JavaScript: Objects, Methods, Prototypes

Lecture 25

What is an Object?

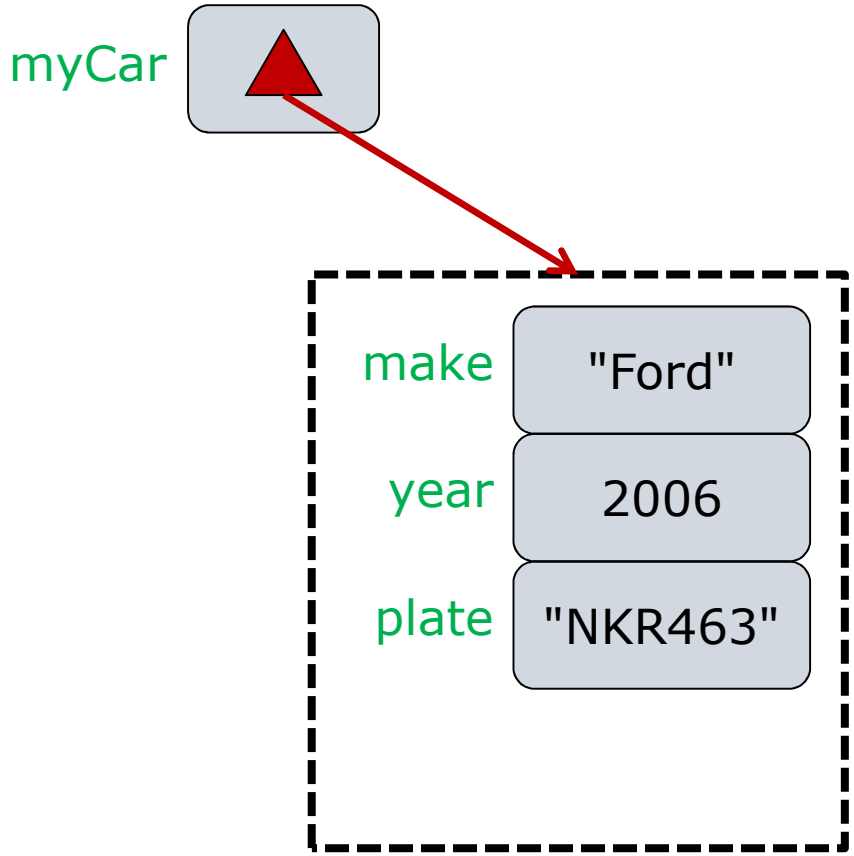
- *Property*: a key/value pair
  - aka name/value pair
- *Object*: a partial map of properties
  - Keys must be unique
- Creating an object, literal notation

```
let myCar = { make: "Acura",
              year: 1996,
              plate: "NKR462" };
```

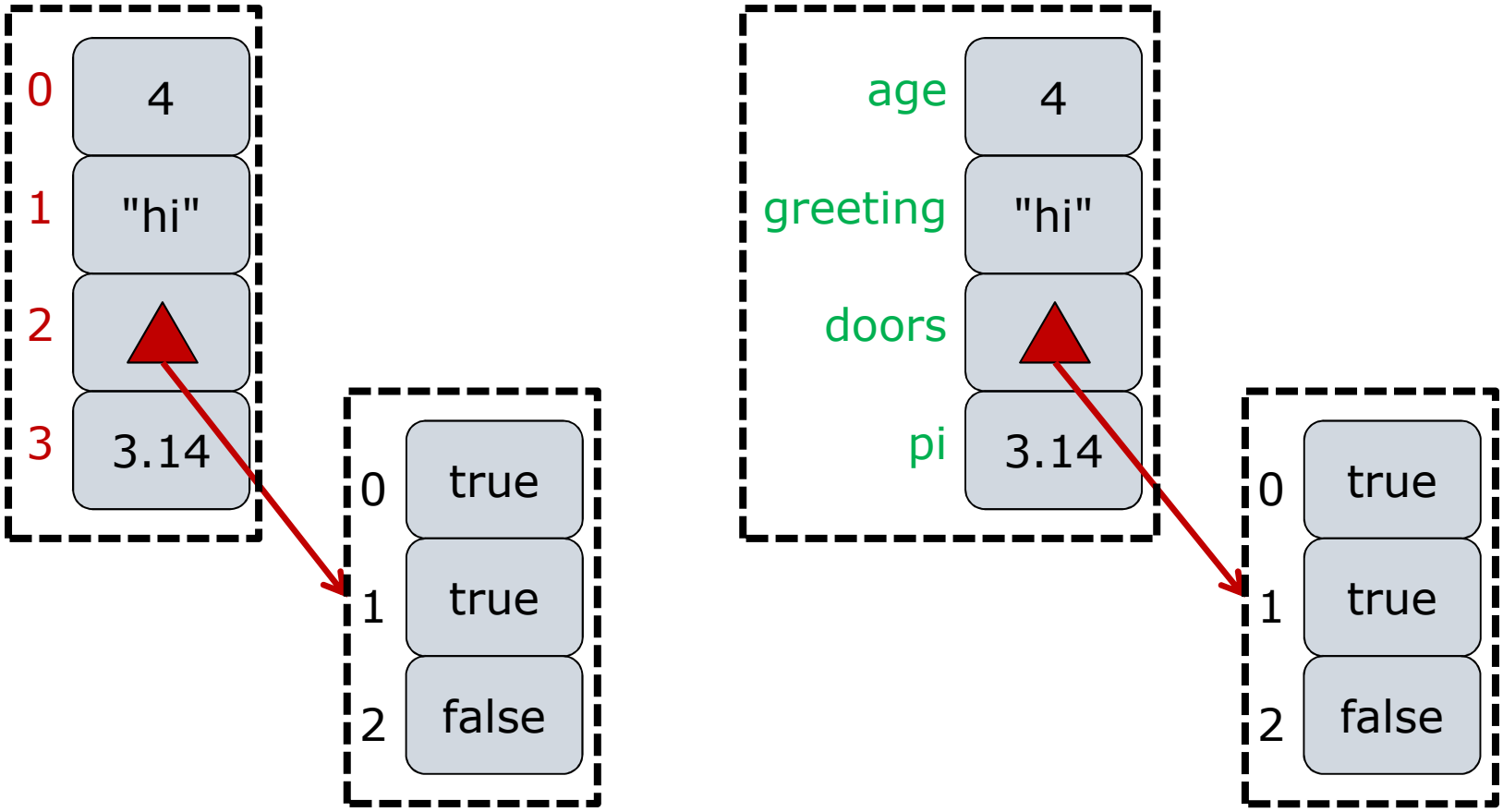
- To access/modify an object's properties:

```
myCar.make = "Ford";  // cf. Ruby
myCar["year"] = 2006;
let str = "ate";
myCar["pl" + str] == "NKR463"; //=> true
```

Object Properties



Arrays vs Associative Arrays



Dynamic Size, Just Like Arrays

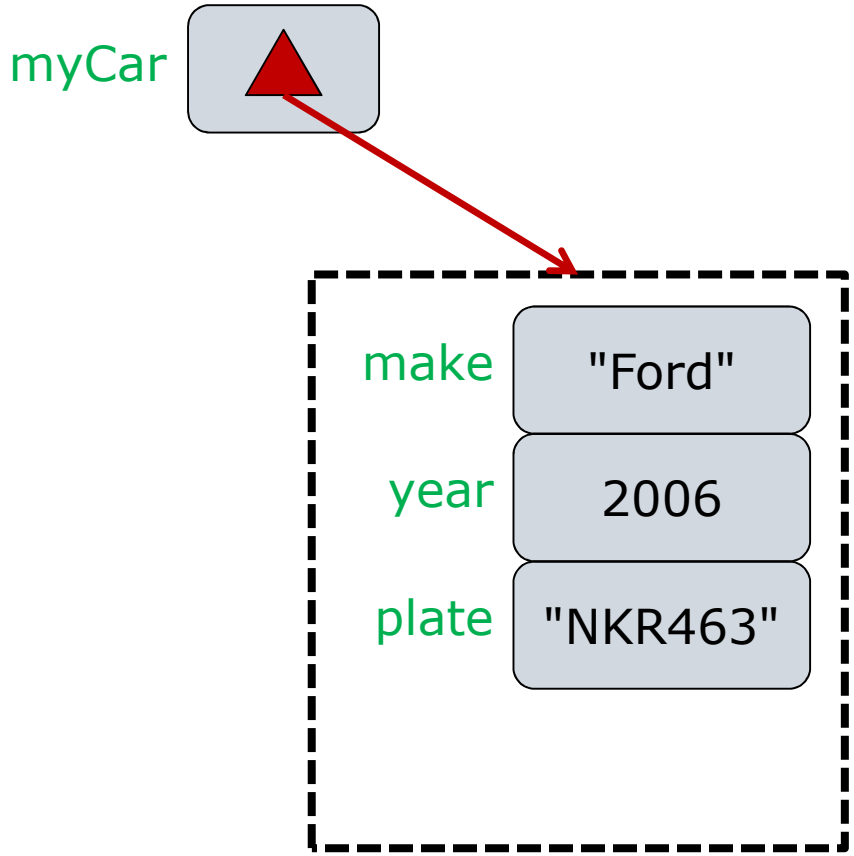
- Objects can grow

```
myCar.state = "OH"; // 4 properties
let myBus = {};
myBus.driver = true; // adds a prop
myBus.windows = [2, 2, 2, 2];
```

