Participant Guide

Contents

MODULE 1: OBJECT ORIENTED JAVA SCRIPT

You are in Module M1: Object Oriented Java Script.

Lesson Structure:

Lesson M1.L1: Primitive and Reference Types
Lesson M1.L2: Functions
Lesson M1.L3: Understanding JavaScript Objects
Lesson M1.L4: Constructors and Prototypes
Lesson M1.L5: Inheritance

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

What are JavaScript Types?

Type simply means data type using which a variables data type can be fixed. In JavaScript the Types are categorized as :

- **Primitive Types :** used as simple storage types
- **Reference Types :**used to store compex types which are mainly objects are really just references to locations in memory.

Some Important Points:

- JavaScript lets us treat primitive types like reference types in order to make the language more consistent for the developer. We will understand the exact differences shortly.

- Unlike traditional programming languages like Java , C#, JavaScript does not use the infrastructure of Heap and Stack to store the Reference Types and Primitive/Value Types

- JavaScript tracks variables for a particular scope with a **variable object**. Primitive values are stored directly on the variable object, while reference values are placed as a pointer in the variable object, which serves as a reference to a location in memory where the object is stored.

Primitive Types

Primitive types represent simple pieces of data that are stored as is, such as true and 25.

There are five primitive types in JavaScript:

Boolean	true or false
Number	Any integer or floating-point numeric value
String	A character or sequence of characters delimited by either single or double quotes (JavaScript has no separate character type)
Null	A primitive type that has only one value, null
Undefined	A primitive type that has only one value, undefined (undefined is the value assigned to a variable that is not initialized)

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
```

Rule of Assignment for Primitives :

" When you assign a primitive value to a variable, the value is copied into that variable. This means that if you set one variable equal to another, each variable gets its own copy of the data."

Example: Consider the following code

```
var color1 = "red";
var color2 = color1;
```

When this code gets executed , the Variable Object creates the following representation:

Variable Object	
color1	"red"
color2	"red"

Observe that :

- the primitives – color1 and color2 has the same value "red"

Now Consider the following code snippet:

```
var color1 = "red";
var color2 = color1;

console.log(color1); // "red"
console.log(color2); // "red"

color1 = "blue";
console.log(color1); // "blue"
console.log(color2); // "red"
```

Observe that :

- In this code, color1 is changed to "blue" and color2 retains its original value of "red".
- Change of value in color1 has no bearning on the variable color2's value .
- This is because there are two different storage locations, one for each variable.
- The above Figure illustrates the variable object for this snippet of code.

Identifying Primitive Types

"The best way to identify primitive types is with the typeof operator, which works on any variable and returns a string indicating the type of data."

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"
```

Special Note:

When you run typeof null, the result is "object".
But why an object when the type is null?
This has been acknowledged as an error by TC39, the committee that designs and maintains JavaScript.
You could reason that null is an empty object pointer, making "object" a logical return value,
but that's still confusing.

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
```

Comparing Without Coercion

Notice that this code uses the triple equals operator (===) instead of the double equals operator. The reason is that triple equals does the comparison without coercing the variable to another type.

To understand why this is important, consider the following:

```
console.log("5" == 5); // true
console.log("5" === 5); // false

console.log(undefined == null); // true
console.log(undefined === null); // false
```

- When you use the double equals, the string "5" and the number 5 are considered equal because the double equals converts the string into a number before it makes the comparison.
- The triple equals operator doesn't consider these values equal because they are two different types.
- Likewise, when you compare undefined and null, the double equals says that they are equivalent, while the triple equals says they are not.
- When you're trying to identify null, use triple equals so that you can correctly identify the type.

Primitive Methods

- Despite the fact that they are primitive types - strings, numbers, and Booleans actually have methods.
- The null and undefined types have no methods.
- Strings, in particular, have numerous methods to help you work with them.

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.

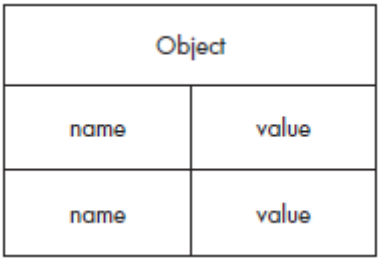
Reference Types

- Reference types represent objects in JavaScript and are the closest things to classes that you will find in the language.
- Reference values are instances of reference types and are synonymous with objects.
- An object is an unordered list of properties consisting of a name (always a string) and a value.
- When the value of a property is a function, it is called a method.
- Functions themselves are actually reference values in JavaScript, so there's little difference between a property that contains an array and one that contains a function except that a function can be executed.
- You must create objects before you can begin working with them.

Creating Objects

The simplest way to understand a JavaScript Object is to relate them to hash tables.

Structure of an object



The core of the object is made of key-value pairs.

There are two ways to create, or instantiate, objects:

- The first is to use the new operator with a constructor
- The second is with the assignment operator by means of assigning one object to another

Creating Objects : Using new Operator with a Constructor

- A constructor is simply a function that uses new operator to create an object—any function can be a constructor
- By convention, **constructors** in JavaScript begin with a **capital letter** to distinguish them from nonconstructor functions.
- For example, this code instantiates a generic object and stores a reference to it in object:

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

Creating Objects : Using Assignment Operator

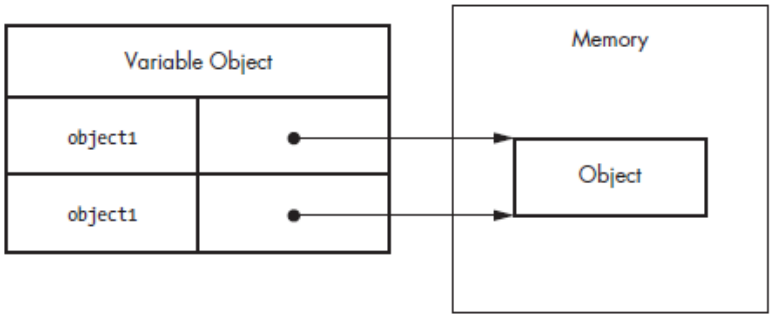
- Reference types do not store the object directly into the variable to which it is assigned, so the object variable in the following code does not actually contain the object instance.

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

- Instead, it holds a pointer (or reference) to the location in memory where the object exists.
- This is the primary difference between objects and primitive values, **as the primitive is stored directly in the variable.**
- When you assign an object to a variable, you are actually assigning a pointer. That means if you assign one variable to another, each variable gets a copy of the pointer, and both still reference the same object in memory.

```
var object1 = new Object();
var object2 = object1;
```

- This code first creates an object (with new operator and constructor) and stores a reference in object1. Next, object2 is assigned the value of object1. There is still only the one instance of the object that was created on the first line, but both variables now point to that object.



Dereferencing Objects

JavaScript is a garbage-collected language. So you don't really need to worry about memory allocations when you use reference types. However, it's best to dereference objects that you no longer need so that the garbage collector can free up that memory. The best way to do this is to set the object variable to null.

