To Ponder

Given: roster of students (an array)

Write a JavaScript program that outputs an html list of students (name and midterm score) whose gpa is > 3.0, such that the list is sorted by midterm score

1. Xi Chen (85)
2. Mary Smith (80)
3. Alessandro Reis (74)

JavaScript: Array API

Lecture 17

Arrays: Basics

- Numbered starting at 0
- Indexed with `[ ]`
- Property length is # of elements

```
let sum = 0;
for (let i = 0; i < n.length; i++) {
  sum += n[i];
}
```

Array Instantiation/Initialization

- Instantiate with new

```
let n = new Array(3);
```

- Initially, each element is undefined
- Note: Elements can be a mix of types

```
n[0] = 10;
n[1] = "hi";
n[2] = new Array(100);
```

- Array literals usually preferred

```
let n = [10, 20, 30, 40];
let m = ["hi", , "world", 3.14];
[3, "hi", 17, [3, 4]].length == 4
```

Dynamic Size

- Arrays can grow

```
let n = ["tree", 6, -2];
n.length == 3 //=> true
n[8] = 17;
n.length == 9 //=> true
```

- Arrays can shrink

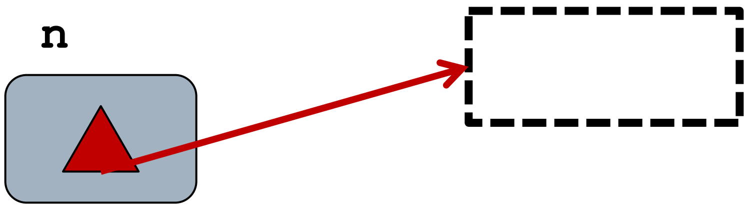
```
n.length = 2;
// n is now ["tree", 6 ]
```

Arrays are Dynamic

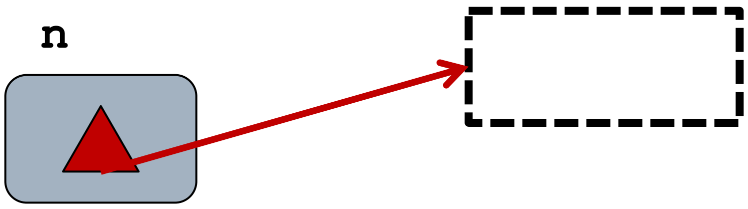
```
let n = [];
```

Arrays are Dynamic

```
let n = [];
```

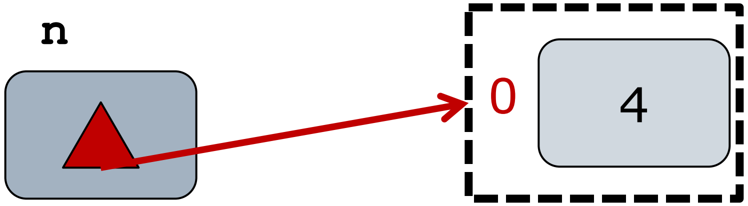


Arrays are Dynamic

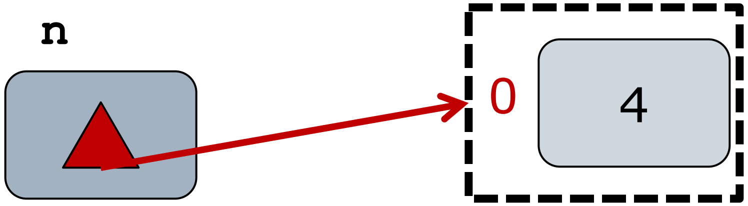


```
n[0] = 4;
```

Arrays are Dynamic

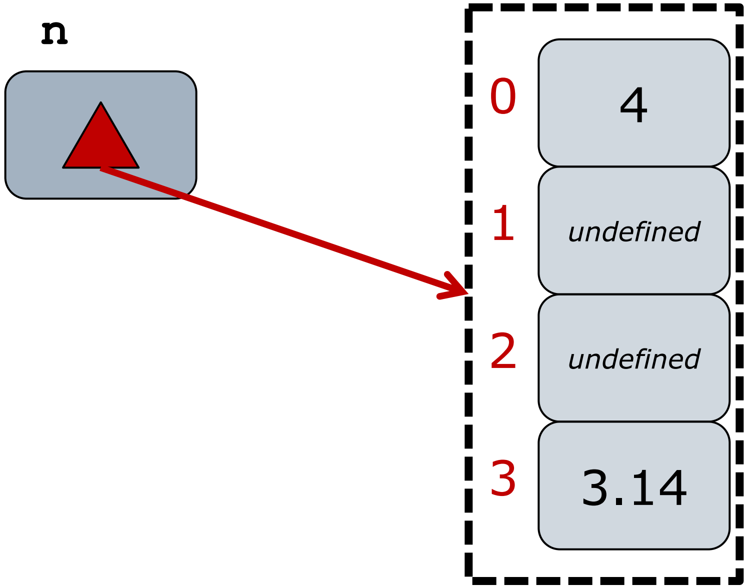


Arrays are Dynamic

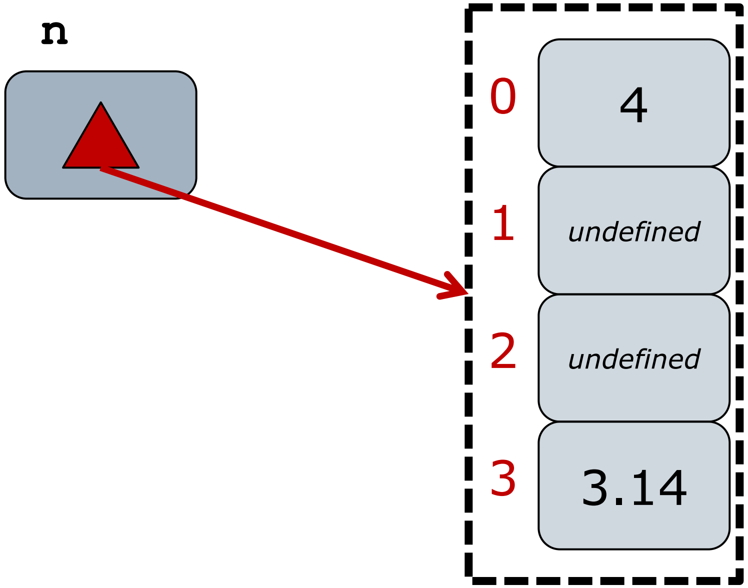


```
n[3] = 3.14;
```

Arrays are Dynamic

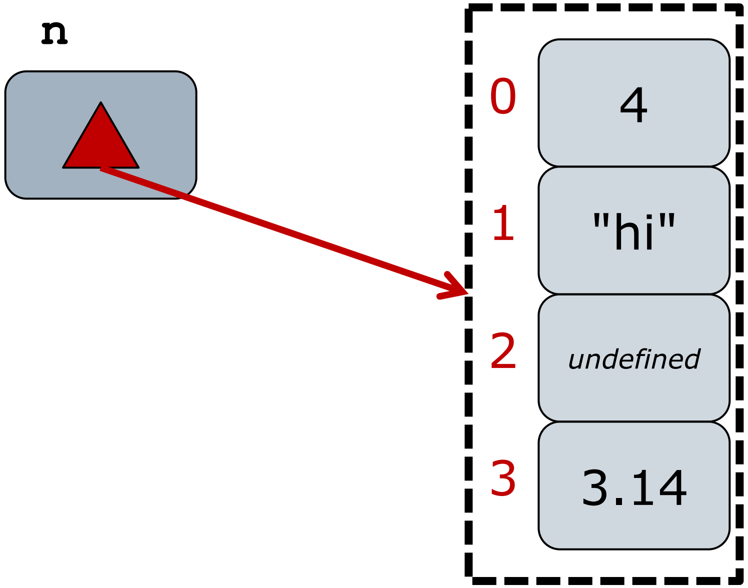


Arrays are Dynamic



```
n[1] = "hi";
```

Arrays are Dynamic



Accessors: Searching

- Find occurrence: indexOf/lastIndexOf
  - Returns -1 if not found

    indexOf(*element*[, *startIndex*])

    lastIndexOf(*element*[, *lastIndex*])
  - Optional parameter: start/end index
  - Uses strict equality (===)

```
let i = n.indexOf(elt);
while (i != -1) {
    report(i);
    i = n.indexOf(elt, i + 1);
}
```

Accessors: Extracting

- None of the following change the array
  - Return a new array/string with result
- Concatenate: **concat**

```
concat(a1, a2, …, aN)
let d = n.concat(n);
```

- Extract a sub-section: **slice**

```
slice(startIndex, endIndex)
k = n.slice(1, 3); // k is n[1], n[2]
```

- Combine into string: **join**

```
join(separator)
s = n.join(" "); // default is ","
```

Mutators: Growing/Shrinking

- Add/remove from end: **`push/pop`**

```
let n = [10, 20];
newLength = n.push(30, 40); //=> 4
lastValue = n.pop(); //=> 40
```

- Add/remove from beginning: **`unshift/shift`**

```
let n = [10, 20];
newLength = n.unshift(30, 40); //=> 4
firstValue = n.shift(); //=> 30
```

- Push/shift gives FIFO queue

Push Example

```
function findAll(n, elt) {
  let indices = [];
  let i = n.indexOf(elt);
  while (i != -1) {
    indices.push(i);
    i = n.indexOf(elt, i + 1);
  }
  return indices;
}
```

Mutators: Delete/Insert/Replace

- Delete/insert/replace sub-array: `splice`

  `splice (`*`index`*`, `*`howMany[, e1, e2, …, eN]`*`)`

  - Modifies array (*cf.* `slice`, an *accessor*)
  - Returns array of removed elements

  ```
  let magic = [34, -17, 6, 4];
  let removed = magic.splice(2, 0, 13);
  // removed is []
  // magic is [34, -17, 13, 6, 4]

  removed = magic.splice(3, 1, "hi", "yo");
  // removed is [6]
  // magic is [34, -17, 13, "hi", "yo", 4]
  ```

Mutators: Rearrange

- Transpose all elements: **`reverse`**

```
let n = [5, 300, 90];
n.reverse(); // n is [90, 300, 5]
```

- Order all elements: **`sort`**

```
let f = ["blue", "beluga","killer"];
f.sort(); // f is
          // ["beluga", "blue", "killer"]
n.sort(); // n is [300, 5, 90]
```

Mutators: Rearrange

- Transpose all elements: **`reverse`**

```
let n = [5, 300, 90];
n.reverse(); // n is [90, 300, 5]
```

- Order all elements: **`sort`**

```
let f = ["blue", "beluga", "killer"];
f.sort(); // f is
          // ["beluga", "blue", "killer"]
n.sort(); // n is [300, 5, 90]
```

- Problem: Default ordering is based on string representation (lexicographic)
- Solution: Use a function that compares

Sorting with Comparator

- A comparator (a, b) returns a number
  - < 0 iff a is *smaller than* b
  - == 0 iff a is *same size* as b
  - > 0 iff a is *greater than* b
- Examples

```
function lenOrder(a, b) {
  return a.length - b.length;
}
function compareNumbers(a, b) {
  return a - b;
}
```

Sorting with Comparator

- □ Optional argument to sort

```
sort([compareFunction])
```

- □ Example

```
names.sort(lenOrder);
n.sort(compareNumbers);

n.sort(function(a, b) {
  return a - b;
});
```

Iteration: Logical Quantification

```
function isBig(elt, index, array) {
  return (elt >= 10);
}
```

