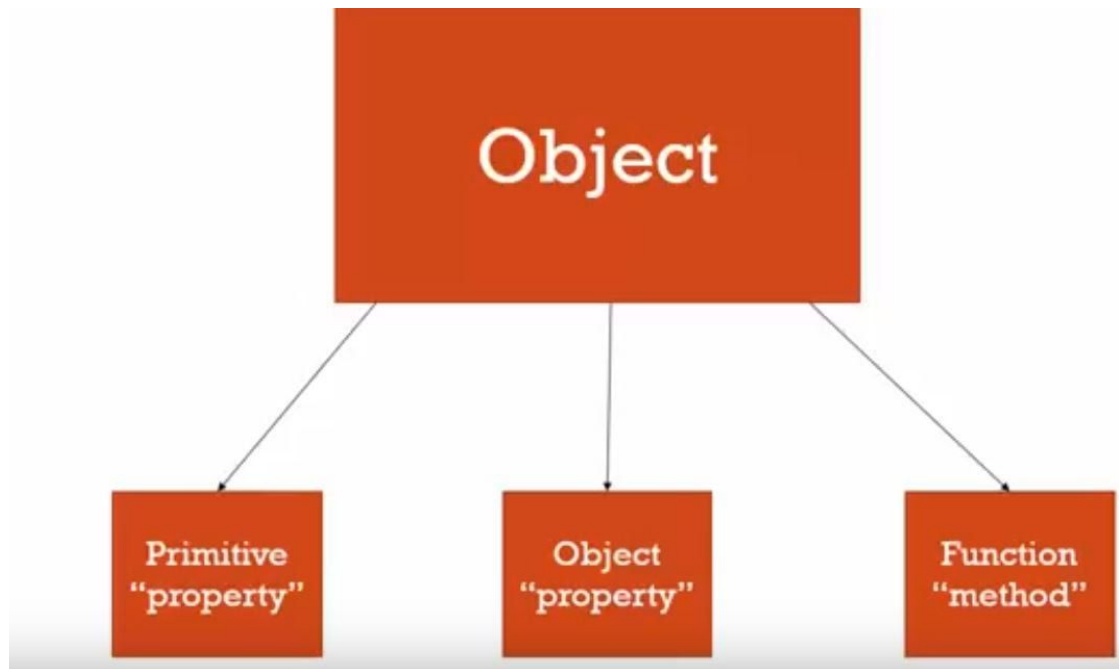


JavaScript

(Part 3)

1. Objects and Dots:

- Structure



- Creating objects and assigning name-value pairs:
//Though new method for object creation is not very useful but let's start with it:

```
Var person = new Object();  
person["firstname"] = "Tony";  
person["lastname"] = "Keto";  
// to view our object  
console.log(person);  
console.name(person.firstname);
```

Console output:

>object (Inside this we have firstname, lastname, and _proto_)

Tony

We may also use the following example as a substitute:

```
1 var person = new Object();
2
3 person["firstname"] = "Tony";
4 person["lastname"] = "Alicea";
5
6 var firstNameProperty = "firstname";
7
8 console.log(person);
9 console.log(person[firstNameProperty]);
```

- Adding an object inside an object:
In the previous case:
person.address = new object();
person.address.street = "111 Queen Park";
person.address.city = "Melbourne";
person.address.state = "Melbourne"
console.log(person.address.state);

//The above line of code involved the use of dot operator but the bracket

operator (a.k.a. Computed member access) can be used in the following way.

```
console.log(person["address"]["state"]);
```

2. Object and object literals

As discussed earlier 'new' method for adding objects is not good. An alternative approach is

```
var person = {};
```

//And an object named person is created and this is an empty object. To initialize it:

```
var person = {  
  firstname:'Jamy' ,  
  lastname:'Hector  
  Address:  
  {  
    Street: "Queen Park",  
    City: "Melbourne"  
  }  
};
```

