JavaScript: Objects, Methods, Prototypes

Lecture 25

What is an Object?

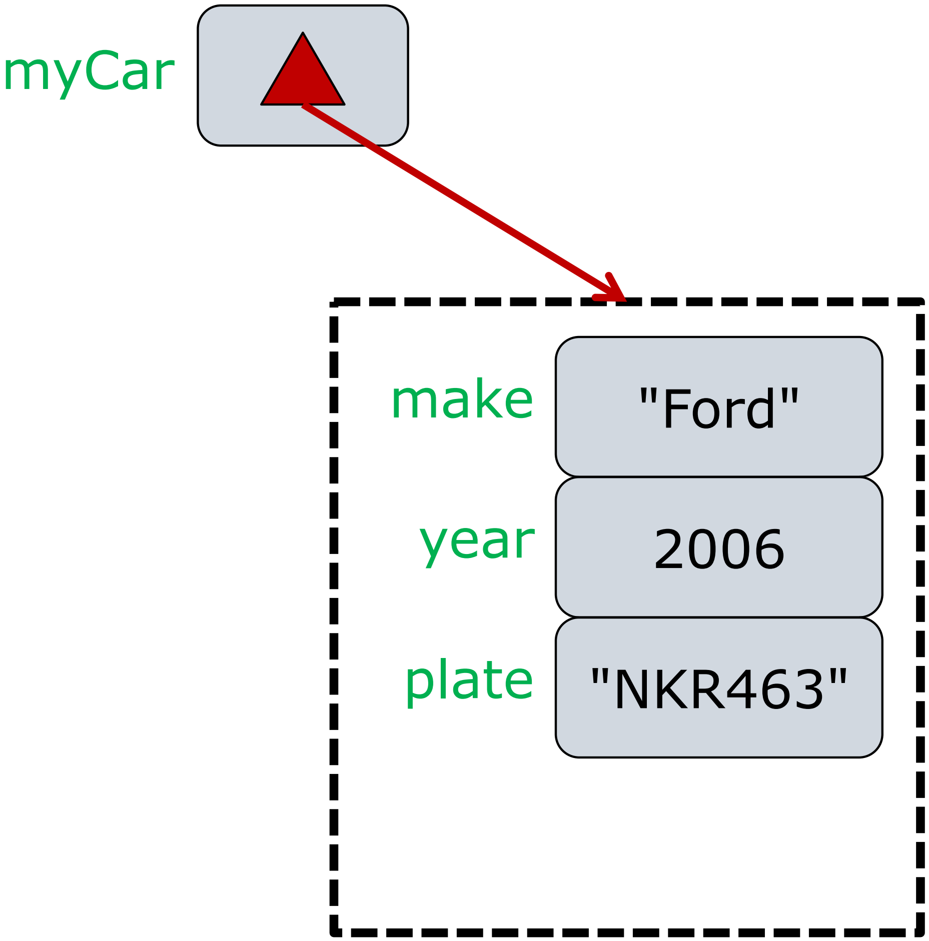
- *Property*: a key/value pair
  - aka name/value pair
- *Object*: a partial map of properties
  - Keys must be unique
- Creating an object, literal notation

```
let myCar = { make: "Acura",
              year: 1996,
              plate: "NKR463" };
```

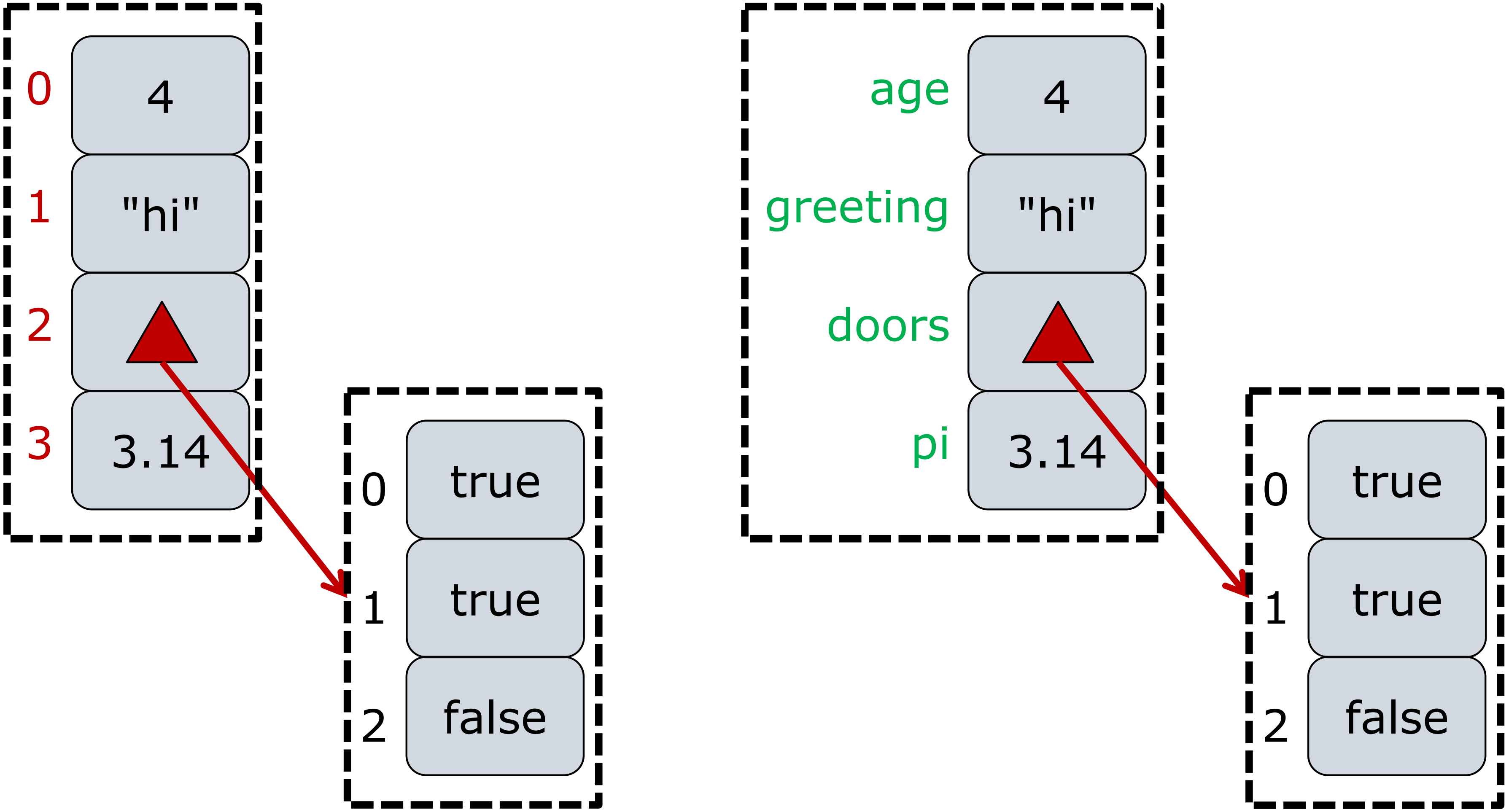
- To access/modify an object's properties:

```
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

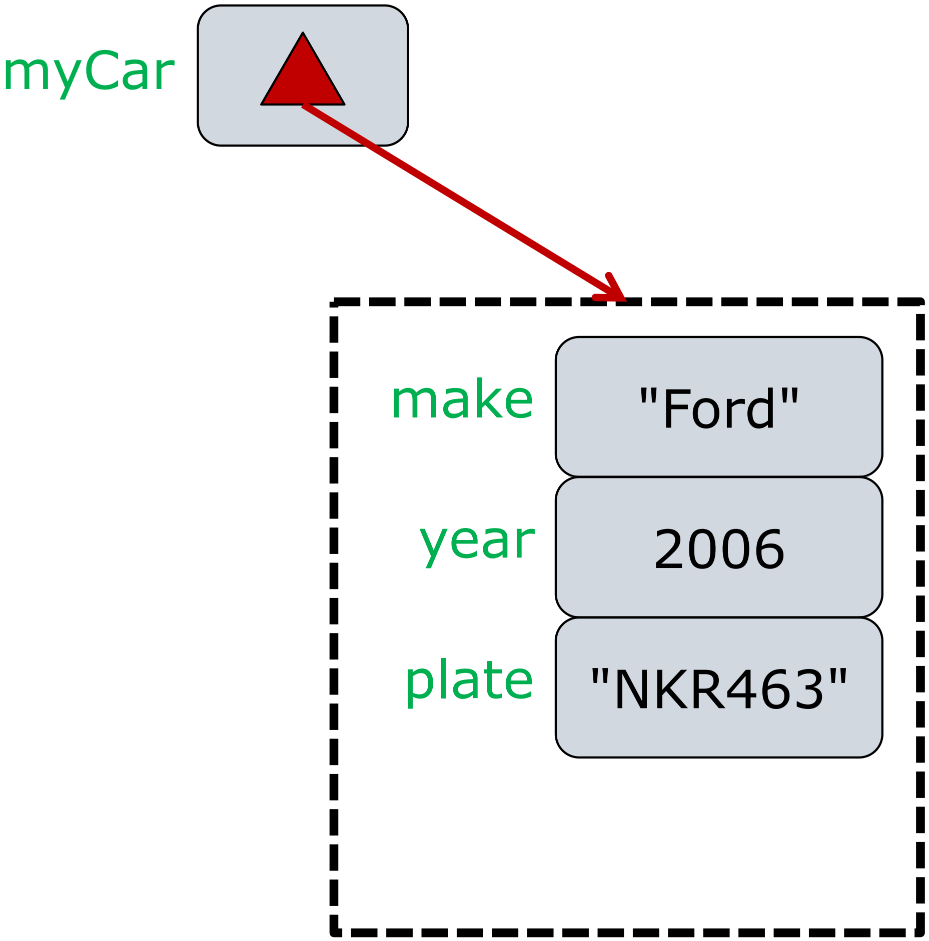
- Objects can grow

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

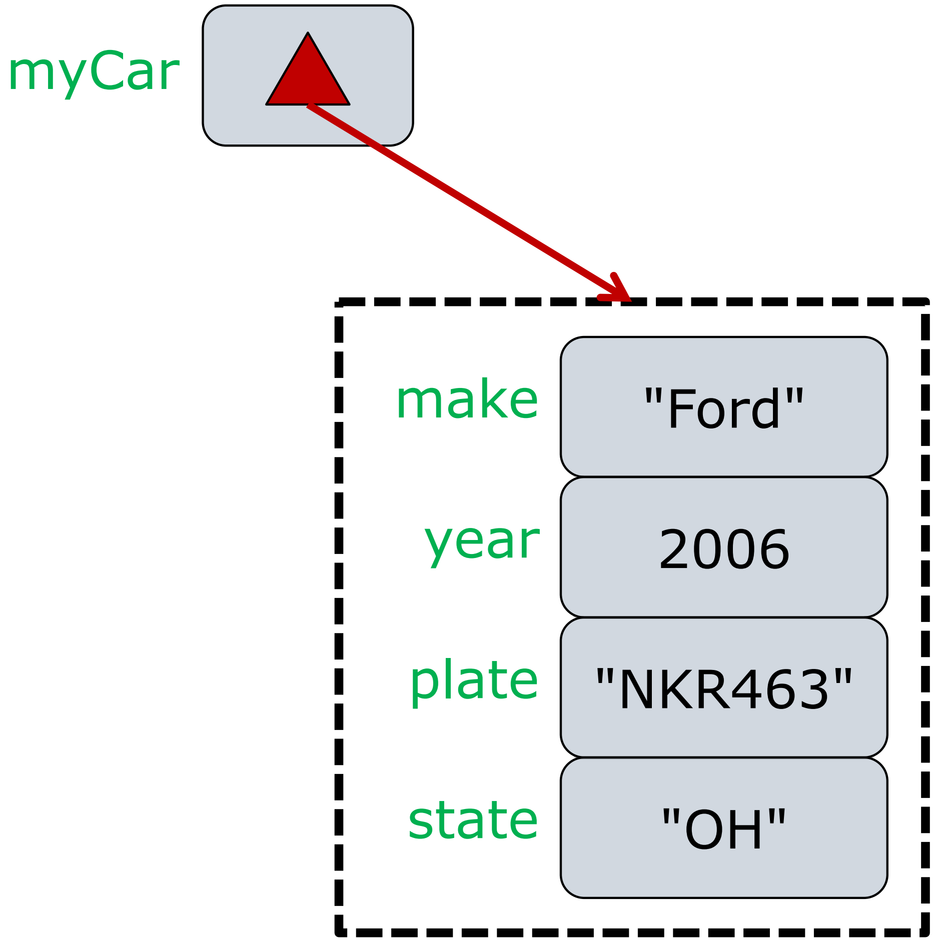
- Objects can shrink

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

Object Properties

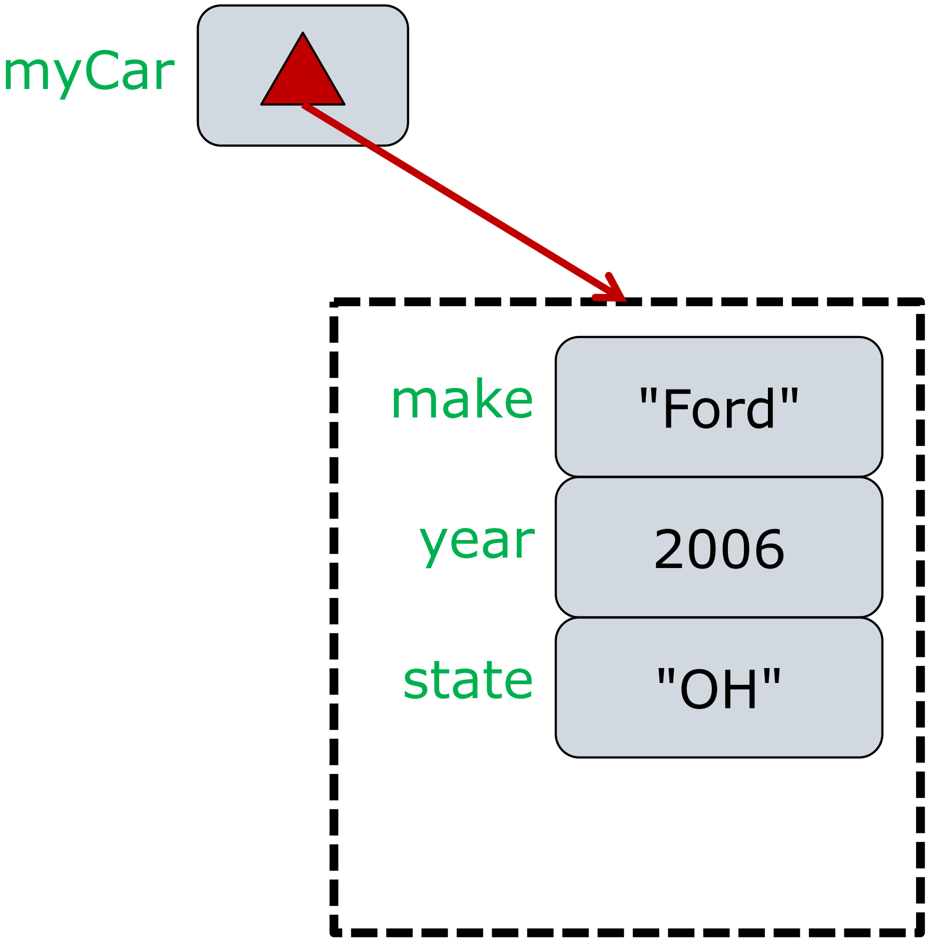


Object Properties