- □ Universal quantification: **every**

```
[5, 8, 13, 44].every(isBig); // false
[51, 18, 13, 44].every(isBig); // true
```

- □ Existential quantification: **some**

```
[5, 8, 13, 44].some(isBig); // true
[5, 8, 1, 4].some(isBig); // false
```

- □ Neither modifies original array

Iteration: Filter

- Pare down an array based on a condition: **`filter`**

  ```
  filter(predicate)
  predicate(element, index, array)
  ```

- Returns a new array, with elements that satisfied the predicate
  - Does not modify the original array
- Example

  ```
  t = [12, 5, 8, 13, 44].filter(isBig);
  ```

Iteration: Map

- Transform an array into a new array, element by element: map
  - E.g. an array of strings into an array of their lengths
  - ["hi", "there", "world"] → [2, 5, 5]

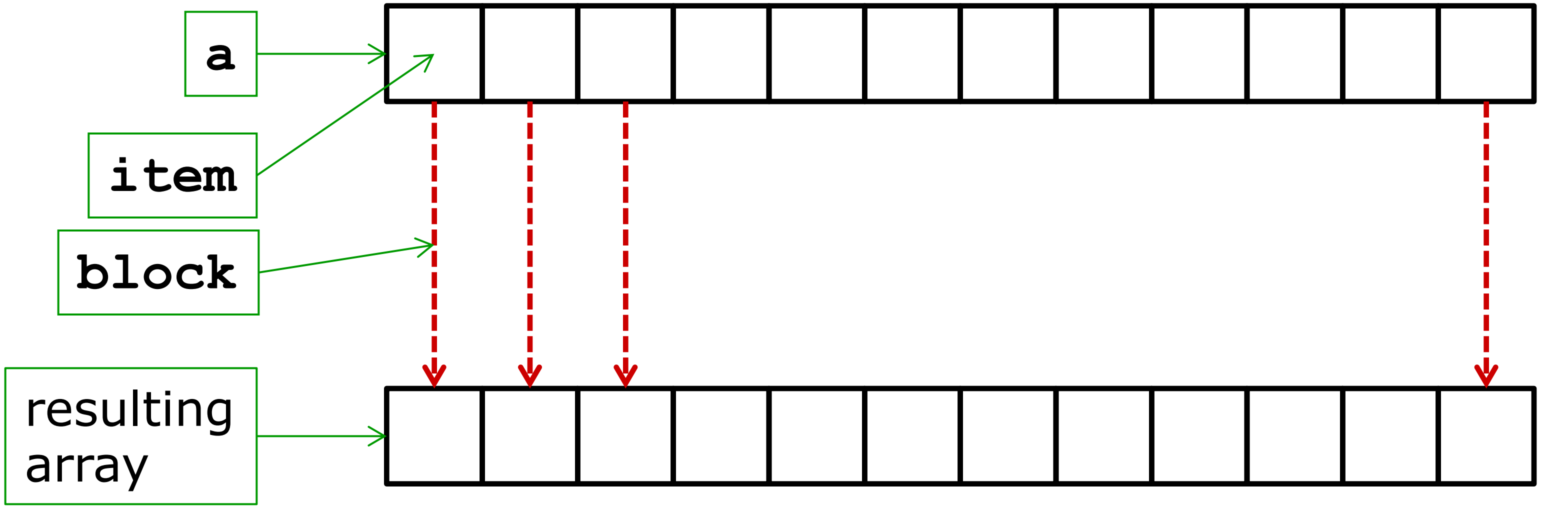
    ```
    map(callback)
    callback(element, index, array)
    ```
- Example

```
len = names.map(function(elt, i, a) {
  return elt.length
});
```

Recall: Ruby Map

- Transform an array into a new array, *element by element*
- Uses *block* to calculate each new value

```
a.map { |item| block }
```



Iteration: For Each

- Similar to map, but preferred for side-effects and changing an array in place

```
forEach(callback)
callback(element, index, array)
```

- Example

```
function logArrayElts(elt, i, array) {
  console.log("[" + i + "] = " + elt);
}
[2, 5, 9].forEach(logArrayElts);
```

Iteration: Reduce

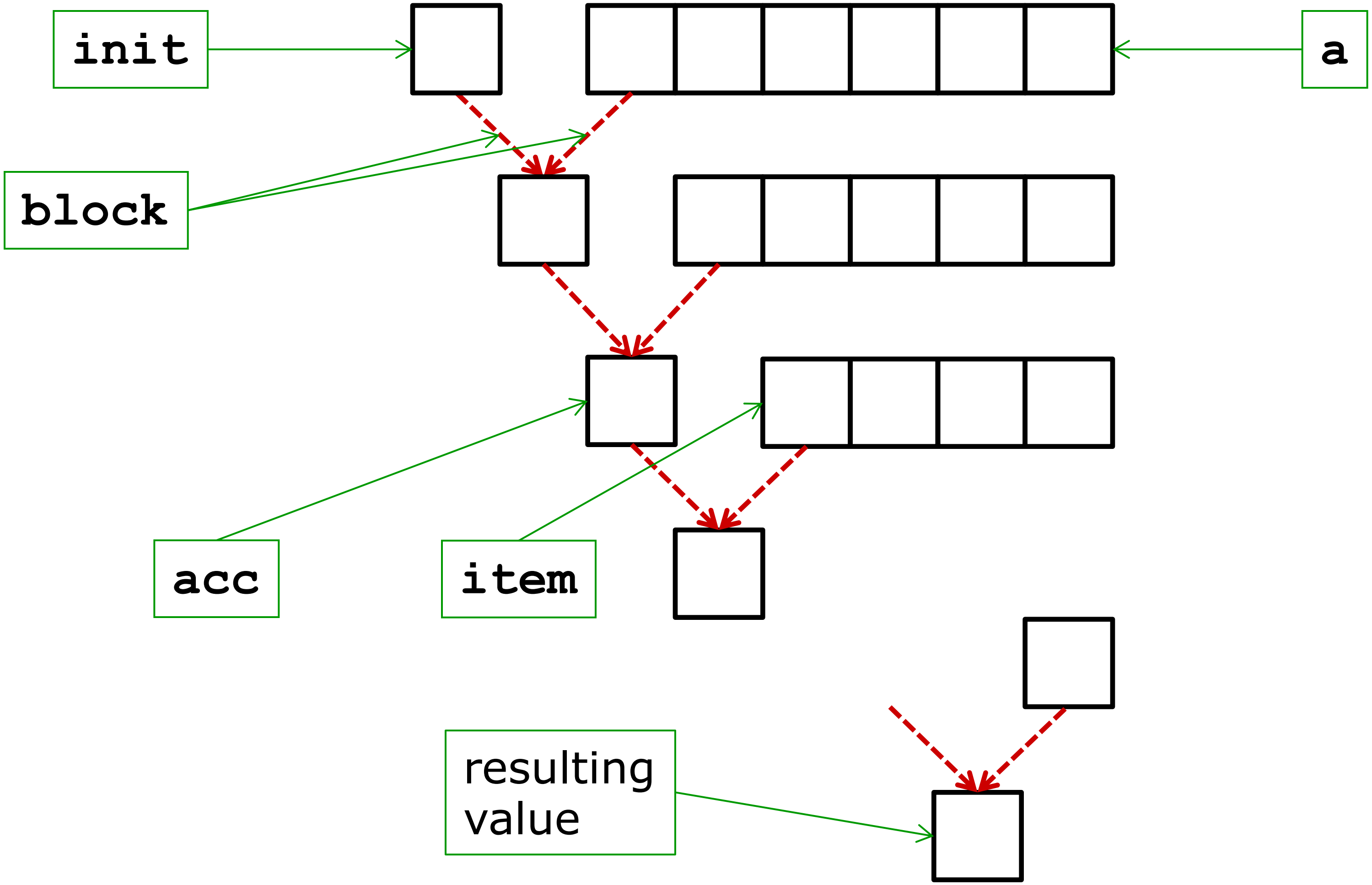
- Applies a binary operator between all the elements of the array
  - E.g., to sum the elements of an array
  - [15, 10, 8] → 0 + 15 + 10 + 8 → 33

  ```
  reduce(callback[, initialValue])
  callback(previous, elt, index, array)
  ```

- Examples

```
function sum(a, b) { return a + b; }
function acc(a, b) { return a + 2 * b; }
[2, 3, 7, 1].reduce(sum)      //=> ?
[2, 3, 7, 1].reduce(sum, 0)   //=> ?
[2, 3, "7", 1].reduce(sum)    //=> ?
[2, 3, 7, 1].reduce(acc)      //=> ?
[2, 3, 7, 1].reduce(acc, 0)   //=> ?
```

Recall: Ruby’s Reduction Chain



Iteration: Reduce

- Examples with anonymous functions

```
[2, 3].reduce( function(a, b) {
      return a + b;
    }); //=> ?
[[0, 1], [2, 3], [4, 5]].reduce(
    function(a, b) {
      return a.concat(b);
    }); //=> ?
```

Your Turn

Given: roster of students (an array)

Write a JavaScript program that outputs an html list of students (name and midterm score) whose gpa is > 3.0, such that the list is sorted by midterm score

1. Xi Chen (85)
2. Mary Smith (80)
3. Alessandro Reis (74)

Example Input

```
let roster =
[    { name: "Mary Smith",
       gpa: 3.7,
       midterm: 80 },
     { name: "Xi Chen",
       gpa: 3.5,
       midterm: 85 },
     { name: "Alessandro Reis",
       gpa: 3.2,
       midterm: 74 },
     { name: "Erin Senda",
       gpa: 3.0,
       midterm: 68 }        ];
```

One Solution

```
document.writeln("<ol><li>");
document.writeln(
  roster.filter(function (e, i, a) {
    return e.gpa > 3.0;
  }).sort(function (a, b) {
    return b.midterm - a.midterm;
  }).map(function (e, i, a) {
    return e.name + " ("
                  + e.midterm + ")";
  }).join("</li><li>"));
document.writeln("</li></ol>");
```

Summary

- Array accessors and mutators
  - Accessors: indexOf, slice
  - Mutators for extraction: push/pop, unshift/shift, splice
  - Mutators for rearranging: reverse, sort
- Array iteration
  - Quantification: every, some, filter
  - Map (foreach for side-effects & mutating)
  - Reduce

To Ponder

Assume:

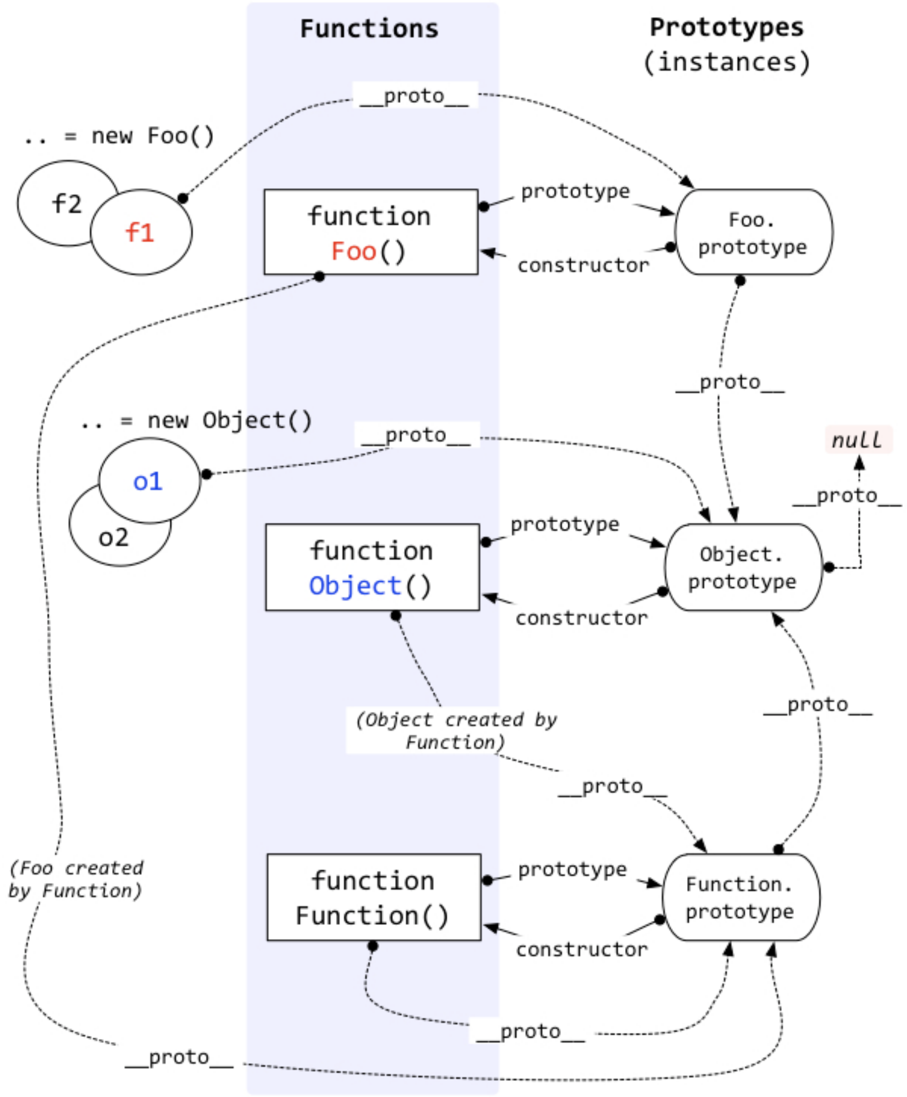
```
var d = new Dog();
d instanceof Dog; //=> true
d instanceof Pet; //=> true
```

Questions:

- What is Dog? (A class? An interface? ...)
- What is Pet?
- How are they related? Draw the hierarchy

To Ponder

JavaScript Object Layout [Hursh Jain/mollypages.org]



JavaScript: Objects, Methods, Prototypes

What is an Object?

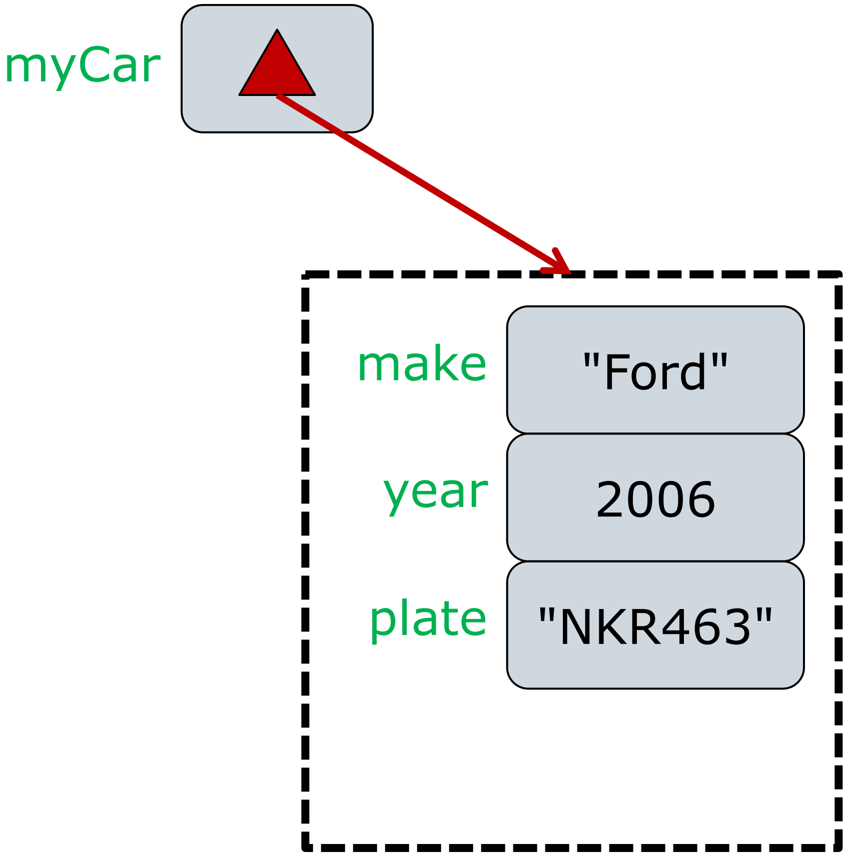
- *Property*: a key/value pair
  - aka name/value pair
- *Object*: a partial map of properties
  - Keys must be unique
- Creating an object, literal notation

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

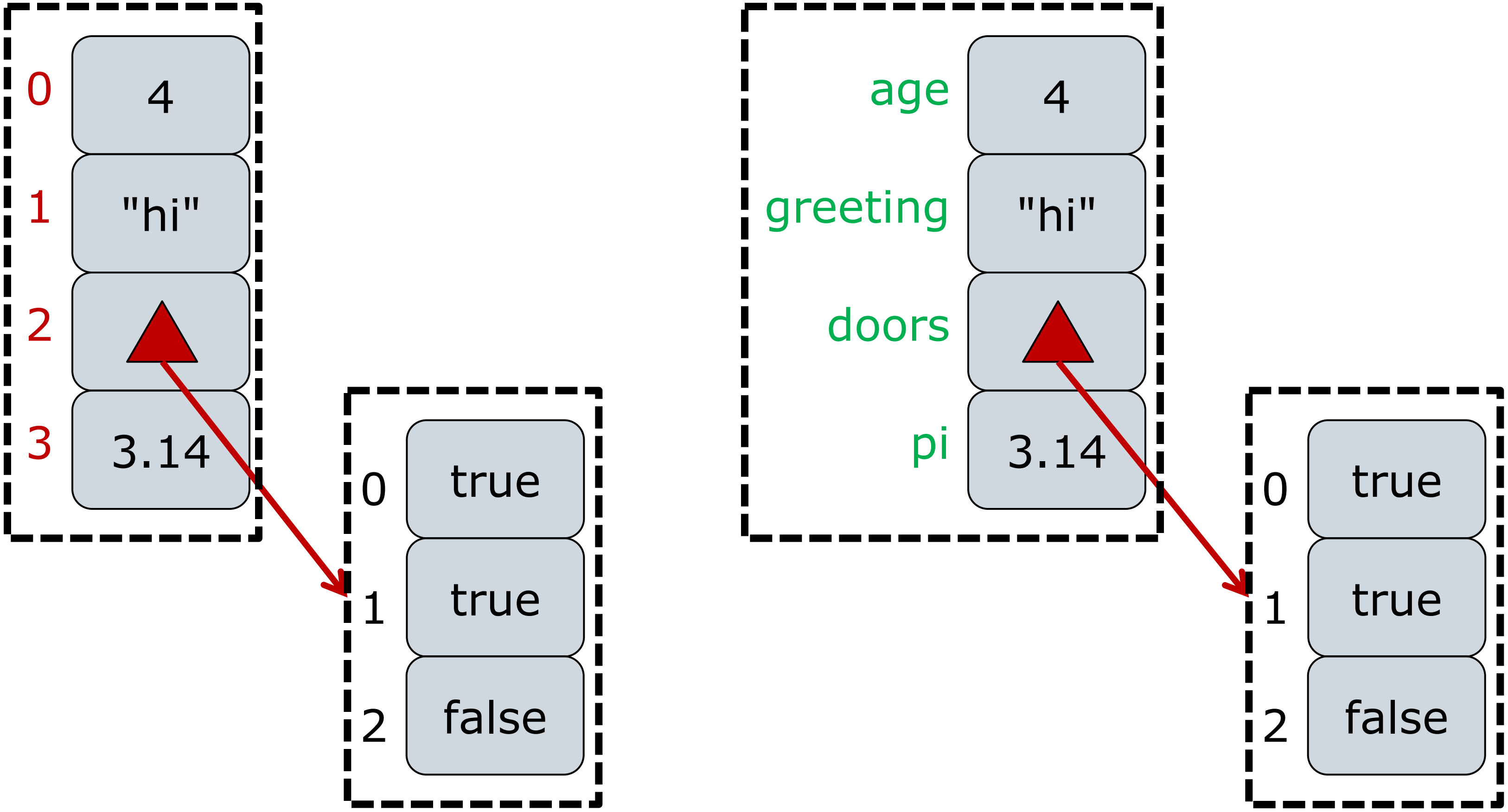
- To access/modify an object's properties:

```
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

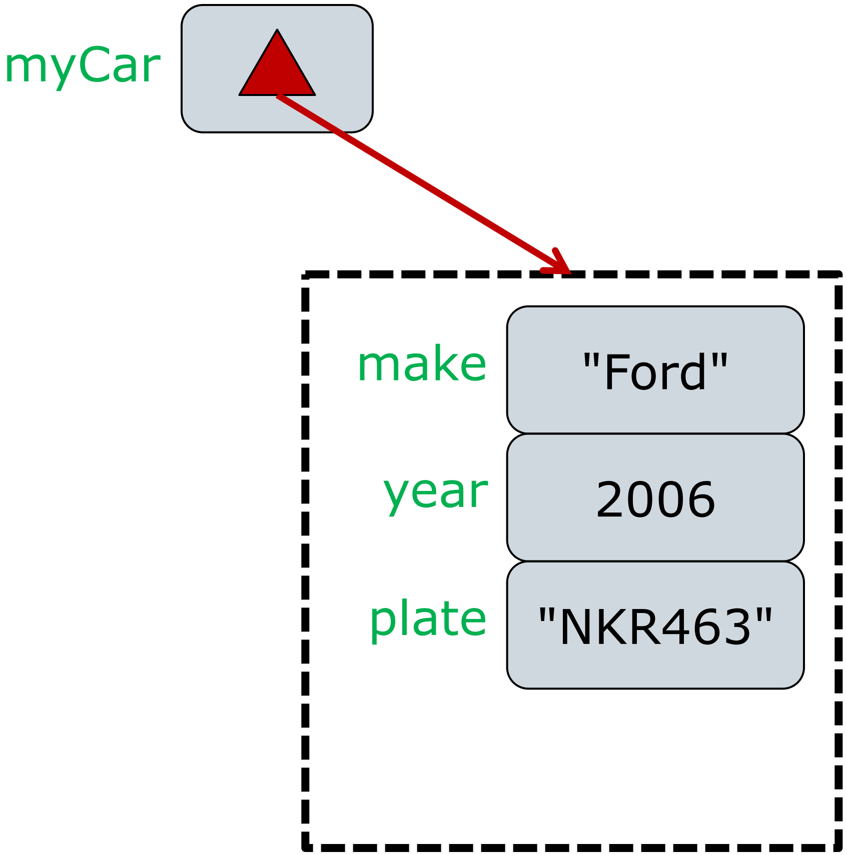
□ Objects can grow

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

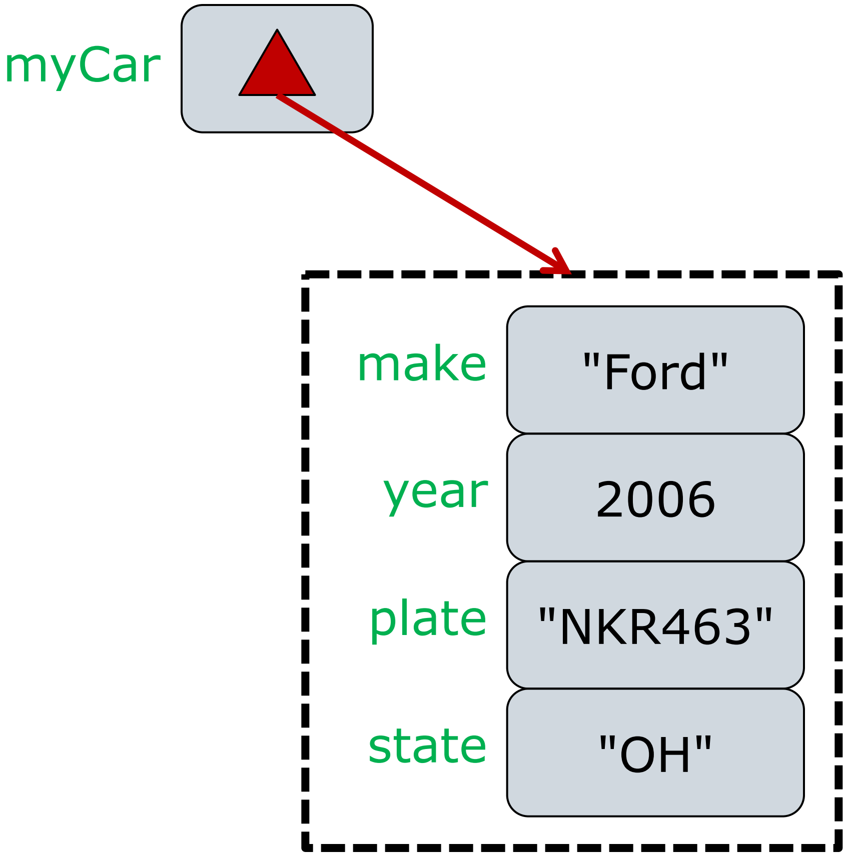
□ Objects can shrink

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

Object Properties

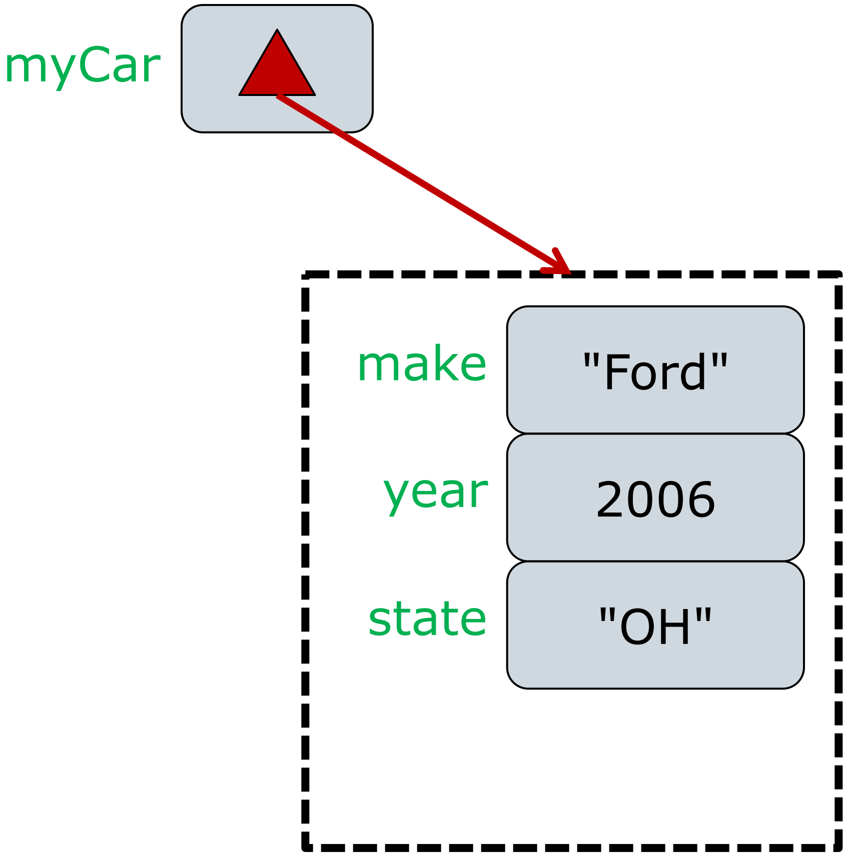


Object Properties