```
var object1 = new Object();

// do something

object1 = null; // dereference
```

- Here, object1 is created and used before finally being set to null.
- When there are no more references to an object in memory, the garbage collector can use that memory for something else.
- Dereferencing objects is especially important in very large applications that use many objects.

Adding or Removing Properties

Another interesting aspect of objects in JavaScript is that you can add and remove properties at any time.

For example:

```
var object1 = new Object();
var object2 = object1;

object1.myCustomProperty = "Awesome!";
console.log(object2.myCustomProperty); // "Awesome!"
```

- Here, myCustomProperty is added to object1 with a value of "Awesome!".
- That property is also accessible on object2 because both object1 and object2 point to the same object.

Instantiating Built-in Types

JavaScript provides many Built-in Types . By creating the objects you can borrow a lot of preexisting functionalities.

The built-in types are:

Array	An ordered list of numerically indexed values
Date	A date and time
Error	A runtime error (there are also several more specific error subtypes)
Function	A function
Object	A generic object
RegExp	A regular expression

You can instantiate each built-in reference type using new, as shown in the code below:

```
var items = new Array();
var now = new Date();
var error = new Error("Something bad happened.");
var func = new Function("console.log('Hi');");
var object = new Object();
var re = new RegExp("\\d+");
```

Literal Forms

Several built-in reference types have literal forms. A literal is syntax that allows you to define a reference value without explicitly creating an object, using the new operator and the object's constructor.

Object Literals

- To create an object with object literal syntax, you can define the properties of a new object inside braces.
- Properties are made up of an identifier or string, a colon, and a value, with multiple properties separated by commas.
- For example:

```
var book = {
name: "The Future Shock",
year: 1970,
author: "Alvin Toffler"
};
```

- You can also use string literals as property names, which is useful when you want a property name to have spaces or other special characters:
- For example:

```
var book = {
"name" : "The Future Shock",
"year" : 1970,
"author name" : "Alvin Toffler"
};


console.log(book["author name"]);
```

- This example is equivalent to the previous one despite the syntactic differences. Both examples are also logically equivalent to the following:

```
var book = {
"name" : "The Future Shock",
"year" : 1970,
"author name" : "Alvin Toffler"
};

console.log(book["author name"]);

var book1 = new Object();
book1.name = "The Future Shock";
book1.year = 1970;
book1["author name"] = "Alvin Toffler";

console.log(book1["author name"]);
```

Array Literals

- You can define an array literal in a similar way by enclosing any number of comma-separated values inside square brackets.
- For example:

```
var colors = [ "red", "blue", "green" ];
console.log(colors[0]); // "red"
```

- This code is equivalent to the following:

```
var colors = new Array("red", "blue", "green")
console.log(colors[0]); // "red"
```

Function Literals

- You almost always define functions using their literal form.
- In fact, using the Function constructor is typically discouraged given the challenges of maintaining, reading, and debugging a string of code rather than actual code.
- Creating functions is much easier and less error prone when you use the literal form.
- For example:

```
// Function Literal Form

function reflect(value) {
return value;
}

// constructor form
// is the same as
var reflect = new Function("value", "return value;");
```

- This code defines the reflect() function, which returns any value passed to it.
- Even in the case of this simple function, the literal form is easier to write and understand than the constructor form.
- Further, there is no good way to debug functions that are created in the constructor form: **These functions are not recognized by JavaScript debuggers and therefore act as a black box in your application.**

Property Access

Properties are name/value pairs that are stored on an object.

Dot notation is the most common way to access properties in JavaScript (as in many object-oriented languages), but you can also access properties on JavaScript objects by using bracket notation with a string.

For example, you could write this code, which uses dot notation:

```
var array = [];
array.push(12345);
```

With bracket notation, the name of the method is now included in a string enclosed by square brackets, as in this example:

```
var array = [];
array["push"](12345);
```

This syntax is very useful when you want to dynamically decide which property to access. For example, here bracket notation allows you to use a variable instead of the string literal to specify the property to access.

```
var array = [];
var method = "push";
array[method](12345);
```

Identifying Reference Types

A function is the easiest reference type to identify because when you use the typeof operator on a function, the operator should return "function":

```
function reflect(value) {
return value;
}
console.log(typeof reflect); // "function"
```

Other reference types are trickier to identify because, for all reference types other than functions, typeof returns "object". That's not very helpful when you're dealing with a lot of different types. To identify reference types more easily, you can use JavaScript's **instanceof operator**.

The **instanceof operator** takes an object and a constructor as parameters. When the value is an instance of the type that the constructor specifies, instanceof returns true; otherwise, it returns false, as you can see here:

```
var items = [];
var object = {};
function reflect(value) {
return value;
}

console.log(items instanceof Array); // true
console.log(object instanceof Object); // true
console.log(reflect instanceof Function); // true
```

The **instanceof operator** can identify **inherited types**. That means every object is actually an instance of Object because every reference type inherits from Object.

To Labnstrate, the following listing examines the three references previously created with instanceof:

```
var items = [];
var object = {};
function reflect(value) {
return value;
}

console.log(items instanceof Array); // true
console.log(items instanceof Object); // true
console.log(object instanceof Object); // true
console.log(object instanceof Array); // false
console.log(reflect instanceof Function); // true
console.log(reflect instanceof Object); // true
```

Identifying Arrays

Although instanceof can identify arrays, there is one exception that affects web developers: JavaScript values can be passed back and forth between frames in the same web page. This becomes a problem only when you try to identify the type of a reference value, because each web page has its own global context—its own version of Object, Array, and all other builtin types. As a result, when you pass an array from one frame to another, instanceof doesn't work because the array is actually an instance of Array from a different frame.

To solve this problem, ECMAScript 5 introduced Array.isArray(), which definitively identifies the value as an instance of Array regardless of the value's origin. This method should return true when it receives a value that is a native array from any context. If your environment is ECMAScript 5 compliant, Array.isArray() is the best way to identify arrays:

```
var items = [];
console.log(Array.isArray(items)); // true
```

The Array.isArray() method is supported in most environments, both in browsers and in Node.js. This method isn't supported in Internet Explorer 8 and earlier.

Primitive Wrapper Types

There are three primitive wrapper types – **String , Number , Boolean**

These special reference types exist to make working with primitive values as easy as working with objects.

The primitive wrapper types are reference types that are automatically created behind the scenes whenever strings, numbers, or Booleans are read. For example, in the first line of this listing, a primitive string value is assigned to name. The second line treats name like an object and calls charAt(0) using dot notation.

```
var name = "Nicholas";
var firstChar = name.charAt(0);
console.log(firstChar); // "N"
```

This is what happens behind the scenes:

```
// what the JavaScript engine does
var name = "Nicholas";
var temp = new String(name);
var firstChar = temp.charAt(0);
temp = null;
console.log(firstChar); // "N"
```

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

What are Functions?

Functions are reusable blocks of code. They can be named or anonymous.

The defining characteristic of a function—what distinguishes it from any other object—is the presence of an **internal property named [[Call]]**. Internal properties are not accessible via code but rather define the behavior of code as it executes. ECMAScript defines multiple internal properties for objects in JavaScript, and these internal properties are indicated by double-square-bracket notation.

The [[Call]] property is unique to functions and indicates that the object can be executed. Because only functions have this property, the **typeof operator** is defined by ECMAScript to return "function" for any object with a [[Call]] property.

Declarations vs. Expressions

There are actually two literal forms of functions:

- Function Declaration
- Function Expression

Function Declaration

You can declare Functions with Function Declaration Literal.

- The function declaration begins with the **function keyword** and includes the **name of the function** immediately following it.
- The contents of the function are enclosed in braces, as shown in this declaration:

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

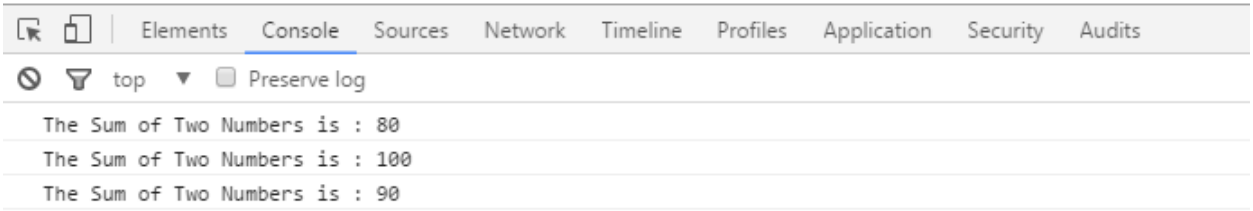
Once a function is declared using the Function Declaration Literal , it can be called any number of times as per the need and context of the program.

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 :



Function Expression

You can declare Functions with Function Expression Literal.

- Function Expression does not require a name after function.
- These functions are considered **anonymous** because the function object itself has no name.
- Instead, function expressions are typically referenced via a variable or property, as in this expression:

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

- This code actually assigns a **function value** to the **variable add**.

- The function expression is almost identical to the function declaration except for the missing name and the semicolon at the end.

- Assignment expressions typically end with a semicolon, just as if you were assigning any other value.

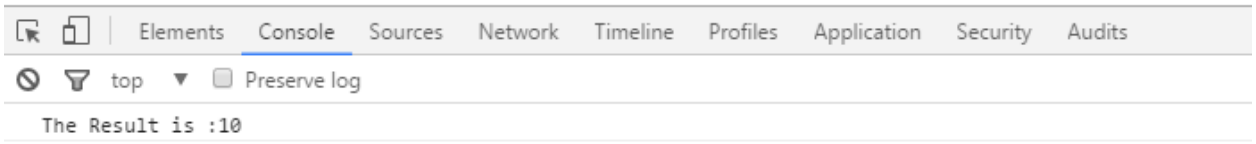
Function Declaration Vs Function Expression : What is the difference?

Although these two forms are quite similar, they differ in a very important way.

- **Function declarations are hoisted to the top of the context when the code is executed.**
- You can actually define a function after it is used in code without generating an error.
- For example:

```
var result = add(5, 5);

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

console.log("The Result is :" + result);
```

- Generates the following output:



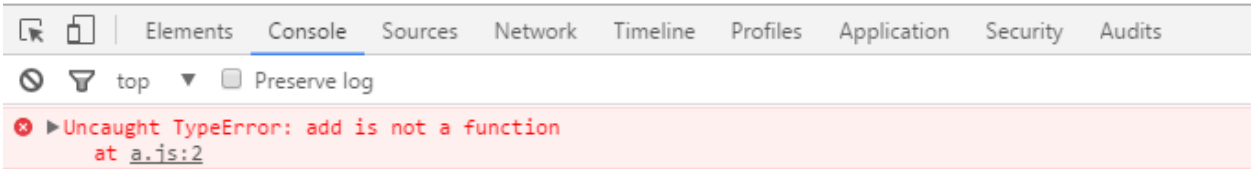
- The above code might look like it will cause an error, but it works just fine because the JavaScript engine hoists the function declaration to the top and actually executes the code as if it were written like this:

```
// how the JavaScript engine interprets the code
function add(num1, num2) {
return num1 + num2;
}
var result = add(5, 5);
```

- **Function expressions, on the other hand, cannot be hoisted because the functions can be referenced only through a variable.**
- For example:

```
// error!
var result = add(5, 5);
var add = function(num1, num2) {
return num1 + num2;
};
```

- So this code causes an error:



So the Rules is

" Function hoisting happens only for function declarations because the function name is known ahead of time. Function expressions, on the other hand, cannot be hoisted because the functions can be referenced only through a variable. As long as you always define functions before using them, you can use either function declarations or function expressions."

Functions as Values

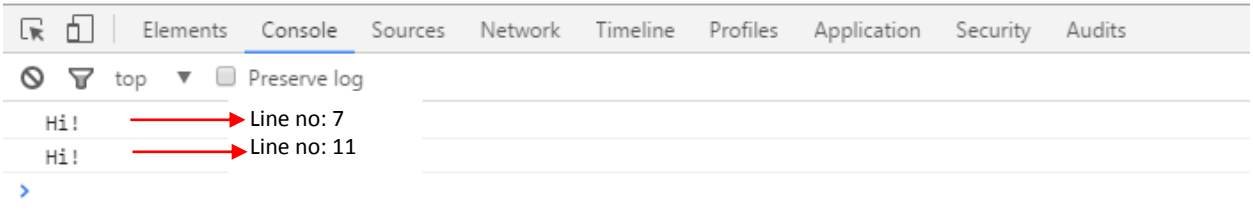
- Because JavaScript has first-class functions, you can use them just as you do any other objects.
- You can assign them to variables, add them to objects, pass them to other functions as arguments, and return them from functions.
- Basically, you can use a function anywhere you would use any other reference value.
- This makes JavaScript functions incredibly powerful.

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:



Parameters

Another unique aspect of JavaScript functions is that you can pass any number of parameters to any function without causing an error.

- The **function parameters** are actually stored as an **array-like** structure called **arguments**.
- Just like a regular JavaScript array, arguments can grow to contain any number of values.
- The values are referenced via numeric **indices**, and there is a **length property** to determine how many values are present.
- The **arguments object** is automatically available inside any function.
- This means named parameters in a function exist mostly for convenience and don't actually limit the number of arguments that a function can accept.

The arguments object is not an instance of Array and therefore doesn't have the same methods as an array; Array.isArray(arguments) always returns false.

On the other hand, JavaScript **doesn't ignore the named parameters** of a function either.

The number of arguments a function expects is stored on the **function's length property**. Remember, a function is actually just an object, so it can have properties. The length property indicates the function's arity, or the number of parameters it expects.

Knowing the function's arity is important in JavaScript because functions won't throw an error if you pass in too many or too few parameters.

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.

Overloading

Most object-oriented languages support function overloading, which is the ability of a single function to have multiple signatures.

A function signature is made up of the function name plus the number and type of parameters the function expects. Thus, a single function can have one signature that accepts a single string argument and another that accepts two numeric arguments. The language determines which version of a function to call based on the arguments that are passed in.

As mentioned previously, JavaScript functions can accept any number of parameters, and the types of parameters a function takes aren't specified at all. That means JavaScript functions don't actually have signatures.

A lack of function signatures also means a lack of function overloading.

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 "

Object Methods

- In JavaScript you can add and remove properties from objects at any time.
- When a property value is actually a function, the property is considered a **method**.
- You can add a method to an object in the same way that you would add a property.
- For example, in the following code, the person variable is assigned an object literal with a name property and a method called sayName.

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

person.sayName(); // outputs "Nicholas"
```

- Note that the syntax for a data property and a method is exactly the same—an identifier followed by a colon and the value.
- In the case of sayName, the value just happens to be a function.
- You can then call the method directly from the object as in person.sayName("Nicholas")

The this Object

Every scope in JavaScript has a **this object** that represents the **calling object for the function**.