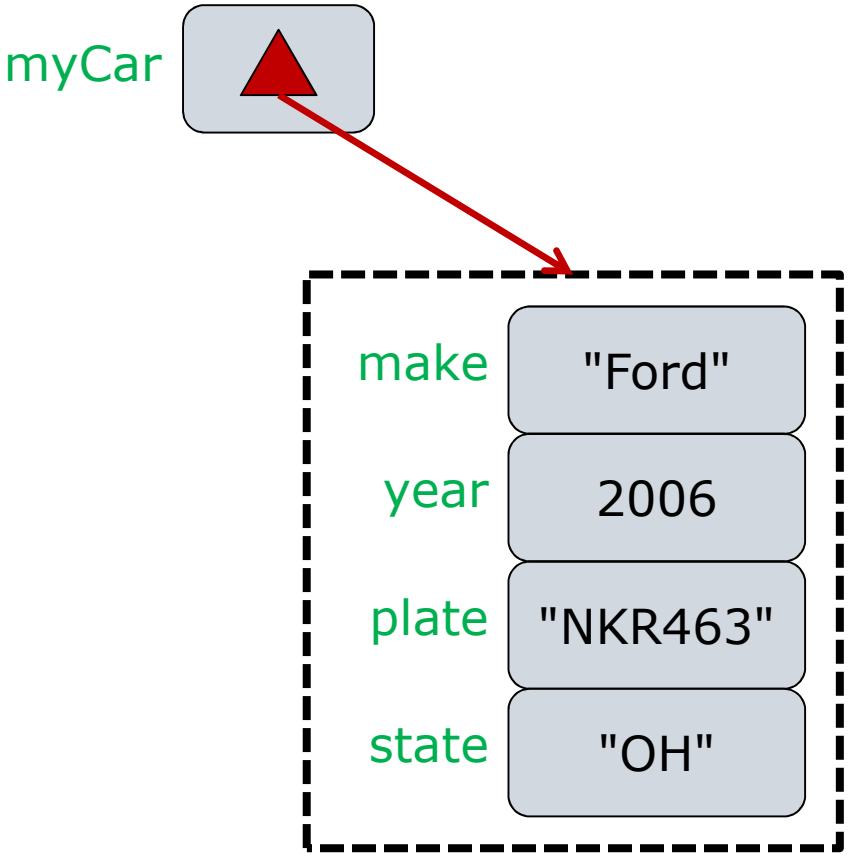
- Objects can shrink

```
delete myCar.plate;
// myCar is now {make: "Ford",
//               year: 2006, state: "OH"}
```

Object Properties

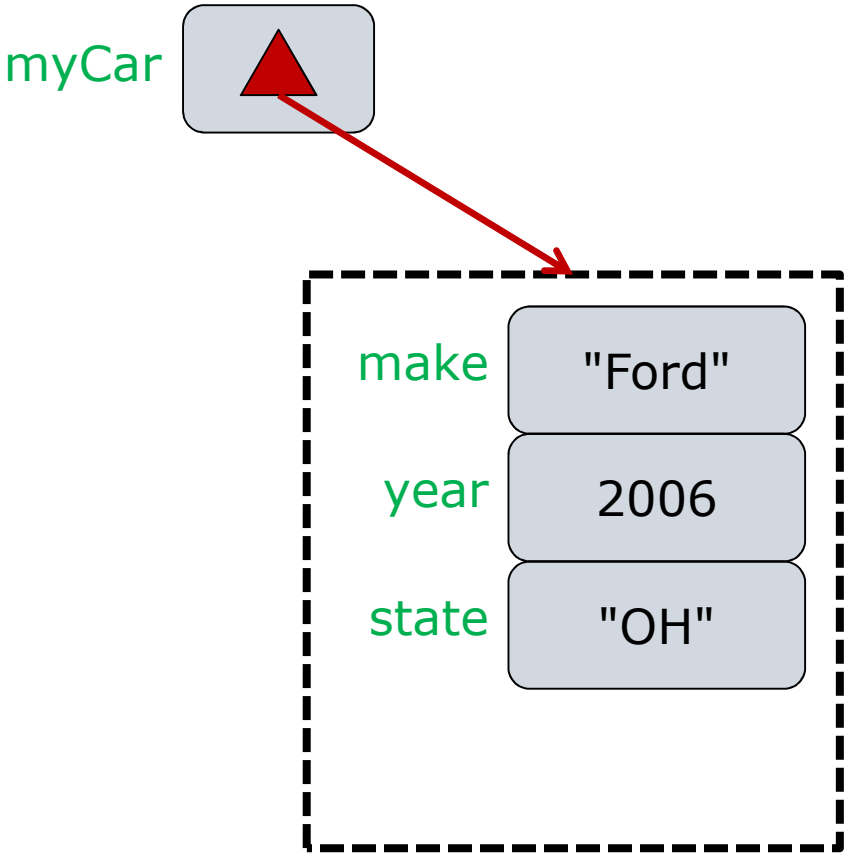


Object Properties




```
myCar.state = "OH";
```

Object Properties




```
delete myCar.plate;
```

Testing Presence of Key

- Boolean operator: *in*

```
propertyName in object
```

- Evaluates to true iff object has the indicated property key

```
"make" in myCar            //=> true
"speedometer" in myCar     //=> false
"OH" in myCar              //=> false
```

  - Property names are strings

Iterating Over Properties

- Iterate using *for…in* syntax

```
for (property in object) {
  …object[property]…
}
```

- Notice `[]` to access each property

```
for (p in myCar) {
  document.write(p + ": " + myCar[p]);
}
```

Methods

- The value of a property can be:
  - A primitive (boolean, number, string, null…)
  - A reference (object, array, *function*)

```
let temp = function(sound) {
  play(sound);
  return 0;
}
myCar.honk = temp;
```

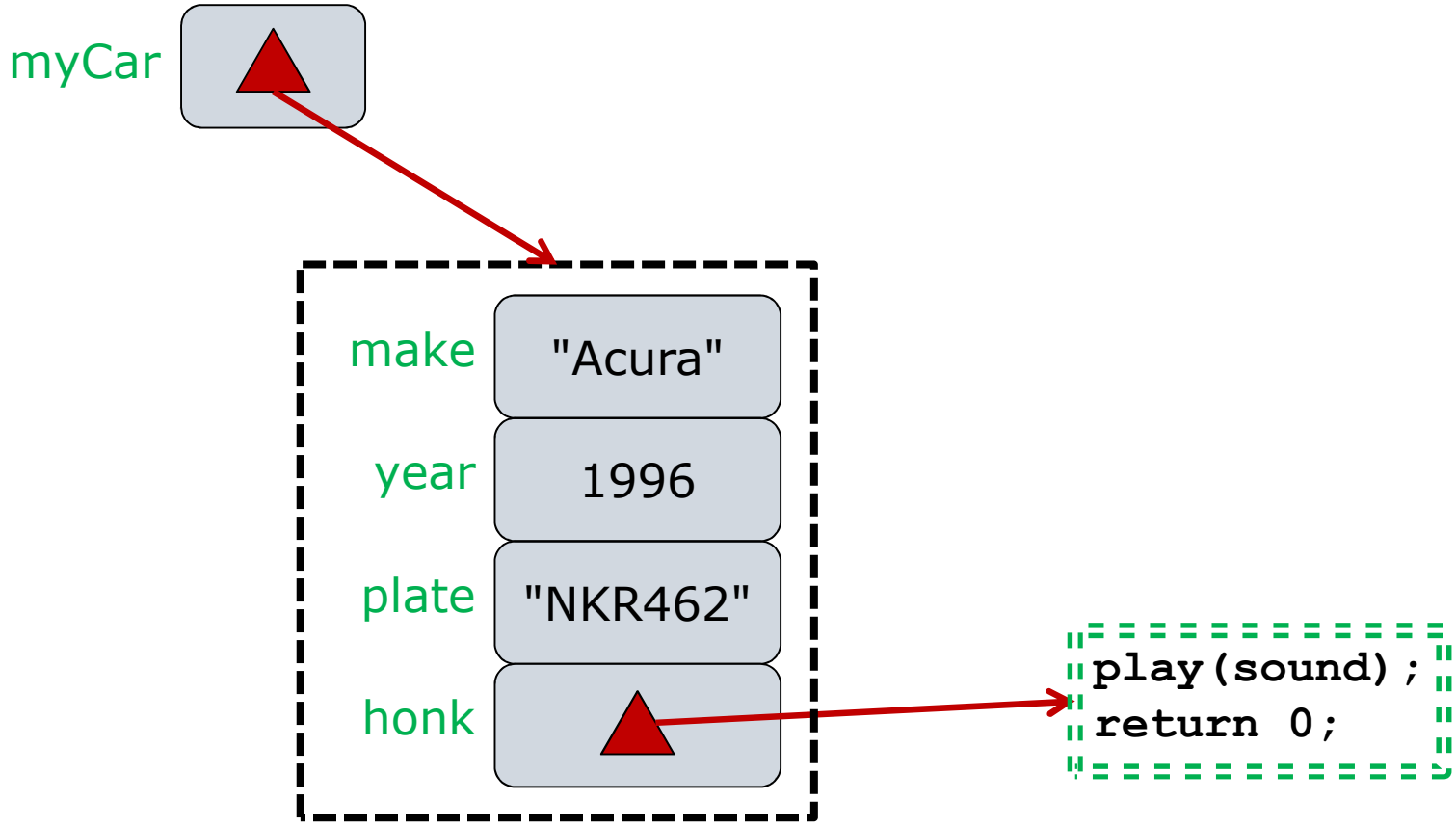
- More succinctly:

```
myCar.honk = function(sound) {
  play(sound);
  return 0;
}
```

Example: Method


```
let myCar = {
    make: "Acura",
    year: 1996,
    plate: "NKR462",
    honk: function(sound) {
        play(sound);
        return 0;
    }
};
```

Object Properties



Keyword "this" in Functions

- Recall *distinguished formal parameter*

```
x.f(y, z); //x is the distinguished argmt.
```