```
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;
}
```

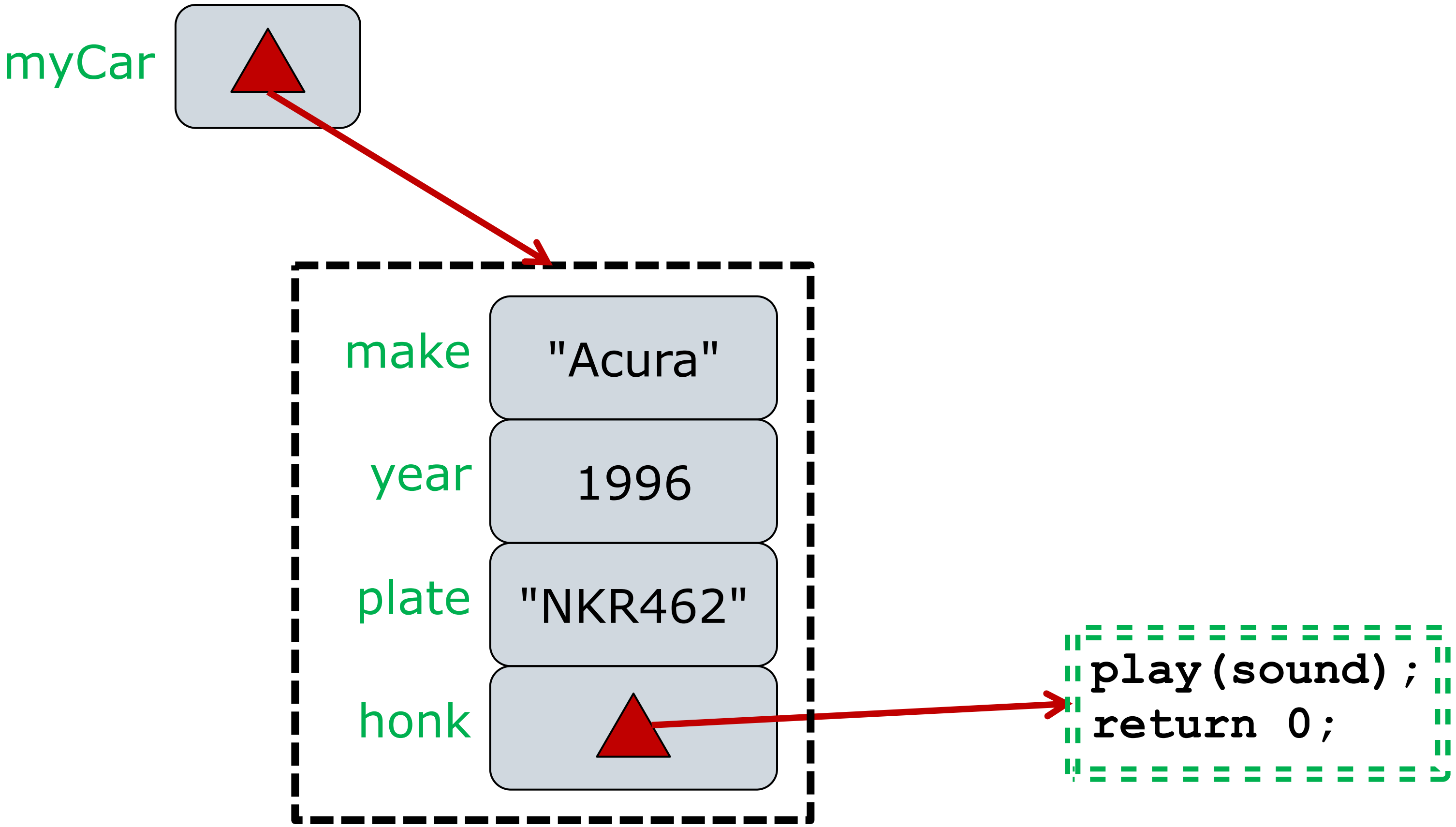
Example: Method

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

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.
```

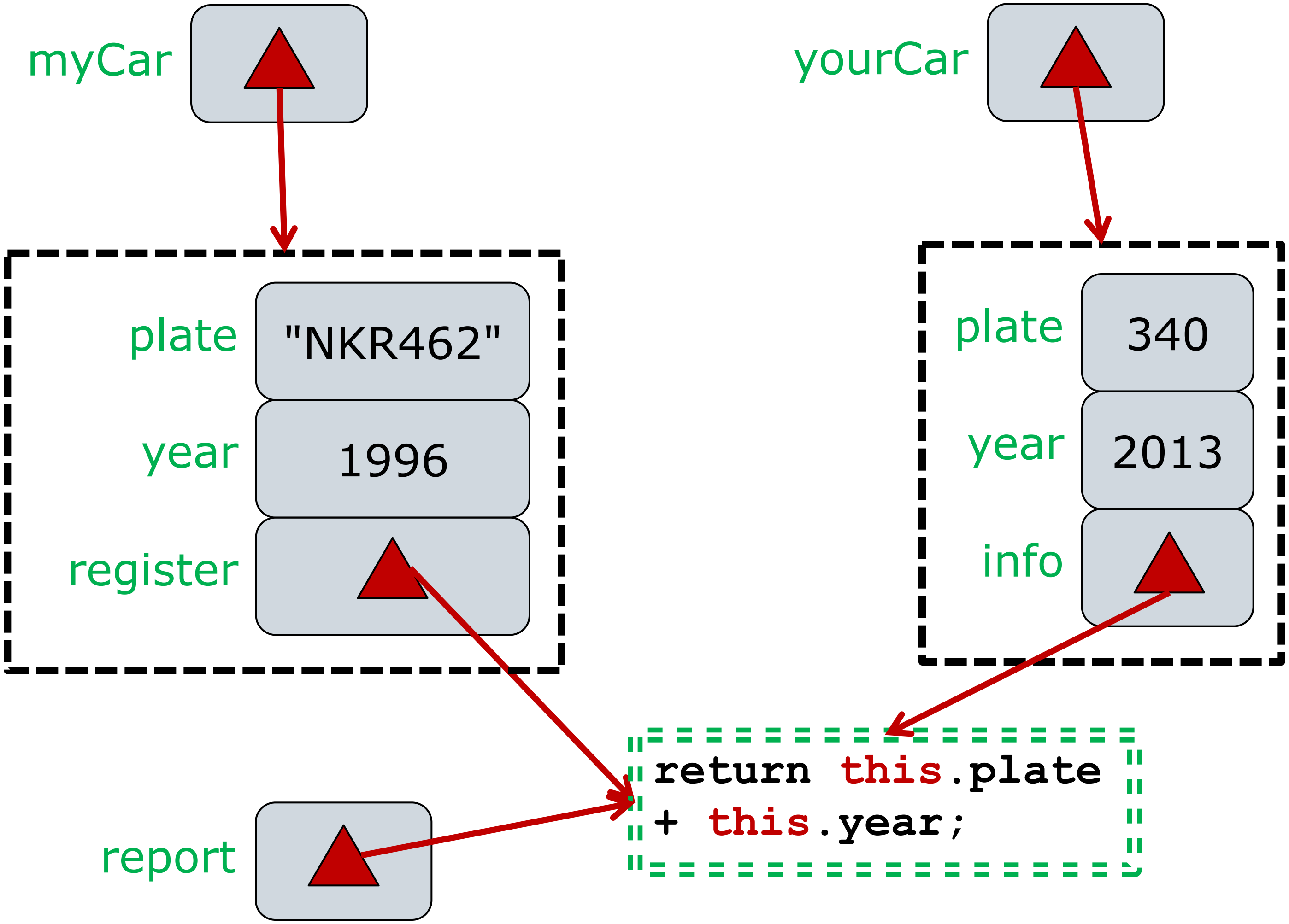
- Inside a function, keyword “this”