- In the global scope, this represents the global object (window in web browsers).
- When a function is called while attached to an object, the v**alue of this is equal to that object by default.**
- So, instead of directly referencing an object inside a method, you can reference this instead.
- For example, you can rewrite the code from the previous example to use this:

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

  person.sayName(); // outputs "Nicholas"
```

- Let us take another example to understand the working of this object:

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().

Defining Properties

There are two basic ways to create your own objects using:

- Object constructor
- Object literal.

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

When a property is first added to an object, JavaScript uses an **internal method called [[Put]]** on the object.

The [[Put]] method creates a spot in the object to store the property. This is simlar to setter / write method for a given property.

This operation specifies not just the initial value, but also some attributes of the property.

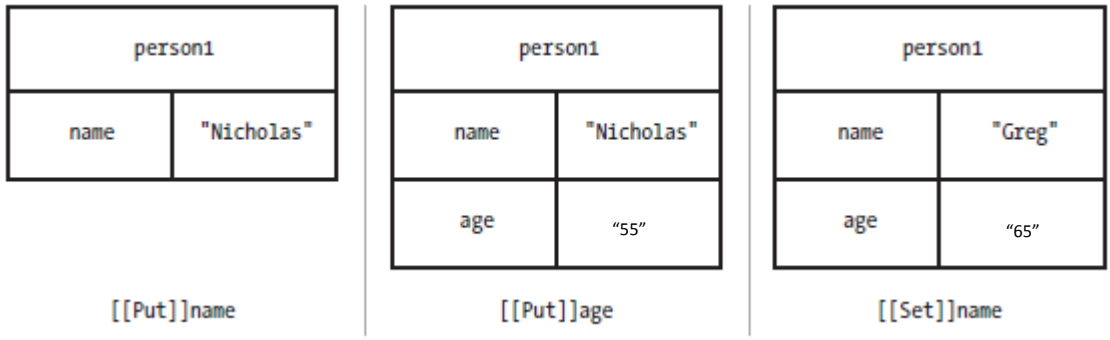
So, in the previous example, when the name and age properties are first defined on each object, the [[Put]] method is invoked for each.

The result of calling [[Put]] is the creation of an own property on the object. An **own property** simply indicates that the specific instance of the object owns that property. The property is stored directly on the instance, and all operations on the property must be performed through that object.

When a new value is assigned to an existing property, a separate operation called [[Set]] takes place. This operation replaces the current value of the property with the new one.

In the previous example, setting name to a second value results in a call to [[Set]].

See the follwing Figure for a step-bystep view of what happened to person1 behind the scenes as its name and age properties were changed.



- In the first part of the diagram, an object literal is used to create the person1 object.
- This performs an implicit [[Put]] for the name property.
- Assigning a value to person1.age performs a [[Put]] for the age property.
- However, setting person1.name to a new value ("Greg") performs a [[Set]] operation on the name property, overwriting the existing property value.

Detecting Properties

Because properties can be added at any time, it is sometimes necessary to check whether a property exists in the object.

The following code is unreliable:

```
// unreliable
if (person1.age) {
// do something with age
}
```

- The problem with this code is how JavaScript's type coercion affects the outcome.
- The if condition evaluates to true if the value is truthy (an object, a nonempty string, a nonzero number, or true) and evaluates to false if the value is falsy (null, undefined, 0, false, NaN, or an empty string).
- Because an object property can contain one of these falsy values, the example code can yield false negatives.
- For instance, if person1.age is 0, then the if condition will not be met even though the property exists.
- A more reliable way to test for the existence of a property is with the in operator.

The **in operator** looks for a property with a given name in a specific object and returns true if it finds it. In effect, the in operator checks to see if the given key exists in the hash table.

For example, here is what happens when in is used to check for some properties in the person1 object:

```
console.log("name" in person1); // true
console.log("age" in person1); // true
console.log("title" in person1); // false
```

Keep in mind that methods are just properties that reference functions, **so you can check for the existence of a method in the same way**.

The following adds a new function, sayName(), to person1 and uses in to confirm the function's presence.

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

In most cases, the in operator is the best way to determine whether the property exists in an object. It has the added benefit of not evaluating the value of the property, which can be important if such an evaluation is likely to cause a performance issue or an error.

Removing Properties

Just as properties can be added to objects at any time, they can also be removed.

Simply setting a property to null doesn't actually remove the property completely from the object. Such an operation calls [[Set]] with a value of null, only replaces the value of the property.

You need to use the delete operator to completely remove a property from an object.

The delete operator works on a single object property and calls an internal operation named [[Delete]].

You can think of this operation as removing a key/value pair from a hash table. When the delete operator is successful, it returns true.

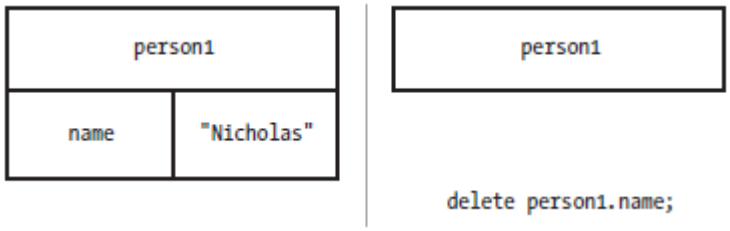
Lab 12 : Removing Properties

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.

Enumeration

By default, all properties that you add to an object are enumerable, which means that you can iterate over them using a for-in loop.

Enumerable properties have their **internal [[Enumerable]] attributes** set to true.

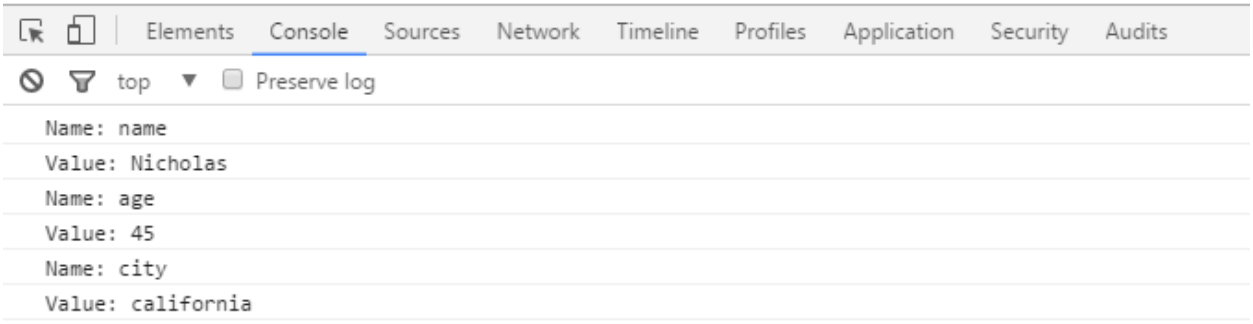
The for-in loop enumerates all enumerable properties on an object, assigning the property name to a variable.

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]);
}
```