- Using an object inside a function:

Ex:

```
var jamy = {  
  firstname:'Jamy' ,  
  lastname:'Hector  
  Address:
```

```
{  
  Street: "Queen Park",  
  City: "Melbourne"  
}  
  
};  
  
function greet(person){  
  console.log('Hi'+person.firstname);  
}  
  
greet(jamy);
```

Output:

Hi Jamy

- Creating an object inside a function parameter column in the above case:
greet({firstname: 'Mary',
lastname: 'Keila'})

3. Faking Namespaces

Namespaces: A container for variables and functions (typically it is used to keep the variables and functions with the same name separate).

However, JS does not have namespaces.

Faking namespaces is used in several libraries and frameworks to avoid overwriting of variables.

Ex:

```
var greet = 'hello';  
var greet = 'Hola';  
console.log(greet);
```

Output:

Hola

To avoid this:

```
var english = {};  
var spanish = {};
```

```
english.greet = 'hello';  
spanish.greet = 'Hola';
```

4. **JSON and object literals:**

JSON stands for JavaScript Object Notation

Earlier data on the internet was sent in the following format:

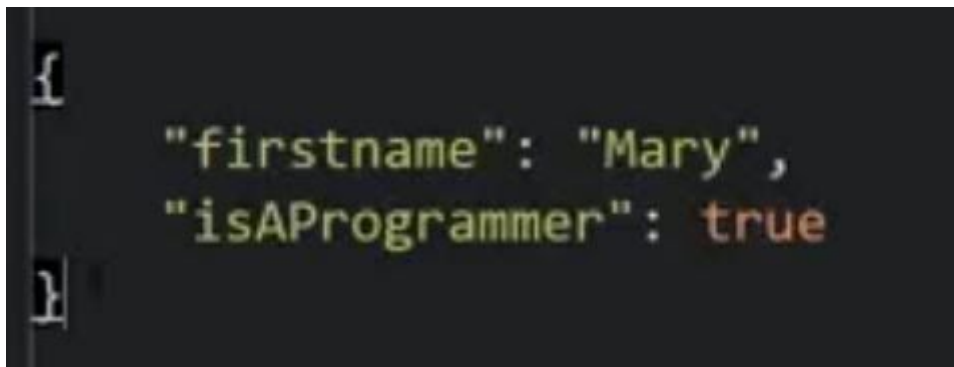
```
<firstname>"John"</firstname>  
<lastname.....
```

However, the problem with this system is that the user has to load a lot of data from the server to get very little info (like two tags involved, etc.)

And then a new way of data handling came into existence and this was inspired by JS objects and is known as JSON.

Difference b/w JS and JSON objects?

In JSON the name of the property is also enclosed within double quotes `""`. However, double quotes aren't used in JS for the property names. Anything that is JSON is also a valid JS object.



JSON file generally looks like this.

Now, since JSON is a valid JS hence JS comes with built-in functions for JSON.

Conversion from JS to JSON:

JS:

```
var objectliteral = {  
  Firstname = 'Mary',  
  Lastname = 'Keila'  
}
```

For conversion to JSON:

```
console.log(JSON.stringify(objectliteral));
```

Resulting JSON looks like:

```
{  
  "Firstname" = "Mary",  
  "Lastname" = "Keila"  
}
```

For conversion from JSON to JS:

```
var jsonvalue = JSON.parse(' {"Firstname" = "Mary",  
  "Lastname" = "Keila" } ')
```

