of how many instances have been created.
- The constructor property is restored on Square.prototype after the original value is overwritten.
- After that, rect is created as an instance of Rectangle, and square is created as an instance of Square.
- Both objects have the getArea() method because it is inherited from Rectangle.prototype.
- The square variable is considered an instance of Square as well as Rectangle and Object because instanceof uses the prototype chain to determine the object type.



The prototype chains for square and rect show that both inherit from Rectangle.prototype and Object.prototype, but only square inherits from Square.prototype.

Square.prototype doesn't actually need to be overwritten with a Rectangle object, though; the Rectangle constructor isn't doing anything that is necessary for Square. In fact, the only relevant part is that Square.prototype needs to somehow link to Rectangle.prototype in order for inheritance to happen.

Let us simplify this example by using Object.create() once again.

```
// inherits from Rectangle
function Square(size) {
  this.length = size;
  this.width = size;
}
Square.prototype = Object.create(Rectangle.prototype, {
  constructor: {
    configurable: true,
    enumerable: true,
    value: Square,
    writable: true
  }
});
Square.prototype.toString = function() {
  return "[Square " + this.length + "x" + this.width + "]";
};
```

- In this version of the code, Square.prototype is overwritten with a new object that inherits from Rectangle.prototype, and the Rectangle constructor is never called.
- That means you don't need to worry about causing an error by calling the constructor without arguments anymore.
- Otherwise, this code behaves exactly the same as the previous code.
- The prototype chain remains intact, so all instances of Square inherit from Rectangle.prototype and the constructor is restored in the same step.

Lab 26 : Accessing Supertype Methods

```
function Rectangle(length, width) {
  this.length = length;
  this.width = width;
}
Rectangle.prototype.getArea = function() {
  return this.length * this.width;
};
Rectangle.prototype.toString = function() {
  return "[Rectangle " + this.length + "x" + this.height + "]";
};
// inherits from Rectangle
function Square(size) {
  Rectangle.call(this, size, size);
}
Square.prototype = Object.create(Rectangle.prototype, {
  constructor: {
    configurable: true,
    enumerable: true,
    value: Square,
    writable: true
  }
});
// call the supertype method
  Square.prototype.toString = function() {
  var text = Rectangle.prototype.toString.call(this);
  return text.replace("Rectangle", "Square");
};
```

- In this version of the code, Square.prototype.toString() calls Rectangle.prototype.toString() by using call().
- The method just needs to replace "Rectangle" with "Square" before returning the resulting text.
- This approach may seem a bit verbose for such a simple operation, but it is the only way to access a supertype's method.