OOP, Prototypes, and Inheritance

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:

```
// 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);
```

Problems with pseudo-constructor

```
function 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 `distanceFromOrigin` method in each `Point` object

Functions as constructors

```
// 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

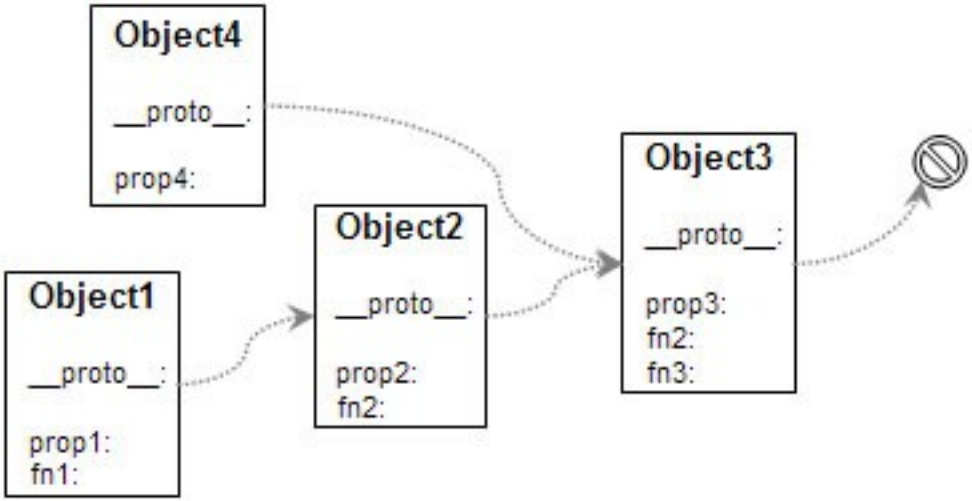
Functions as constructors

- 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?

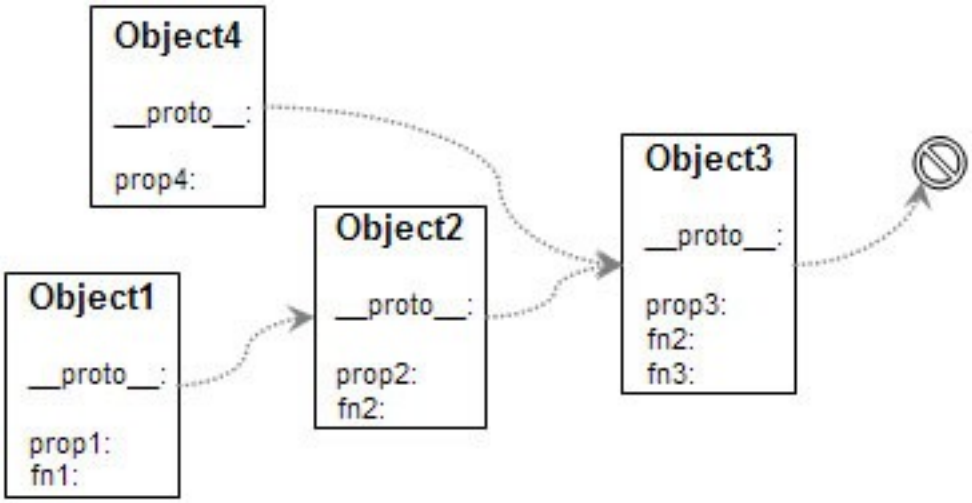
```
> var p = Point(4, -3);
```

Prototypes



- **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

Functions and prototypes

```
// also causes Point.prototype to be
  defined
function Point(xValue, yValue) {
    ...
}
```