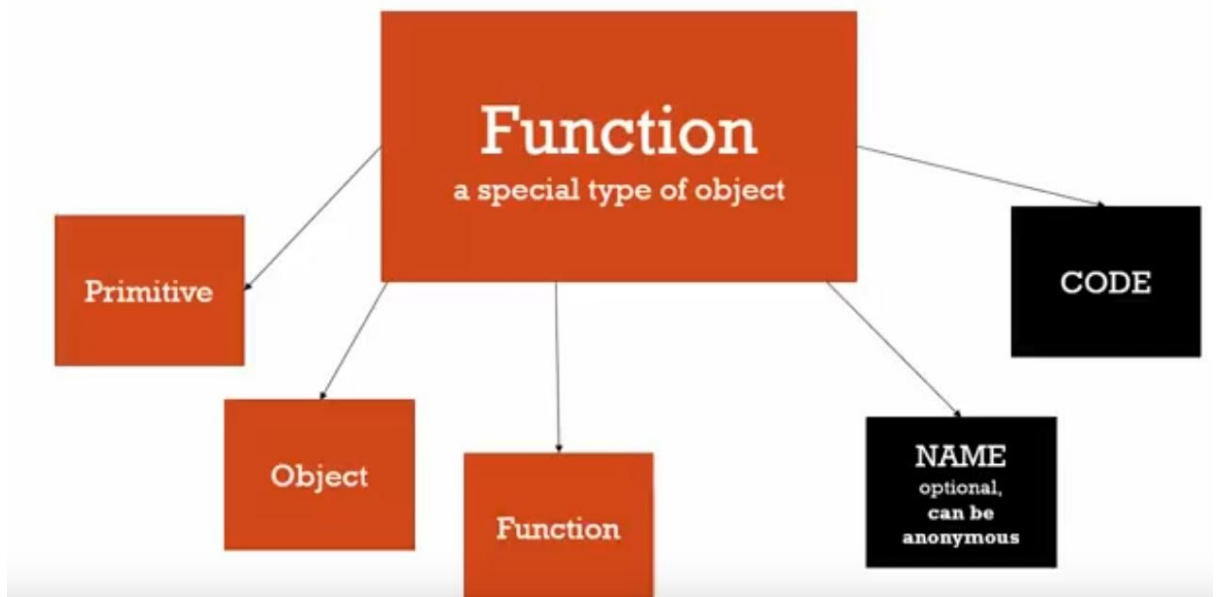
And note before parsing to the JSON.parse the JSON file was stringified i.e. single quotes are added.

Thus we can say all JSON is valid JS but vice versa may or may not be true.

5. Functions are just objects(In JS):

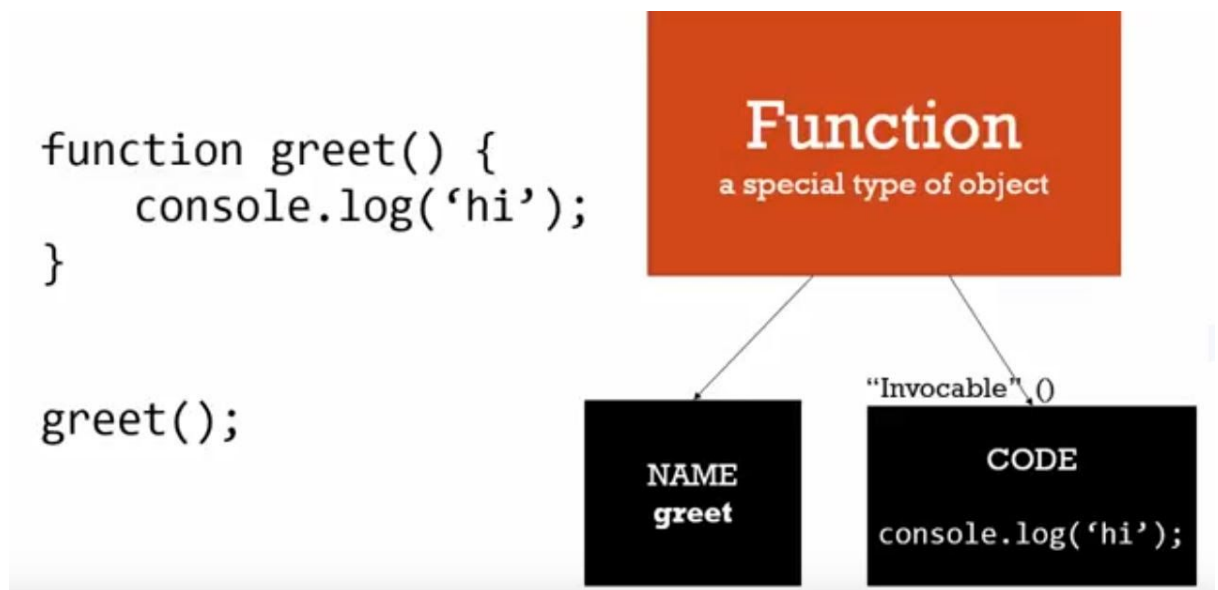
First class functions: Everything that can be done with data types, can be done with functions. Assign them to variables, pass them around, create them on the fly.

