Review Objects

- JavaScript Objects are class-free.
- An object is a container of properties.
- A property has a name and a value.
- The property values of an object can be another object, which can be function, array or object.
- An object inherits the properties of another object directly through prototype.

Create Objects

```javascript
var flight = {
  airline: "Oceanic",
  number: 815,
  departure: { ... },
  arrival: { ... }
};
```

```javascript
Create Objects

var flight = new Object();
flight.airline="Oceanic";
flight.number=815;
flight.departure={...};
flight.arrival={...};
flight.travel=function(){...}
```

Constructor Function

- We can also implement a function to construct an object and use new operator to call the constructor function.
- ```javascript
function Rectangle(w, h) {
  this.width = w;
  this.height = h;
  this.area = function(){return this.width * this.height;}
  //Note: “this” is important, means the w and h are assigned to the current object that the function is creating.
  // Note: no return statement here
}
```
Creating Objects Using Constructor

```javascript
var rect1 = new Rectangle(2, 4);
var rect2 = new Rectangle(3, 5);
rect1.width = 2;
rect1.height = 4;
rect1.area(); //?
rect1.prototype.area = function() { /*...*/
rect2.width = 3;
rect2.height = 5;
rect2.area(); //?
rect2.prototype.area = function() { /*...*/
```

Prototype Inheritance

- Every object has a property called `prototype`, which contains the properties that the object inherits. A JavaScript object inherits the prototype of its Constructor Function.

```
function Rectangle(x, y)
  this.width = x;
  this.height = y;
Rectangle.prototype.area = function() {
  return this.width * this.height;
}
var rect1 = new Rectangle(2, 4);
rect1.area(); //?
rect1.hasOwnProperty("width"); //?
rect1.hasOwnProperty("area"); //?
```

Prototype Inheritance - Continue

```
function Rectangle(w, h) {
  this.width = w;
  this.height = h;
} 
Rectangle.prototype.area = function() {
  return this.width * this.height;
}
var rect1 = new Rectangle(2, 4);
rect1.area(); //?
rect1.hasOwnProperty("width"); //?
rect1.hasOwnProperty("area"); //?
```

What should be put in a Constructor Function Prototype?

- All properties in the prototype of a Constructor Function are shared by all instances that call the Constructor Function.
- Prototype is therefore an ideal place for:
  - Constants, such as `pi`
  - Methods that are shared by all objects created through calling the constructor function.
Extending Built-in Types

• We can add new methods to the prototype of built-in types Constructor Function.
• In this way, all objects created by calling the built-in type Constructor Function can share the method.
• Don’t extend Object prototype:
  Object.prototype.method=function(){
  }
• This messes up for(var i in obj)

Extend Array Prototype

• In LISP, car returns the first element of a list.
  Array.prototype.car=function(){return this[0];}
  var numbers=[1,2,3,4];
  var first=numbers.car();//numbers? first?
• In LISP, cdr returns the list with its first element removed.
  Array.prototype.cdr=function(){return this.slice(1);}
  var rest=numbers.cdr();//numbers? rest?

More Array Prototype Extension

Array.prototype.mymap=function(f){
  var result=[];
  for(var i=0;i<this.length; i++){
    result.push(f(this[i]));
  }
  return result;
}
//add1 is a function takes 1 argument
function add1(x){return ++x;}
document.writeln(numbers.mymap(add1));

Recursion Version

Array.prototype.recursiveMap=function(f){
  function recCall(result, list){
    if(list.length==0){return result;}
    else{
      result.push(f(list[0]));
      return recCall(result, list.slice(1));
    }
  }
  return recCall([],this);
}
function add1(x){return ++x;}
document.writeln(numbers.recursiveMap(add1));
Assignment 6

- Extends Array Prototype Methods:
  - cons: a function takes an element (can be a list) and returns a new list with the element inserted before the given list.
    
    ```javascript
    var z = [1,2,3,4].cons(0);
    //z=[0,1,2,3,4]
    ```

foldr (reduce)

- The function takes 3 arguments: a dyadic function, a terminate value and a list.
- When evaluated, foldr places the dyadic function between each of the list elements and adds the terminate value at the end.
- It then evaluates the result from the right to left.
- In JavaScript, the list is the array to be reduced.
  
  ```javascript
  function div(x,y){return x/y;}
  var z = [1,2,4].foldr(div,40);
  //z=1/2/4/40=1/2/0.1=1/20=0.05.
  var z = [10].foldr(div,10); //z=1
  ```

foldl (accumulate)

- foldl(accumulate): Similar to foldr but placing the terminate value in the front and evaluating the result of the expression from the left to right.

  ```javascript
  function div(x,y){return x/y;}
  var z = [1,2,4].foldl(div,40);
  //z=40/1/2/4/40/2/4=20/4=5.
  var z = [10].foldl(div,10); //z=1
  //foldr and foldl don’t always produce the same result.
  ```

Combine LISP function

- [1,2,3,4].foldr(cons,[])
  - =1 cons 2 cons 3 cons 4 cons []
  - =1 cons 2 cons 3 cons [4]
  - =1 cons 2 cons [3,4]
  - =1 cons [2,3,4]
  - =[1,2,3,4]