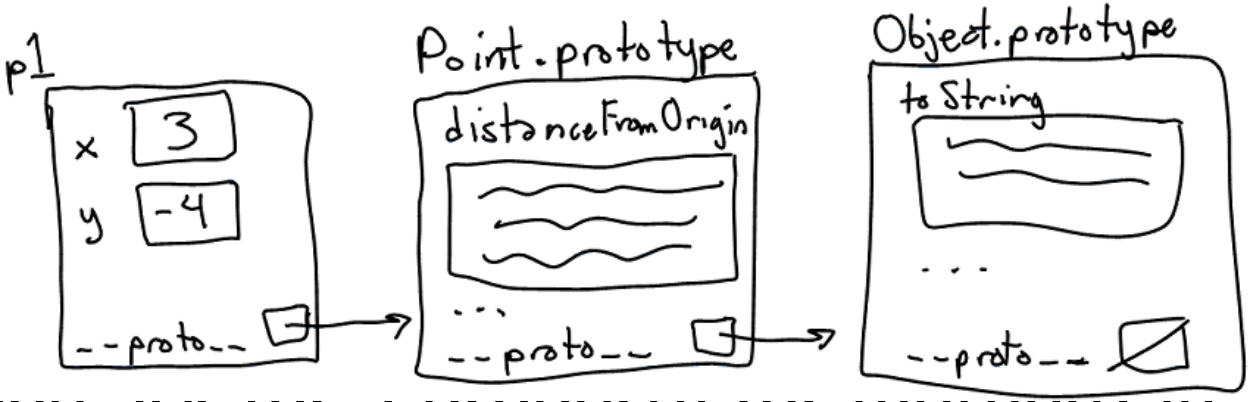
- every function stores a **`prototype`** object property in it
  - example: when we define our Point function (constructor), that creates a `Point.prototype`
  - initially this object has nothing in it (`{}`)
  - every object you construct will use the function's prototype object as its prototype
    - 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

```
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.

Pseudo class-based-inheritance

```
function SuperClassName(parameters) { ... }

function SubClassName(parameters)   { ... }

SubClassName.prototype =                    // connect
  them
     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;
```

Pseudo-inheritance example

```
// 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 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.

```
var d = Point.prototype.

distanceFromOrigin.apply(this);
```

- solution: many JS libraries add class creation syntax, e.g.

```
Class.create(name, superclass, ...)
```

The instanceof keyword

expr `instanceof` ***ConstructorFunction***

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

```
> 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

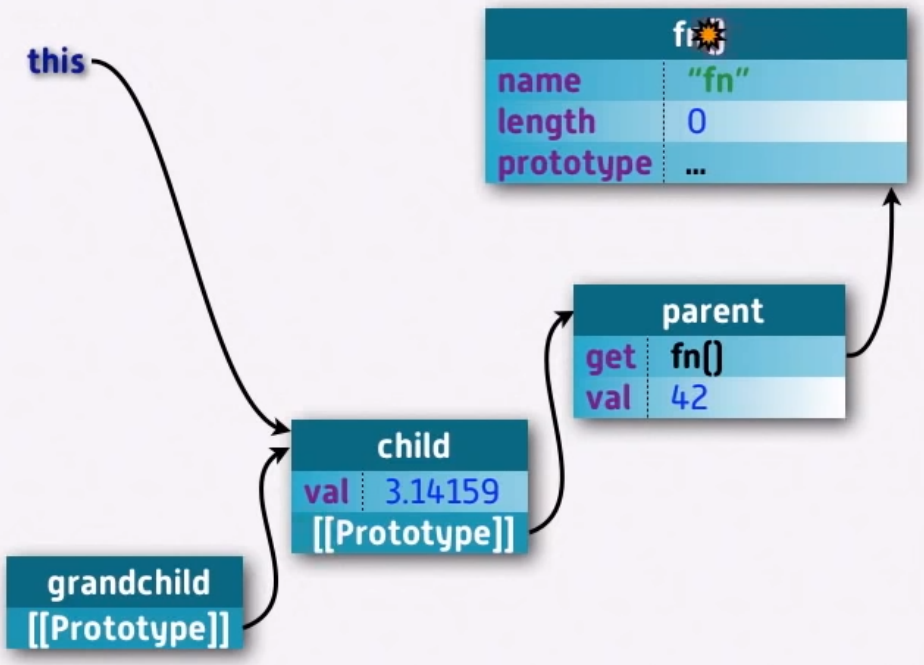
The base2 library