Objects can accommodate various data types and functions. (that's why function is a special type of object).

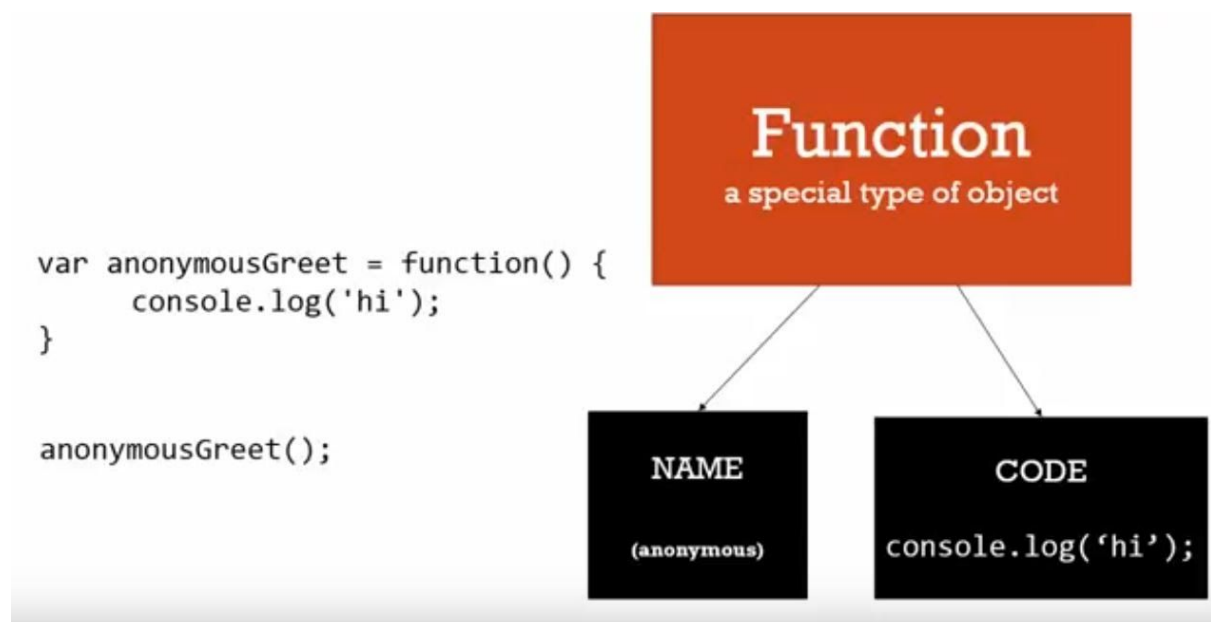


```
function greet(){  
    console.log('hi');  
}
```

```
greet.language = 'english';  
console.log(greet.language);
```

Making anonymous functions using the fact that they are objects:



When the name of the function is declared then it becomes a function statement and when an anonymous function is declared then this becomes expression as it results in a value and results in object creation.