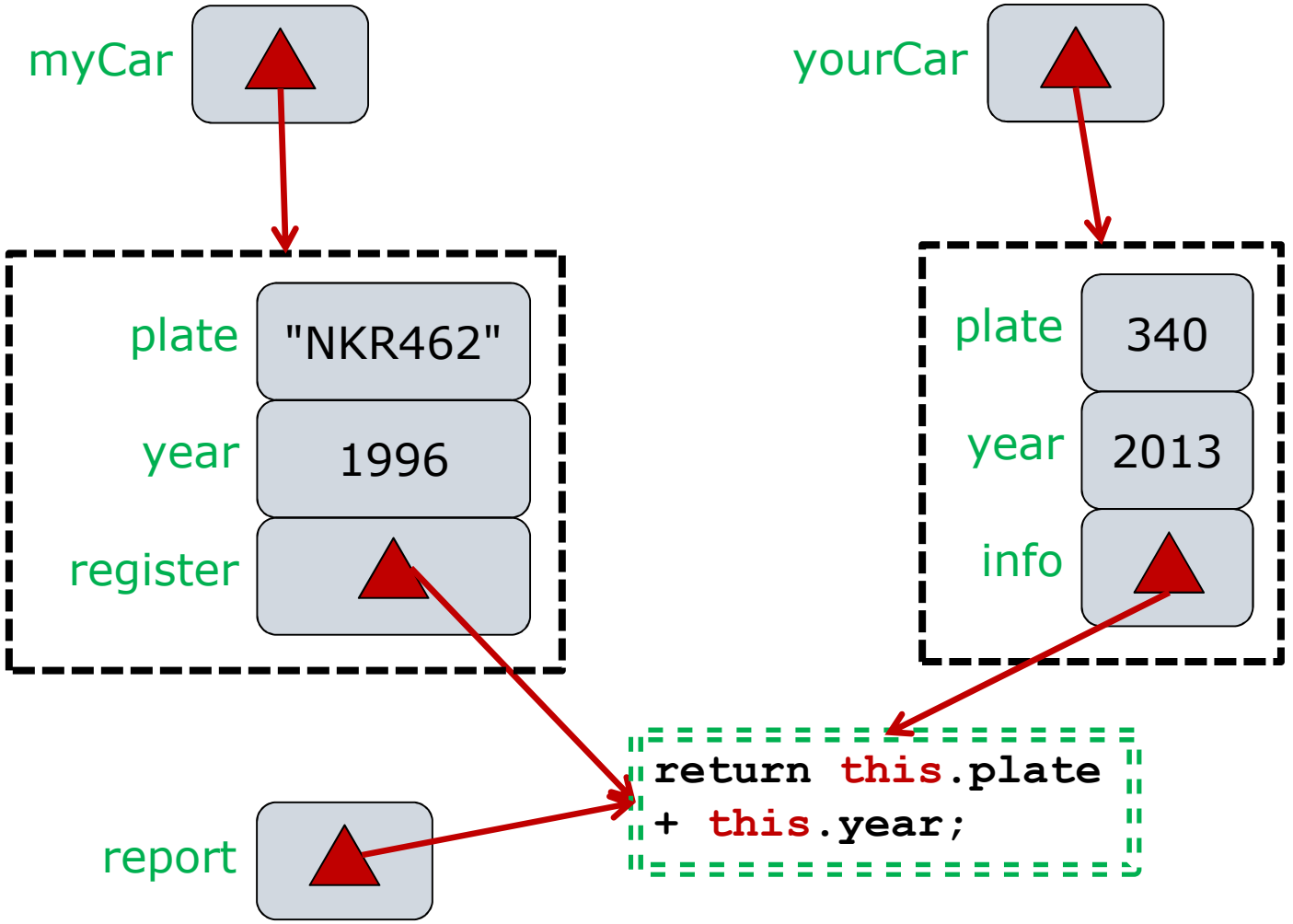
- Inside a function, keyword "this"

```
function report() {
  return this.plate + this.year;
}
```

- At run-time, "this" is set to the *distinguished argument* of invocation

```
myCar = { plate: "NKR462", year: 1996 };
yourCar = { plate: 340, year: 2013 };
myCar.register = report;
yourCar.info = report;
myCar.register();      //=> "NKR4621996"
yourCar.info();        //=> 2353
```

Object Properties



Constructors

- *Any* function can be a constructor
- When calling a function with "new":
  1. Make a brand new (empty) object
  2. Call the function, with the new object as the distinguished parameter
  3. Implicitly return the new object to caller
- A "constructor" often adds properties to the new object simply by assigning them

```
function Dog(name) {
  this.name = name;  // adds 1 property
  // no explicit return
}
let furBall = new Dog("Rex");
```

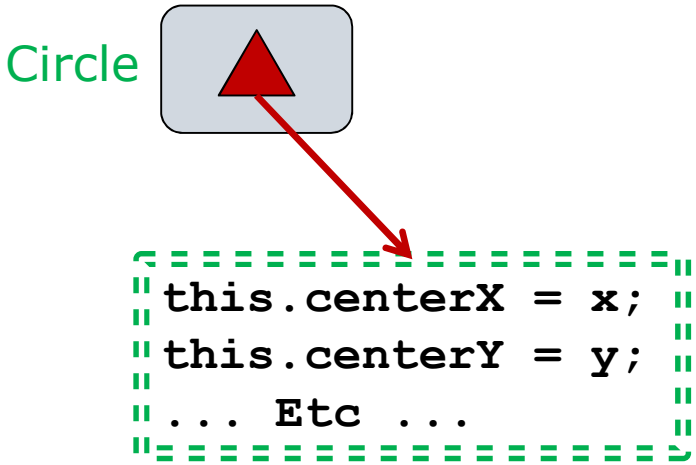
- Naming convention: Functions intended to be constructors are capitalized

Example


```
function Circle(x, y, radius) {
  this.centerX = x;
  this.centerY = y;
  this.radius = radius;
  this.area = function() {
    return Math.PI * this.radius *
           this.radius;
  }
}
let c = new Circle(10, 12, 2.45);
```

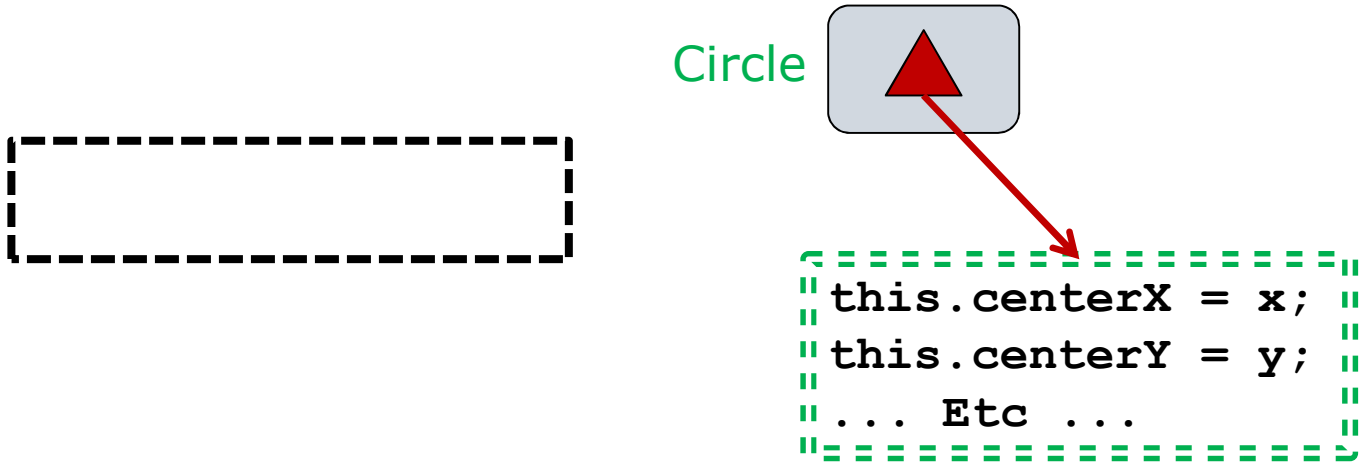
Creating a Circle Object


```
let c = new Circle(10, 12, 2.45);
```



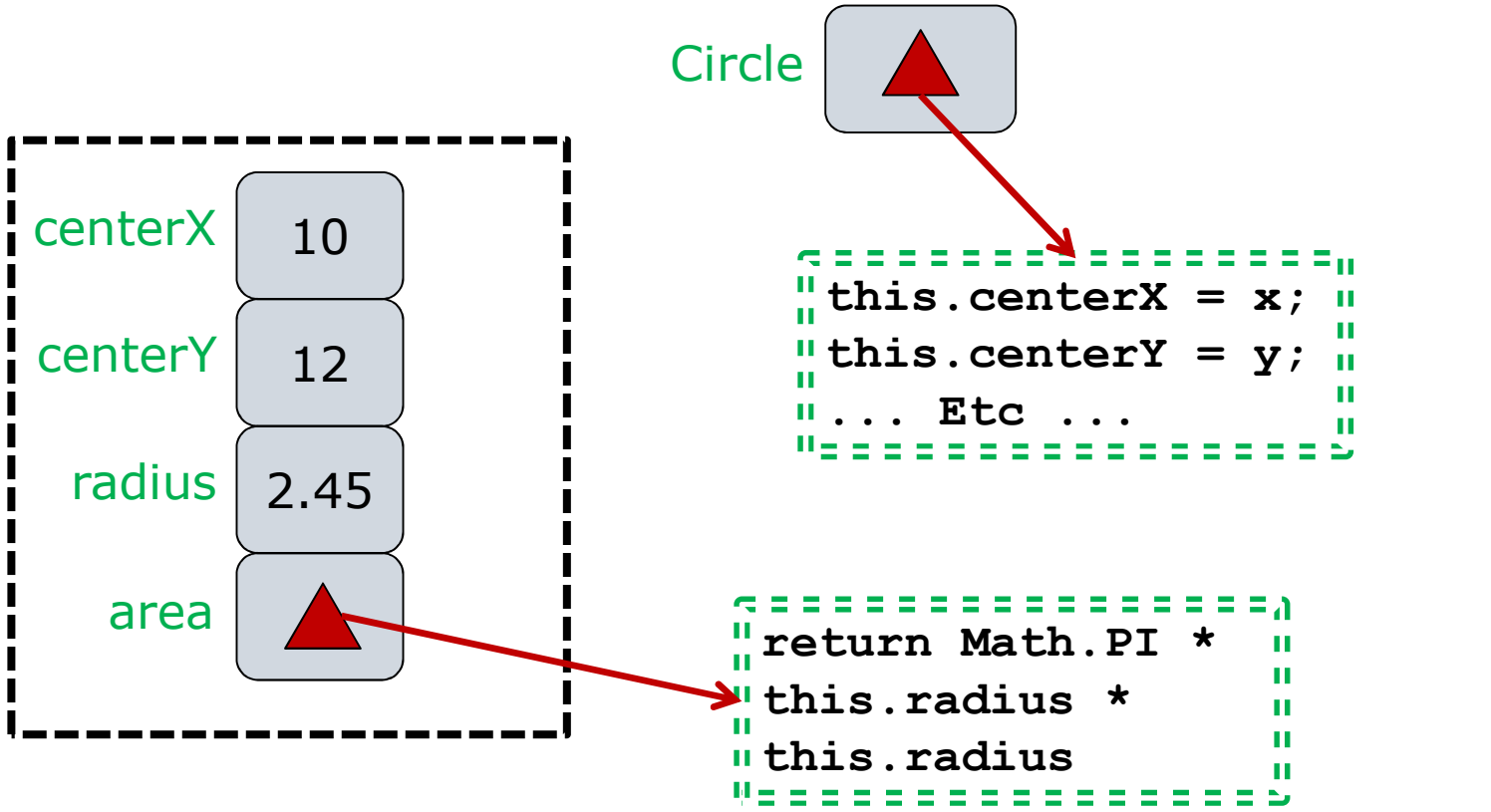
Creating a Circle Object


```
let c = new Circle(10, 12, 2.45);
```