```
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`

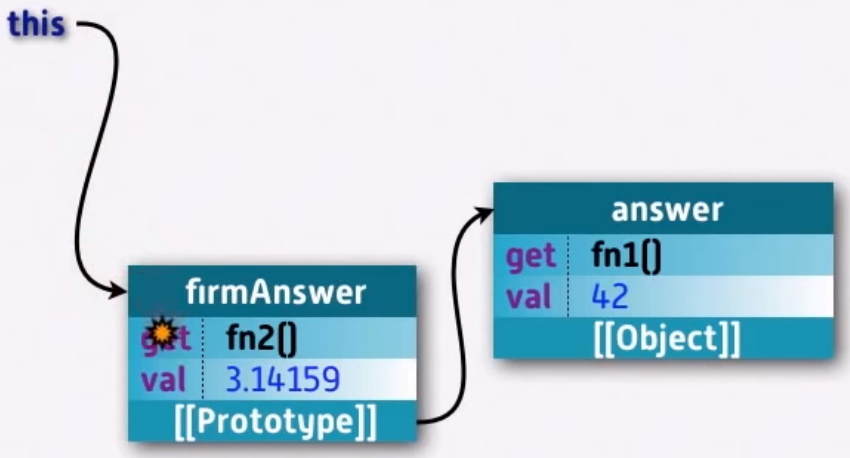
Sample Code (1)

```
var parent = {
  get: function fn() {
    return this.val;
  },
  val: 42
};

var child = Object.create(parent);
child.val = 3.14159;

var grandchild = Object.create(child);

parent.get();       // →42
child.get();        // →3.14159
```



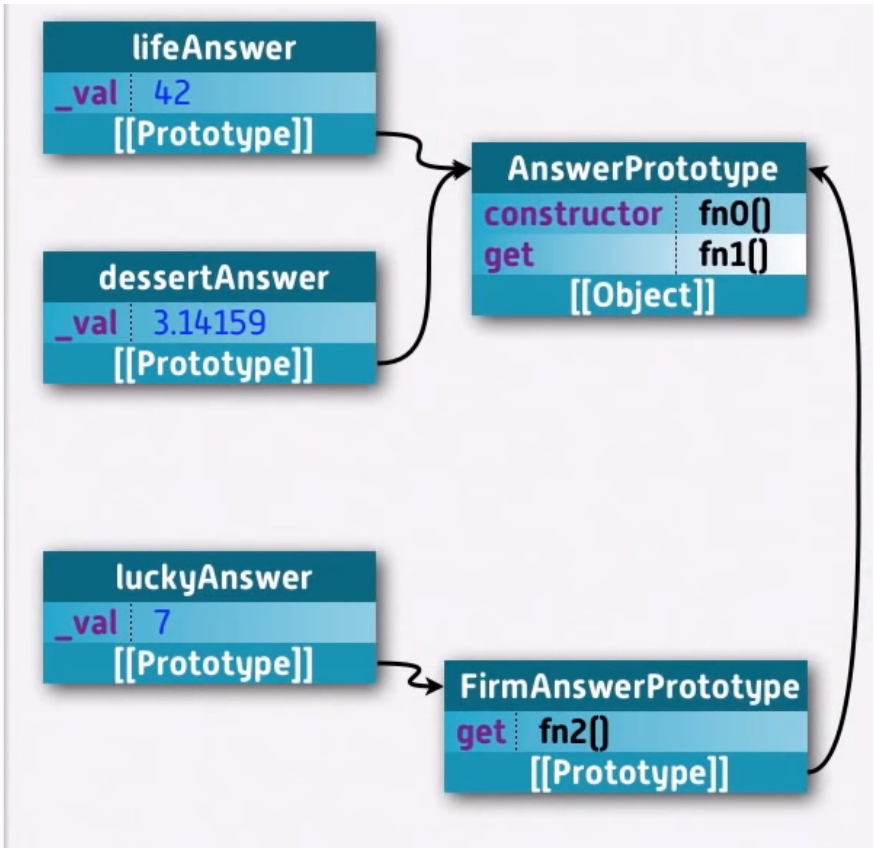
Sample Code (2)

```
var answer = {
  get: function fn1() {
    return this.val;
  },
  val: 42
};

var firmAnswer = Object.create(answer);
firmAnswer.get = function fn2() {
  return answer.get.call(this) + "!!";
};

firmAnswer.val = 3.14159;
firmAnswer.get();  // →"3.14159!!"
```