Remember hoisting is allowed in case of function as a trivial function but when function is declared as an object (anonymous function), hoisting doesn't work.

```
anonymousGreet();  
  
var anonymousGreet = function() {  
    console.log('hi');  
}
```

This gives 'undefined' as output. Because here function is declared as a variable that indeed creates an object and variables aren't hoisted.

Remember first class functions can be created on the fly and can be passed to other functions.

Code:

```
function log(a){  
    console.log(a);  
}  
// Creating function on the fly and can be passed on to  
other functions  
log(function(){console.log('hi');});
```

Output:

```
log(function(){console.log('hi');})
```

How to invoke the function created like this:

Remember here the function declared on the fly inside log is nothing but 'a'. So to invoke it declare a() inside the main function scope like:

Code:

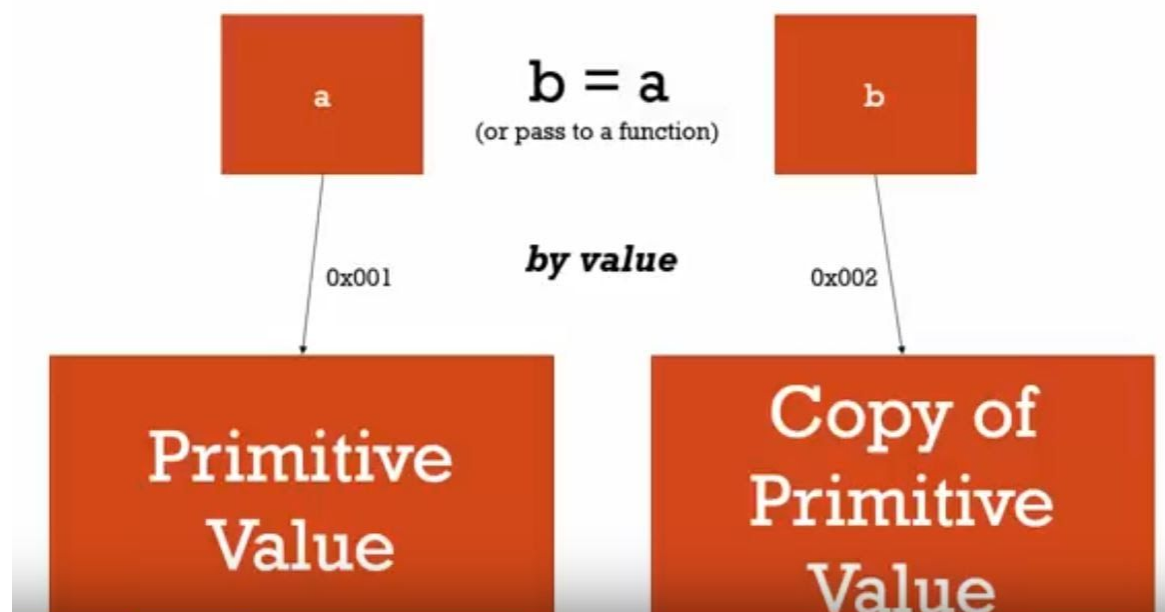
```
function log(a){  
    a();  
}
```

// Creating function on the fly and can be passed on to other functions

```
log(function(){console.log('hi');});
```

6. By value vs By reference

- By value: It simply means copying an existing value to a new variable.