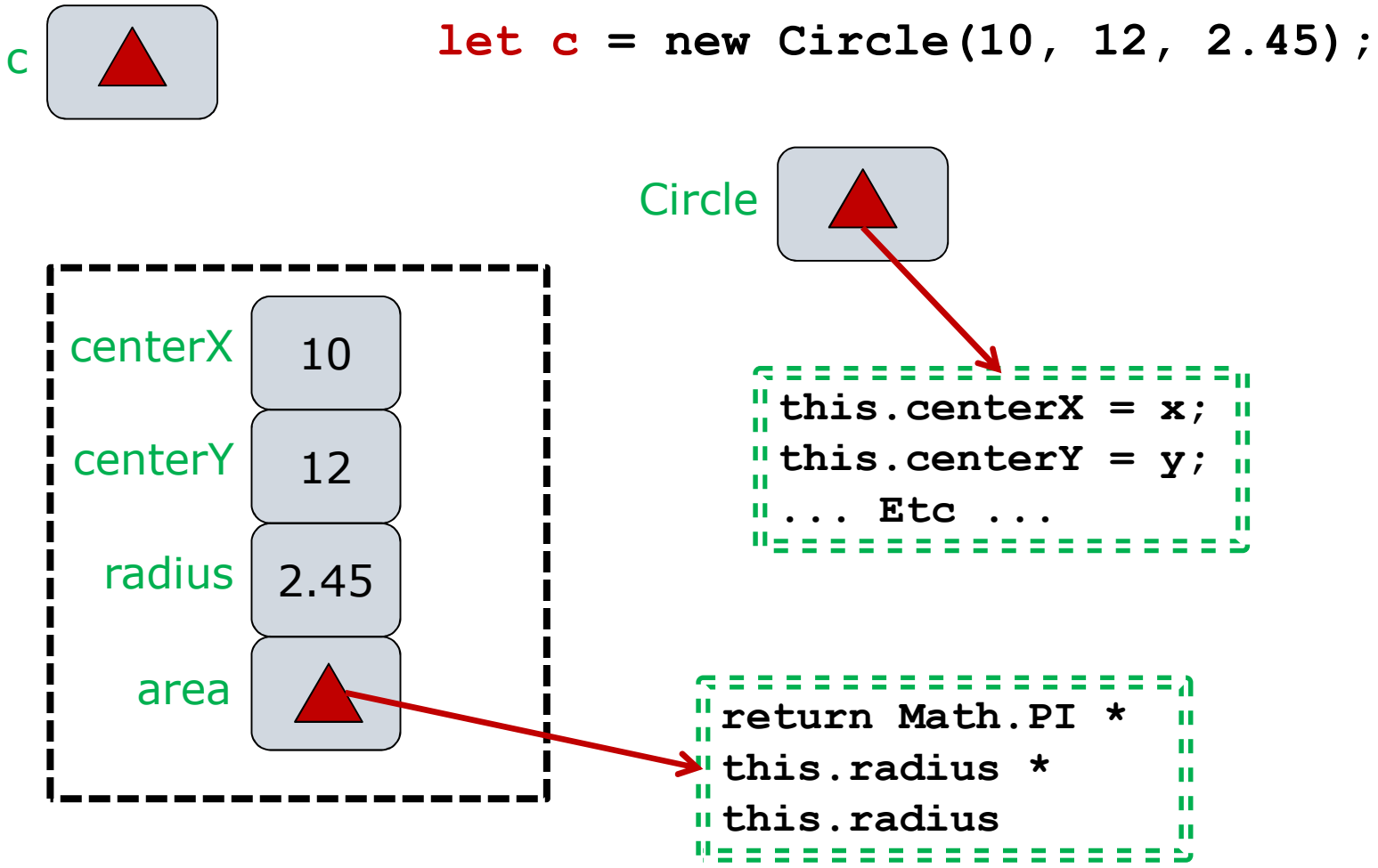


Creating a Circle Object

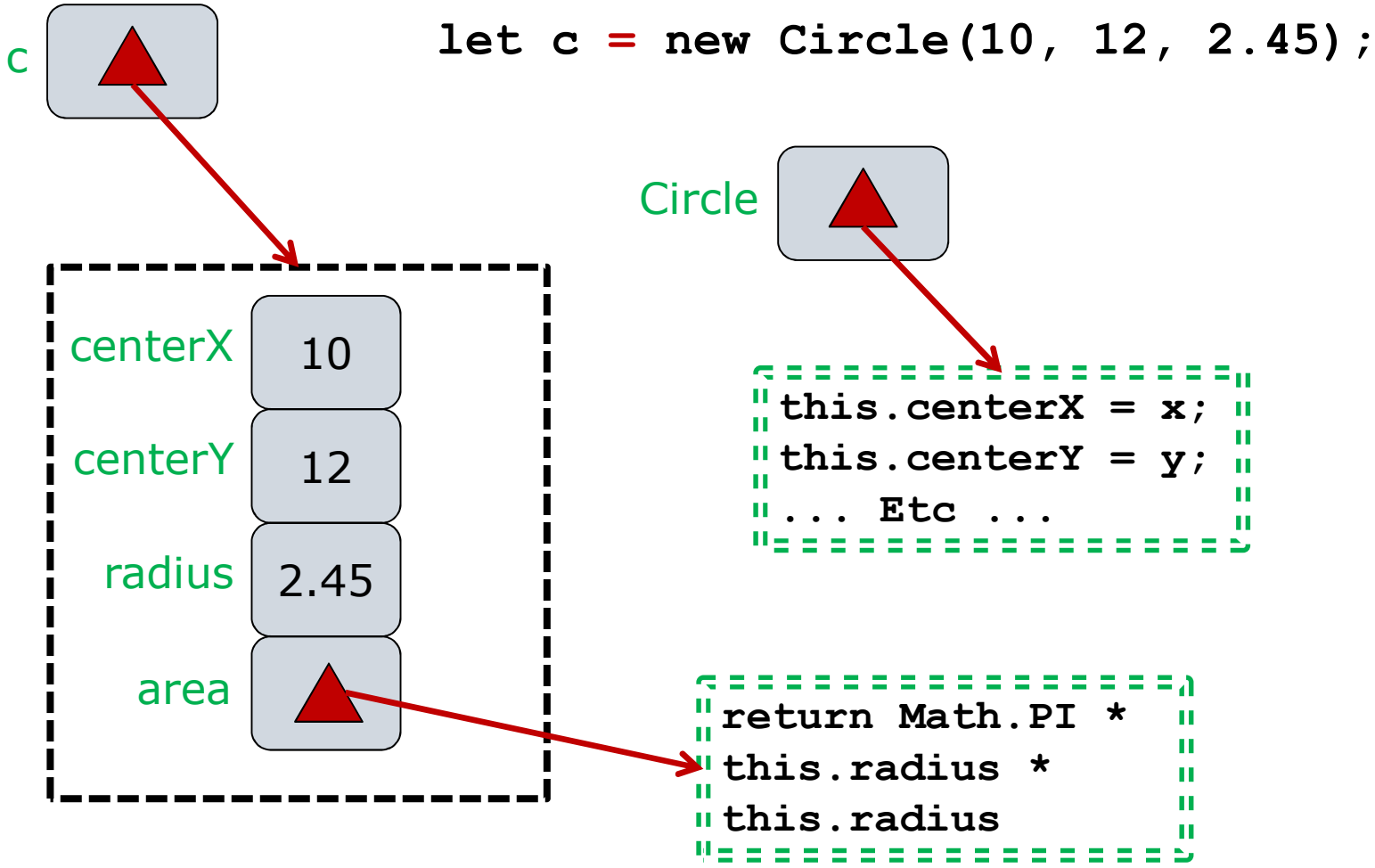

```
let c = new Circle(10, 12, 2.45);
```