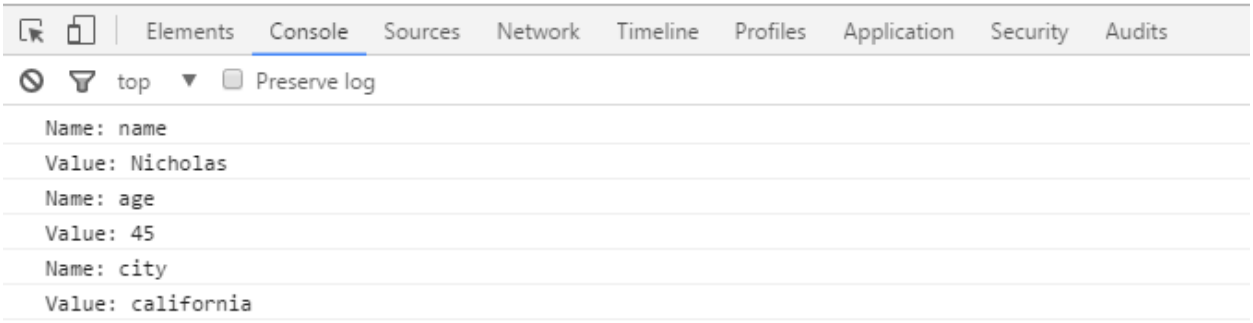
Observe the output:



If you just need a list of an object's properties to use later in your program, ECMAScript 5 introduced the Object.keys() method to retrieve an array of enumerable property names, as shown here:

```
var properties = Object.keys(person1);
// if you want to mimic for-in behavior
var i, len;
for (i=0, len=properties.length; i < len; i++){
console.log("Name: " + properties[i]);
console.log("Value: " + person1[properties[i]]);
}
```

Output:



This example uses:

- Object.keys() to retrieve the enumerable properties from an object .
- A for loop is then used to iterate over the properties and output the name and value.
- Typically, you would use Object.keys() in situations where you want to operate on an array of property names and for-in when you don't need an array.

Types of Properties

There are two different types of properties:

- Data properties
- Accessor properties.

Data properties contain a value, like the name property.

Accessor properties don't contain a value but instead define a function to call when the property is read (called a getter), and a function to call when the property is written to (called a setter).

The default behavior of the [[Put]] method is to create a data property.

Accessor properties only require either a getter or a setter, though they can have both.

There is a special syntax to define an accessor property using an object literal:

```
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;
  }
};

console.log(person1.name); // "Reading name" then "Nicholas"
person1.name = "Greg";
console.log(person1.name); // "Setting name to Greg" then "Greg"
```

- This example defines an accessor property called **name**.
- There is a data property called **_name** that contains the actual value for the property.
- The syntax used to define the getter and setter for name looks a lot like a function but without the function keyword.
- The special keywords get and set are used before the accessor property name, followed by parentheses
- and a function body.
- Getters are expected to return a value, while setters receive the value being assigned to the property as an argument.
- **It is important to note that any reading operation on a property will always invloke the get and writing operation on a property will invoke set automatically. Also this results in extremely simplifed natural syntax.**

Property Attributes

This section covers in detail the attributes of both data and accessor properties, starting with the ones they have in common.

Common Attributes

There are two property attributes shared between data and accessor properties:

- One is **[[Enumerable]]** which determines whether you can iterate over the property.
- The other is **[[Configurable]]** which determines whether the property can be changed.

You can remove a configurable property using delete and can change its attributes at any time. By default, all properties you declare on an object are both enumerable and configurable.

If you want to change property attributes, you can use the **Object .defineProperty()** method.

This method accepts three arguments:

- the object that owns the property,
- the property name, and
- a property descriptor object containing the attributes to set.

The descriptor has properties with the same name as the internal attributes but without the square brackets. So you use enumerable to set [[Enumerable]], and configurable to set [[Configurable]].

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.

Data Property Attributes

Data properties possess two additional attributes that accessors do not:

- The first is **[[Value]],** which holds the property value. This attribute is filled in automatically when you create a property on an object. All property values are stored in [[Value]], even if the value is a function.

- The second attribute is **[[Writable]],** which is a Boolean value indicating whether the property can be written to. By default, all properties are writable unless you specify otherwise.

With these two additional attributes, you can fully define a data property using Object.defineProperty() even if the property doesn't already exist.

Consider this code:

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

You've seen this snippet throughout this lesson; it adds the name property to person1 and sets its value.

You can achieve the same result using the following (more verbose) code:

```
var person1 = {};

Object.defineProperty(person1, "name", {
value: "Nicholas",
enumerable: true,
configurable: true,
writable: true
});
```

- When Object.defineProperty() is called, it first checks to see if the property exists. If the property doesn't exist, a new one is added with the attributes specified in the descriptor. In this case, name isn't already a property of person1, so it is created.

When you are defining a new property with Object.defineProperty(), it's important to specify all of the attributes because Boolean attributes automatically default to false otherwise.

For example, the following code creates a name property that is nonenumerable, nonconfigurable, and nonwritable because it doesn't explicitly make any of those attributes true in the call to Object.defineProperty().

```
var person1 = {};

Object.defineProperty(person1, "name", {
value: "Nicholas"
});

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, you can't do anything with the name property except read the value; every other operation is locked down. If you're changing an existing property, keep in mind that only the attributes you specify will change.

Accessor Property Attributes

Accessor properties also have two additional attributes. Accessors have **[[Get]] and [[Set]],** which contain the getter and setter functions, respectively. As with the object literal form of getters and setters, you need only define one of these attributes to create the property.

The advantage of using accessor property attributes instead of object literal notation to define accessor properties is that you can also define those properties on existing objects. If you want to use object literal notation, you have to define accessor properties when you create the object. As with data properties, you can also specify whether accessor properties are configurable or enumerable.

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.

Defining Multiple Properties

It is also possible to define multiple properties on an object simultaneously if you use Object.defineProperties() instead of Object.defineProperty().

This method accepts two arguments:

- the object to work on and
- an object containing all of the property information.

The keys of that second argument are property names, and the values are descriptor objects defining the attributes for those properties.

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.

Retrieving Property Attributes

If you need to fetch property attributes, you can do so in JavaScript by using:
Object.getOwnPropertyDescriptor()

As the name suggests, this method works only on own properties.

This method accepts two arguments**: the object to work on and the property name to retrieve.**

If the property exists, you should receive a descriptor object with four properties: configurable, enumerable, and the two others appropriate for the type of property.

Even if you didn't specifically set an attribute, you will still receive an object containing the appropriate value for that attribute.

For example, this code creates a property and checks its attributes:

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

var descriptor = Object.getOwnPropertyDescriptor(person1, "name");
console.log(descriptor.enumerable); // true
console.log(descriptor.configurable); // true
console.log(descriptor.writable); // true
console.log(descriptor.value); // "Nicholas"
```

- Here, a property called name is defined as part of an object literal.
- The call to Object.getOwnPropertyDescriptor() returns an object with enumerable, configurable, writable, and value, even though these weren't explicitly defined via Object.defineProperty().

Preventing Object Modification

Objects, just like properties, have internal attributes that govern their behavior.

One of these attributes is [[Extensible]], which is a Boolean value indicating if the object itself can be modified. All objects you create are extensible by default, meaning new properties can be added to the object at any time.

By setting [[Extensible]] to false, you can prevent new properties from being added to an object.

There are three different ways to accomplish this:

- Preventing Extensions
- Sealing Objects
- Freezing Objects

Preventing Extensions

One way to create a nonextensible object is with Object.preventExtensions().

This method accepts a single argument, which is the object you want to make nonextensible.

Once you use this method on an object, you'll never be able to add any new properties to it again.

You can check the value of [[Extensible]] by using Object.isExtensible().