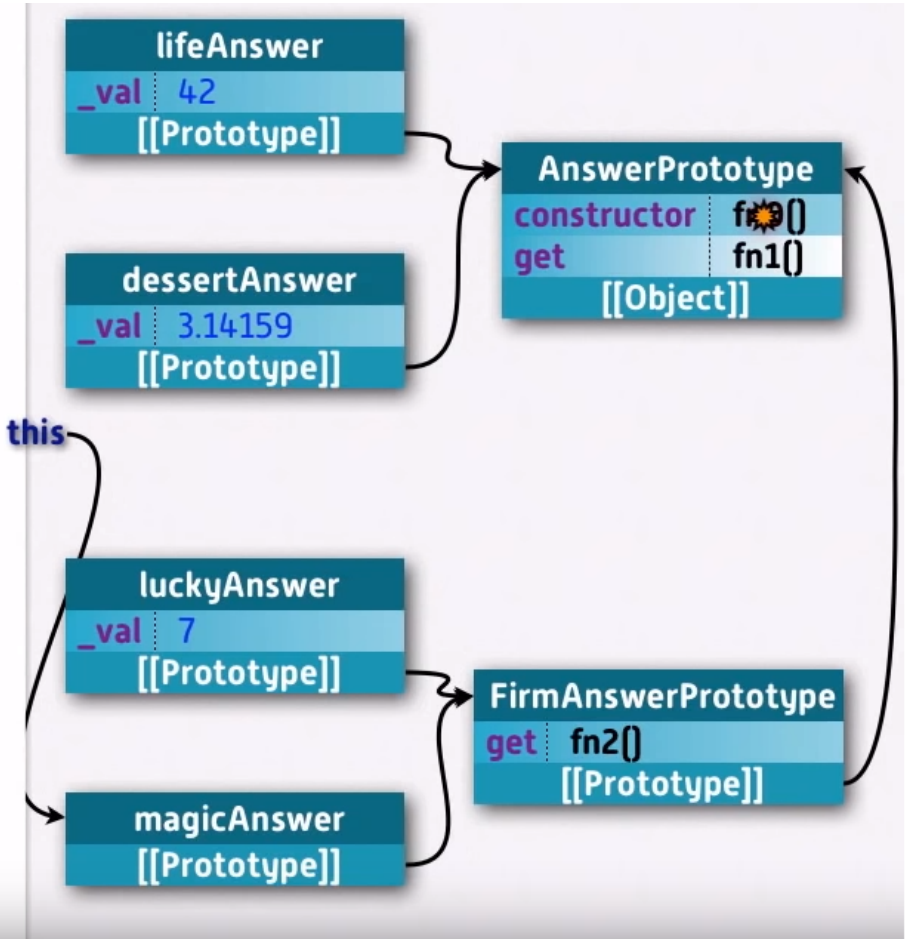
Sample Code (3)

```
var AnswerPrototype = {
  constructor: function fn0(value) {
    this._val = value;
  },
  get: function fn1() {
    return this._val;
  }
};

1. var lifeAnswer = Object.create(AnswerPrototype);
2. lifeAnswer.constructor(42);
   lifeAnswer.get();  // →42

1. var dessertAnswer = Object.create(AnswerPrototype);
2. dessertAnswer.constructor(3.14159);
   dessertAnswer.get();  // →3.14159


var FirmAnswerPrototype = Object.create(AnswerPrototype);
FirmAnswerPrototype.get = function fn2() {
  return AnswerPrototype.get.call(this) + "!!";
};

1. var luckyAnswer = Object.create(FirmAnswerPrototype);
2. luckyAnswer.constructor(7);
   luckyAnswer.get();  // →"7!!"
```