Creating a Circle Object



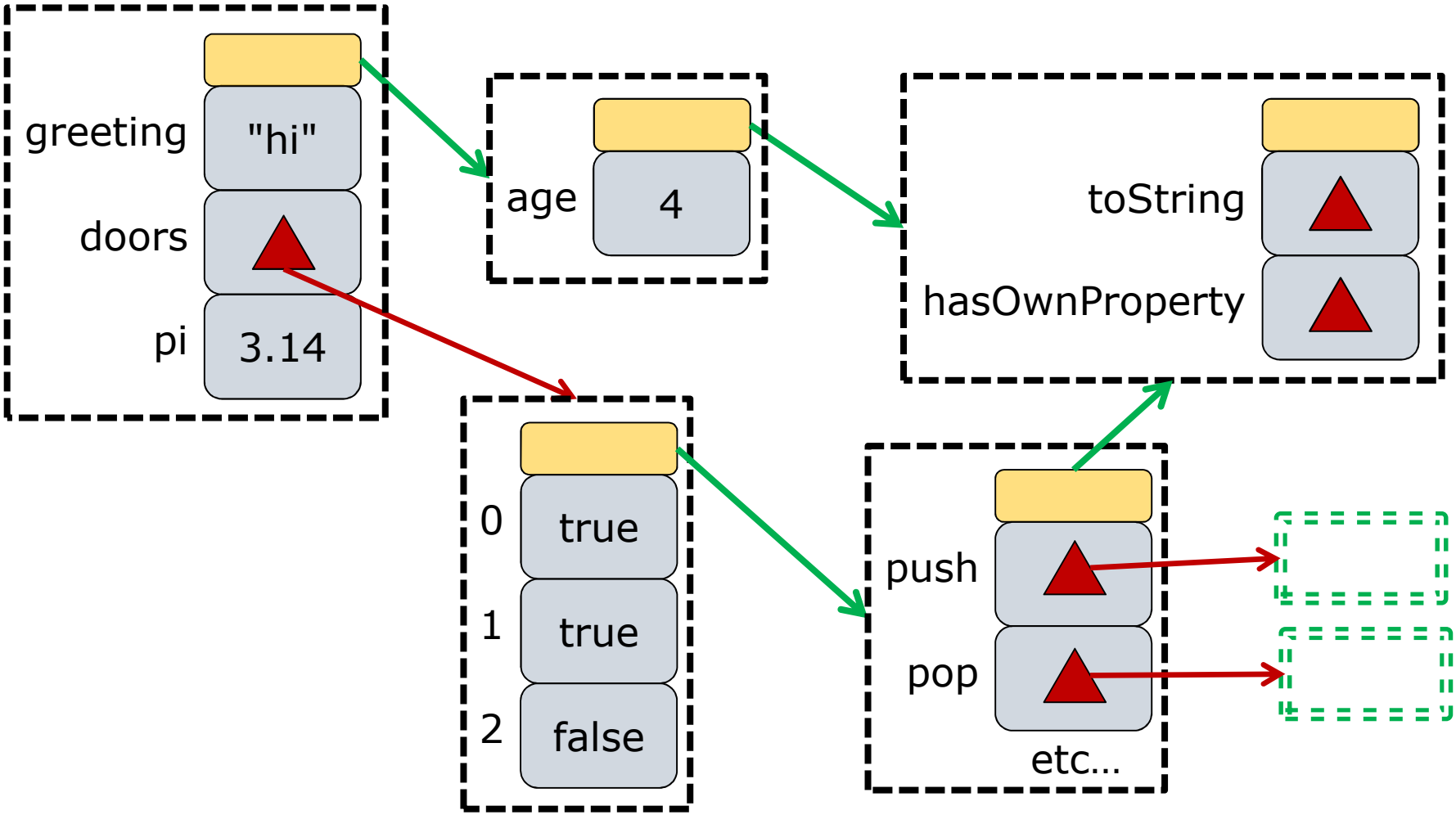
Creating a Circle Object



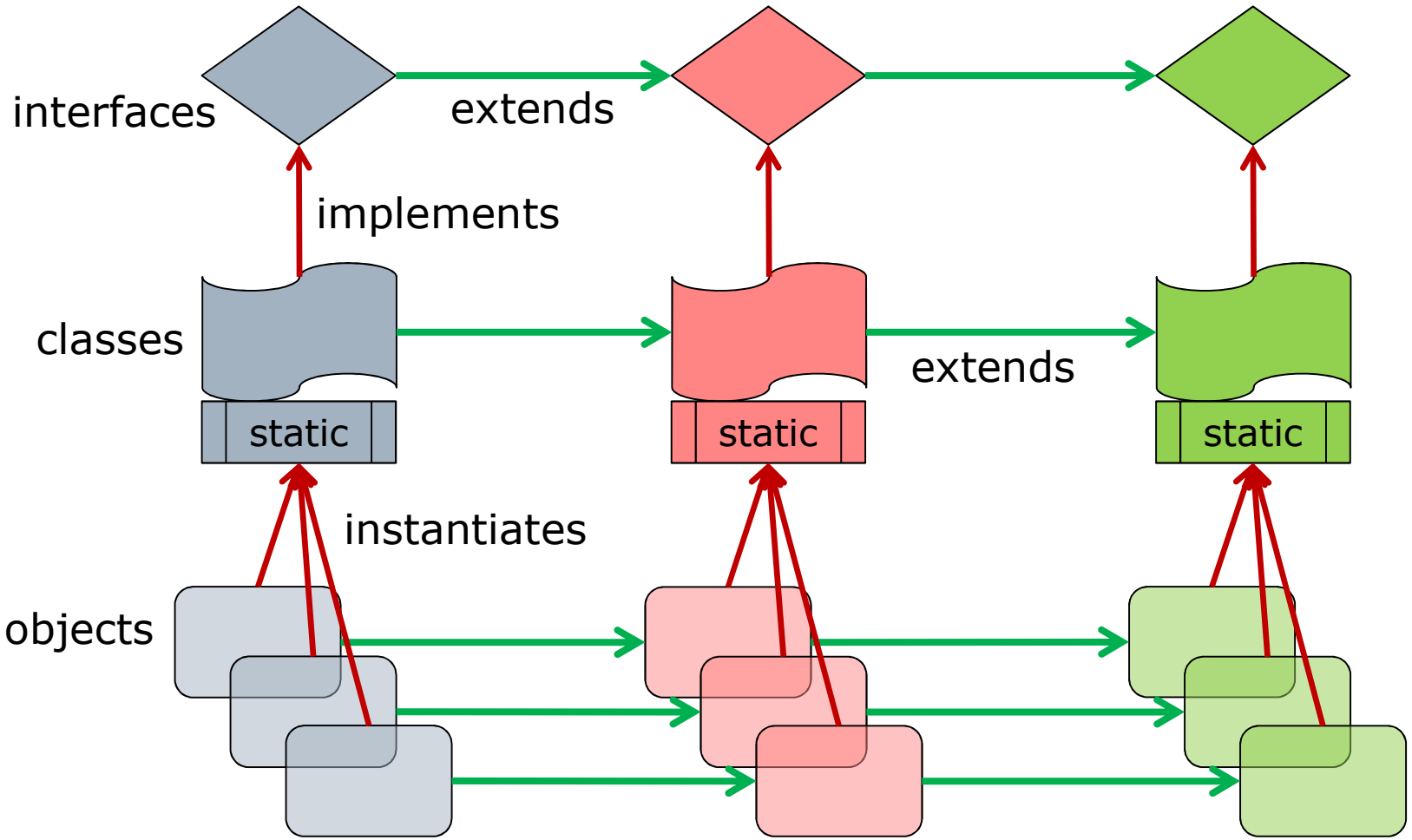
Prototypes

- Every object has a *prototype*
  - A hidden, indirect property ([[Prototype]])
- What is a prototype?
  - Just another object! Like any other!
- When accessing a property (*i.e.* `obj.p`)
  - First look for `p` in `obj`
  - If not found, look for `p` in `obj`'s prototype
  - If not found, look for `p` in that object's prototype!
  - And so on, until reaching the basic system object

Prototype Chaining



Class-Based Inheritance



Example

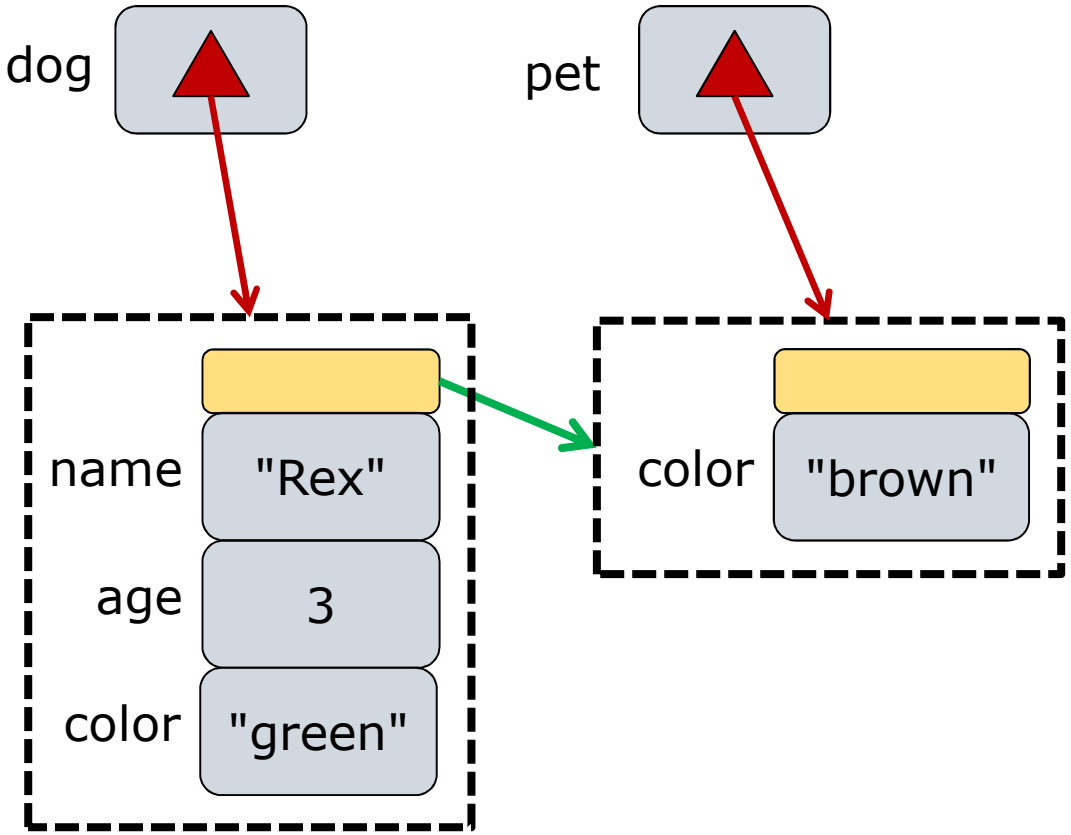
- Consider two objects

```
let dog = { name: "Rex", age: 3 };
let pet = { color: "blue" };
```