The following code shows examples of both methods at work.

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

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

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

console.log("sayName" in person1); // false
```

After creating person1, this example checks the object's [[Extensible]] attribute before making it unchangeable .

Now that person1 is nonextensible, the sayName() method w is never added to it.

Sealing Objects

The second way to create a nonextensible object is to seal the object.

A sealed object is nonextensible, and all of its properties are nonconfigurable.

That means not only can you not add new properties to the object, but you also can't remove properties or change their type (from data to accessor or vice versa). If an object is sealed, you can only read from and write to its properties.

You can use the **Object.seal()** method on an object to seal it. When that happens, the [[Extensible]] attribute is set to false, and all properties have their [[Configurable]] attribute set to false.

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.

Freezing Objects

The last way to create a nonextensible object is to freeze it.

If an object is frozen, you can't add or remove properties, you can't change properties' types, and you can't write to any data properties.

In essence, a frozen object is a sealed object where data properties are also read-only.

Frozen objects can't become unfrozen, so they remain in the state they were in when they became frozen.

You can freeze an object by using Object.freeze() and determine if an object is frozen by using Object.isFrozen().

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

Constructors

A constructor is simply a function that is used with new to create an object.

Up to this point, you have seen several of the built-in JavaScript constructors, such as Object, Array, and Function.

The advantage of constructors is that objects created with the same constructor contain the same properties and methods. If you want to create multiple similar objects, you can create your own constructors and therefore your own reference types.

Because a constructor is just a function, you define it in the same way. **The only difference is that constructor names should begin with a capital letter, to distinguish them from other functions.**

For example, look at the following empty Person function:

```
function Person() {
// intentionally empty
}
```

This function is a constructor, but there is absolutely no syntactic difference between this and any other function. The clue that Person is a constructor is in the name—the first letter is capitalized.

After the constructor is defined, you can start creating instances, like the following two Person objects:

```
var person1 = new Person();
var person2 = new Person();
```

When you have **no parameters** to pass into your constructor, you can even omit the parentheses:

```
var person1 = new Person;
var person2 = new Person;
```

Even though the Person constructor doesn't explicitly return anything, both person1 and person2 are considered instances of the new Person type.

The new operator automatically creates an object of the given type and returns it.

That also means you can use the instanceof operator to deduce an object's type.

The following code shows instanceof in action with the newly created objects:

```
console.log(person1 instanceof Person); // true
console.log(person2 instanceof Person); // true
```

Because person1 and person2 were created with the Person constructor, instanceof returns true when it checks whether these objects are instances of the Person type.

You can also check the type of an instance using the **constructor property**.

Every object instance is automatically created with a **constructor property** that contains a reference to the constructor function that created it.

For generic objects (those created via an object literal or the Object constructor), constructor is set to Object; for objects created with a custom constructor, constructor points back to that constructor function instead.

For example, Person is the constructor property for person1 and person2:

```
console.log(person1.constructor === Person); // true
console.log(person2.constructor === Person); // true
```

The console.log function outputs true in both cases, because both objects were created with the Person constructor.

Even though this relationship exists between an instance and its constructor, you are still advised to use **instanceof** to check the type of an instance. **This is because the constructor property can be overwritten and therefore may not be completely accurate.**

Of course, an empty constructor function isn't very useful. The whole point of a constructor is to make it easy to create more objects with the same properties and methods.

To do that, simply add any properties you want to this inside of the constructor, as in the following example:

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

- This version of the Person constructor accepts a single named parameter, name, and assigns it to the name property of the this object .
- The constructor also adds a sayName() method to the object .
- The this object is automatically created by new when you call the constructor, and it is an instance of the constructor's type. (In this case, this is an instance of Person.)
- There's no need to return a value from the function because the new operator produces the return value.

Now you can use the Person constructor to create objects with an initialized name property:

```
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"
```

Each object has its own name property, so sayName() should return different values depending on the object on which you use it.

Constructors allow you to initialize an instance of a type in a consistent way, performing all of the property setup that is necessary before the object can be used.

For example, you could also use **Object.defineProperty()** inside of a constructor to help initialize the instance:

```
function Person(name) {
  Object.defineProperty(this, "name", {
    get: function() {
      return name;
    },
    set: function(newName) {
      name = newName;
    },
    enumerable: true,
    configurable: true
  });
  this.sayName = function() {
    console.log(this.name);
  };
}
```

In this version of the Person constructor, the name property is an accessor property that uses the name parameter for storing the actual name.

This is possible because named parameters act like local variables.

Make sure to always call constructors with **new**; otherwise, you risk changing the global object instead of the newly created object.

Consider what happens in the following code:

```
var person1 = Person("Nicholas"); // note: missing "new"
console.log(person1 instanceof Person); // false
console.log(typeof person1); // "undefined"
console.log(name); // "Nicholas"
```

- When Person is called as a function without new, the value of this inside of the constructor is equal to the global this object.
- The variable person1 doesn't contain a value because the Person constructor relies on new to supply a return value.
- Without new, Person is just a function without a return statement.
- The assignment to this.name actually creates a global variable called name, which is where the name passed to Person is stored.

Prototypes

Constructors allow you to configure object instances with the same properties, but constructors alone don't eliminate code redundancy.

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

In the example code thus far, each instance has had its own sayName() method even though sayName() doesn't change.

That means if you have 100 instances of an object, then there are 100 copies of a function that do the exact same thing, just with different data.

It would be much more efficient if all of the instances shared one method, and then that method could use this.name to retrieve the appropriate data.

- This is where prototypes come in.
- You can think of a prototype as a recipe for an object.
- Almost every function (with the exception of some built-in functions) has a prototype property that is used during the creation of new instances.
- That prototype is shared among all of the object instances, and those instances can access properties of the prototype.
- For example, the hasOwnProperty() method is defined on the generic Object prototype, but it can be accessed from any object as if it were an own property.

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.

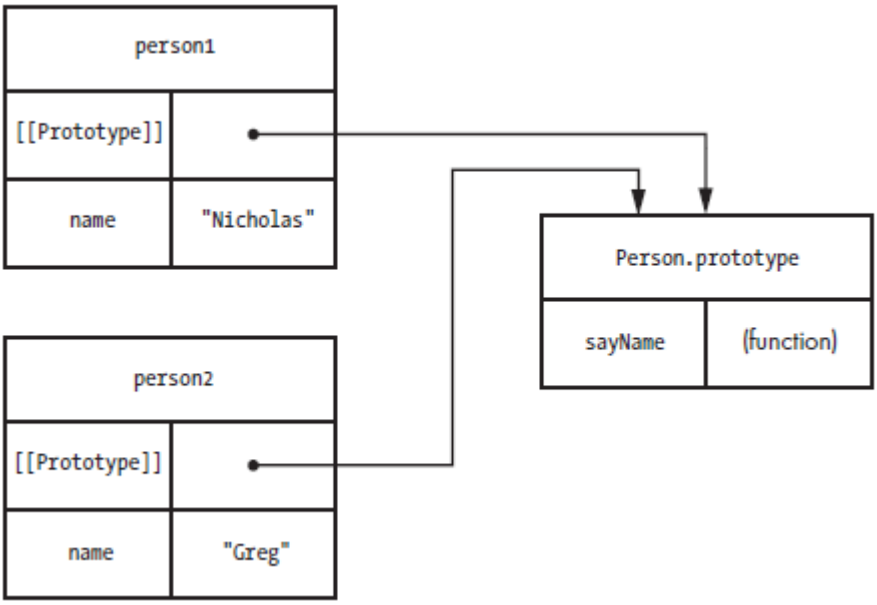
The [[Prototype]] Property

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.

The following Figure shows how the **[[Prototype]] property** lets multiple instances of an object type refer to the same prototype, which can reduce code duplication.



The [[Prototype]] properties for person1 and person2 point to the same prototype.

You can read the value of the [[Prototype]] property by using the Object.getPrototypeOf() method on an object.

For example, the following code checks the [[Prototype]] of a generic, empty object.