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

- At run-time, “this” is 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



```
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");
```

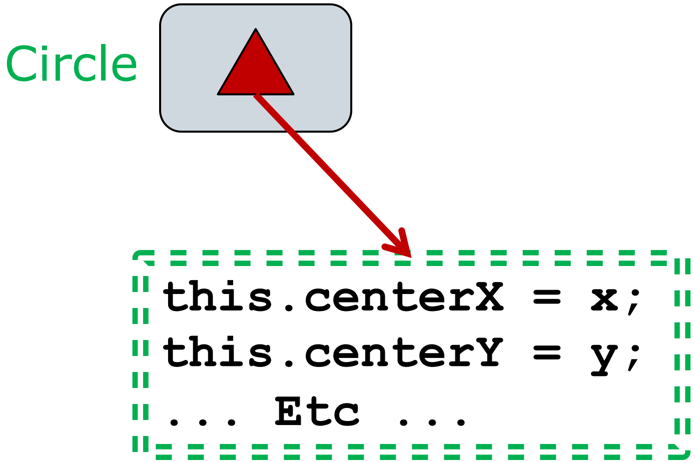
- Naming convention: Functions intended to be constructors are capitalized

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

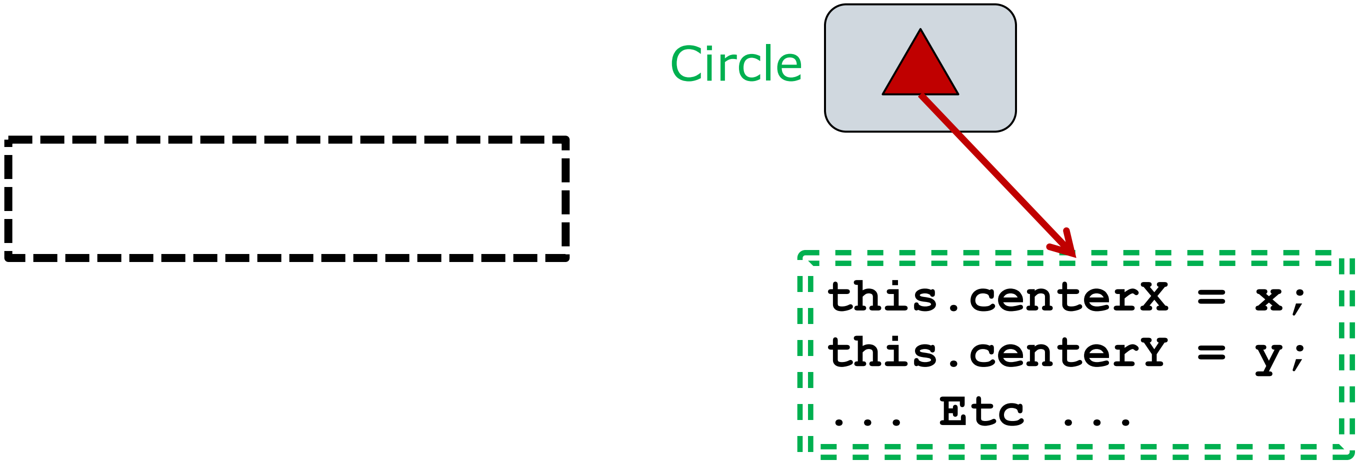
```
let c = new Circle(10, 12, 2.45);
```



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

Creating a Circle Object

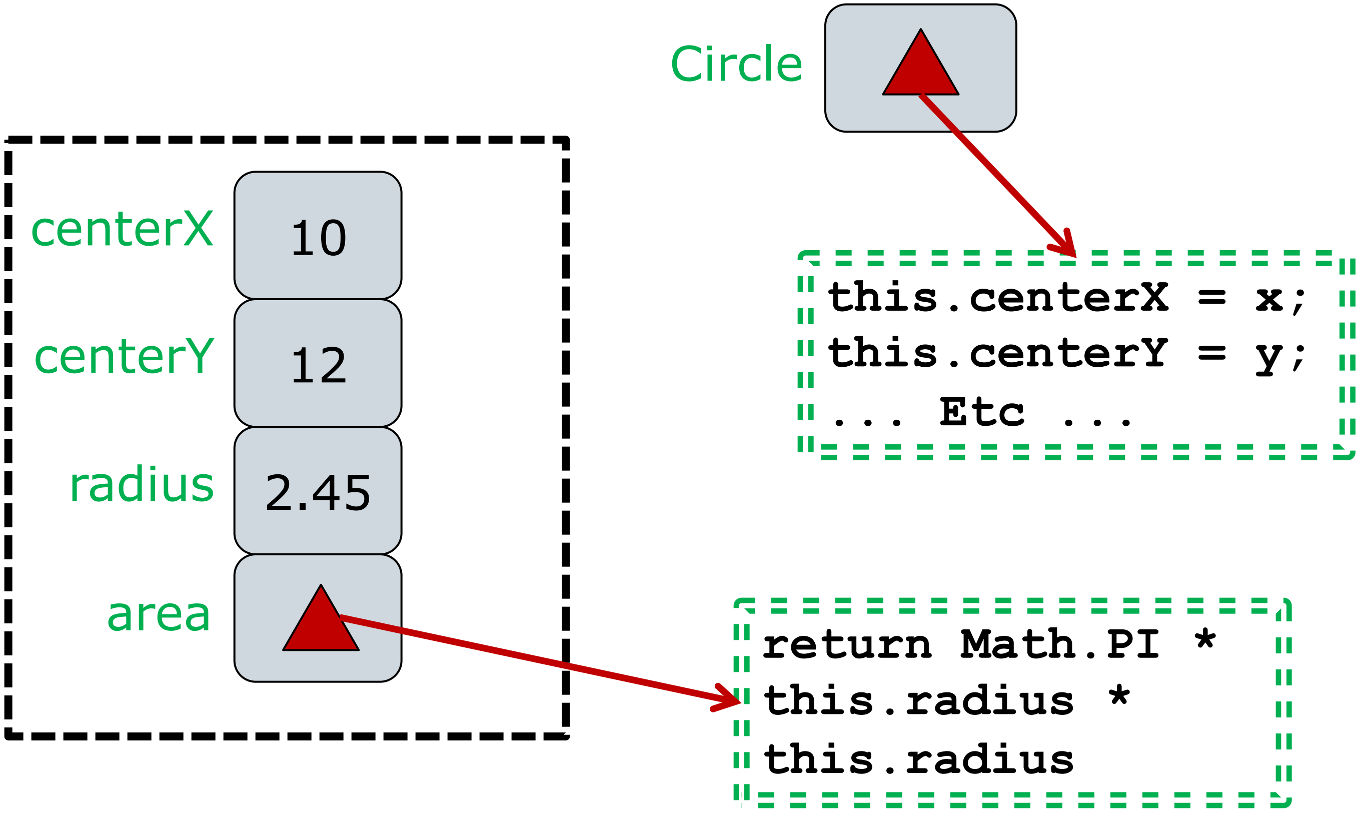
```
let c = new Circle(10, 12, 2.45);
```



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

Creating a Circle Object

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

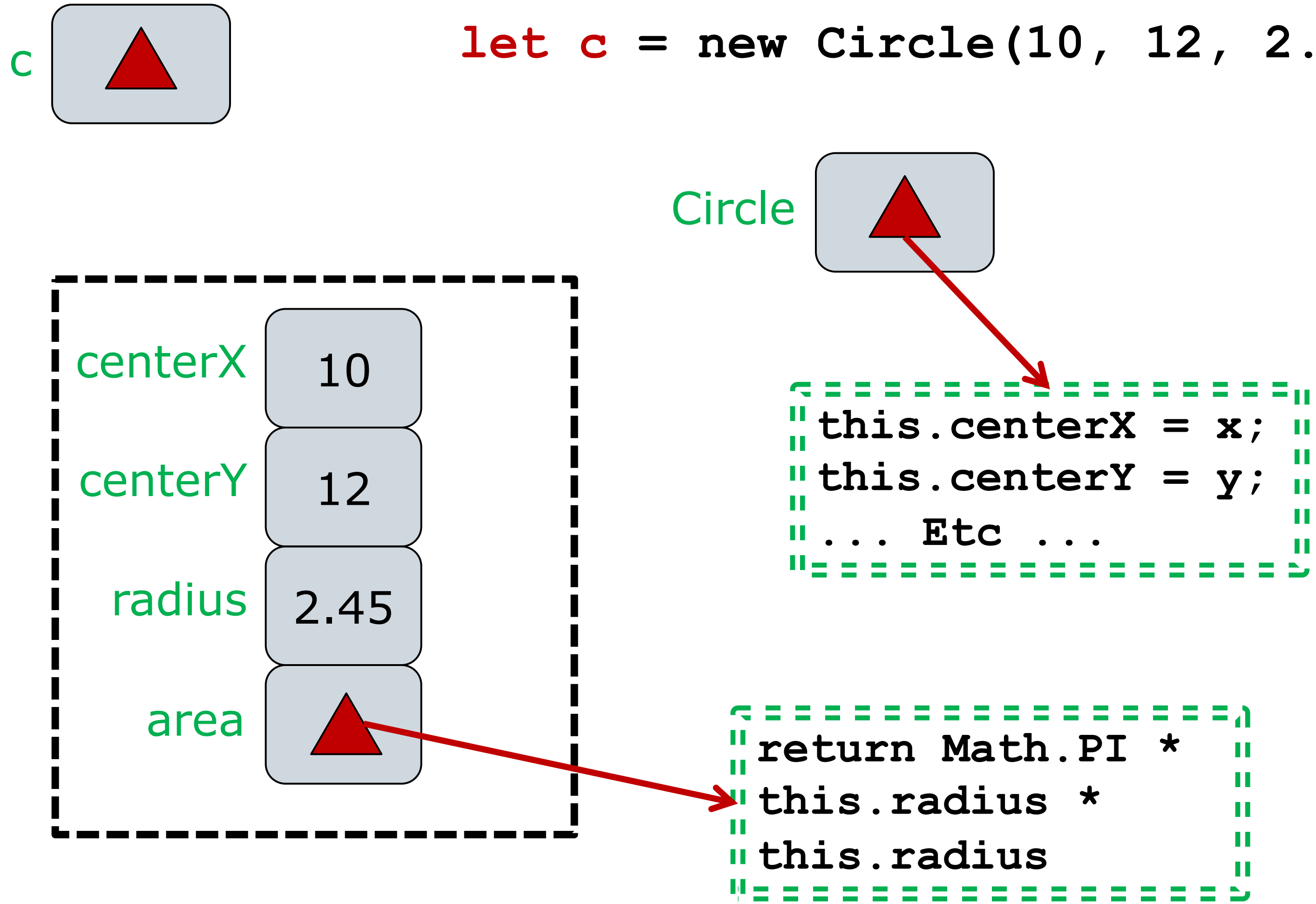


```
this.centerX = x;
this.centerY = y;
... Etc ...
```