- Assume **pet** is **dog**'s prototype

```
// dog.name is "Rex"
// dog.color is "blue" (follow chain)
pet.color = "brown";
// dog.color is "brown" (prop changed)
dog.color = "green";
// pet.color is still "brown" (hiding)
```

Delegation to Prototype

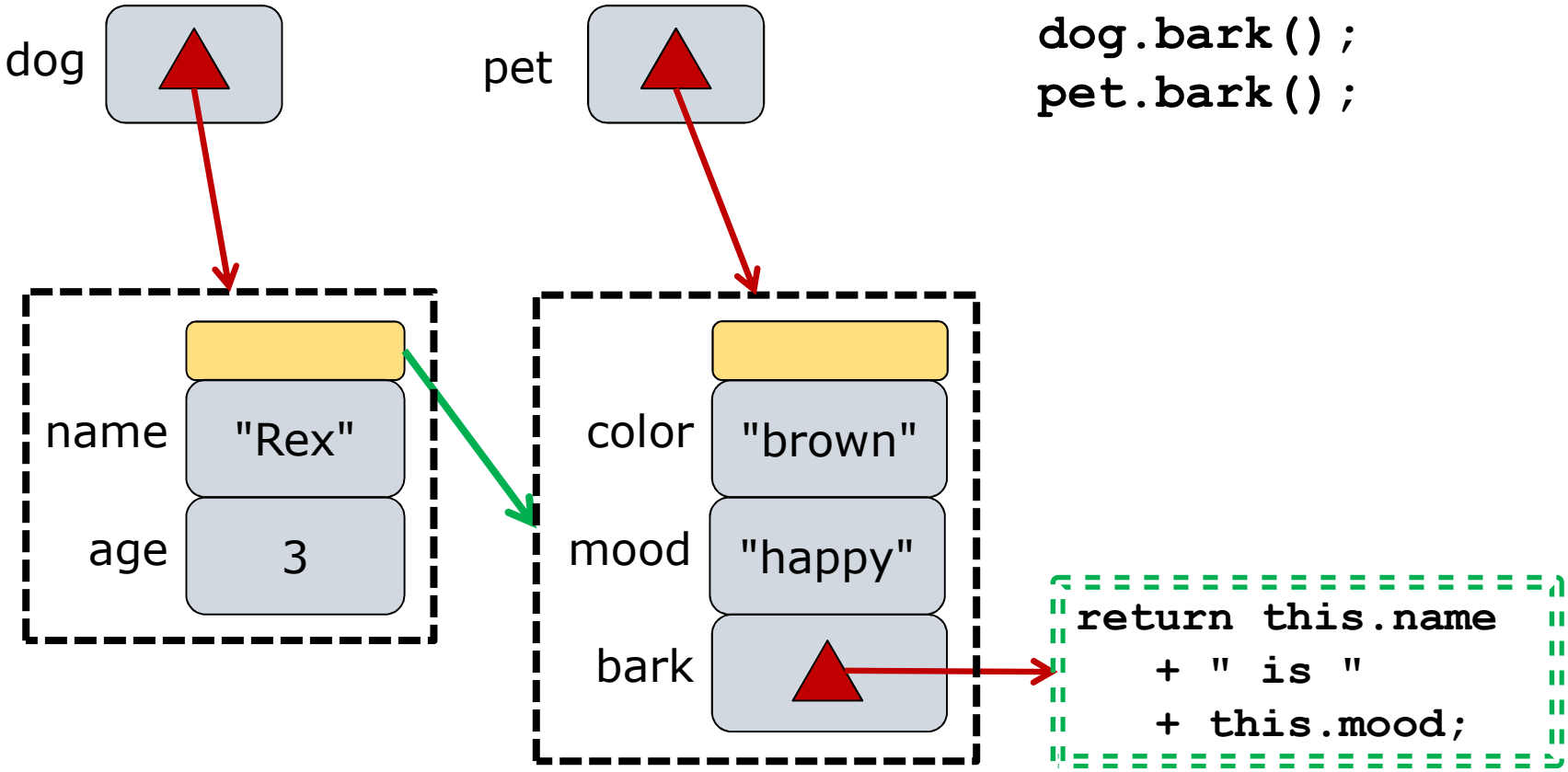


Prototypes Are Dynamic Too

- Prototypes can add/remove properties
- Changes are felt by all children

```
// dog is { name: "Rex", age: 3 }
// dog.mood & pet.mood are undefined
pet.mood = "happy"; // add to pet
// dog.mood is now "happy" too
pet.bark = function() {
  return this.name + " is " + this.mood;
}
dog.bark(); //=> "Rex is happy"
pet.bark(); //=> "undefined is happy"
```