```
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 over keys with *for…in* syntax

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

- Notice `[]` to access each property

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

- Loop over *iterable* (eg array) with *for…of*

```
for (let elt of roster) {
  document.write(`name: ${elt}`);
}
```

Destructuring Assignment

- Objects can have many properties, and many levels of nesting

```
const result = someGiantObject();
// only care about 2 of result's properties
report(result.car);
combine(result.car, result.bus);
```

- Alternative: destructuring assignment

```
let {car, bus} = someGiantObject();
report(car);
combine(car, bus);
let {car: c, bus: b} = someGiantObject();
combine(c, b);
```

  - Eliminates unneeded variable **result**
  - Simplifies access to properties of interest

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;
}
```

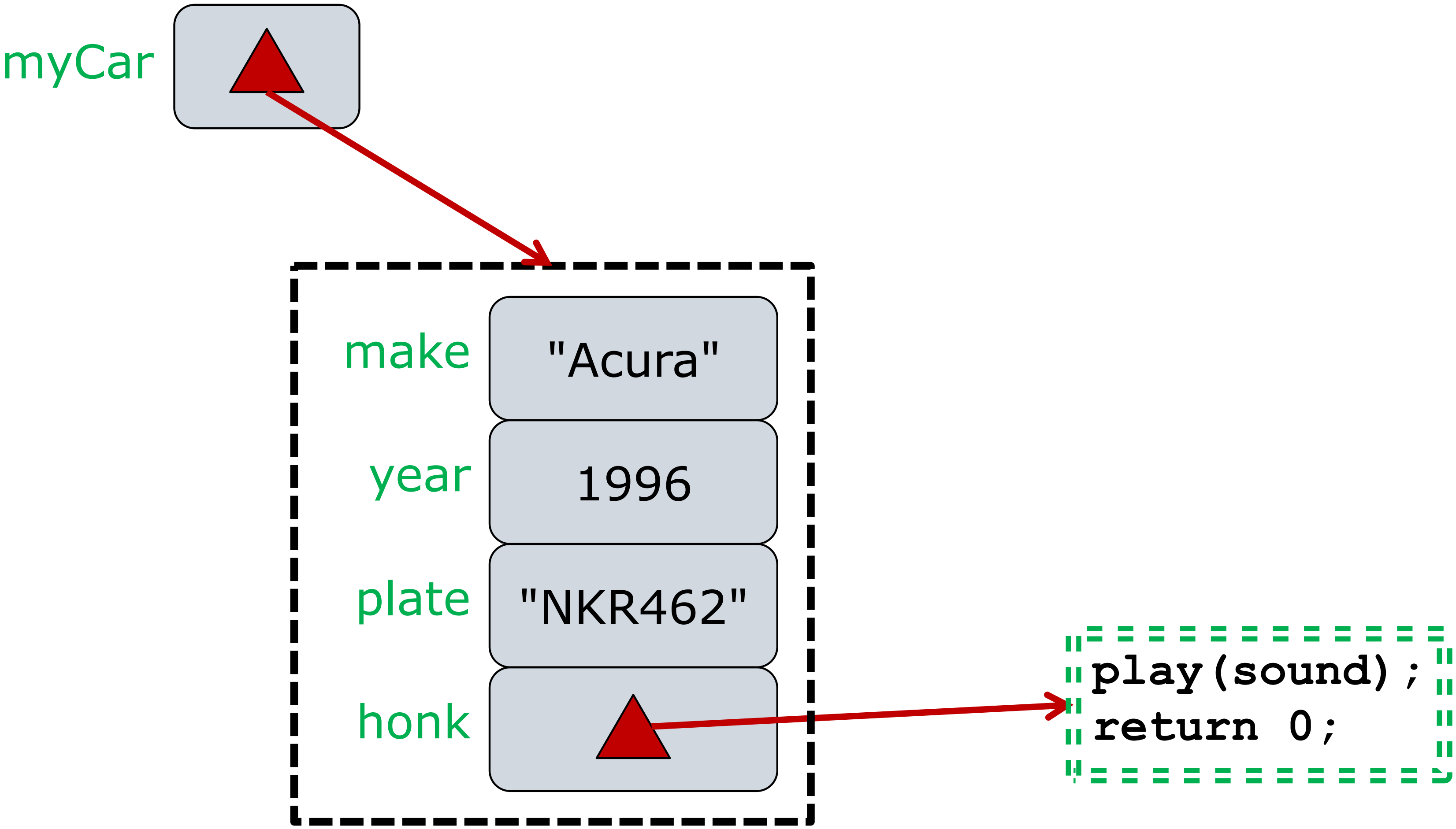
Example: Method

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

Example: Method (with Sugar)

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

Object Properties



```
play(sound);
return 0;
```

Keyword “this” in Functions

- Recall *distinguished formal parameter*

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

- Inside a function, keyword “this”

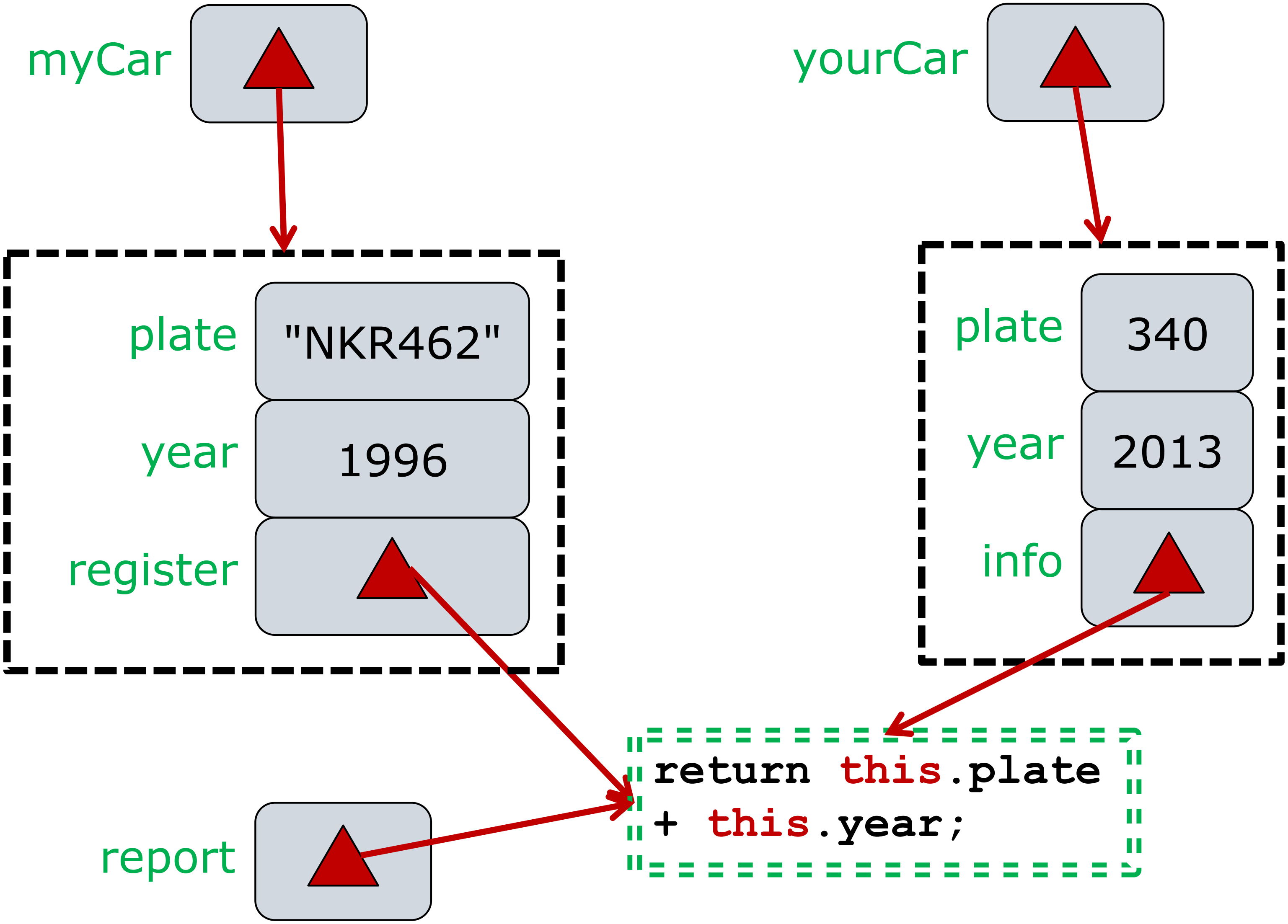
```
    function report() {
      return this.plate + this.year;
    }
```

- At run-time, “this” is the *distinguished argument* of the invocation

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

- Note: arrow functions work differently!
  - Do not have their own this, use enclosing lexical scope

Object Properties



```
return this.plate
+ this.year;
```

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");
```

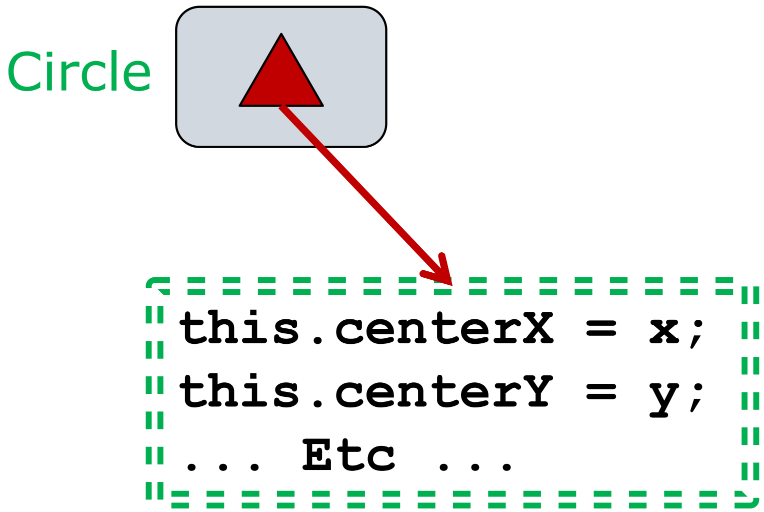
- Naming convention: Functions intended to be constructors are capitalized

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

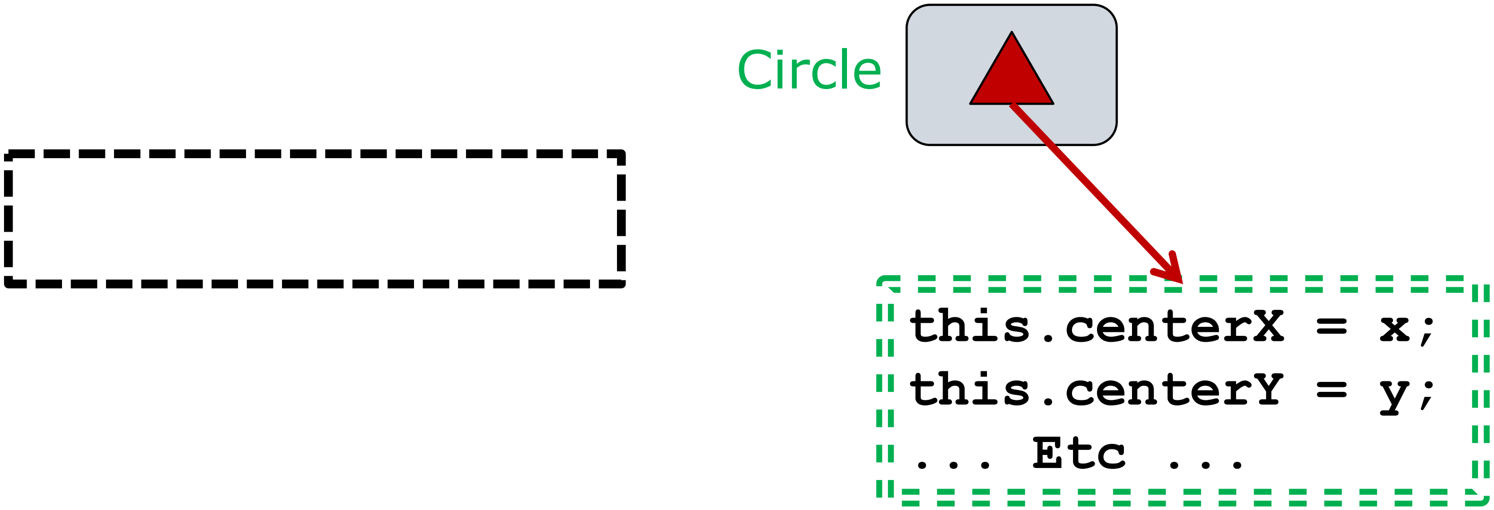
```
let c = new Circle(10, 12, 2.45);
```



```
this.centerX = x;
this.centerY = y;
... Etc ...
```