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

Creating a Circle Object

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

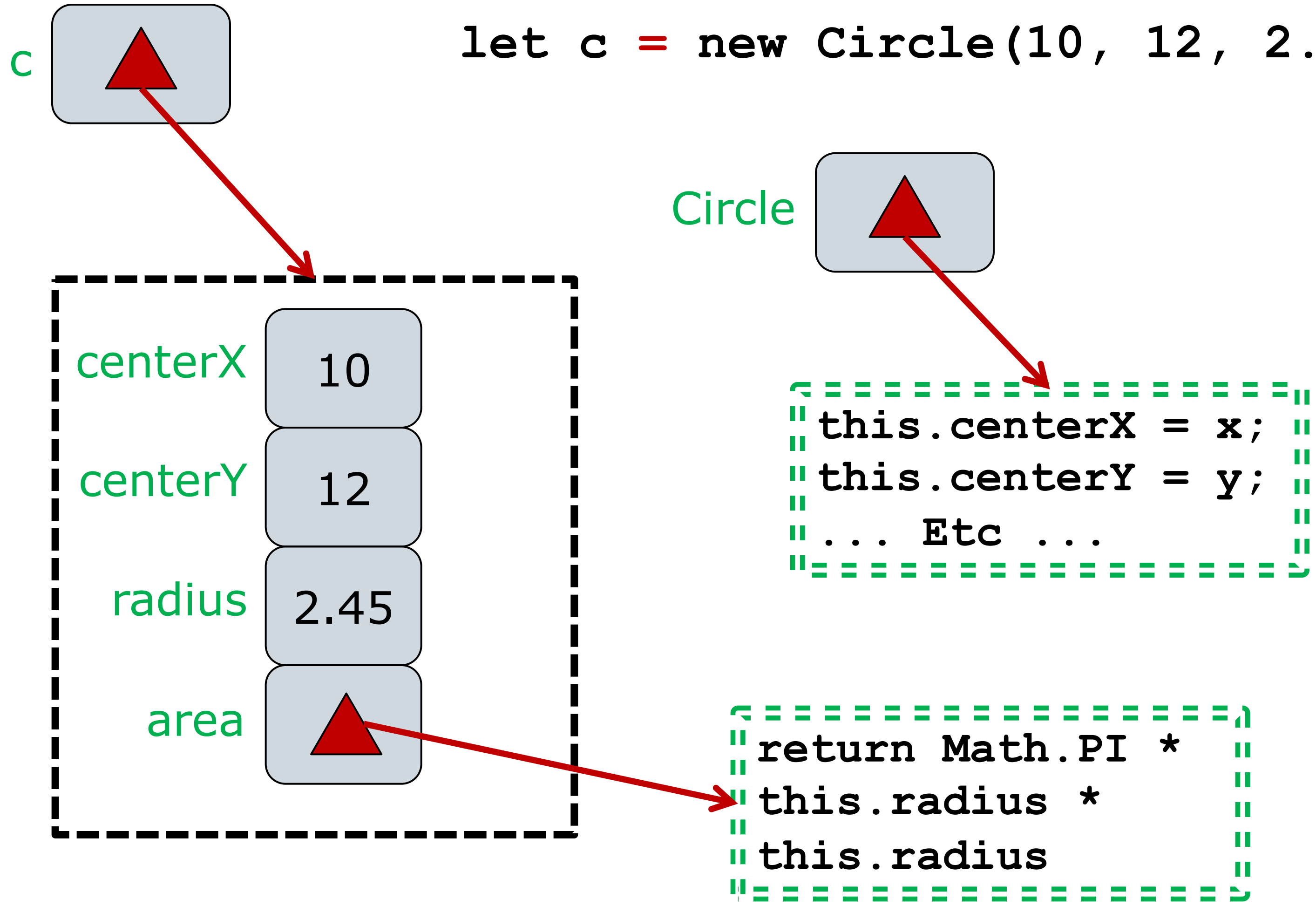


```
this.centerX = x;
this.centerY = y;
... Etc ...
```

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

Creating a Circle Object

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



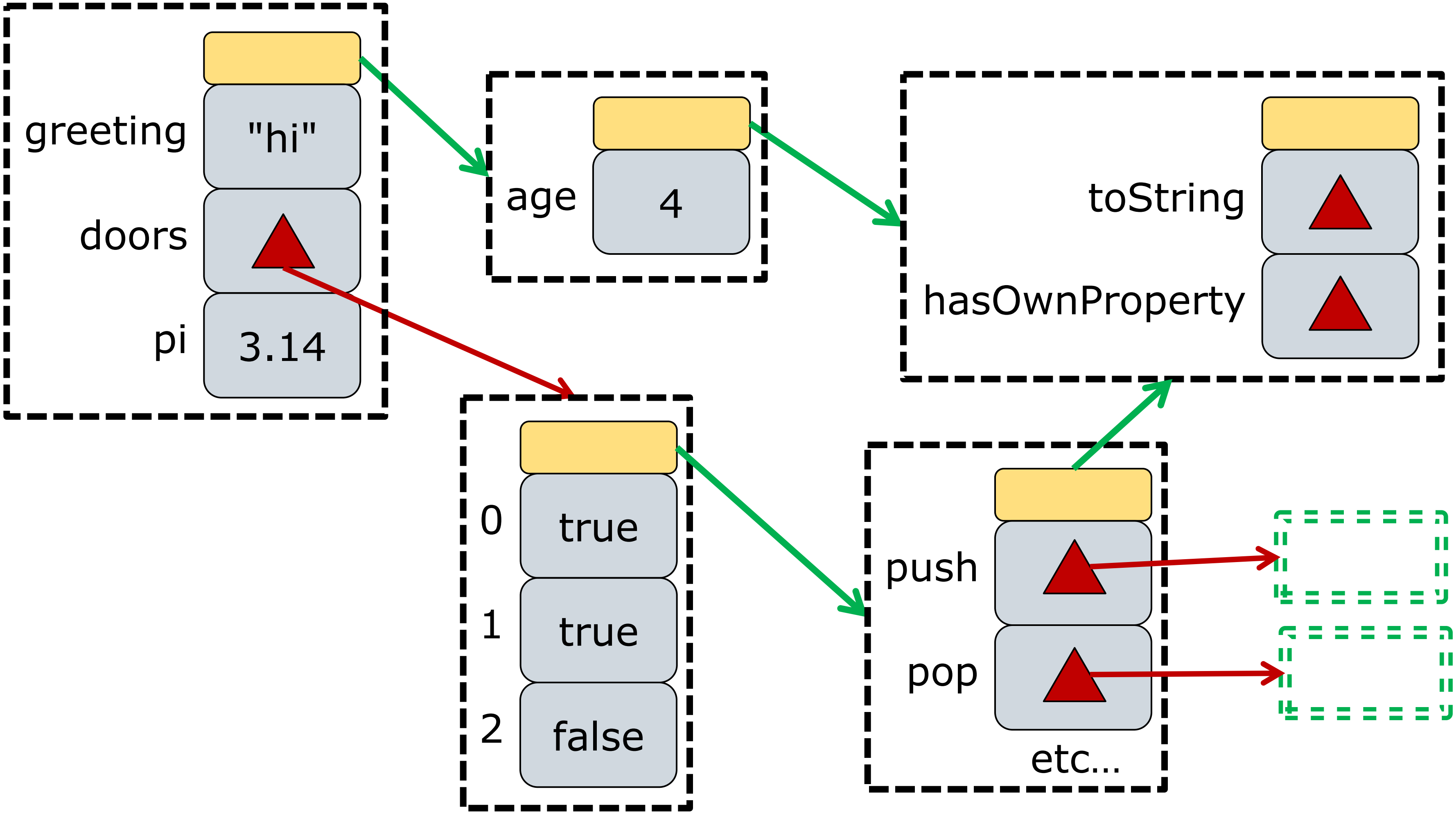
```
this.centerX = x;
this.centerY = y;
... Etc ...
```