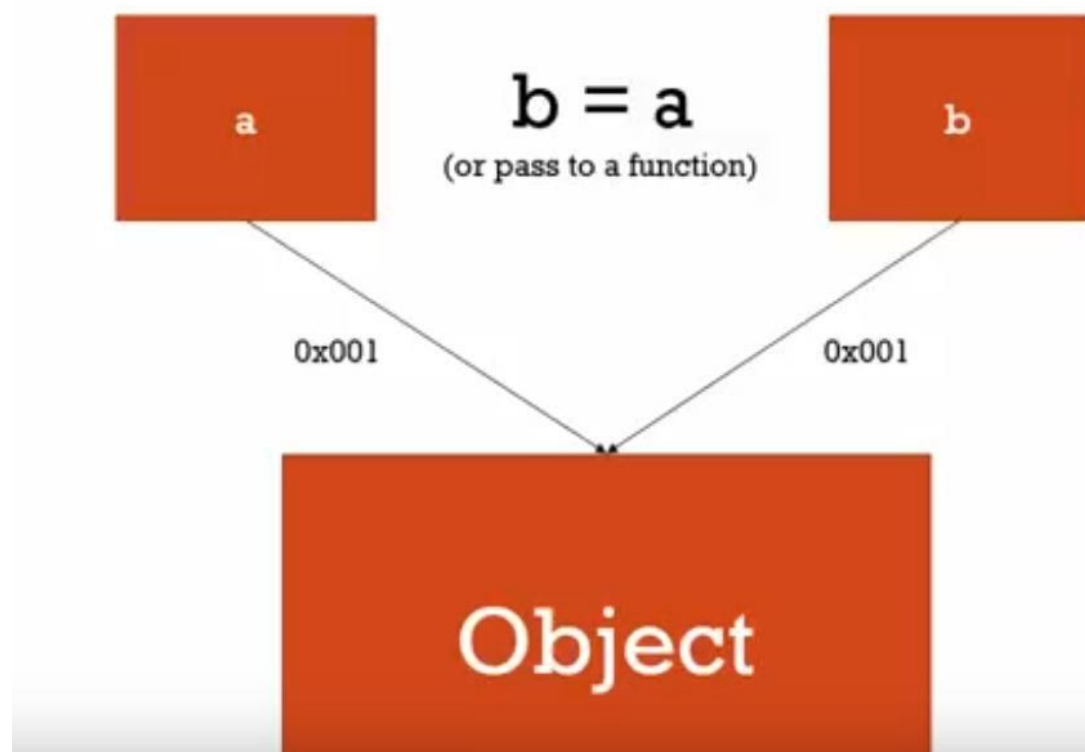


Here, which was an existing primitive data type was assigned to 'b' a new variable. This led to copying of the value of 'a' to b.

- By reference:

Similarly, an object also occupies a space in memory which can be accessed.

However, when a new variable is created and is set equal to the preexisting variable (which had an object stored in it), it leads to both the variables pointing at the same object.



All objects act by-reference

Attributes of calling by name and by reference:

One value can be overwritten and this doesn't affect the other assigned variable.

```
1 // by value (primitives)
2 var a = 3;
3 var b;
4
5 b = a;
6 a = 2;
7
8 console.log(a);
9 console.log(b);
10
11 // by reference (all objects (including functions))
12 var c = { greeting: 'hi' };
13 var d;
14
15 d = c;
16 c.greeting = 'hello'; // mutate
```

In this example, even 'd' is changed when we change 'c'.

Calling by reference using parameters:

```
// by reference (even as parameters)
function changeGreeting(obj) {
    obj.greeting = 'Hola'; // mutate
}

changeGreeting(d);
console.log(c);
console.log(d);
```

In this case, the name-value pair in c and d are mutated to greetings: “Hola” when ‘d’ is passed into the function. However, = (equal to operator) sets up new memory space when used. For ex:

```
// equals operator sets up new memory space (new address)
c = { greeting: 'howdy' };|
console.log(c);
console.log(d);
```

In this way, c becomes {greeting: ‘howdy’} and ‘d’ remains {greeting: ‘Hola’}.

Thus, all objects are by reference and all primitive types are by value.

7. ‘this’

- Impact of ‘this’ in global environment:

Code:

```
console.log(this);
```

Output:

Window

- Impact of 'this' when in the environment of variables:

```
1 function a() {  
2     console.log(this);  
3 }  
4  
5 var b = function() {  
6     console.log(this);  
7 }  
8  
9 a();  
10 b();
```

Output:

>window

>window

Now if inside a() we insert the line of code:

this.greet = 'Hello';

'greet' has been created as a global object.