Delegation to Prototype



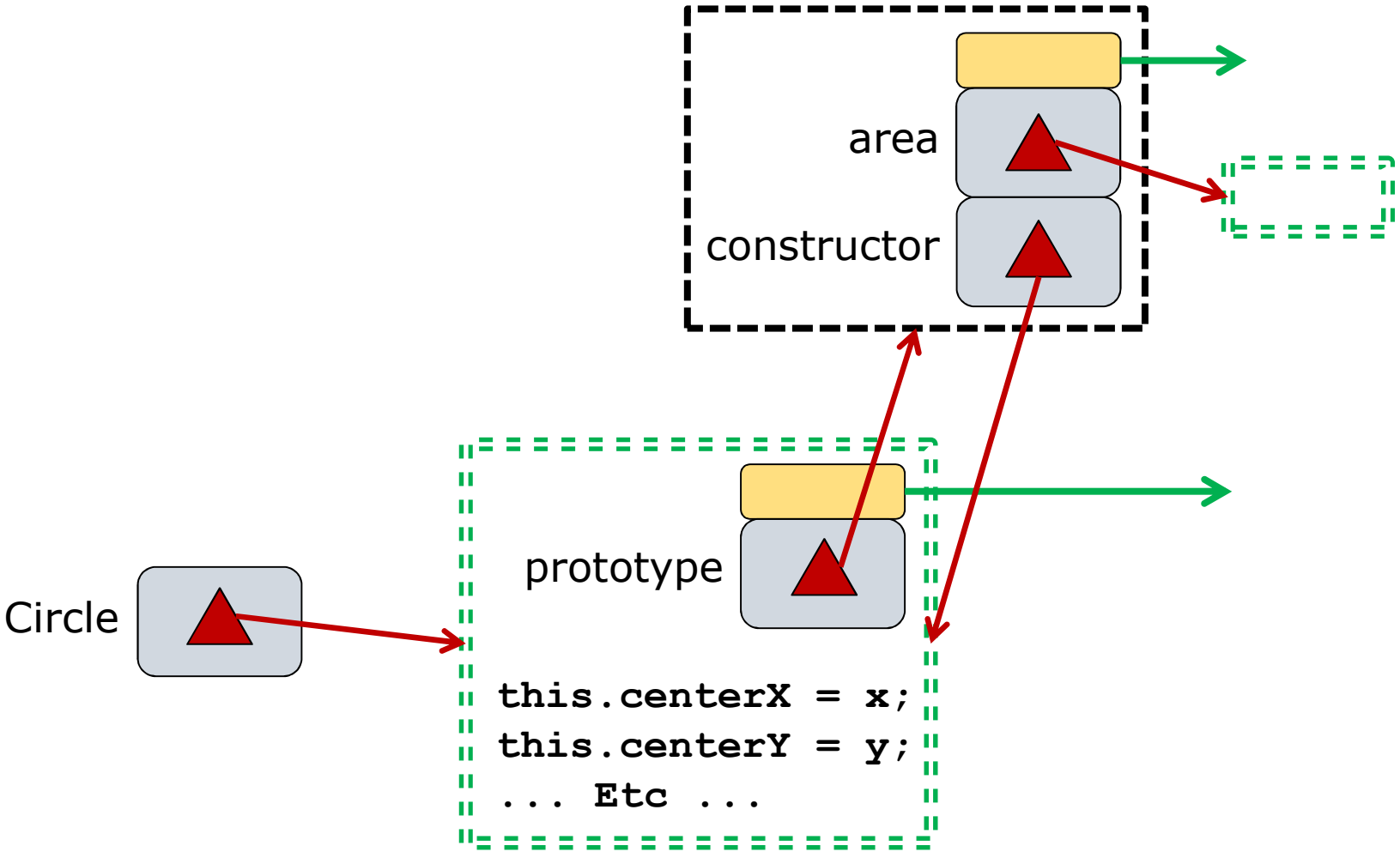
Connecting Objects & Prototypes

- How does an object get a prototype?

```
let c = new Circle();
```

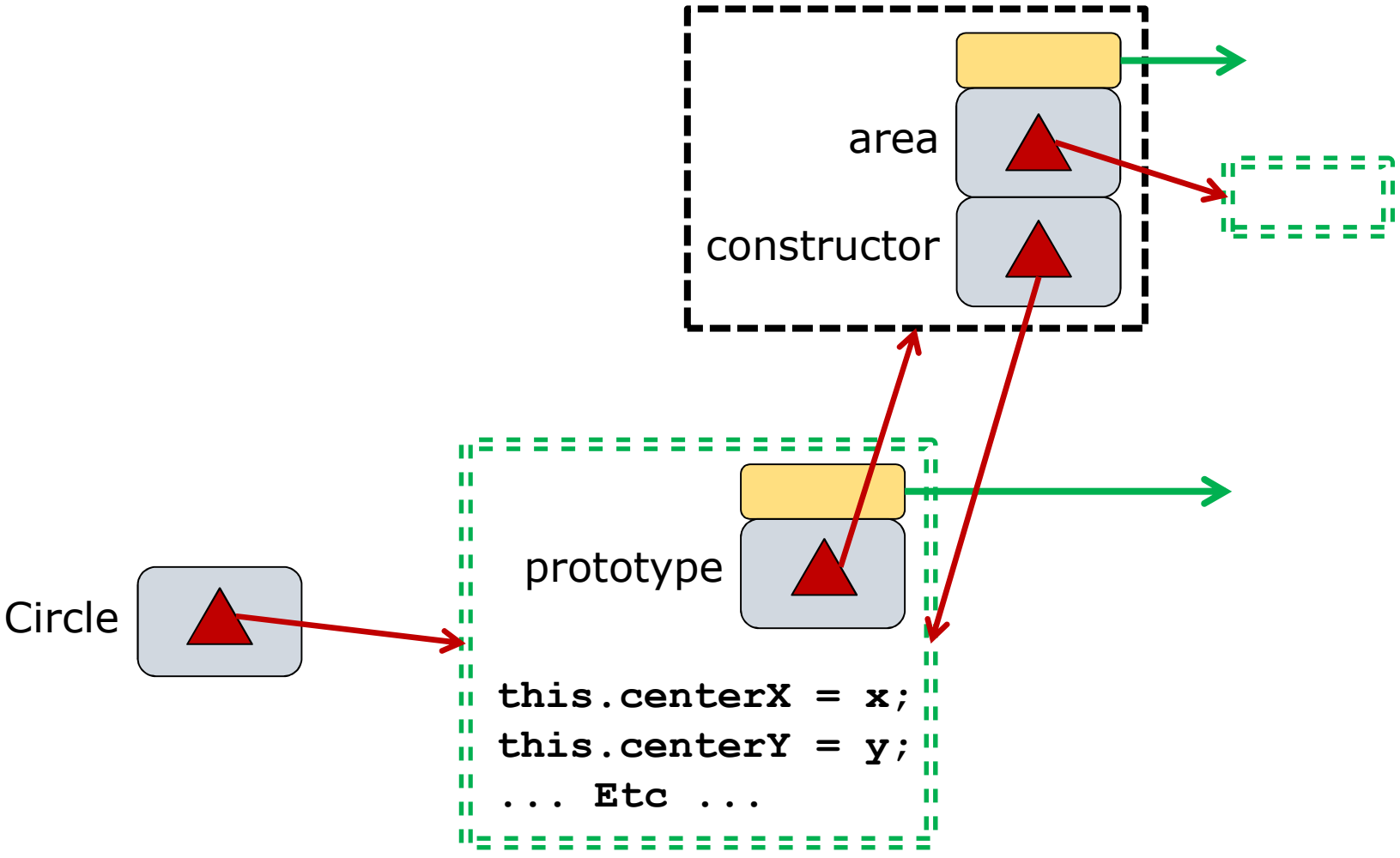
- Answer
  1. Every function has a prototype *property*
     - Do not confuse with hidden **`[[Prototype]]`**!
  2. Object's prototype *link*—**`[[Prototype]]`**—is set to the function's prototype *property*
- When a function **`Foo`** is used as a constructor, *i.e.* **`new Foo()`**, the value of **`Foo`**'s prototype property is the prototype object of the created object

Prototypes And Constructors

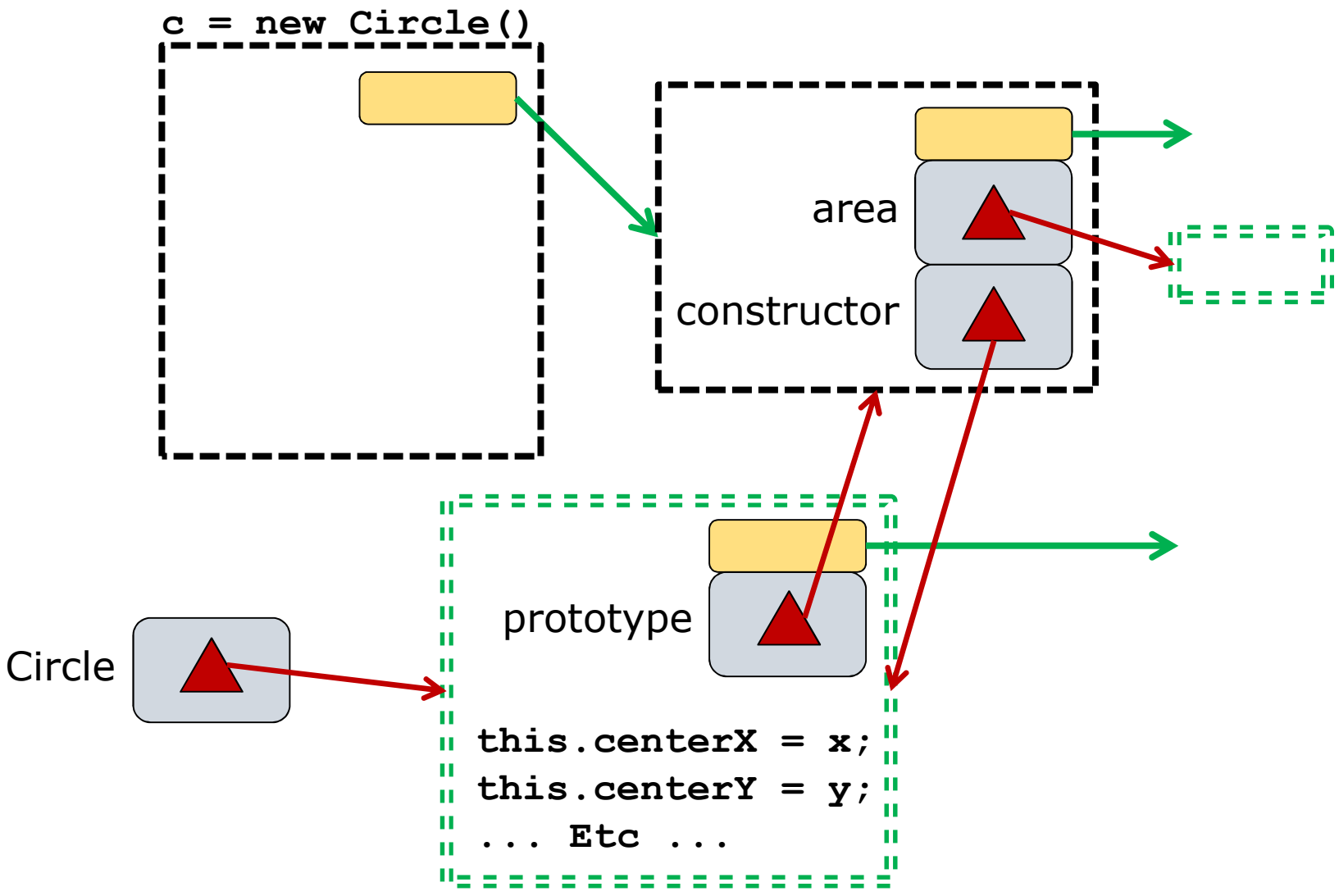


Prototypes And Constructors

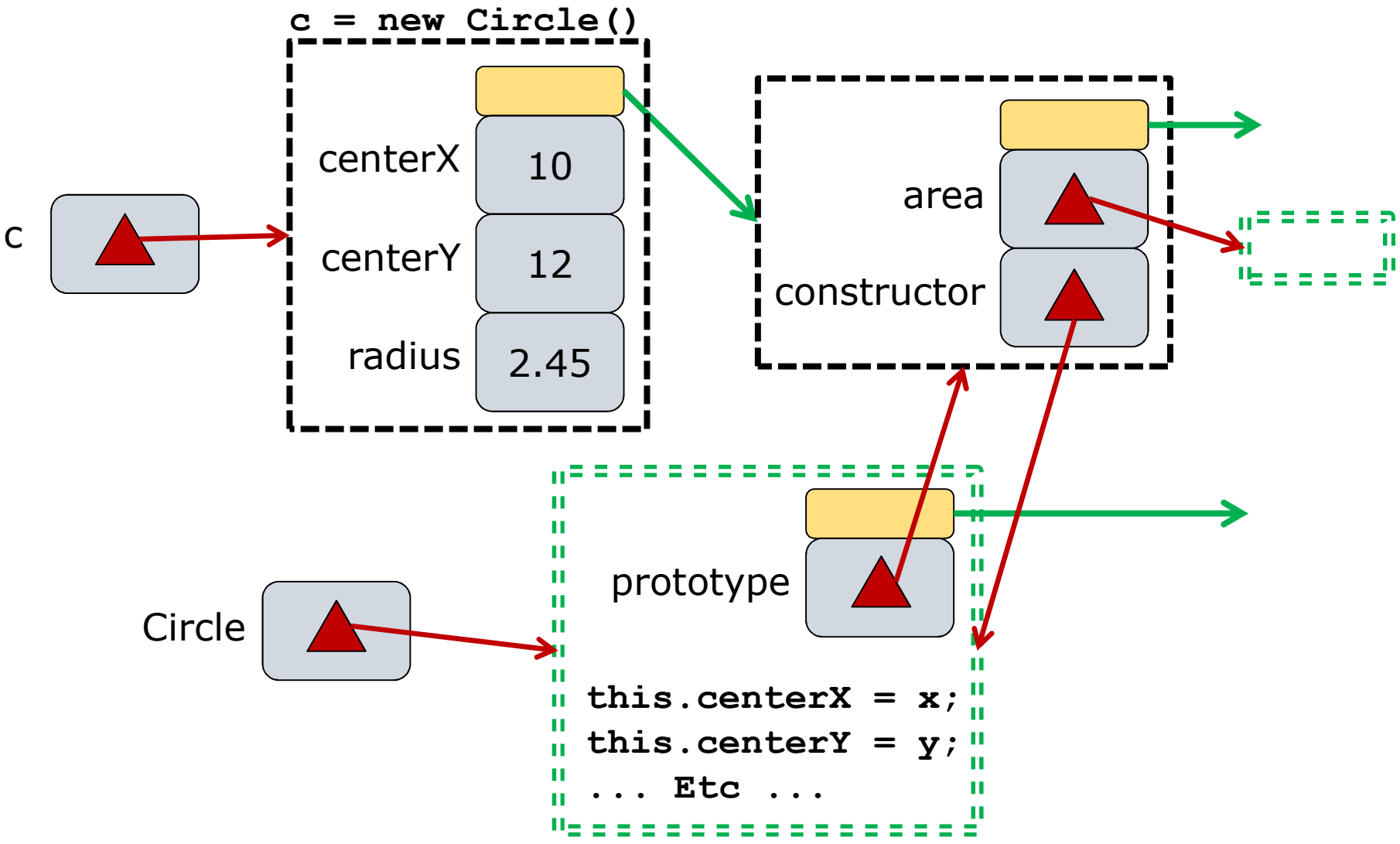

```
c = new Circle()
```



Prototypes And Constructors



Prototypes And Constructors



Idiom: Methods in Prototype


```
function Dog(n, a) {
    this.name = n;
    this.age = a;
};

let canine = {
    bark: function(sound) {
      return this.name + "says" + sound;
    }
};

Dog.prototype = canine;
```