Creating a Circle Object

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

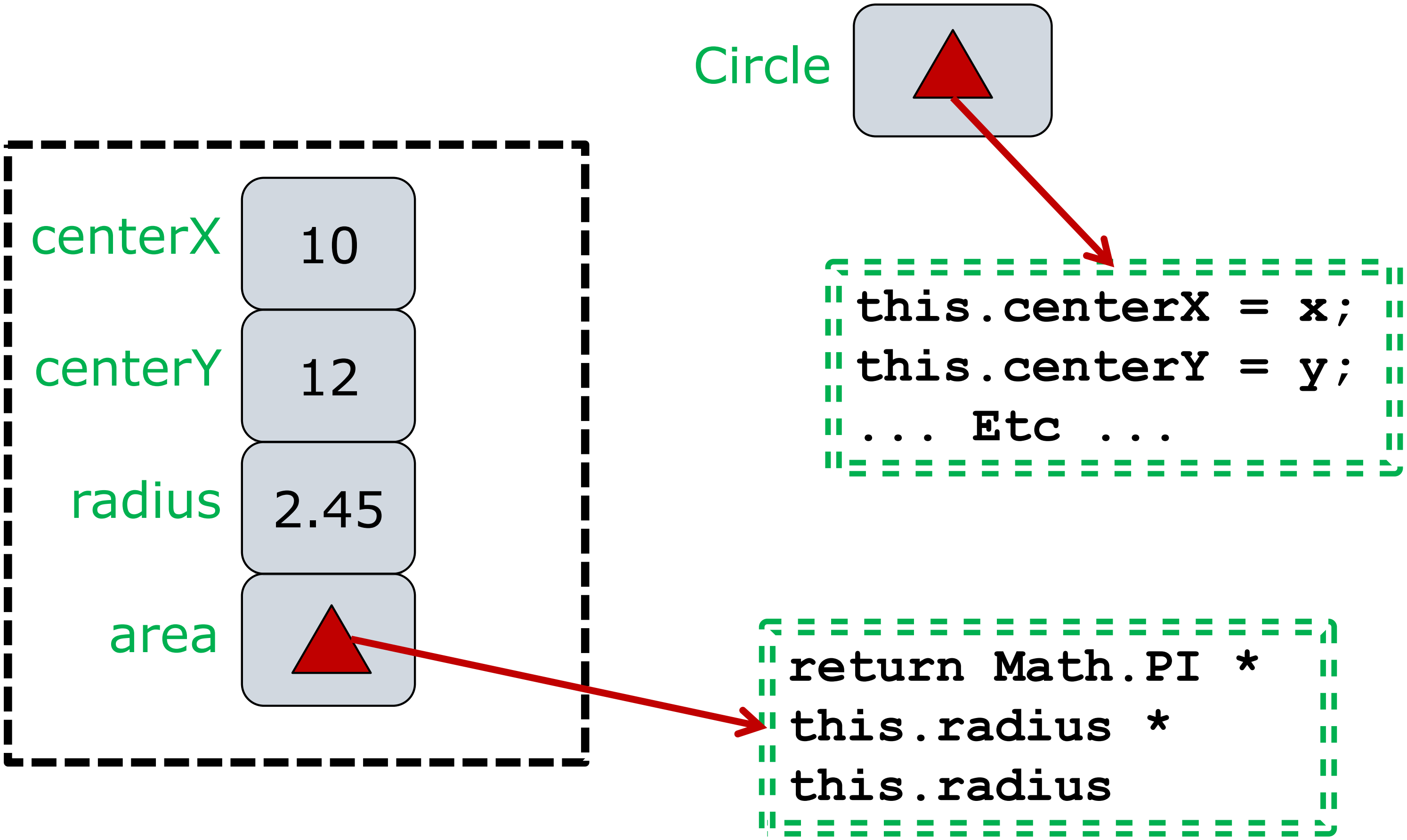


Circle

```
this.centerX = x;
this.centerY = y;
... Etc ...
```

Creating a Circle Object

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

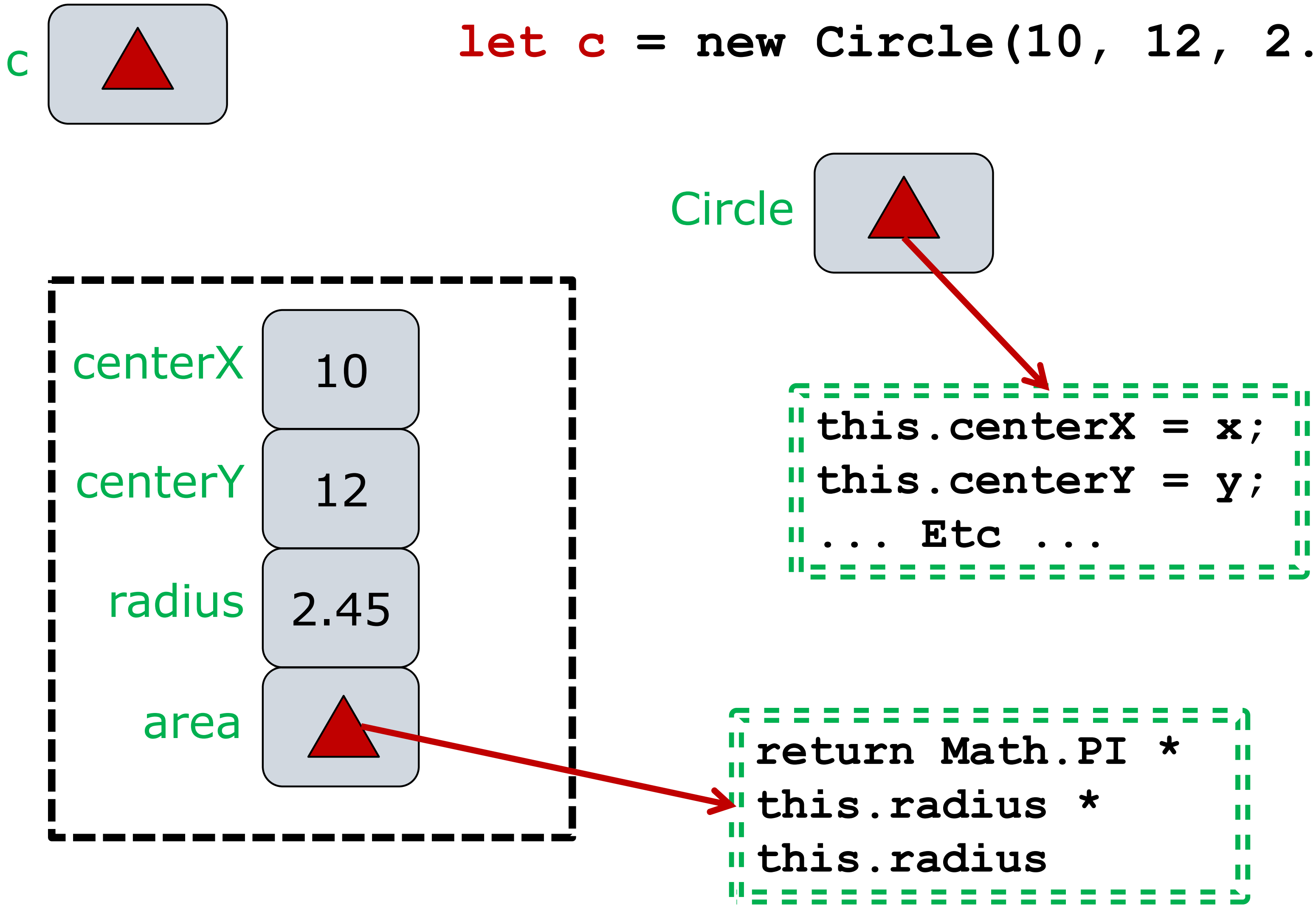


```
this.centerX = x;
this.centerY = y;
... Etc ...
```

```
return Math.PI *
this.radius *
this.radius
```

Creating a Circle Object

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

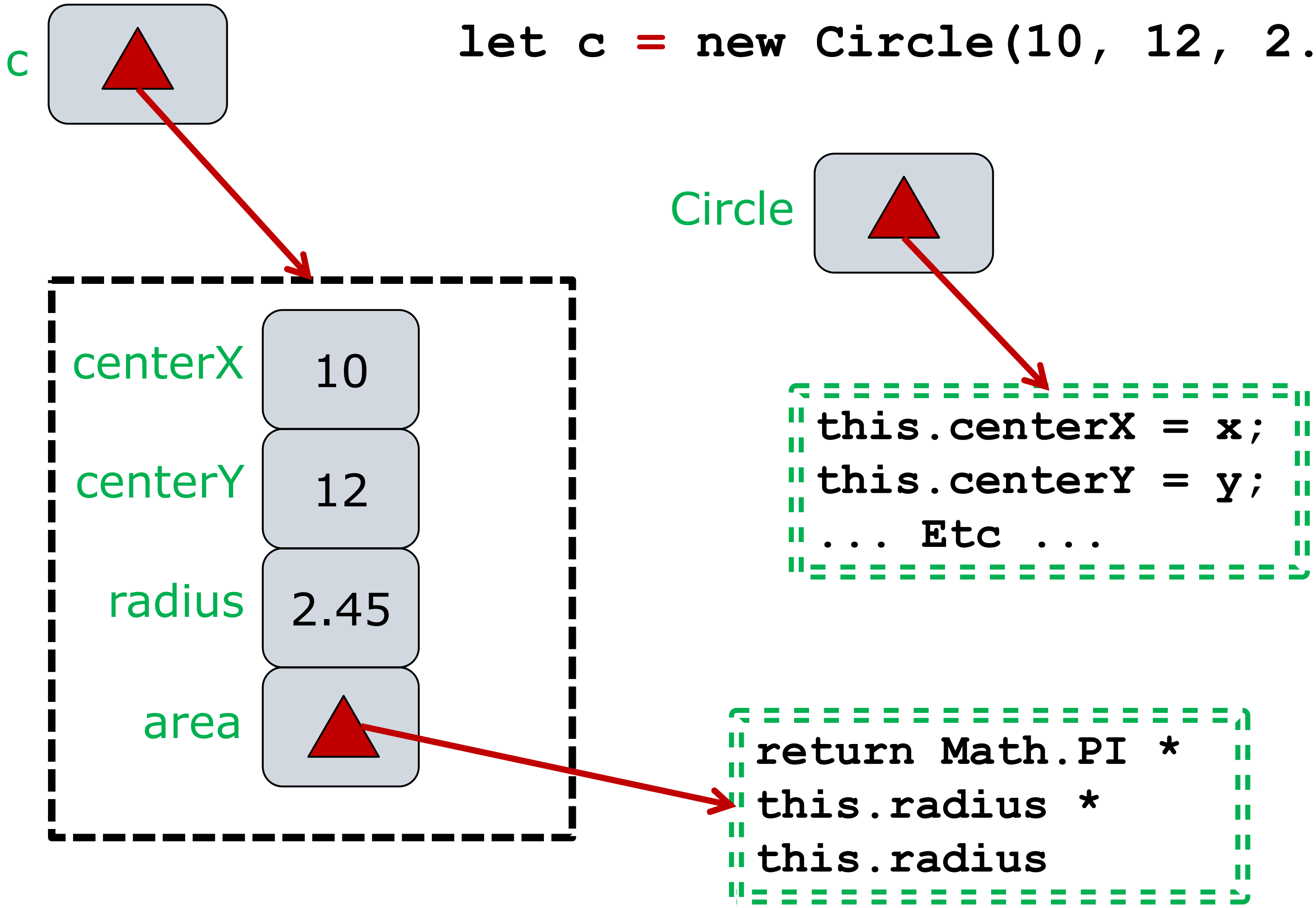


```
this.centerX = x;
this.centerY = y;
... Etc ...
```

```
return Math.PI *
this.radius *
this.radius
```