And can be accessed globally anywhere.

But it is not necessary that 'this' always points to 'window'.

For example:

If 'this' is invoked inside an object then

'this' would start pointing at the object in

which it resides.

```
var c = {  
  name: 'The c object',  
  log: function() {  
    console.log(this);  
  }  
}  
  
c.log();
```

Output:

```
► Object {name: "The c object", log: function}
```

2.

```
var c = {  
  name: 'The c object',  
  log: function() {  
    this.name = 'Updated c object';  
    console.log(this);  
  }  
}  
  
c.log();
```

output:

```
► Object {name: "The c object", log: functi
```


3. Assumed bug in JS:

```
16 var c = {
17     name: 'The c object',
18     log: function() {
19         this.name = 'Updated c object';
20         console.log(this);
21     }
22     var setname = function(newname) {
23         this.name = newname;
24     }
25     setname('Updated again! The c object');
26     console.log(this);
27 }
28 }
29
30 c.log();
```

Output:

```
► Object {name: "Updated c object", log: function}
► Object {name: "Updated c object", log: function}
```

It was expected that the name of the object would update but this doesn't happen. However, a 'setname' variable can be seen in global variable space in this case.

Thus, using 'this' inside a function set in the object does point out to the global environment not the object.

To resolve this problem:

```

var c = {
  name: 'The c object',
  log: function() {
    var self = this;

    self.name = 'Updated c object';
    console.log(self);

    var setname = function(newname) {
      self.name = newname;
    }
    setname('Updated again! The c object');
    console.log(self);
  }
}

```

Here, 'self' keyword is first declared and set equals to the object 'this' and then it is used throughout the code. Thus everywhere in the code 'this' points out to the environment of the object 'c'.

Output:

```

► Object {name: "Updated c object", log: function}
► Object {name: "Updated again! The c object", log: function}

```

No programming language is perfect.

8. Prompt and alert

```
var age = prompt("Enter your age:");
```

This way the age of a user can be taken.

We can use `document.write("Your age is " + age);` to print this on screen.

9. 'typeof' operator

This is used to find the category of primitive data type of a given variable. Just use `console.log(typeof varname);`

10. Ternary operator:

`(condition) ? (case executed when condition is true):(executed when condition is false);`

Ex:

```
var wantDrinks = (age>=18)?console.log('have  
some drinks') : console.log('have some juice');
```

11. Switch Statements

1.

```
var job = 'teacher';  
switch (job)  
{  
    case 'teacher':  
        console.log("You are a teacher");  
        break;  
    case 'doctor':  
        console.log("Hello doc!!!");  
        break;  
    default:  
        console.log("Get employed buddy!!");  
}
```

2.

```
switch(true)  
{  
    case age<13:
```

```
        console.log("Teen");
    case age>13 && age<21:
        console.log("Youngster");
    case age<21:
        console.log("Man");

}
```

12. Arrays

Methods of declaring arrays:

1. var names = ['John', 'Keto', 'Jane'];
2. Var years = new Array(1998,1967,1980);

- To find array size:

```
console.log(names.length);
```

- Arrays are mutable.
- To add a last element:
names[names.length] = 'Jason'
- Arrays can accommodate different data types.
- To 'push' elements into an array:
 - Used to add new entries to our array at the end.
 - eg:
var john = ['John', 28, 'Not married',
'teacher'];
john.push('blue');
//blue is added to the array john.
- 'Unshift'

- Adds new entries to the starting of the array
- 'pop'
 - Removes the last element of an array.
- 'shift'
 - Removes the first element of an array.
- 'indexOf'
 - Gives position or index number of the element passed in.
 - eg:

```
console.log(john.indexOf(28));
```

```
// returns 1
```

If we get a negative value that means that element is not present in the array.