```
var object = {};
var prototype = Object.getPrototypeOf(object);
console.log(prototype === Object.prototype); // true
```

For any generic object like this one , **[[Prototype]] is always a reference to Object.prototype.**

You can also test to see if one object is a prototype for another by using the isPrototypeOf() method, which is included on all objects:

```
var object = {};
console.log(Object.prototype.isPrototypeOf(object)); // true
```

Because object is just a generic object, its prototype should be Object.prototype, meaning isPrototypeOf() should return true.

When a property is read on an object, the JavaScript engine first looks for an own property with that name. If the engine finds a correctly named own property, it returns that value.

If no own property with that name exists on the target object, JavaScript searches the [[Prototype]] object instead. If a prototype property with that name exists, the value of that property is returned.

If the search concludes without finding a property with the correct name, undefined is returned.

Consider the following, in which an object is first created without any own properties:

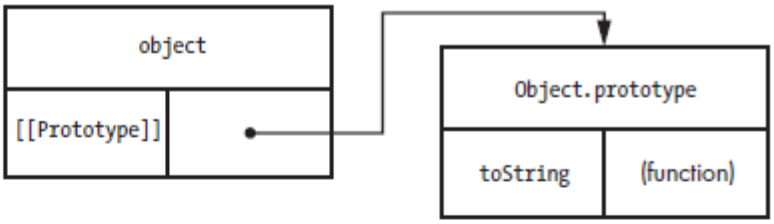
```
var object = {};

console.log(object.toString()); // "[object Object]"
object.toString = function() {
  return "[object Custom]";
};

console.log(object.toString()); // "[object Custom]"
// delete own property
delete object.toString;
console.log(object.toString()); // "[object Object]"
// no effect - delete only works on own properties
delete object.toString;
console.log(object.toString()); // "[object Object]"
```

- In this example, the toString() method comes from the prototype and returns "[object Object]" by default.
- If you then define an own property called toString(), that own property is used whenever toString() is called on the object again .
- The own property shadows the prototype property, so the prototype property of the same name is no longer used.
- The prototype property is used again only if the own property is deleted from the object .

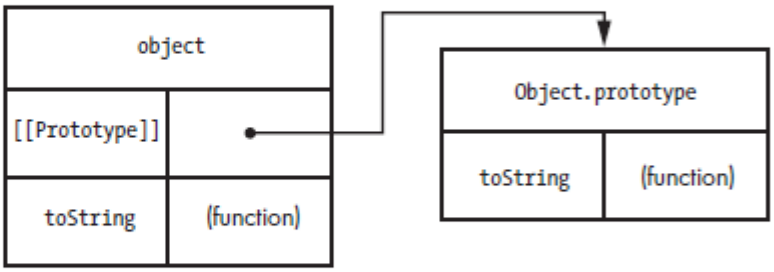
This example also highlights an important concept:
You cannot assign a value to a prototype property from an instance.

As you can see in the middle section of Figure, assigning a value to toString creates a new own property on the instance, leaving the property on the prototype untouched.

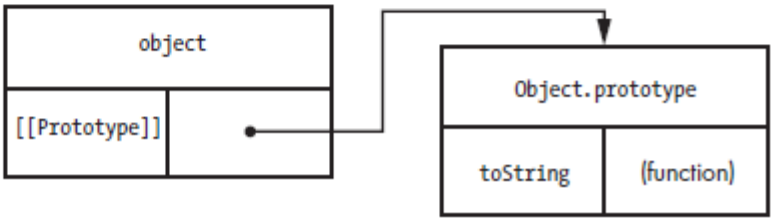
```
var object = {};
```




```
object.toString = function() {
    return "[object Custom]";
};
```




```
delete object.toString;
```



Using Prototypes with Constructors

The shared nature of prototypes makes them ideal for defining methods once for all objects of a given type. Because methods tend to do the same thing for all instances, there's no reason each instance needs its own set of methods.

It's much more efficient to put the methods on the prototype and then use this to access the current instance.

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.

This example shows what can happen when you don't watch where your reference values are pointing:

```
function Person(name) {
  this.name = name;
}

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

Person.prototype.favorites = [];
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
person1.favorites.push("pizza");
person2.favorites.push("quinoa");
console.log(person1.favorites); // "pizza,quinoa"
console.log(person2.favorites); // "pizza,quinoa"
```

- The favorites property is defined on the prototype, which means person1.favorites and person2.favorites point to the same array.
- Any values you add to either person's favorites will be elements in that array on the prototype.
- That may not be the behavior that you actually want, so it's important to be very careful about what you define on the prototype.

Even though you can add properties to the prototype one by one, many developers use a more succinct pattern that involves replacing the prototype with an object literal:

```
function Person(name) {
  this.name = name;
}
Person.prototype = {
  sayName: function() {
    console.log(this.name);
  },
  toString: function() {
    return "[Person " + this.name + "]";
  }
};
```

- This code defines two methods on the prototype, sayName() and toString() .
- This pattern has become quite popular because it eliminates the need to type Person.prototype multiple times.

Changing Prototypes

Because all instances of a particular type reference a shared prototype, you can augment all of those objects together at any time.

Remember, the [[Prototype]] property just contains a pointer to the prototype, and any changes to the prototype are immediately available on any instance referencing it.

That means you can literally add new members to a prototype at any point and have those changes reflected on existing instances, as in this example:

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.

Built-in Object Prototypes

At this point, you might wonder if prototypes also allow you to modify the built-in objects that come standard in the JavaScript engine. The answer is yes.

All built-in objects have constructors, and therefore, they have prototypes that you can change.

For instance, adding a new method for use on all arrays is as simple as modifying Array.prototype.

```
Array.prototype.sum = function() {
  return this.reduce(function(previous, current) {
    return previous + current;
  });
};
var numbers = [1, 2, 3, 4, 5, 6];
var result = numbers.sum();
console.log(result); // 21
```

- This example creates a method called sum() on Array.prototype that simply adds up all of the items in the array and returns the result.
- The numbers array automatically has access to that method through the prototype.
- Inside of sum(), this refers to numbers, which is an instance of Array, so the method is free to use other array methods such as reduce().

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

Prototype Chaining and Object.prototype

JavaScript's built-in approach for inheritance is called **prototype chaining**, or **prototypal inheritance**. As you learned, prototype properties are automatically available on object instances, which is a form of inheritance.

The object instances inherit properties from the prototype. Because the prototype is also an object, it has its own prototype and inherits properties from that. **This is the prototype chain: An object inherits from its prototype, while that prototype in turn inherits from its prototype, and so on.**

All objects, including those you define yourself, automatically inherit from Object unless you specify otherwise . More specifically, all objects inherit from Object.prototype.

Any object defined via an object literal has its [[Prototype]] set to Object.prototype, meaning that it inherits properties from Object.prototype, just like book in this example:

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

var prototype = Object.getPrototypeOf(book);
console.log(prototype === Object.prototype); // true
```

- Here, book has a prototype equal to Object.prototype.
- No additional code was necessary to make this happen, as this is the default behavior when new objects are created.
- This relationship means that book automatically receives methods from Object.prototype.