Creating a Circle Object

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

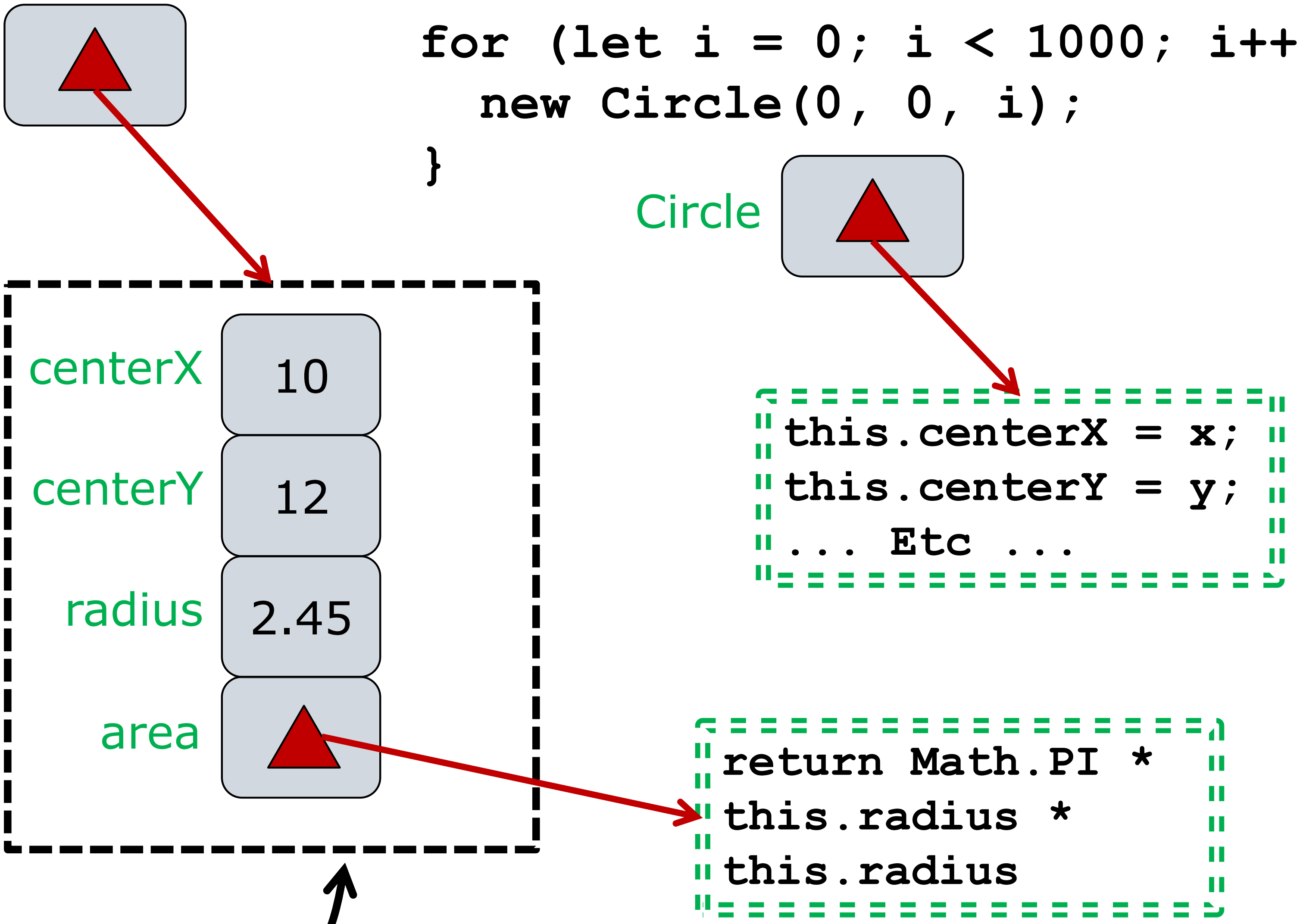


```
this.centerX = x;
this.centerY = y;
... Etc ...
```

```
return Math.PI *
this.radius *
this.radius
```

Creating Many Circle Objects

```
for (let i = 0; i < 1000; i++) {
  new Circle(0, 0, i);
}
```



```
this.centerX = x;
this.centerY = y;
... Etc ...
```

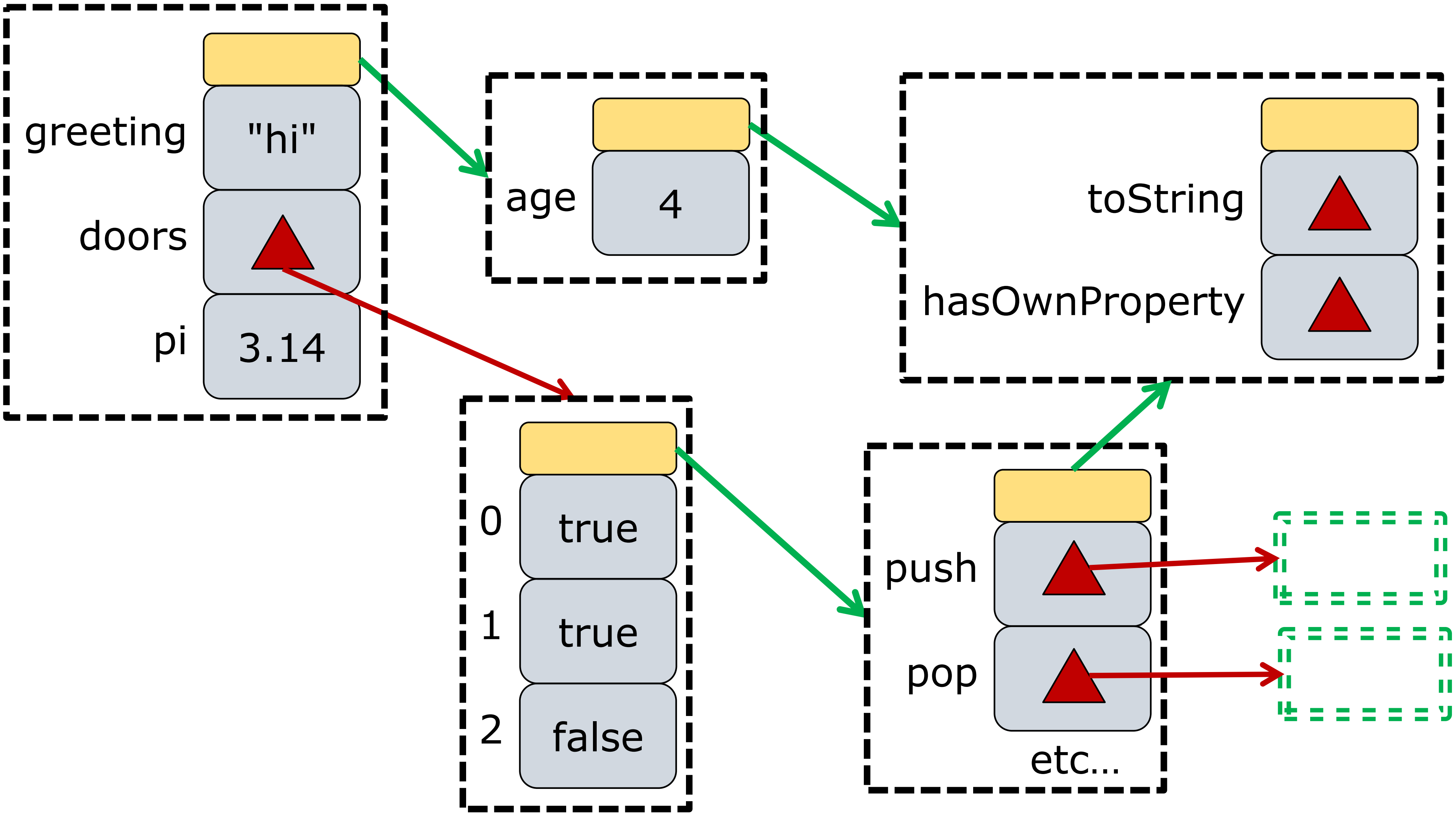
```
return Math.PI *
this.radius *
this.radius
```

How many of these?

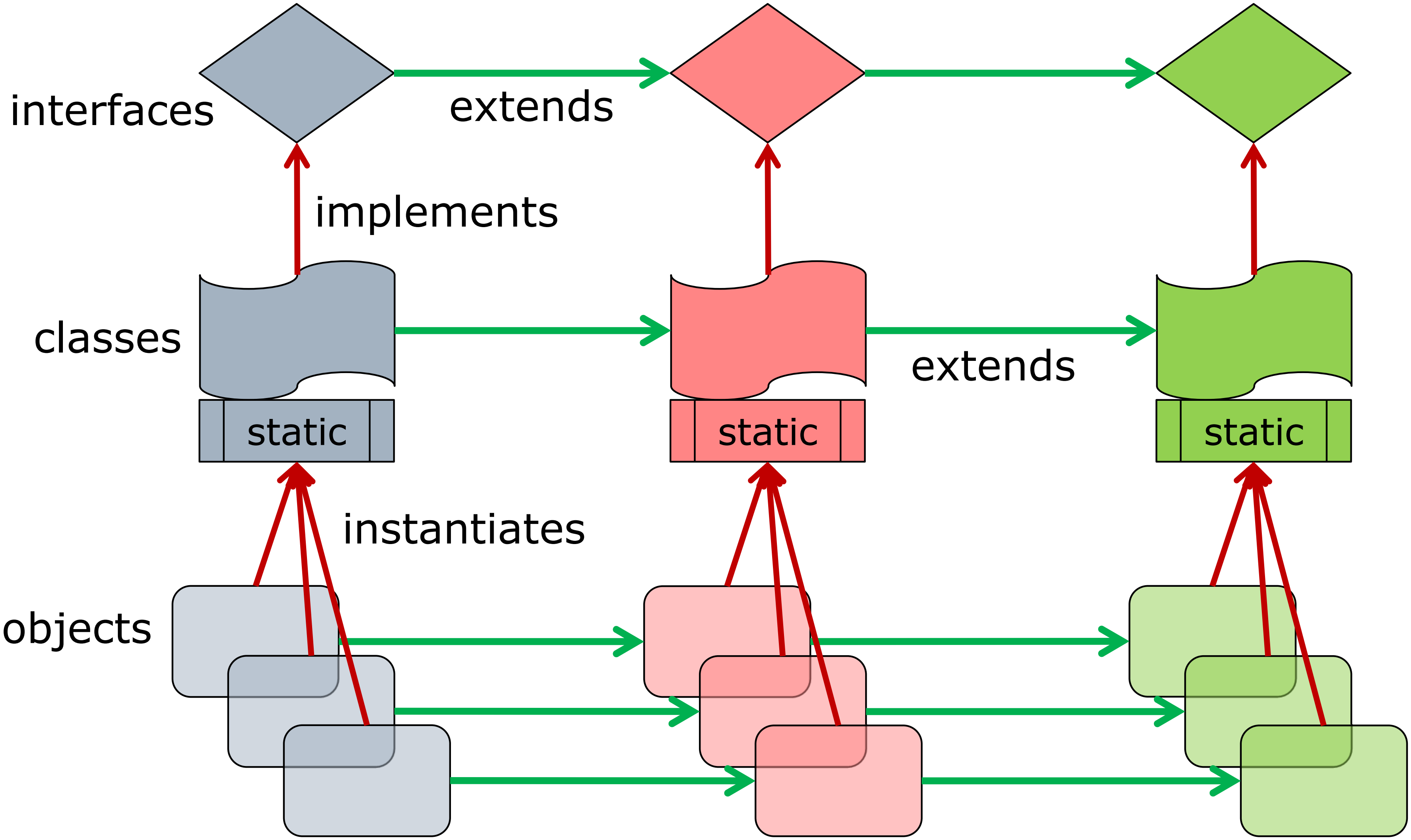
Prototypes

- 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



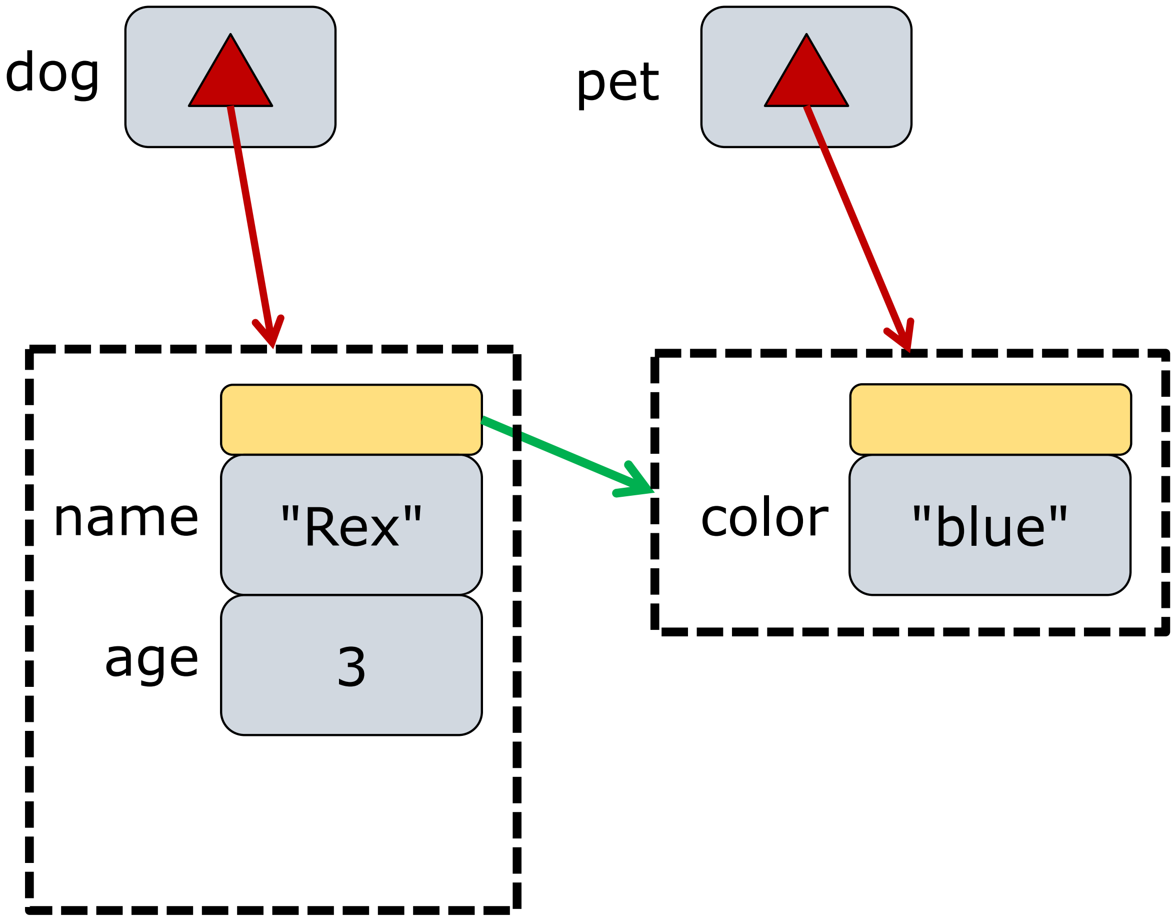
Prototype: Get vs Set of Proprty

- Consider two objects

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

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

Delegation to Prototype



Prototype: Get vs Set of Proprty

- Consider two objects

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

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

```
// dog.name == ?
// dog.color == ?
pet.color = "brown";
// dog.color is ?
dog.color = "green";
// dog.color is ?
// pet.color is ?
```

