How to get a "class"?

• What if we want to create a class, not just one object?
  ▪ JavaScript, unlike Java, does NOT have classes
  ▪ we could emulate a constructor with a function:

```javascript
// Creates and returns a new Point object.
function constructPoint(xValue, yValue) {
  // bad code
  return {
    x: xValue,  y: yValue,
    distanceFromOrigin: function() {
      return Math.sqrt(this.x * this.x +
                      this.y * this.y);
    }
  };
}

> var p = constructPoint(4, -3);
```
function \texttt{constructPoint}(xValue, yValue) { // bad code
    return {
        x: xValue,  y: yValue,
        distanceFromOrigin: function() {
            return Math.sqrt(this.x * this.x +
                             this.y * this.y);
        }
    };
}

- ugly
- doesn't match the "new" syntax we're used to
- wasteful; stores a separate copy of the \texttt{distanceFromOrigin} method in each Point object
// Constructs and returns a new Point object.
function Point(xValue, yValue) {
    this.x = xValue;
    this.y = yValue;
    this.distanceFromOrigin = function() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    };
}

> var p = new Point(4, -3);

- a constructor is just a normal function!
- called with new like in Java
Functions as constructors

• in JavaScript, any function can be used as a constructor!
  ▪ by convention, constructors' names begin in uppercase
  ▪ when a function is called w/ new, it implicitly returns this

function Point(x, y) {
  this.x = x;
  this.y = y;
}

• all global "classes" (Number, String, etc.) are functions acting as constructors, that contain useful properties
Functions as constructors

- any function can be called as a constructor or a function
- when any function called with `new`, JavaScript:
  - creates a new empty anonymous object
  - uses the new empty object as `this` within the call
  - implicitly returns the new object at the end of the call
- if you call a "constructor" without `new`, `this` refers to the global object instead
  - what happens if our "constructor" is called this way?

```javascript
> var p = Point(4, -3);
```
• **prototype**: an ancestor of a JavaScript object
  - like a "super-object" instead of a superclass
  - a parent at the object level rather than at the class level
Prototypes

• every object contains a reference to its prototype
  ▪ default: `Object.prototype`; strings → `String.prototype`; etc.

• a prototype can have a prototype, and so on
  ▪ an object "inherits" all methods/data from its prototype(s)
  ▪ doesn't have to make a copy of them; saves memory
  ▪ prototypes allow JavaScript to mimic classes, inheritance
// also causes Point.prototype to be defined
function Point(xValue, yValue) {
    ...
}

• every function stores a **prototype** object property in it
  • example: when we define our Point function (constructor),
    that creates a Point.prototype
  • initially this object has nothing in it ( {} )
  • every object you construct will use the function's
    prototype object as its prototype
    – e.g. every new Point object uses Point.prototype
How constructors work

- when any function called with new, JavaScript:
  - creates a new empty anonymous object
  - uses the new empty object as this within the call
  - attaches the function's .prototype property to the new object as its internal prototype
  - implicitly returns the new object at the end of the call
The prototype chain

```javascript
var p1 = new Point(4, -3);
```

- when you ask for a property (or method) in an object, JS:
  - sees if the **object itself** contains that property
  - if not, recursively checks the object's **prototype** for it
  - if not found, continues up the "prototype chain" until it finds the property or gives up with undefined
Augmenting a type via prototypes

// adding a method to the prototype
function.prototype.name = function(params) {
    statements;
};

Point.prototype.distanceFromOrigin = function() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
};

• adding a property to a prototype will give it to all objects that use that prototype
  ▪ better than manually adding each method to each object
What goes in a prototype?

- generally only **methods** and **constants** (variables)
  - not objects' fields!
  - can also add "static" methods meant to be called on the prototype itself, e.g. `Math.abs`

- What would happen if we put the `x` and `y` fields in `Point.prototype`?

- **Exercise**: Add `distance` and `toString` methods.
Exercise solutions

// Distance between this point and the given point.
Point.prototype.distance = function(p) {
    var dx = this.x - p.x;
    var dy = this.y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
};

// A string version of this object, e.g. "(3, -4)".
Point.prototype.toString = function() {
    return "(" + this.x + ", " + this.y + ")";
};
Modifying built-in prototypes

// add a 'contains' method to all String objects
String.prototype.contains = function(text) {
    return this.indexOf(text) >= 0;
};

• ANY prototype can be modified, including existing types
  ▪ many JS add-on libraries do this to augment the language
  ▪ not quite the same as adding something to a single object

• Exercise: Add a reverse method to all strings.
• Exercise: Add a shuffle method to all arrays.
function *SuperClassName*(*parameters*) { ... }

function *SubClassName*(*parameters*) { ... }

*SubClassName*.prototype = new *SuperClassName*(*parameters*);

- to make a "subclass", tell its constructor to use an object of a "superclass" as its prototype

- why not just write it this way?
  
