Arrays for Lists

Depending on the type of data you want to store, JavaScript offers a number of options. **Arrays** are probably the best option for information that's sequential; that is, each item is one in a sequence and they need to be kept in order.

The important thing to remember about arrays is that they are zero-indexed, so the first item in an array is always item zero. So arrays are perfect for keeping track of lists of anything. Arrays store sequential values, and they can be of any type, and a single array can have items that are different types as well. You can even nest arrays inside of arrays, and that can be used to store more complex data types, such as tables of data that need to be retained in order.

There are a few other things to remember about arrays in JavaScript:

- They can shrink or grow dynamically
- They do that automatically as items are added or removed
- They are also reference types and that means that you can't compare them without iterating through all of their contents. Changing the properties for an array, such as its length, can alter the contents of that array.

What Is Sequential Data

Sequential data is automatically numbered, stored and maintained in a specific order. Arrays are perfect for data types such as queues and stacks, and we'll give you some examples of that in a minute.

To start with, let's give you an example of something that you'd want to maintain in order. A classic example for ordered information would be the steps that you have to go through in order to achieve a process.

```
var steps = ["brainstorm", "narrow", "prototype", "test", "propose"];
var stepTerm;
for(var counter = 0; counter < steps.length; counter++) {
  switch(counter) {
    case 0:
      stepTerm = "First";
      break;
    case steps.length - 1:
      stepTerm = "Finally";
      break;
    default:
```

```
      stepTerm = "Then";
  }
  console.log(stepTerm + " we " + steps[counter]);
}
//"First we brainstorm"
//"Then we narrow"
//"Then we prototype"
//"Then we test"
//"Finally we propose"
```

Using a `for` loop is one of the most common ways of iterating through the items inside of an array. One of the reasons for that is because you're looping through, in order, from beginning to end, and you have a counter that's also looping through in order, from beginning to end. You can keep track of the count and you can make sure that each item you're presenting will always be presented in the same sequence.

Adding and Removing Data From An Array

Arrays are very easy to manipulate too. You can read or add, remove, insert, or replace items from anywhere in an array, or assert the items based on their values. Native arrays in JavaScript support methods that allow you to do all of these things directly.

```
var steps = ["brainstorm", "narrow", "prototype", "test", "propose"];
console.log(steps[0]); //"brainstorm"

steps[2] = "experiment";
console.log(steps); //["brainstorm", "narrow", "experiment", "test", "propose"]
```

We now have a modified array with "experiment" instead of prototype as the third element.

JavaScript will also let you sort the elements in an array, but the important thing to remember is that the `sort` in JavaScript on the array is destructive, and that means that it actually modifies the values in the original array:

```
var alphaSteps = steps.sort();
console.log(alphaSteps); //["brainstorm", "experiment", "narrow", "propose", "test"]
console.log(steps); //["brainstorm", "experiment", "narrow", "propose", "test"]
```

So we haven't created a copy on an array. What we've actually done is modify the original one in place. There is a way to get around this, though - just insert a `slice` command in the chain of methods being applied to steps.

Another thing to be aware of with arrays is that you can alter the properties just by setting them. Altering the properties of an array actually changes the array itself.

Changing a Length Of An Array

For example, you can change the length of an array, but that can delete items in an array or add extra items that are undefined in an array, and you might not expect that.

```
var steps = ["brainstorm", "narrow", "prototype", "test", "propose"];
console.log(steps.length); //5

steps.length = 10;
console.log(steps.length); //10
console.log(steps[6]); //undefined

steps.length = 3;
console.log(steps[4]); //undefined
console.log(steps); //["brainstorm", "narrow", "prototype"]
```

Item number six is definitely beyond the original length of the array, so we see that it has the value `undefined`. In fact, every value beyond the first five is just set to `undefined`, but still the length of the array is considered to be 10.

When we set length to 3, the fourth element in our original `steps` array is now `undefined`. There's no way to get those back after you've done that. So if you reset the length of your array shorter than the values that you've actually been maintaining in your array, you could lose data.