Prototype: Get vs Set of Proprty

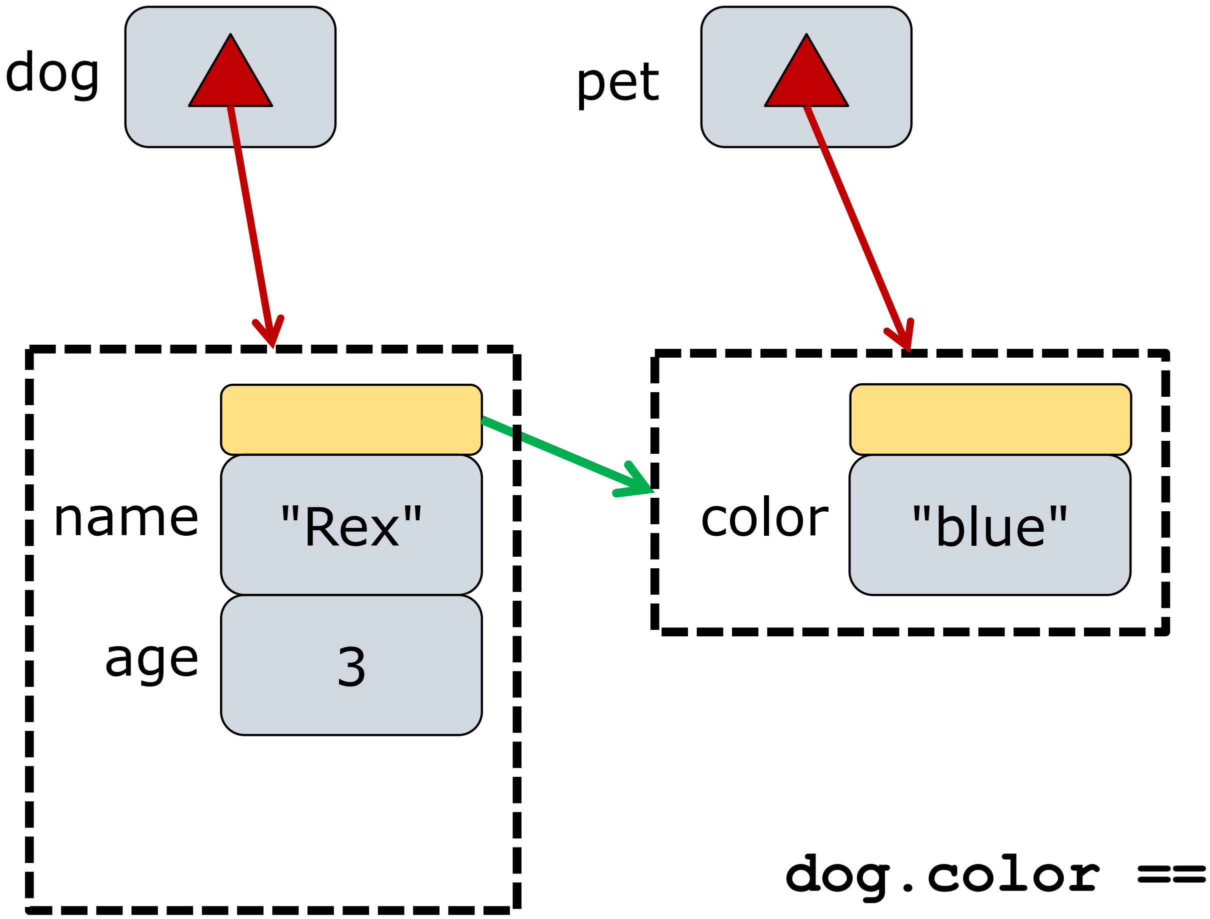
- □ Consider two objects

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

- □ Assume `pet` is `dog`'s prototype

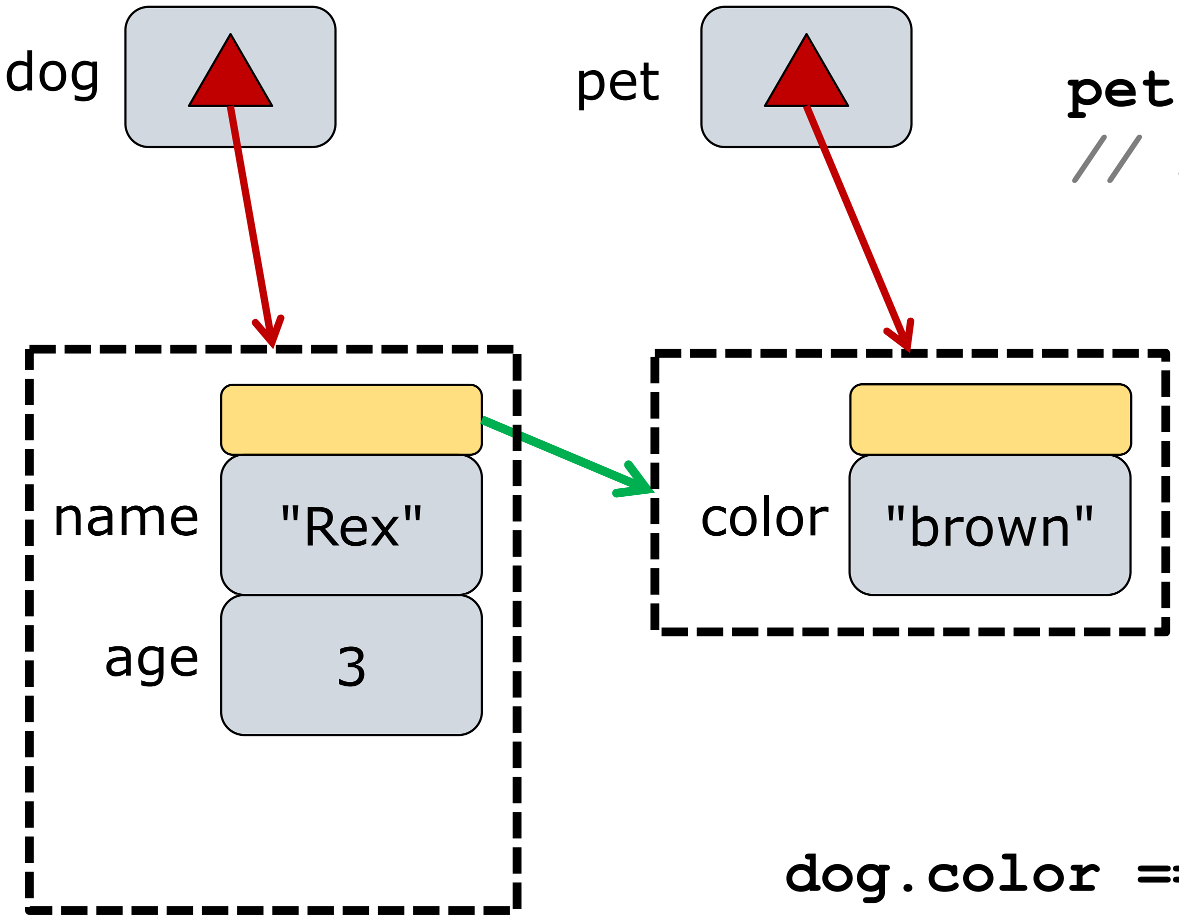
```
// dog.name == "Rex"
// dog.color == "blue" (follow chain)
pet.color = "brown"; // set in proto
// dog.color is "brown" (prop changed)
dog.color = "green"; // set in object
// dog.color is "green"
// pet.color is still "brown" (hiding)
```

Delegation to Prototype



```
dog.color == ?
// get follows prototype chain
```

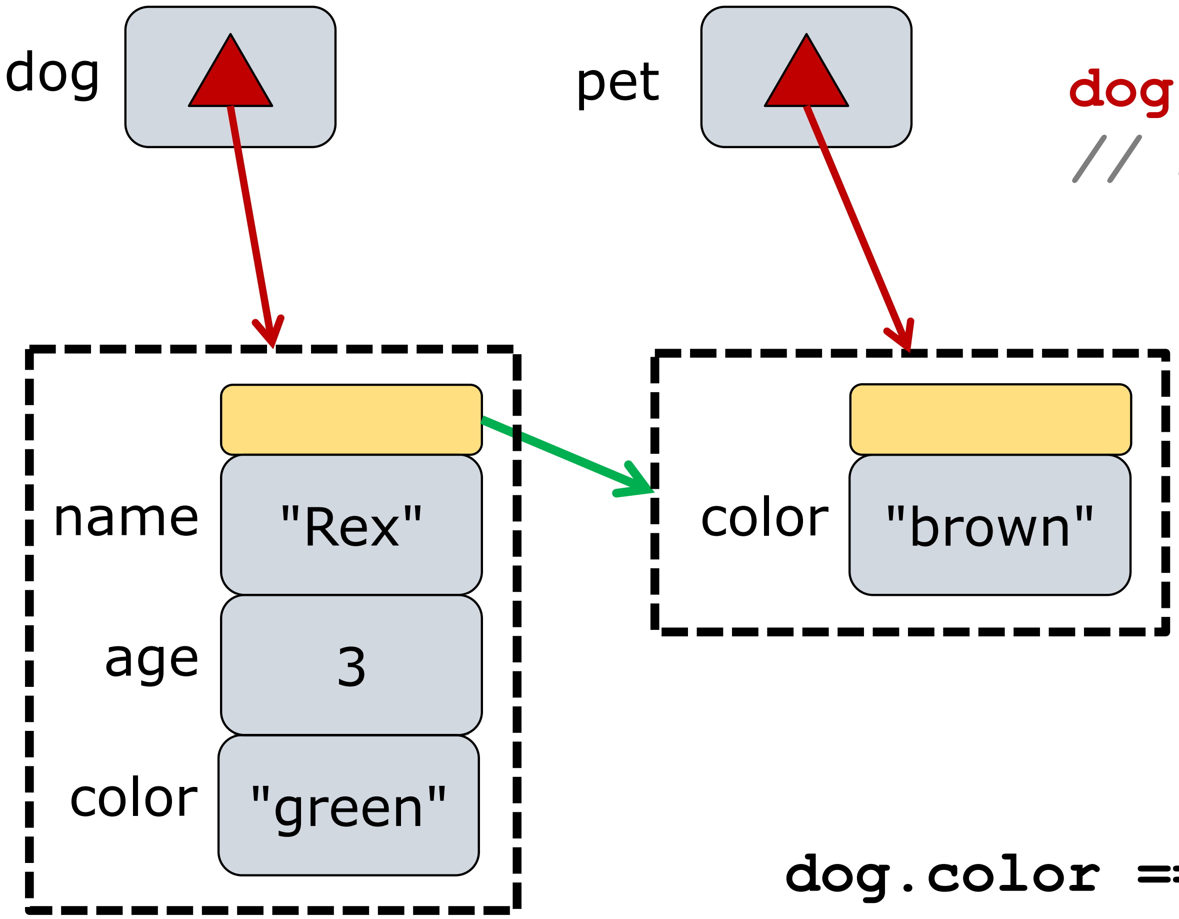
Delegation to Prototype



```
pet.color = "brown";
// set changes object
```

```
dog.color == ?
// get follows prototype chain
```

Delegation to Prototype



```
dog.color = "green";
// set changes object!
```

```
dog.color == ?
// get follows prototype chain
```

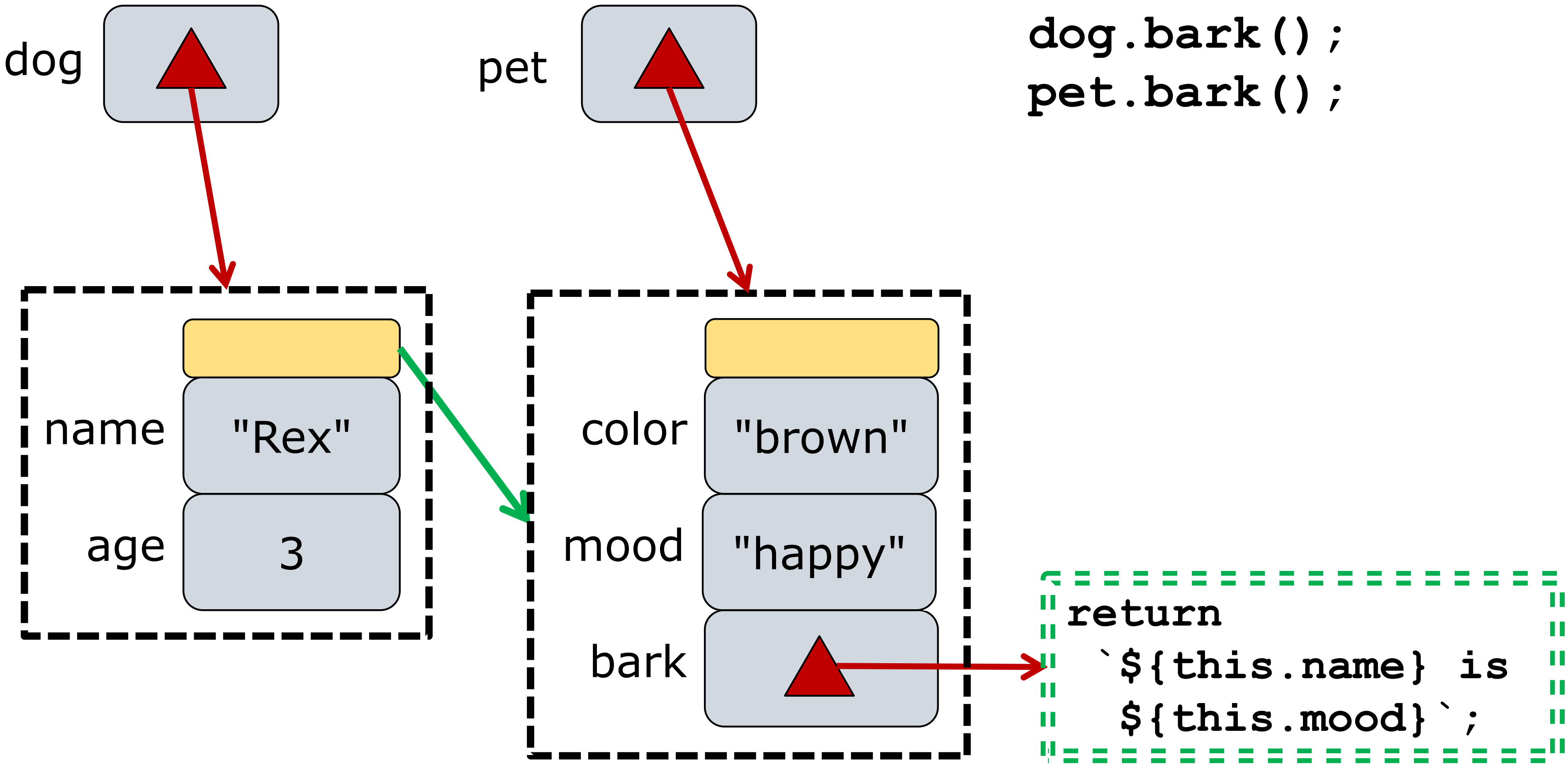
Prototypes Are Dynamic Too

- 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

```
dog.bark();
pet.bark();
```



```
return
  `${this.name} is
   ${this.mood}`;
```