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

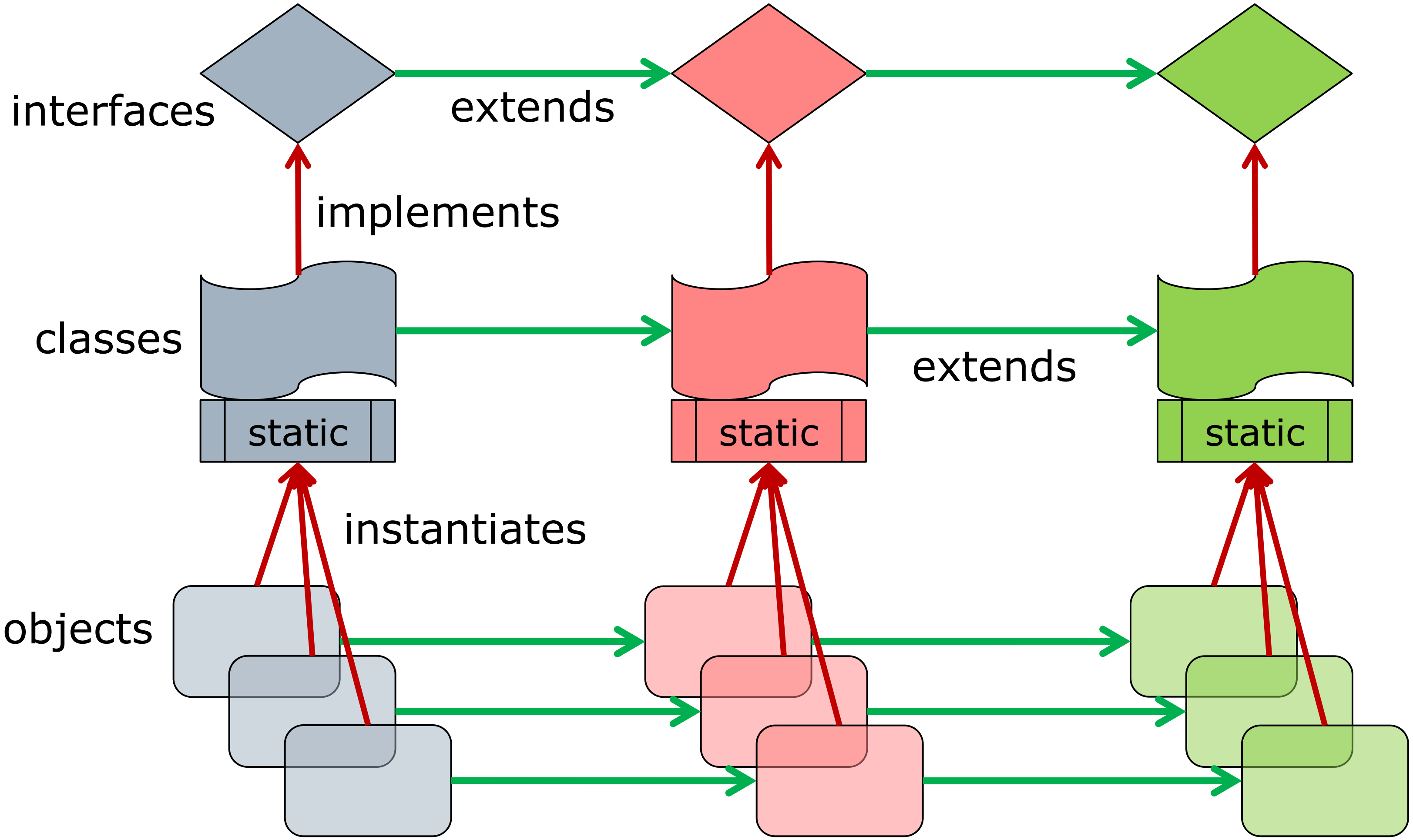
Prototypes

- 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

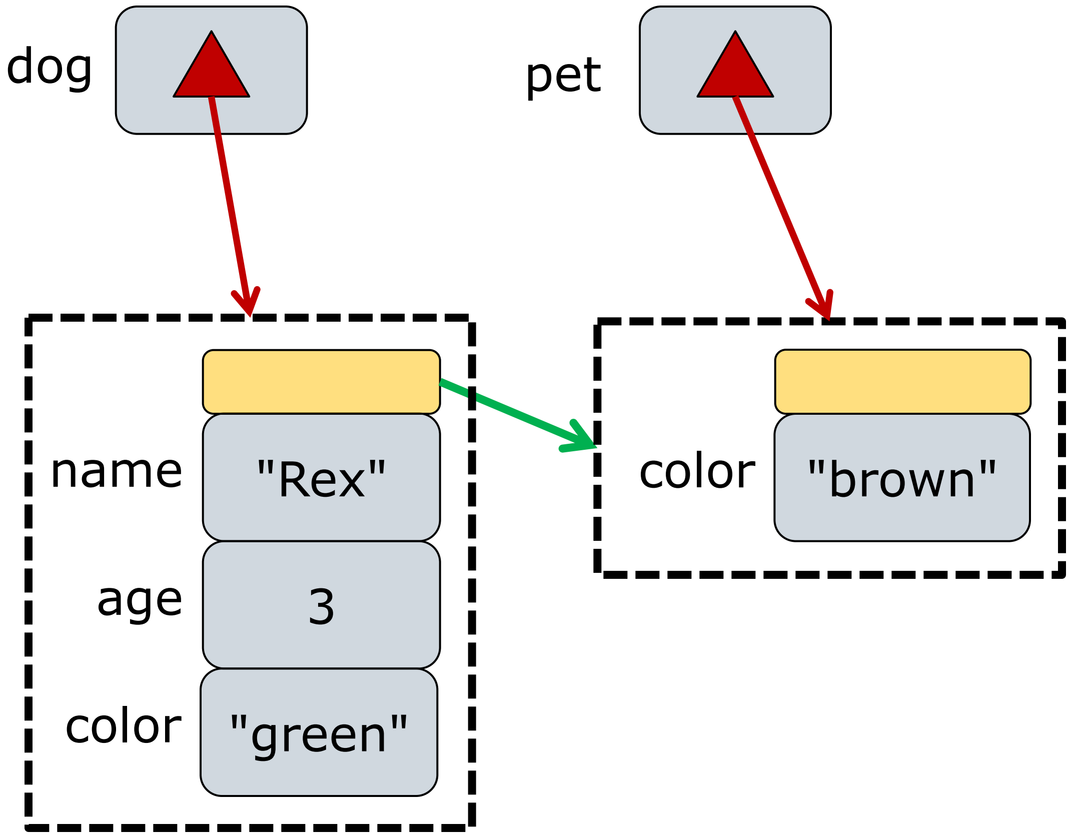
- Consider two objects

```
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

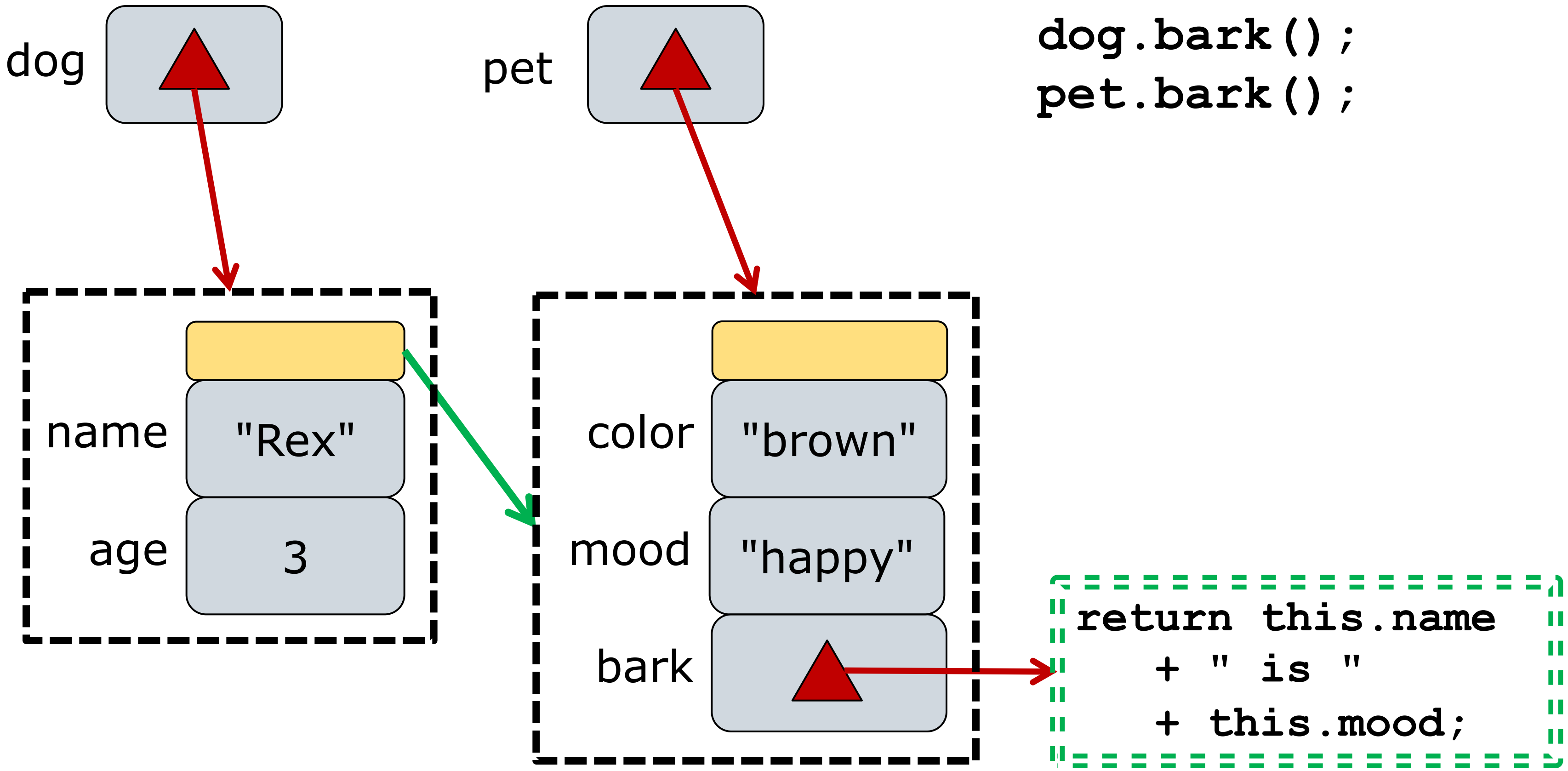


Prototypes Are Dynamic Too

- 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