Idiom: Methods in Prototype


```
function Dog(n, a) {
    this.name = n;
    this.age = a;
};

let canine = {
    bark: function(sound) {
      return this.name + "says" + sound;
    }
};

Dog.prototype = canine;
```

Idiom: Methods in Prototype


```
function Dog(n, a) {
    this.name = n;
    this.age = a;
};

Dog.prototype = {
    bark: function(sound) {
      return this.name + "says" + sound;
    }
};
// set prototype to new anonymous object
```

Idiom: Methods in Prototype


```
function Dog(n, a) {
    this.name = n;
    this.age = a;
};

Dog.prototype.bark = function(sound) {
    return this.name + "says" + sound;
};


// better: extend existing prototype
```

Idiom: Methods in Prototype

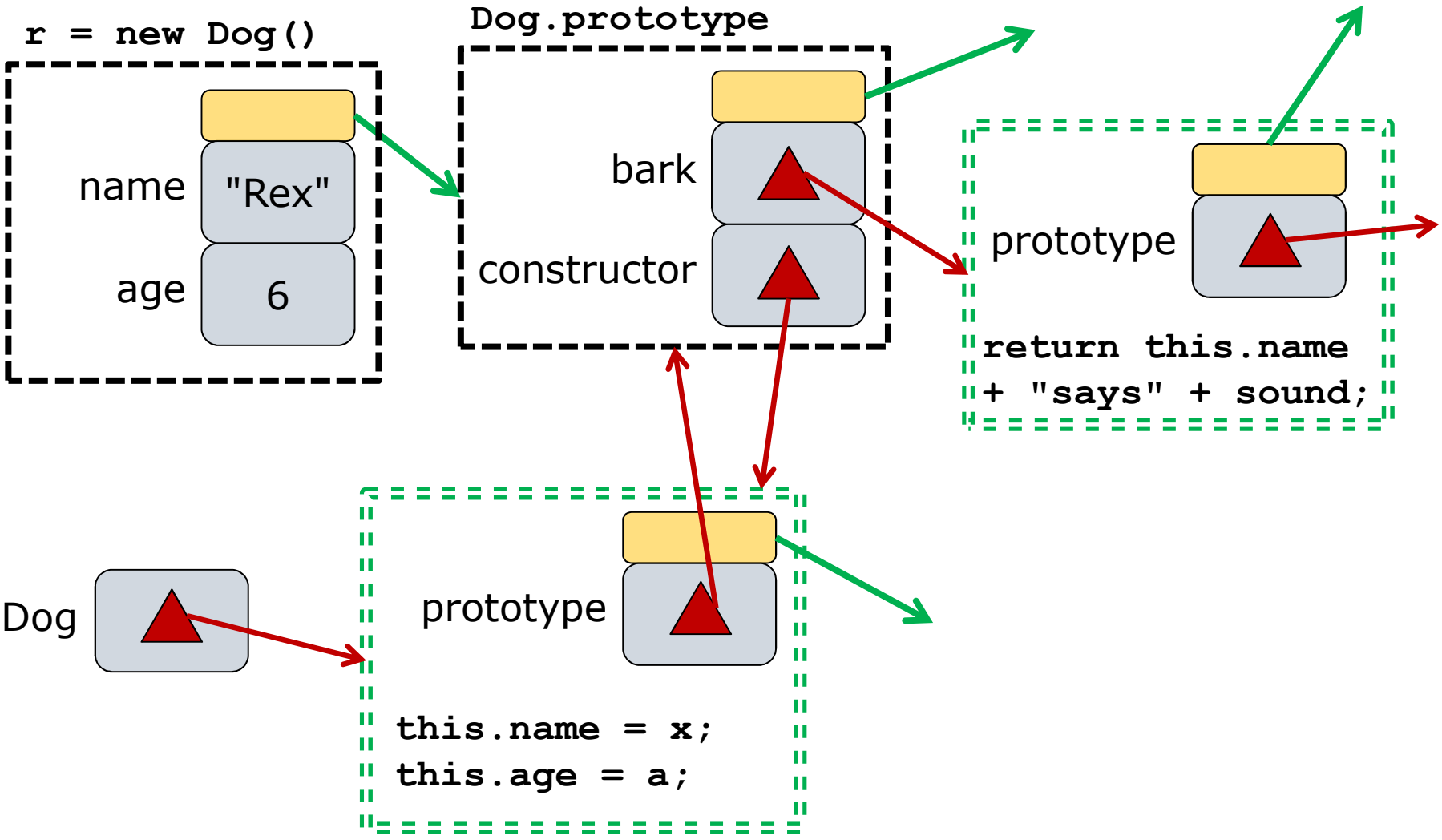

```
class Dog {
  constructor(n, a) {
    this.name = n;
    this.age = a;
  }

  bark(sound) {
    return this.name + "says" + sound;
  }

}

// best: ES6 classes (syntactic sugar)
```

Methods in Prototype



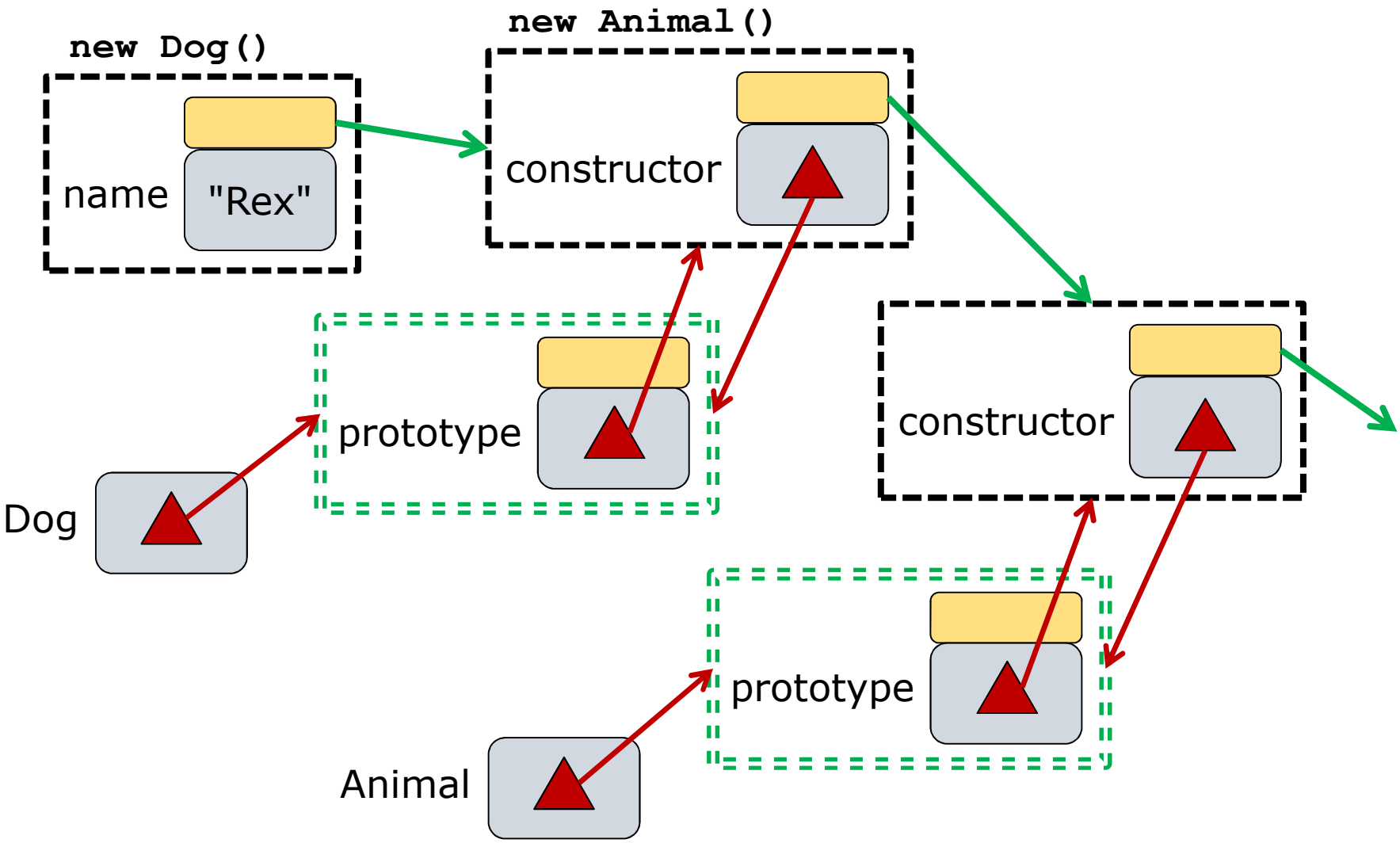
Idiom: Classical Inheritance


```
function Animal() { ... };
function Dog() { ... };

Dog.prototype = new Animal();
  // create prototype for future dogs

Dog.prototype.constructor = Dog;
  // set prototype's constructor
  // properly (ie should point to Dog())
```

Setting up Prototype Chains



Summary

- Objects as associative arrays
  - Partial maps from *keys* to *values*
  - Can dynamically add/remove properties
  - Can iterate over properties
- Method = function-valued property
  - Keyword this for distinguished parameter
- Constructor = any function
- Prototypes are "parent" objects
  - Delegation up the chain of prototypes
  - Prototype is determined by constructor
  - Prototypes can be modified