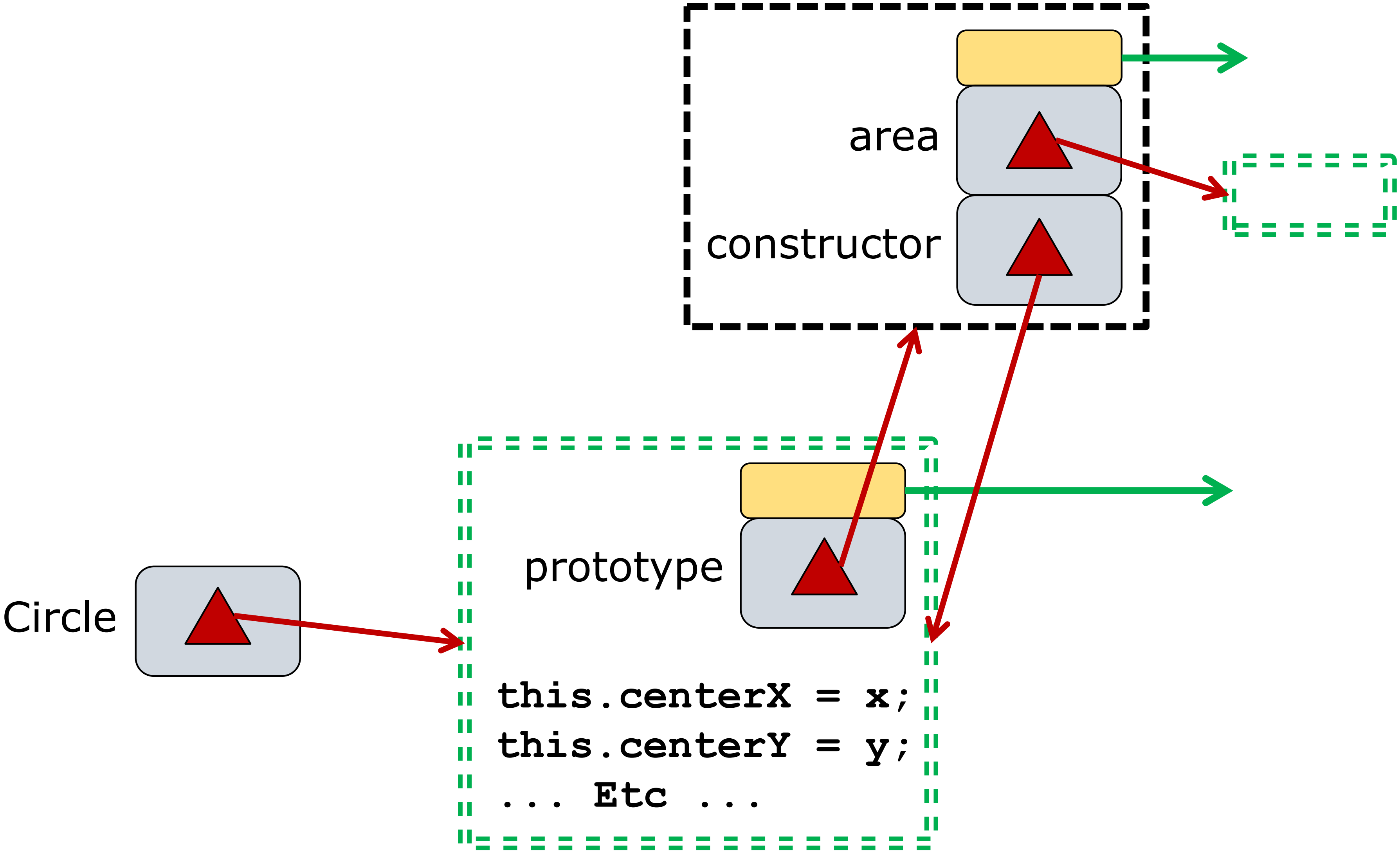
Connecting Objects & Prototypes

- 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

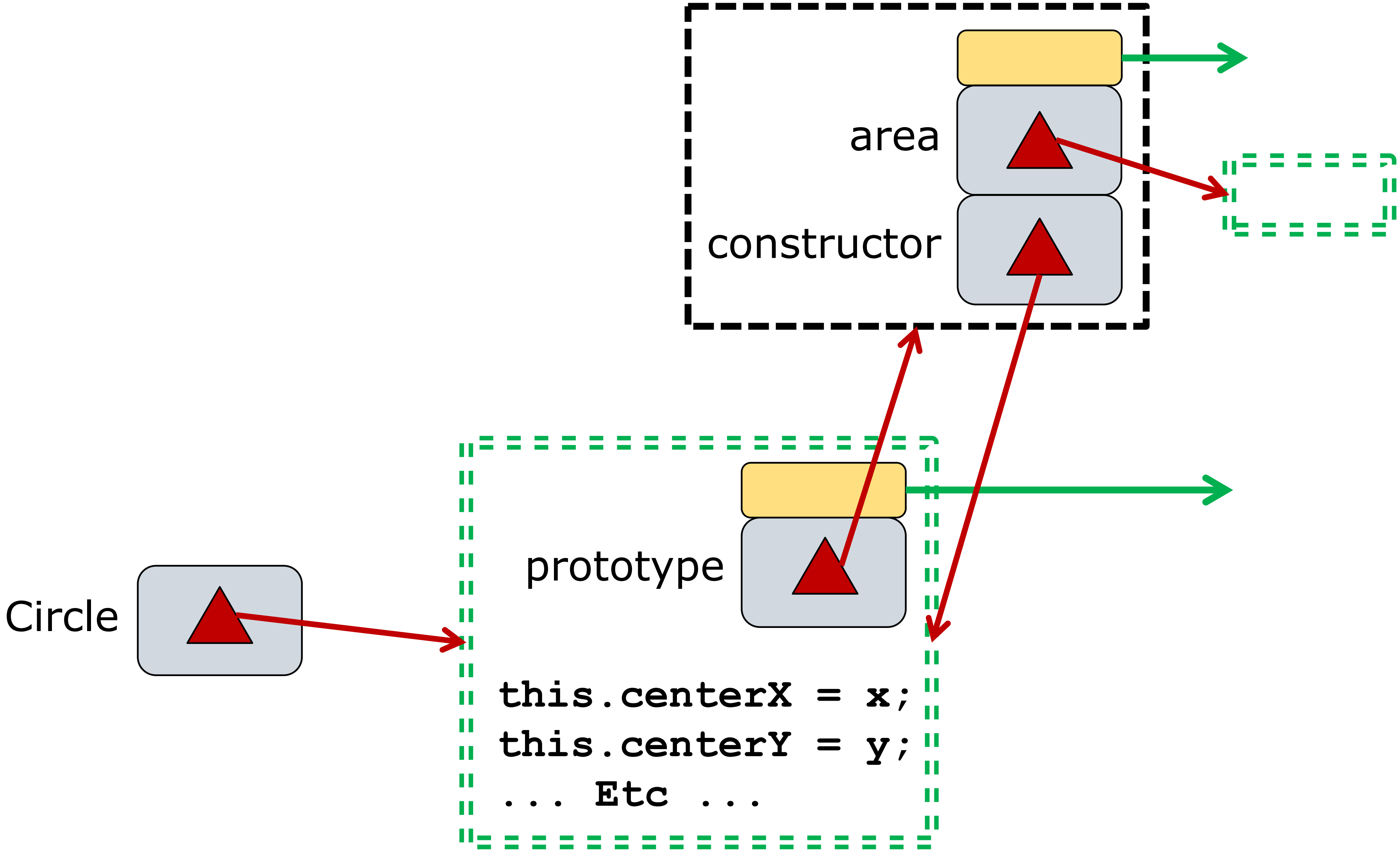
Prototypes And Constructors



```
this.centerX = x;
this.centerY = y;
... Etc ...
```