Methods Inherited from Object.prototype

Several of the methods used in the past couple of chapters are actually defined on Object.prototype and are therefore inherited by all other objects.

Those methods are:

hasOwnProperty()	Determines whether an own property with the given name exists
propertyIsEnumerable()	Determines whether an own property is enumerable
isPrototypeOf()	Determines whether the object is the prototype of another
valueOf()	Returns the value representation of the object
toString()	Returns a string representation of the object

These five methods appear on all objects through inheritance. The last two are important when you need to make objects work consistently in JavaScript, and sometimes you might want to define them yourself.

Modifying Object.prototype

All objects inherit from Object.prototype by default, so changes to Object.prototype affect all objects. That's a very dangerous situation. You were advised not to modify built-in object prototypes in the earlier lesson, and that advice goes double for Object.prototype.

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.

Object Inheritance

The simplest type of inheritance is between objects.

All you have to do is specify what object should be the new object's [[Prototype]].

Object literals have Object.prototype set as their [[Prototype]] implicitly, but you can also explicitly specify [[Prototype]] with the Object.create() method.

The Object.create() method accepts two arguments:

- The first argument is the object to use for [[Prototype]] in the new object.
- The optional second argument is an object of property descriptors in the same format used by Object.defineProperties() .

Consider the following:

```
var book = {
  title: "Future Shcok"
};

// is the same as
var book = Object.create(Object.prototype, {
  title: {
    configurable: true,
    enumerable: true,
    value: "Future Shock",
    writable: true
  }
});
```

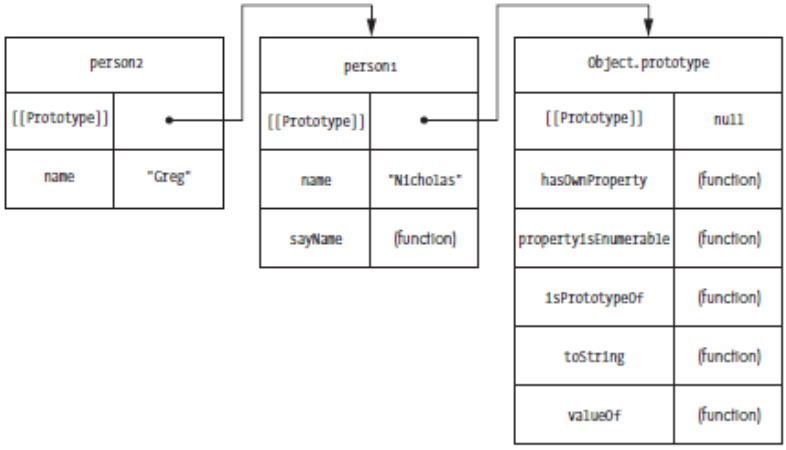
- The two declarations in this code are effectively the same.
- The first declaration uses an object literal to define an object with a single property called title. That object automatically inherits from Object.prototype, and the property is set to be configurable, enumerable, and writable by default.
- The second declaration takes the same steps but does so explicitly using Object.create(). The resulting book object from each declaration behaves the exact same way. But you'll probably never write code that inherits from Object.prototype directly, because you get that by default.

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.

Constructor Inheritance

Object inheritance in JavaScript is also the basis of constructor inheritance.

Recall that almost every function has a prototype property that can be modified or replaced.

The prototype property is automatically assigned to be a new generic object that inherits from Object.prototype and has a single own property called constructor.

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.

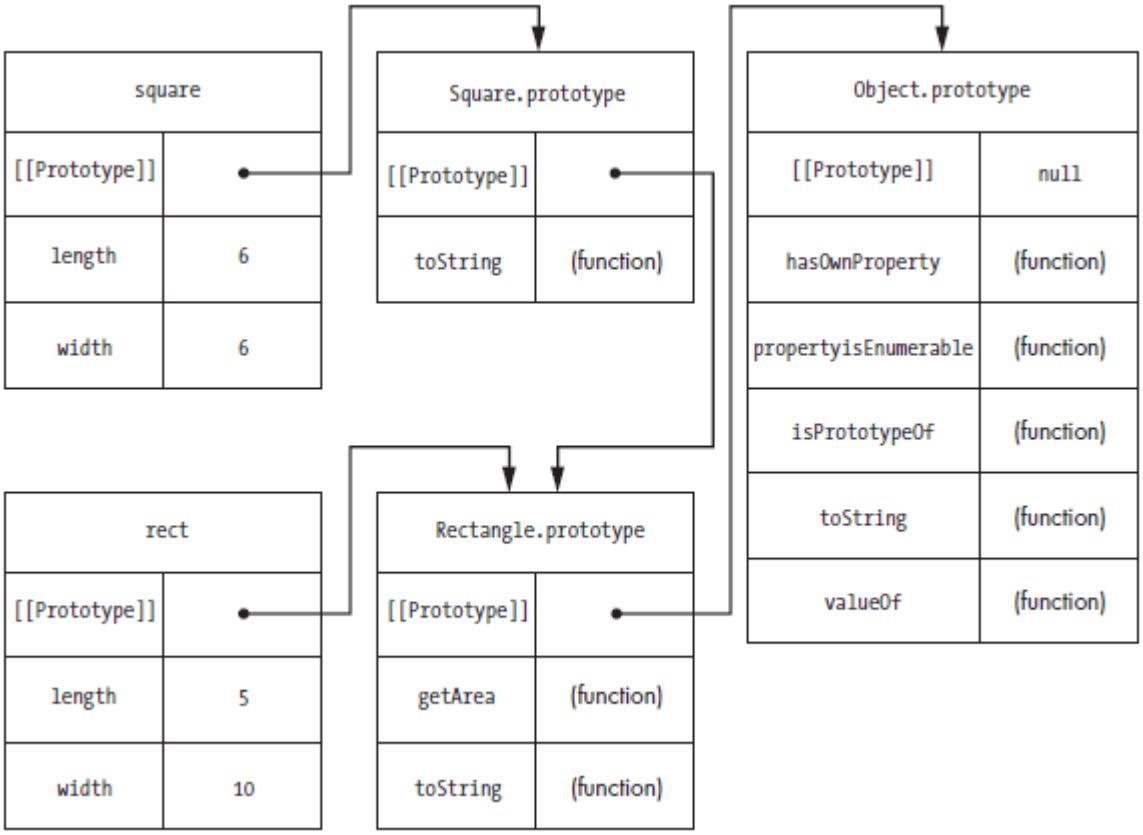
Because the prototype property is writable, you can change the prototype chain by overwriting it.

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.

Accessing Supertype Methods

It is fairly common to override supertype methods with new functionality in the subtype, but what if you still want to access the supertype method?

In other languages, you might be able to say super.toString(), but JavaScript doesn't have anything similar.

Instead, you can directly access the method on the supertype's prototype and use either call() or apply() to execute the method on the subtype object.

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.