Sample Code (4)

```
var AnswerPrototype = {
  constructor: function fn0(value) {
    this._val = value;
  },
  get: function fn1() {
    return this._val;
  }
};

var lifeAnswer = Object.create(AnswerPrototype);
lifeAnswer.constructor(42);
lifeAnswer.get();  // →42

var dessertAnswer = Object.create(AnswerPrototype);
dessertAnswer.constructor(3.14159);
dessertAnswer.get();  // →3.14159


var FirmAnswerPrototype = Object.create(AnswerPrototype);
FirmAnswerPrototype.get = function fn2() {
  return AnswerPrototype.get.call(this) + "!!";
};

var luckyAnswer = Object.create(FirmAnswerPrototype);
luckyAnswer.constructor(7);
luckyAnswer.get();  // →"7!!"

var magicAnswer = Object.create(FirmAnswerPrototype);
magicAnswer.constructor(3);
```



Prototypal Vs Classical Model

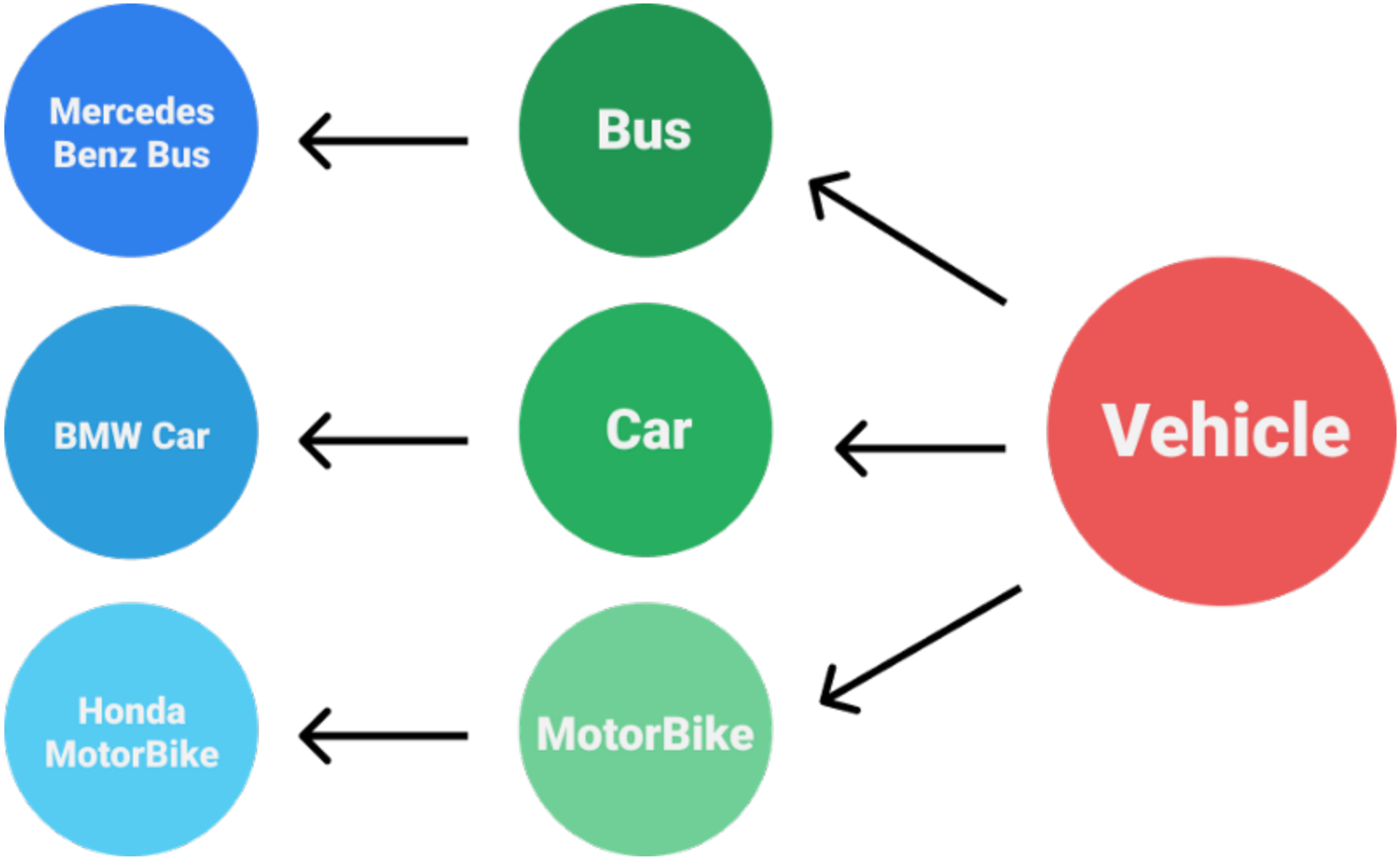
Prototypal Model

```
var AnswerPrototype = {
1.  constructor: function fn0(value) {
      this._val = value;
    },
2.  get: function fn1() {
      return this._val;
    }
};

3. var lifeAnswer = Object.create(AnswerPrototype);
   lifeAnswer.constructor(42);
   lifeAnswer.get();  // →42

4. var dessertAnswer = Object.create(AnswerPrototype);
   dessertAnswer.constructor(3.14159);
   dessertAnswer.get();  // →3.14159


5. var FirmAnswerPrototype = ↵
                        Object.create(AnswerPrototype);

6. FirmAnswerPrototype.get = function fn2() {
     return AnswerPrototype.get.call(this) + "!!";
   };

7. var luckyAnswer = Object.create(FirmAnswerPrototype);
   luckyAnswer.constructor(7);
   luckyAnswer.get();  // →"7!!"

8. var magicAnswer = Object.create(FirmAnswerPrototype);
   magicAnswer.constructor(3);
   magicAnswer.get();  // →"3!!"
```