Prototypes And Constructors

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



area

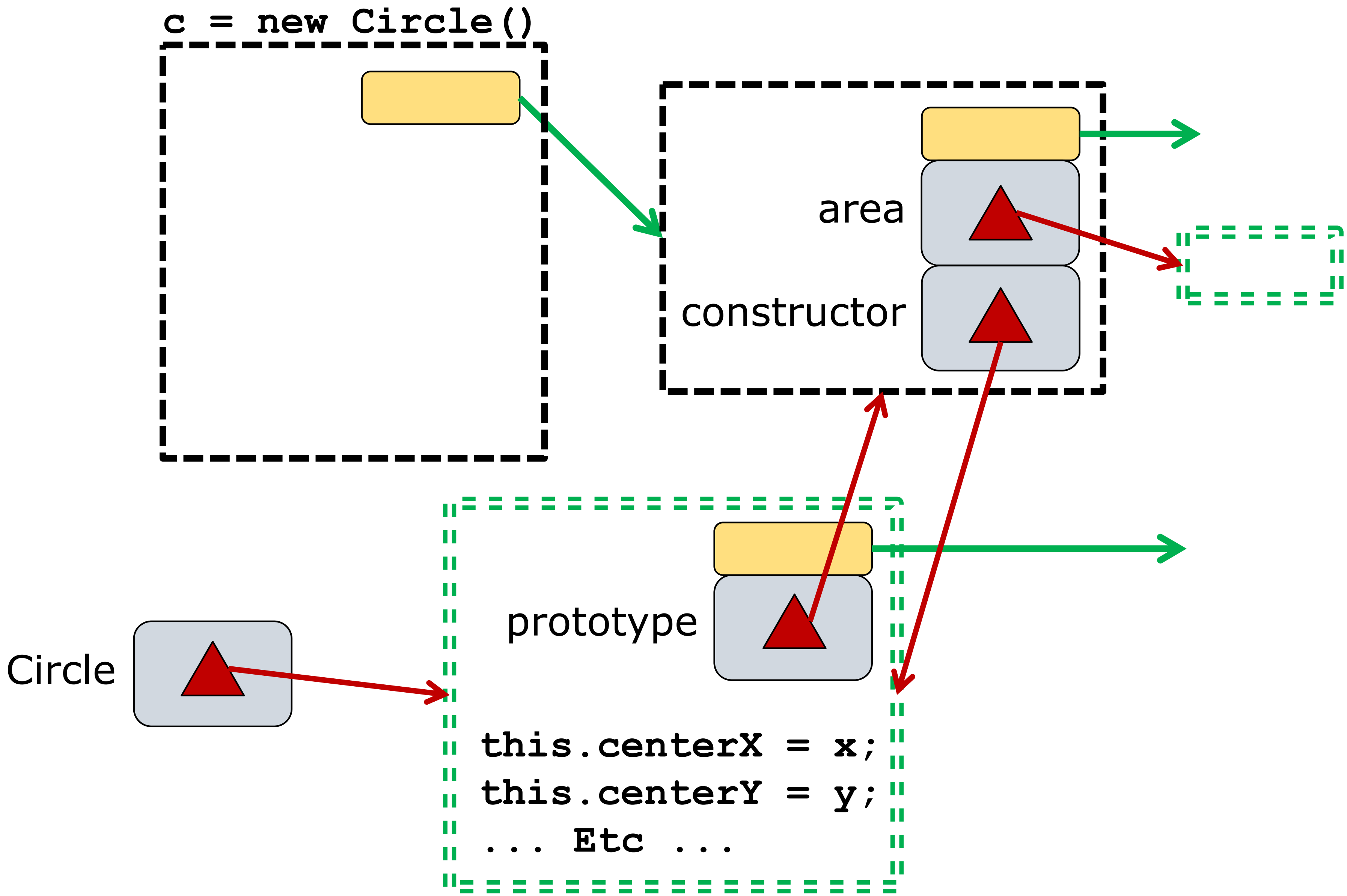
constructor

prototype

Circle

```
this.centerX = x;
this.centerY = y;
... Etc ...
```

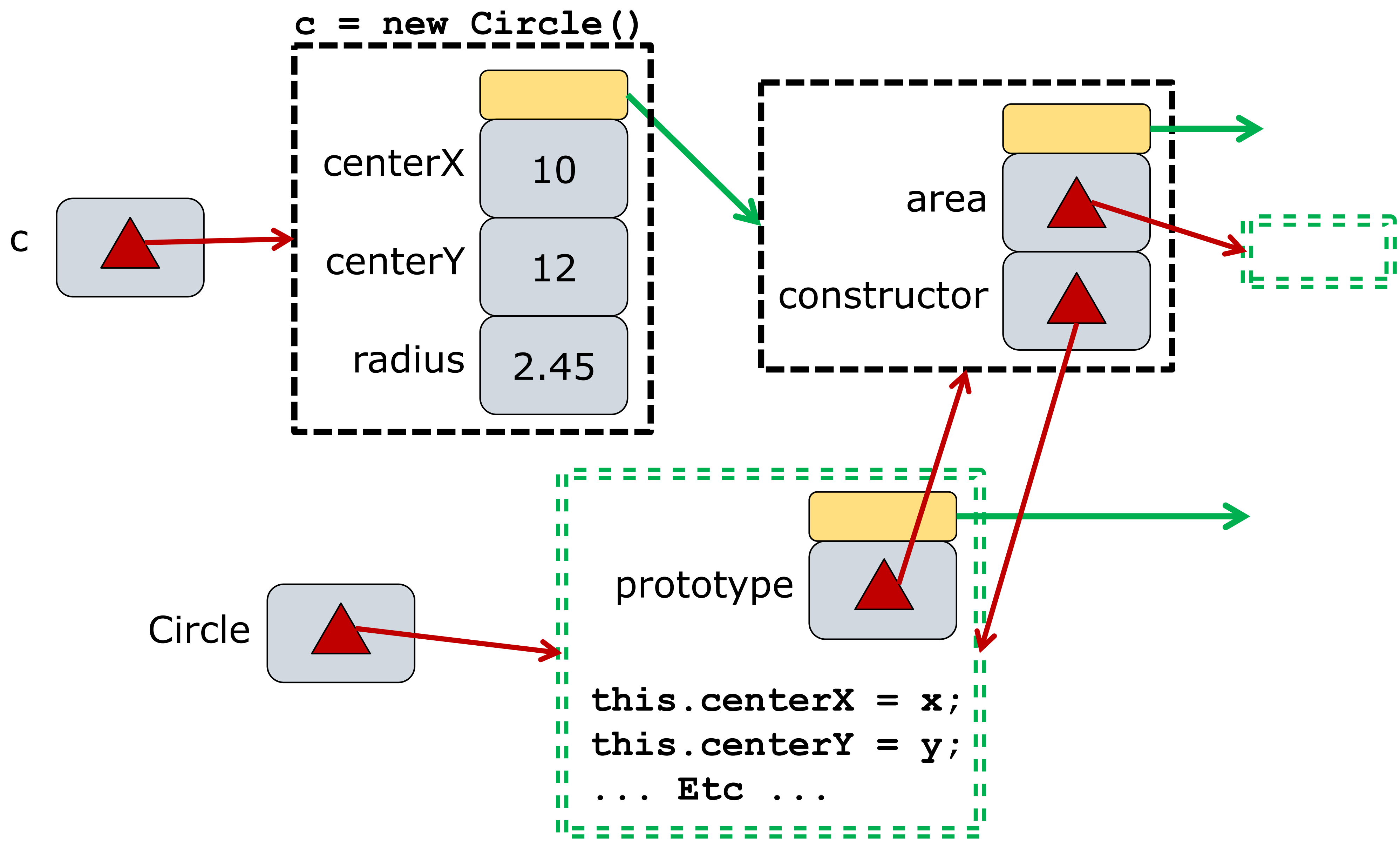
Prototypes And Constructors



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

```
this.centerX = x;
this.centerY = y;
... Etc ...
```

Prototypes And Constructors



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

```
this.centerX = x;
this.centerY = y;
... Etc ...
```

Idiom: Put Methods in Prototype

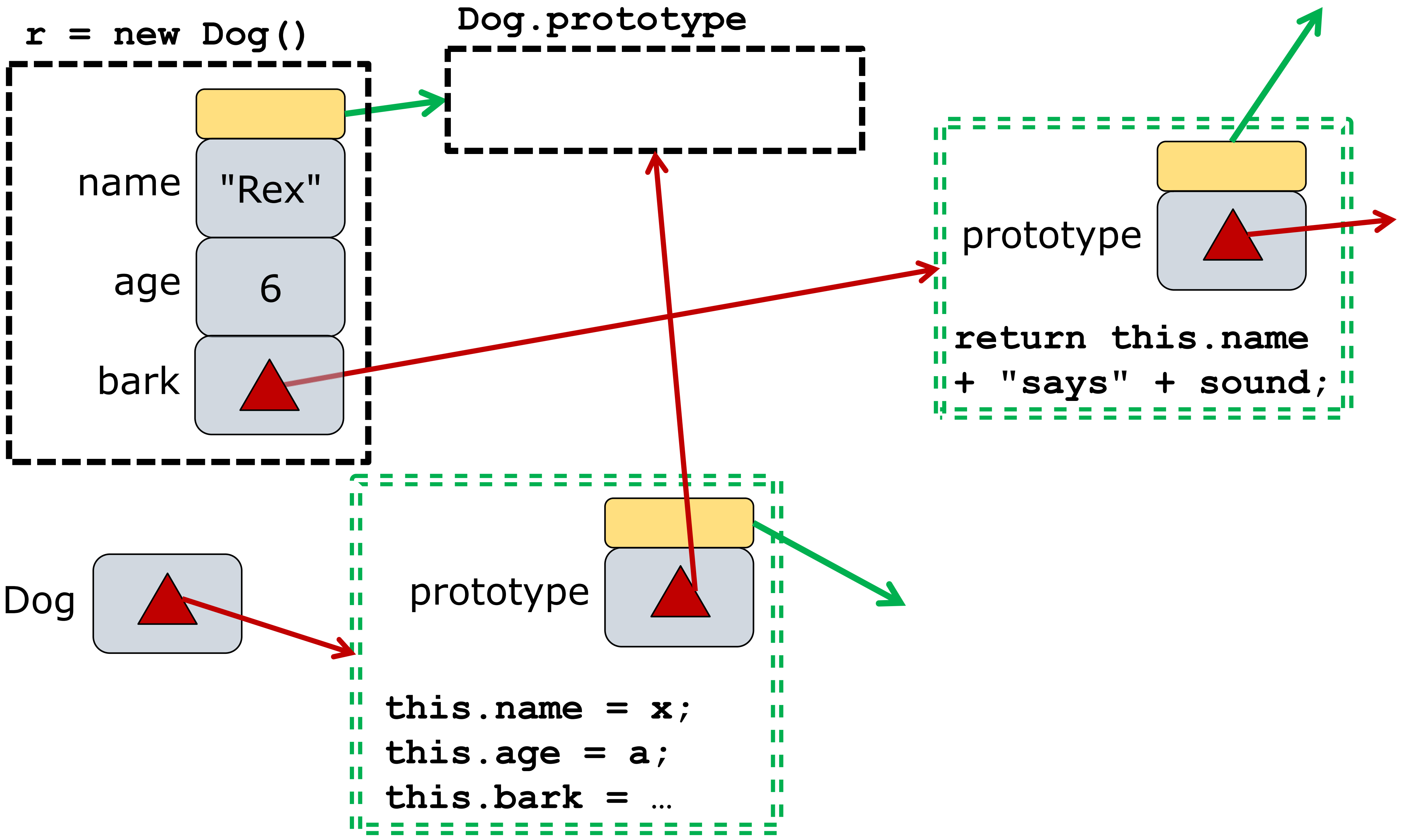
```
function Dog(n, a) {
  this.name = n;
  this.age = a;

  this.bark = function(sound) {
    return `${this.name} says ${sound}`;
  }
};



// bad: method is added to object itself
```

Method is in 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}`;
};


// good: add method to prototype
```

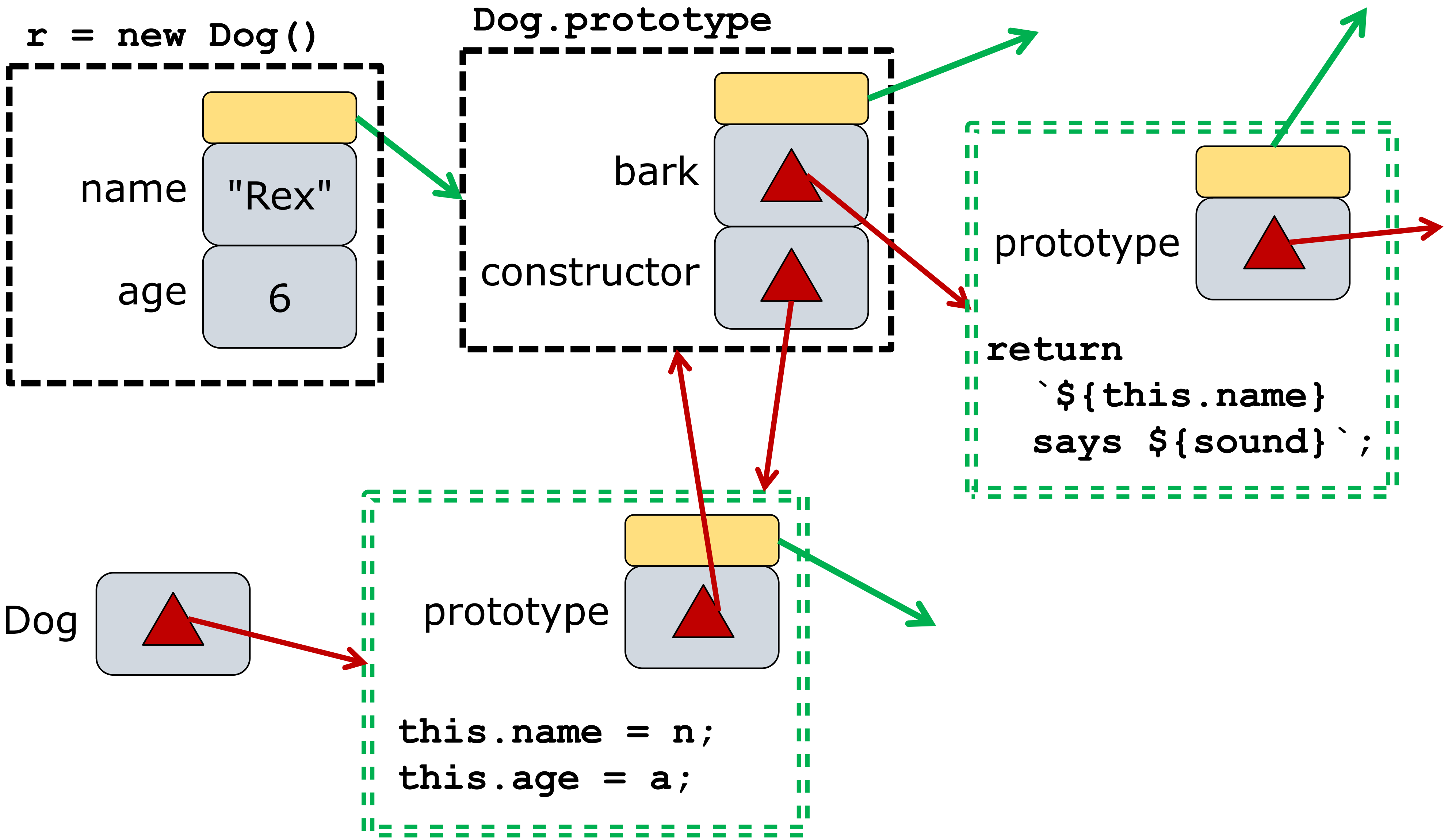
Idiom: Methods in Prototype

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

  bark(sound) {
    return `${this.name} says ${sound}`;
  }
}


// best: ES6 class syntax
```

Methods in Prototype



Class With Instance Fields

```
class Dog {
  name = "Fur"; // property of object
  age;

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

  bark(sound) {
    return `${this.name} says ${sound}`;
  }
}
```

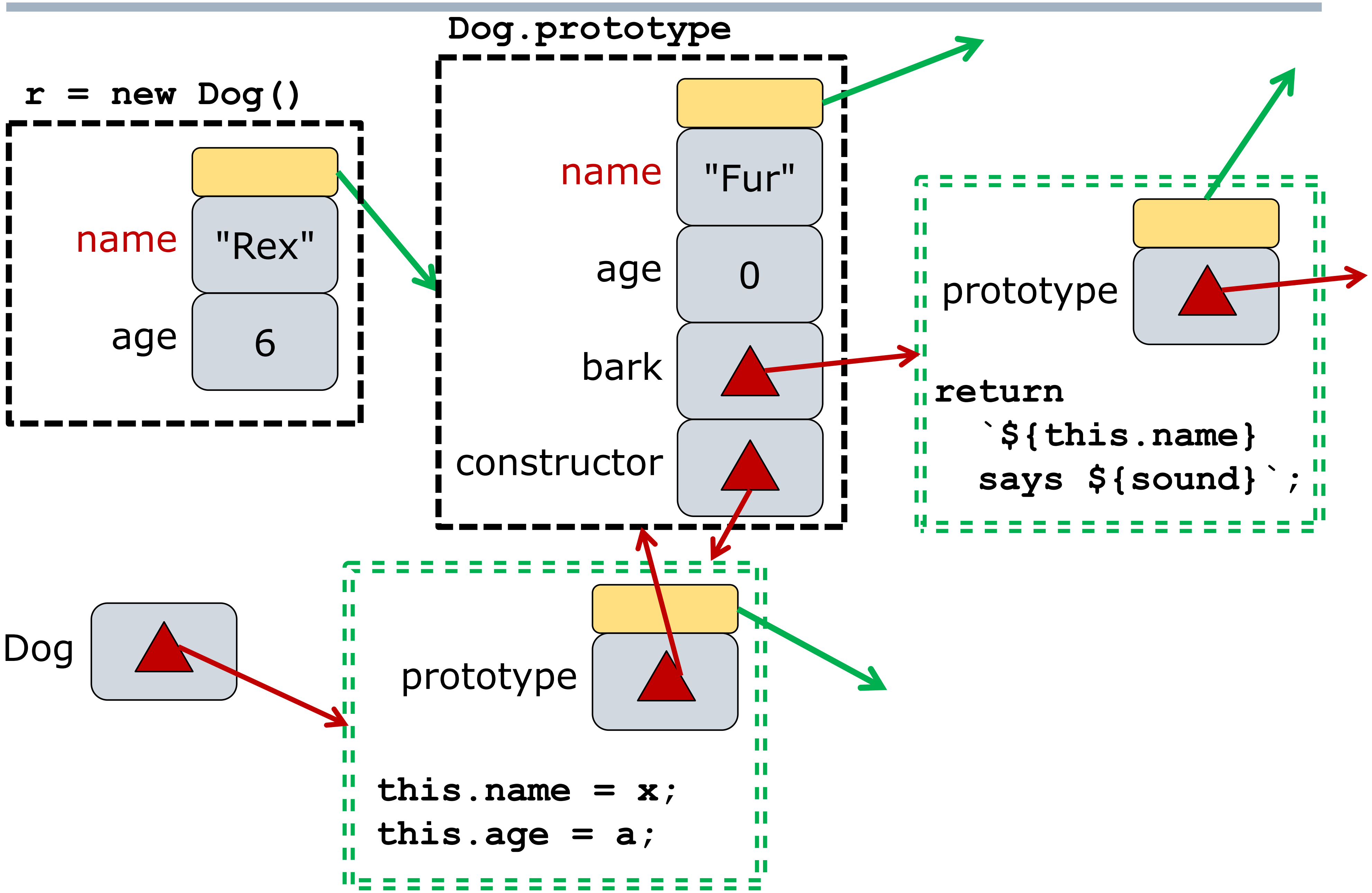
Careful: Class Properties

```
class Dog {
  name: "Fur"; // property is in prototype!
  age: 0;

  constructor(n, a) {
    this.name = n; // hides prototype property
    this.age = a;
  }

  bark(sound) {
    return `${this.name} says ${sound}`;
  }
}
```

