Keyed Data

In addition to arrays, which are useful for sequential lists of items, JavaScript offers the **object** as a data storage type and objects are great for keyed data. That isn't necessarily stored in any particular order, but can be accessed by the key associated with each element in the object.

The objects are sets of related information and they can store values of any type. Objects can also grow and shrink dynamically as items are added or removed. Objects are reference types, which means that you can't compare them without iterating through all of their contents.

Another thing about objects that's similar to arrays is that you can change the properties on an object and that can have an affect on the data stored in that object.

What Is Key Data

Objects are great for associating arbitrary string keys with arbitrary values and each new string, then becomes a property on that object and you access each value by associating it with that string. So, a good example of using key data to store an object in JavaScript might be keeping track of all of the different fruits that you're selling and all of their prices.

```
var fruits = {};
fruits.apple = {"color":"red", "price":0.75};
fruits.orange = {"color":"orange", "price":0.65};
fruits.pear = {"color":"green", "price":0.95};
console.log(fruits.apple.color); //"red"
console.log(fruits);
/*
[object Object] {
  apple: [object Object] {
    color: "red",
    price: 0.75
  },
  orange: [object Object] {
    color: "orange",
    price: 0.65
  },
  pear: [object Object] {
```

```
    color: "green",
    price: 0.95
  }
*/
```

The thing to keep in mind about is it doesn't matter if oranges are first or apples are first. What's important is that you can track the values to their properties and follow the logic of the data through from properties to values until you arrive atthe value that you're looking for. The order is completely arbitrary.

Native Object Prototypes

Objects are very convenient, if you know exactly which property you're trying to go for in order to find a value. But if you're trying to iterate through the items in an object, you can be tripped up by the editable prototype property. Altering the prototype on the object creates new enumerable properties. If you alter the prototype on a base object, that adds new properties to every single object that's inherited from that. So don't do that.

```
var vanilla = {"color":"neutral", "size":"medium"};
for (item in vanilla) {
  console.log("vanilla." + item + " = " + vanilla[item]);
}
// "vanilla.color = neutral"
// "vanilla.size = medium"

Object.prototype.extender = "Hahaha";
for (item in vanilla) {
  console.log("vanilla." + item + " = " + vanilla[item]);
}
// "vanilla.color = neutral"
// "vanilla.size = medium"
// "vanilla.extender = Hahaha"
```

We add something to our object prototype by using `Object.prototype` and set `extender` property to `Hahaha`. `extender` doesn't exist inside of `vanilla`, but it does exist on the prototype of the base object, so it gets outputted to the console.

When you're working in JavaScript, you never can be sure that somebody else who might have been creating libraries for you might not have altered the prototype of the base object. Because of that, whenever you loop over the items in an object, you might find properties there that you wouldn't have expected.

Fortunately, JavaScript does offer you a backdoor for safety `if (vanilla.hasOwnProperty())`:

```
for (item in vanilla) {
  if (vanilla.hasOwnProperty(item)) {
    console.log("vanilla." + item + " = " + vanilla[item]);
  }
}
```

```
// "vanilla.color = neutral"
// "vanilla.size = medium"
```

So, unless you're in an environment where you're absolutely sure that no libraries outside of the code that you've written yourself could possibly have touched, use `hasOwnProperty`.

Adding and Removing Object Data

You can also add, remove or change data easily for objects by referencing the explicit key on that object. The string keys that have spaces have to be accessed using bracket notation, but string keys the don't have spaces in them, you can access with dot notation. The thing to remember though is that the order of the data being returned is completely arbitrary.

Testing Code

```
var fruits = {};
fruits.apple = {"skin color":"red", "price":0.75};
fruits.orange = {"skin color":"orange", "price":0.65};
fruits.pear = {"skin color":"green", "price":0.95};
delete fruits.pear.price;
console.log(fruits.pear.price); //undefined
```

`skin color` property has a space in it. That means that, that needs to be accessed using the bracket notation. We can also use the `delete` command and delete `fruits.pear.price`.

```
fruits.apple.price = 0.79;
fruits.apple["skin color"] = "green";
for (var key in fruits.apple) {
  if( fruits.apple.hasOwnProperty(key) ) {
    console.log(key + " = " + fruits.apple[key]);
  }
}
//(IN NO PARTICULAR ORDER)
//"skin color = green"
//"price = 0.79"
```

The important thing to remember is the data will not be printed out in any particular order. Objects don't fundamentally store data in any particular sequence.

ECMAScript 6 Options: Map

The only thing that you can count on with an object is that the property will be associated with the value. If you properly reference the property,you will get the value you're expecting.

ECMAScript 6 offers an alternative data type called **Map** and it's only an option if you're using an ECMAScript 6 or later environment. But Map can be a better alternative, if you want key data storage. If follows most of the same conventions as an object for data storage, but maps don't include a prototype by default. So, all of the key-value pairs inside of a Map are going to be data specific to that object and a Map can use any type for a key, whereas objects rely on only strings for keys. Maps also have a built-in size method and that means that you don't need to keep track of the size manually, as you would with an object.