Classical Model

```
1. function Answer(value) {
     this._val = value;
   }
2. Answer.prototype.get = function fn1() {
     return this._val;
   };

3. var lifeAnswer = new Answer(42);
   lifeAnswer.get();  // →42

4. var dessertAnswer = new Answer(3.14159);
   dessertAnswer.get();  // →3.14159


5. function FirmAnswer(value) {
     Answer.call(this, value);
   }
   FirmAnswer.prototype = ↵
                   Object.create(Answer.prototype);
   FirmAnswer.prototype.constructor = FirmAnswer;

6. FirmAnswer.prototype.get = function fn2() {
     return Answer.prototype.get.call(this) + "!!";
   };

7. var luckyAnswer = new FirmAnswer(7);
   luckyAnswer.get();  // →"7!!"

8. var magicAnswer = new FirmAnswer(3);
   magicAnswer.get();  // →"3!!"
```

Practice Problem



```
function Vehicle(vehicleType) {  //Vehicle Constructor
this.vehicleType = vehicleType;}
Vehicle.prototype.blowHorn = function () {
console.log('Honk! Honk! Honk!'); // All Vehicle can blow Horn}
function Bus(make) { // Bus Constructor
Vehicle.call(this, "Bus");
this.make = make}
Bus.prototype = Object.create(Vehicle.prototype); // Make Bus
constructor inherit properties from Vehicle Prototype Object
Bus.prototype.noOfWheels = 6; // Let's assume all buses have 6 wheels
Bus.prototype.accelerator = function() {    console.log('Accelerating
Bus'); //Bus accelerator}
Bus.prototype.brake = function() {
console.log('Braking Bus'); // Bus brake}
function Car(make) {  Vehicle.call(this, "Car");  this.make = make;}
Car.prototype = Object.create(Vehicle.prototype);
Car.prototype.noOfWheels = 4;
Car.prototype.accelerator = function() {
console.log('Accelerating Car');}
Car.prototype.brake = function() {    console.log('Braking Car');}
function MotorBike(make) {  Vehicle.call(this, "MotorBike");
this.make = make;}
MotorBike.prototype = Object.create(Vehicle.prototype);
MotorBike.prototype.noOfWheels = 2;
MotorBike.prototype.accelerator = function() {
console.log('Accelerating MotorBike');}
MotorBike.prototype.brake = function() {    console.log('Braking
MotorBike');}
var myBus = new Bus('Mercedes');
var myCar = new Car('BMW');
var myMotorBike = new MotorBike('Honda')
```