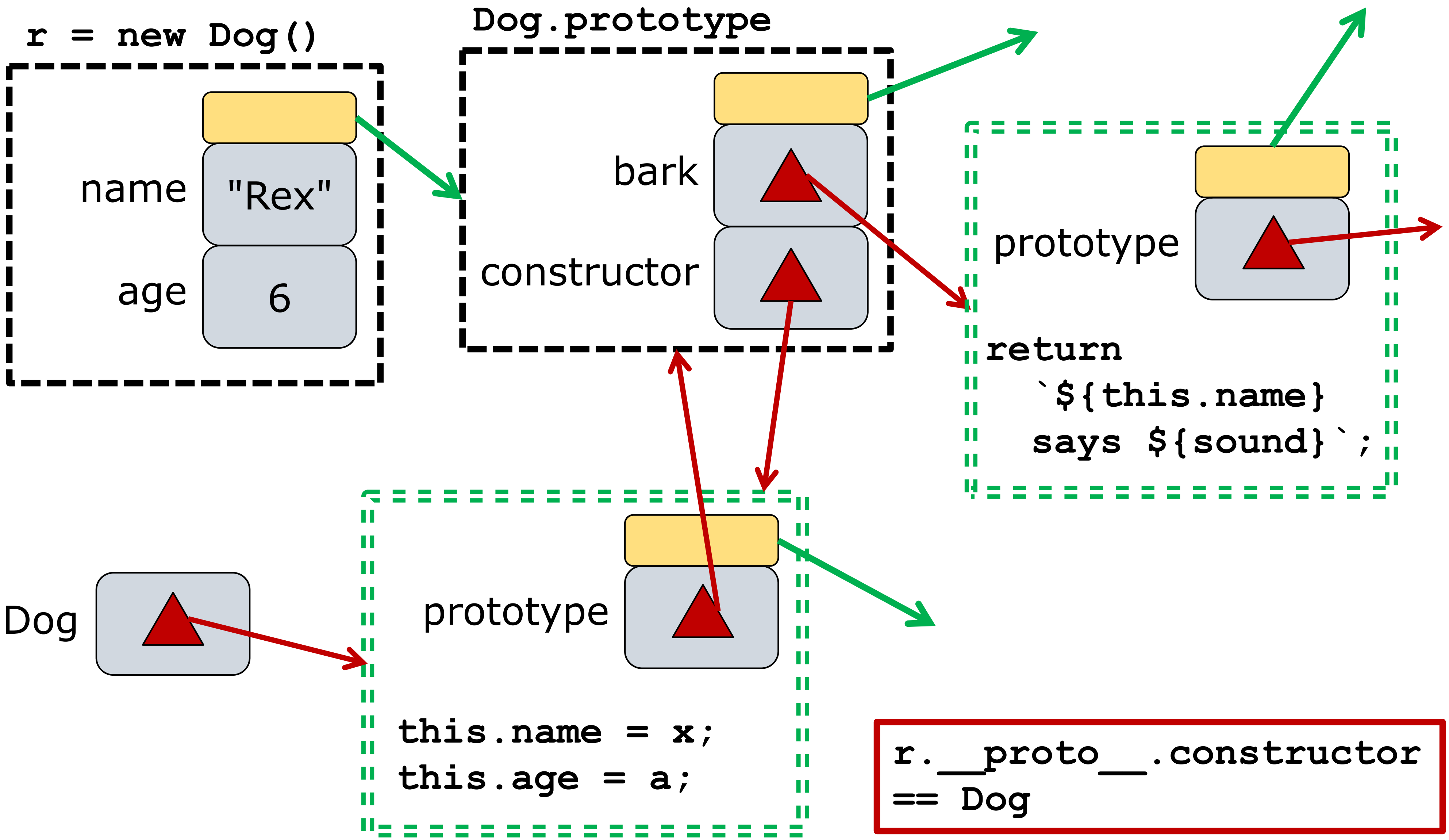
Class Properties



```
return
  `${this.name}
  says ${sound}`;
```

```
this.name = x;
this.age = a;
```

Meaning of `r instanceof Dog`



```
r.__proto__.constructor
== Dog
```

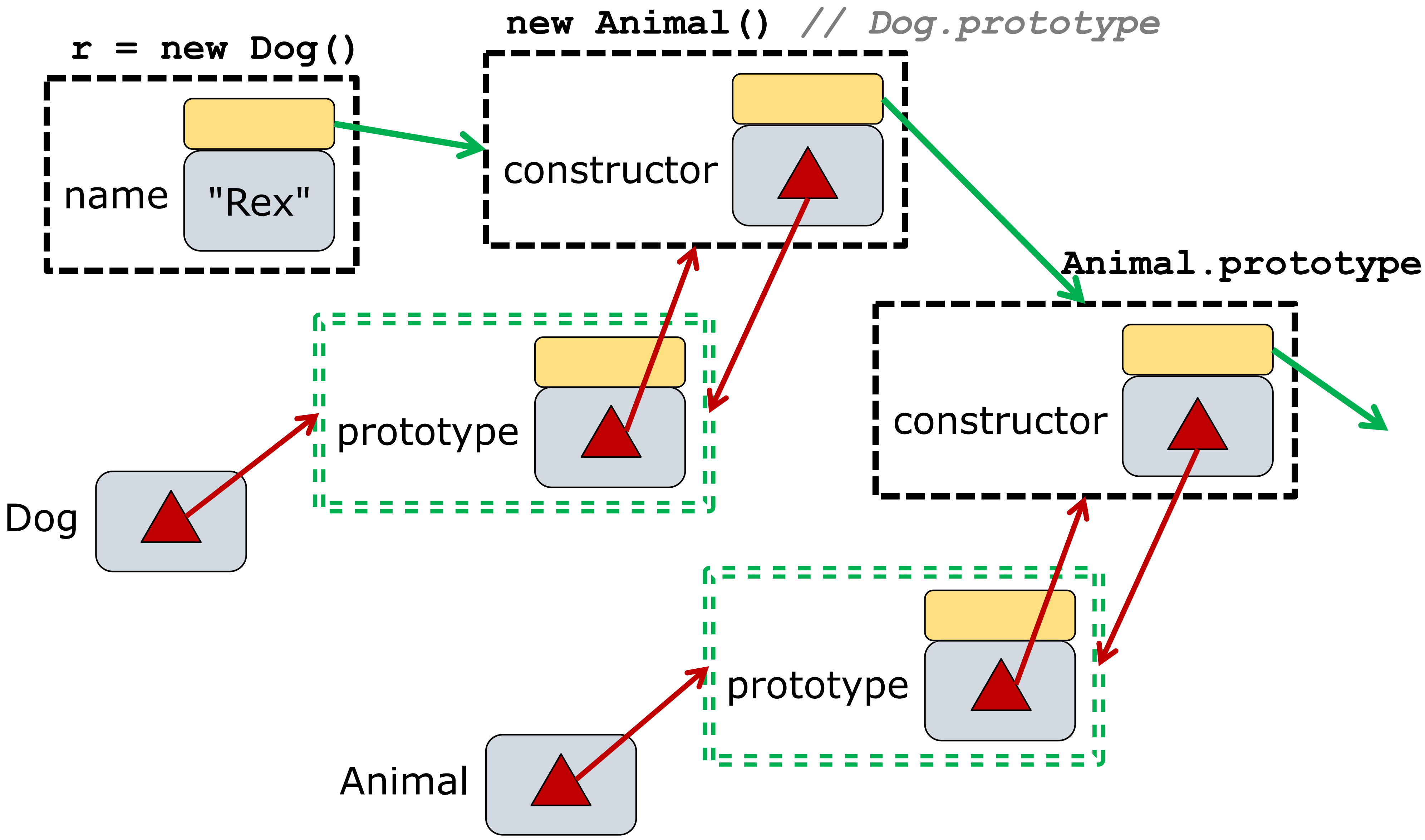
Idiom: Classical Inheritance

```
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



Prototype Chains

- instanceOf is checked transitively up the prototype chain

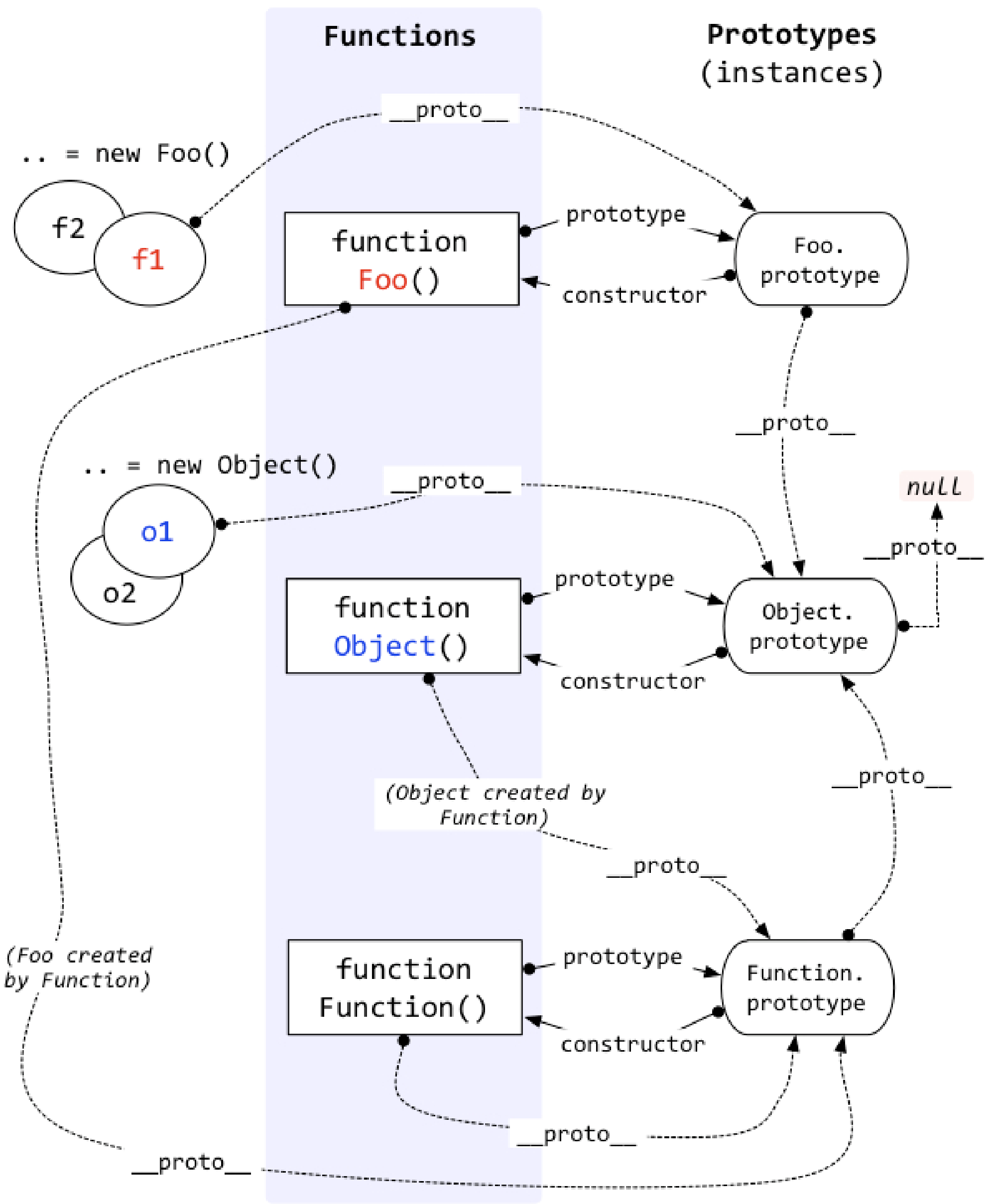
```
r instanceOf Dog //=> true
r instanceOf Animal //=> true
r instanceOf Object //=> true
```

- Q: Identify in the previous diagram

```
r.__proto__.__proto__.constructor
Dog.prototype.__proto__
   .constructor.prototype
```

To Ponder

JavaScript Object Layout [Hursh Jain/mollypages.org]



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
- Any function can be a constructor
- Prototypes are "parent" objects
  - Delegation up the chain of prototypes
  - Prototype is determined by constructor
  - Prototypes can be modified