  *SubClassName*.prototype = *SuperClassName*.prototype;
// Constructor for Point3D "subclass"
function Point3D(x, y, z) {
    this.x = x;
    this.y = y;
    this.z = z;
}

// set it to be a "subclass" of Point
Point3D.prototype = new Point(0, 0);

// override distanceFromOrigin method to be 3D
Point3D.prototype.distanceFromOrigin = function() {
    return Math.sqrt(this.x * this.x + this.y * this.y + this.z * this.z);
};
Problems with pseudo-inheritance

- there is no equivalent of the `super` keyword
  - no easy way to call the superclass's constructor

- no built-in way to call an overridden superclass method
  - have to write it manually, e.g.
    ```javascript
    var d = Point.prototype.
    distanceFromOrigin.apply(this);
    ```

- solution: many JS libraries add class creation syntax, e.g.
  ```javascript
  Class.create(name, superclass, ...)
  ```
The `instanceof` keyword

\[ expr \text{ instanceof } \text{ConstructorFunction} \]

- returns `true` if the given object was constructed by the given constructor, or is in the object's prototype chain

```javascript
> var p = new Point(3, -4);
> var p3d = new Point3D(3, -4, 5);
> p instanceof Point
true
> p3d instanceof Point3D
true
> p3d instanceof Point
true
> "hello" instanceof Point || {} instanceof Point
false
```
Another type test: .constructor

> var p1 = new Point(3, -4);
> p1.constructor
function Point(xValue, yValue) { ... }
> var o = {};
> o.constructor
function Object() {[native code for Object.Object]}

• every object has a constructor property that refers to the function used to construct it (with new)
  ▪ if the object was created without a constructor using {}, its .constructor property refers to the Object() function
  ▪ constructor can be changed; instanceof will still work
The base2 library

load("base2.js");  // http://code.google.com/p/base2/
var Animal = Base.extend({
    constructor: function(name) {
        this.name = name;
    },
    name: "",
    eat: function() {
        this.say("Yum!");
    },
    say: function(message) {
        print(this.name + ": " + message);
    }
});

- intended to make inheritance/subtyping easier
- all classes extend a common constructor called Base
Java within JavaScript

• the Rhino VM is written in Java
  ▪ it implements a layer of JavaScript on top of Java

• Rhino lets you use Java classes in JavaScript
  ▪ combine Java's rich class library with JavaScript's dynamism and simpler syntax

• current trend: languages that on top of the JVM
  ▪ **Clojure**: a Lisp dialect
  ▪ **Scala**: an ML-like functional language
  ▪ **Groovy**: a scripting language
  ▪ JVM adaptations: JRuby, Jython, Erjang, JScheme, ...
Using Java classes in Rhino

```javascript
importPackage(Packages.package);
importClass(Packages.package);
var name = new JavaClassName(params);
```

- Example:

```javascript
> importPackage(Packages.java.util);
> var s = new TreeSet();
> s.addAll(Arrays.asList([2,7,1,2,4,1,2,4]));
> s
[1.0, 2.0, 4.0, 7.0]
```
Accessing class properties

`JavaClassName.property`
`JavaClassName["property"]`

• Example:

```javascript
> var console = new Scanner(System.in);
js: "<stdin>", line 44: missing name after . operator
js: var console = new Scanner(System.in);
js: ..................................................^
> var console = new Scanner(System["in"]);
```
Some Java ↔ JS quirks

- JS Numbers are sometimes doubles when used in Java
  
  ```java
  Arrays.asList([1, 2, 3])
  [1.0, 2.0, 3.0]       <-- ArrayList<Double>
  ```

- to force usage of int, use Integer objects
  
  ```java
  var list = new ArrayList();
  list.add(1);
  list.add(new Integer(2));
  list
  [1.0, 2]
  ```

- char, long, short, byte are treated as Numbers in JS
  
  ```java
  var s = new java.lang.String("hello");
  s.charAt(0)
  104
  ```
More Java ↔ JS quirks

• sometimes JS → Java can't tell what type to use:

  ```javascript
  > var a = [4, 1, 7, 2];
  > Arrays.sort(a);
  The choice of Java constructor sort matching JavaScript argument types (object) is ambiguous; candidate constructors are:
  void sort(java.lang.Object[])
  void sort(long[])
  void sort(int[])
  ...
  ```

• Java collections/arrays DO have bounds checking

  ```javascript
  > var list = new ArrayList();
  > list.get(7);
  java.lang.IndexOutOfBoundsException: Index:7, Size:0
  ```
Implementing and extending

```javascript
new InterfaceOrSubclass(object) // or,
new JavaAdapter(Packages.superclass, interface1, ..., interfaceN, object)
```

- Example:
  ```javascript
  > var o = { compare: function(s1, s2) {
  ... return s1.length() - s2.length(); }
  }
  > var comp = new Comparator(o);
  > var set = new TreeSet(comp);
  > set.add("goodbye");
  > set.add("what");
  > set.add("bye");
  > set.add("hello");
  > set
  [bye, what, hello, goodbye]
  ```
Other direction: JS within Java

- Java 1.6 adds `javax.script` package to run JS code:

```java
import java.io.*;
import javax.script.);

public class RunJS {
    public static void main(String[] args) throws Throwable {
        ScriptEngine engine = new ScriptEngineManager().
            getEngineByName("javascript");

        for (String arg : args) {
            engine.eval(new FileReader(arg));
        }
    }
}
```