```
return this.name
    + " is "
    + this.mood;
```

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

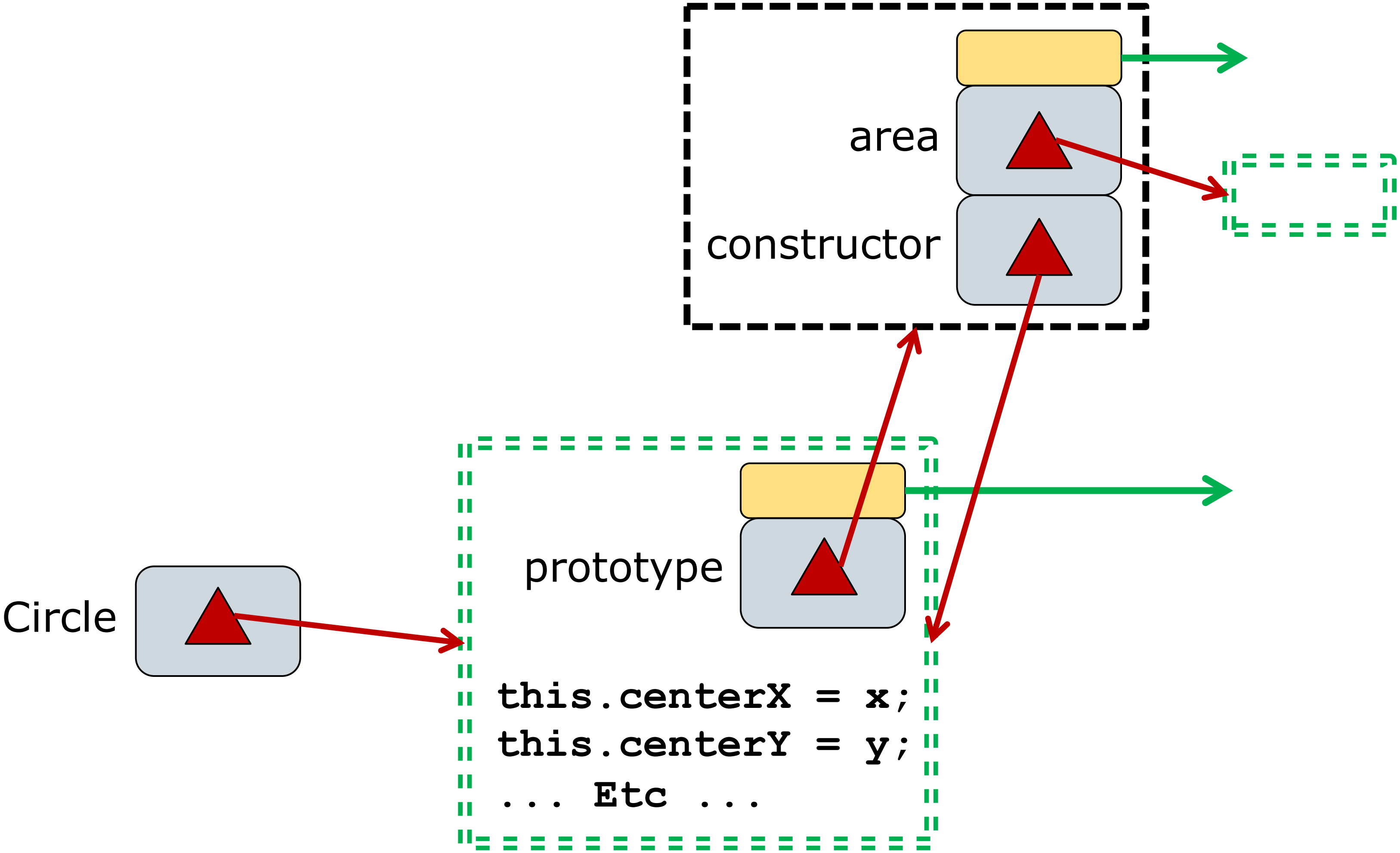
Connecting Objects & Prototypes

- □ 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



area

constructor

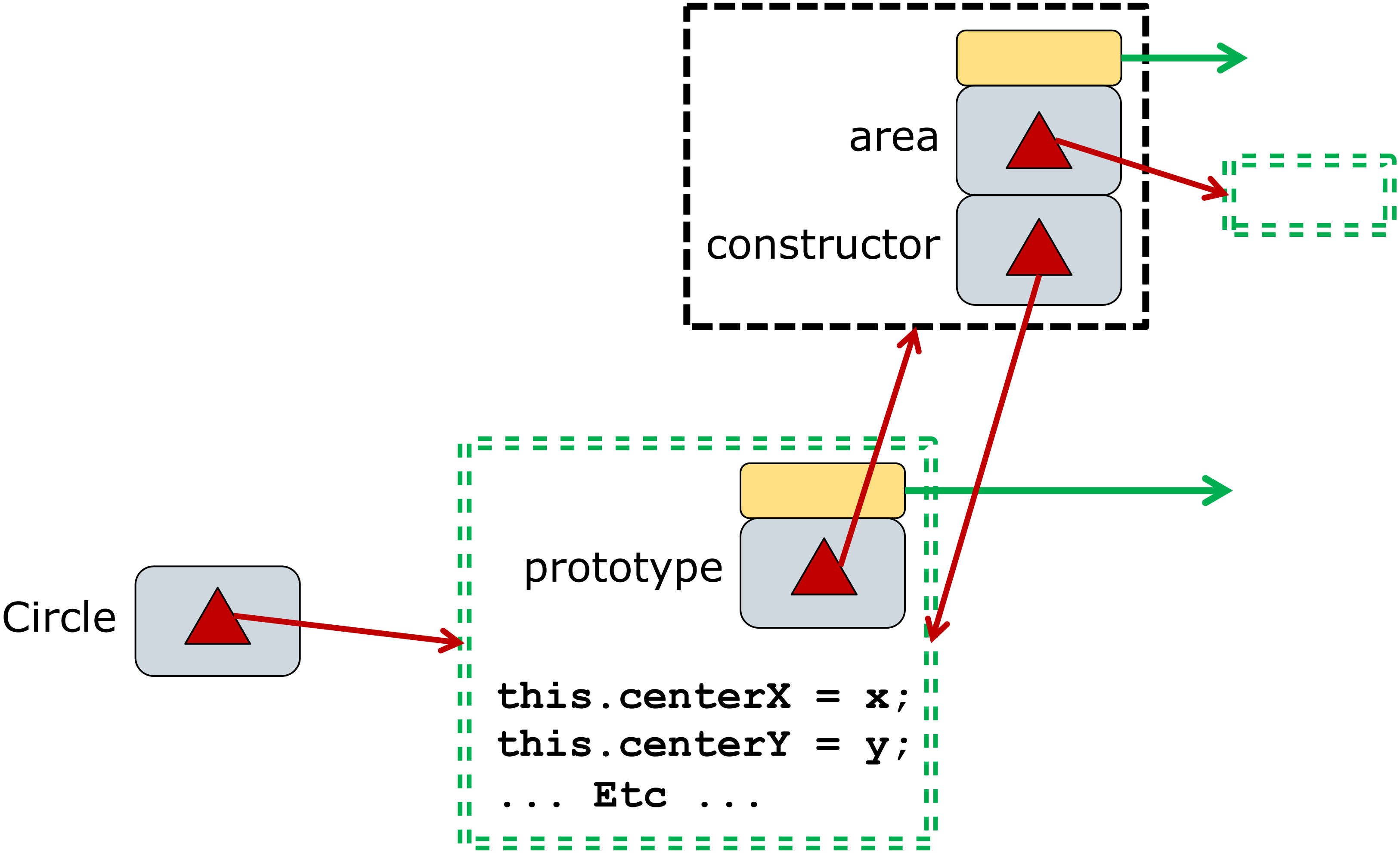
prototype

Circle

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

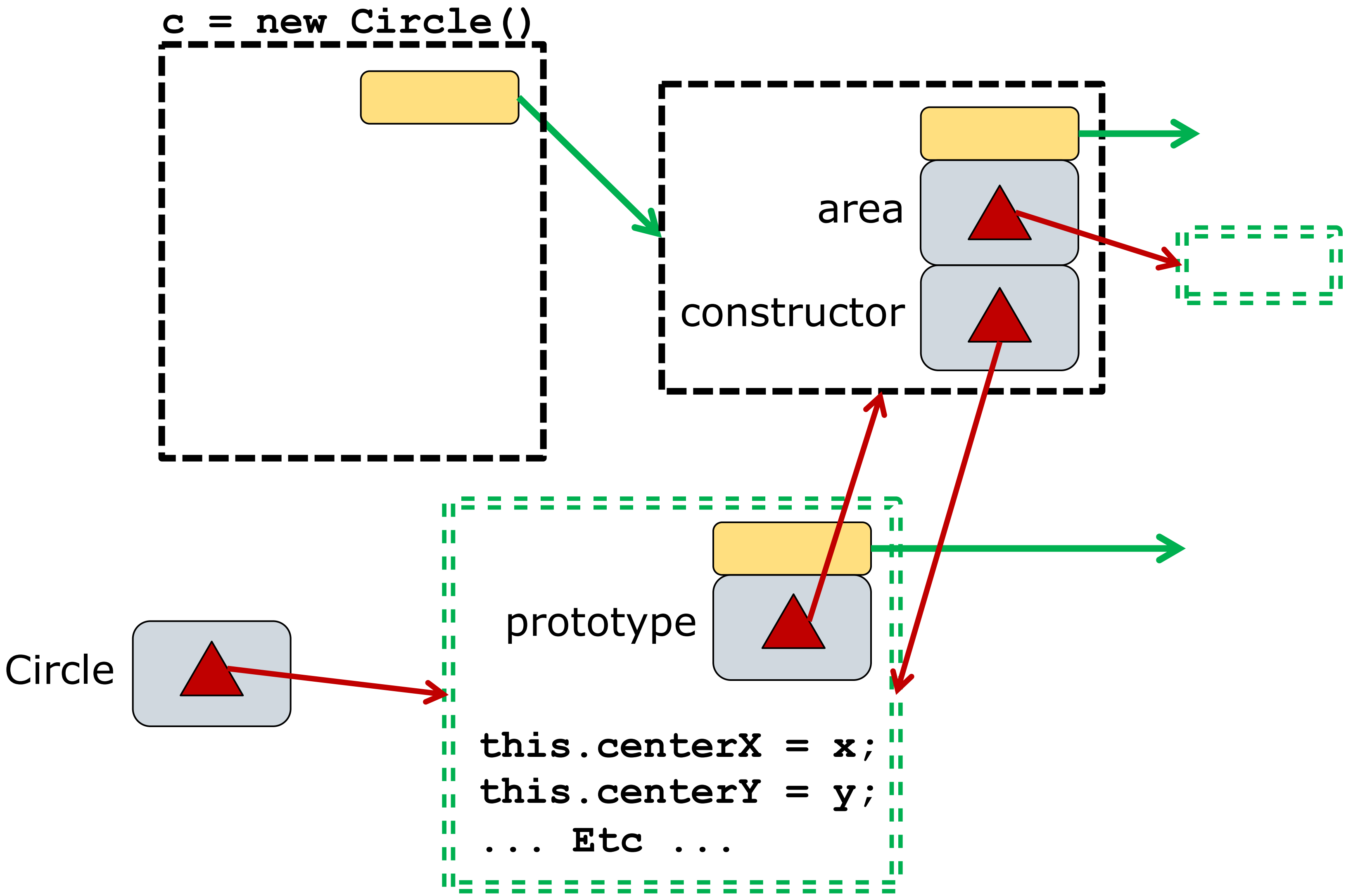
Prototypes And Constructors

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



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

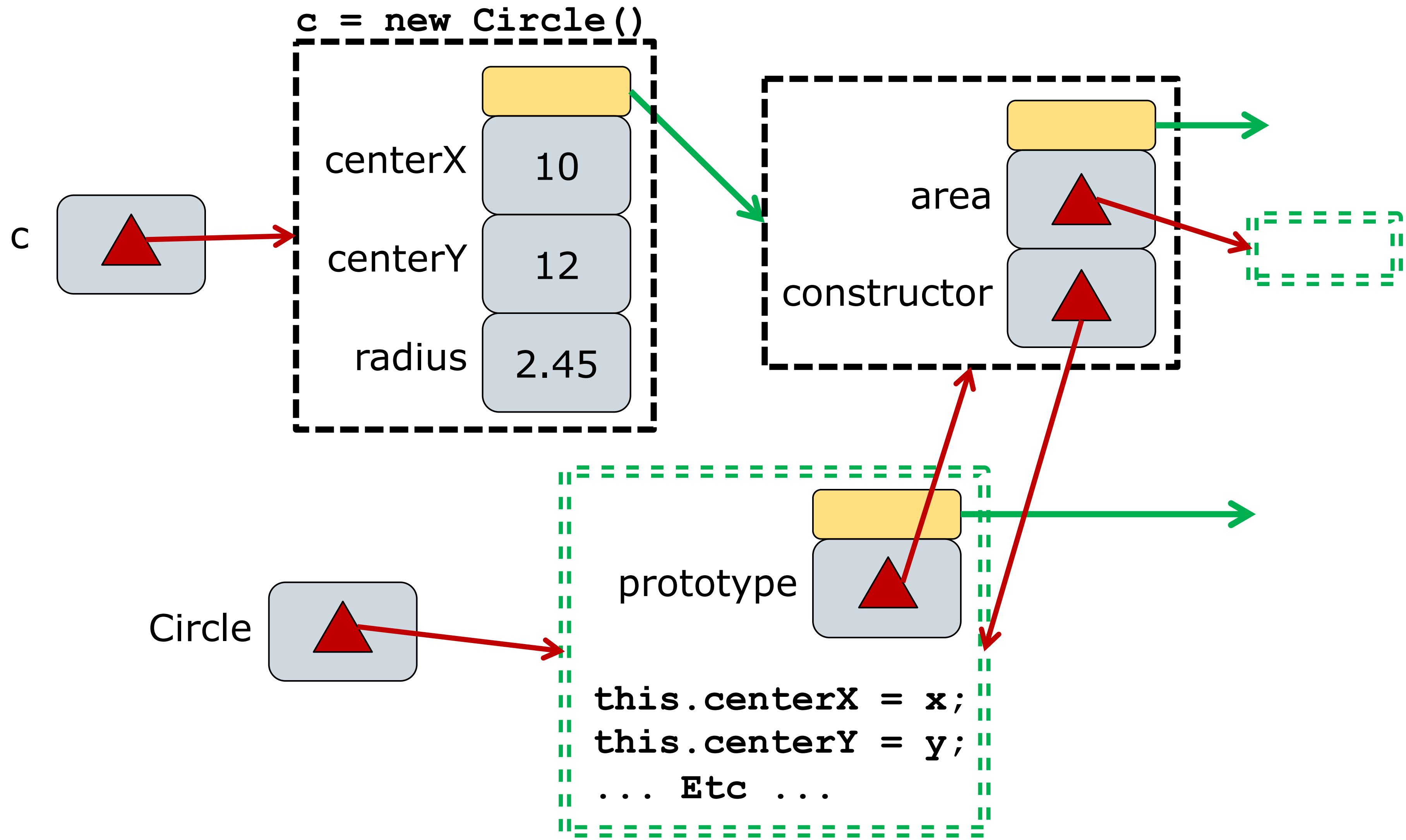
Prototypes And Constructors



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

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

Prototypes And Constructors



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

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

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
```

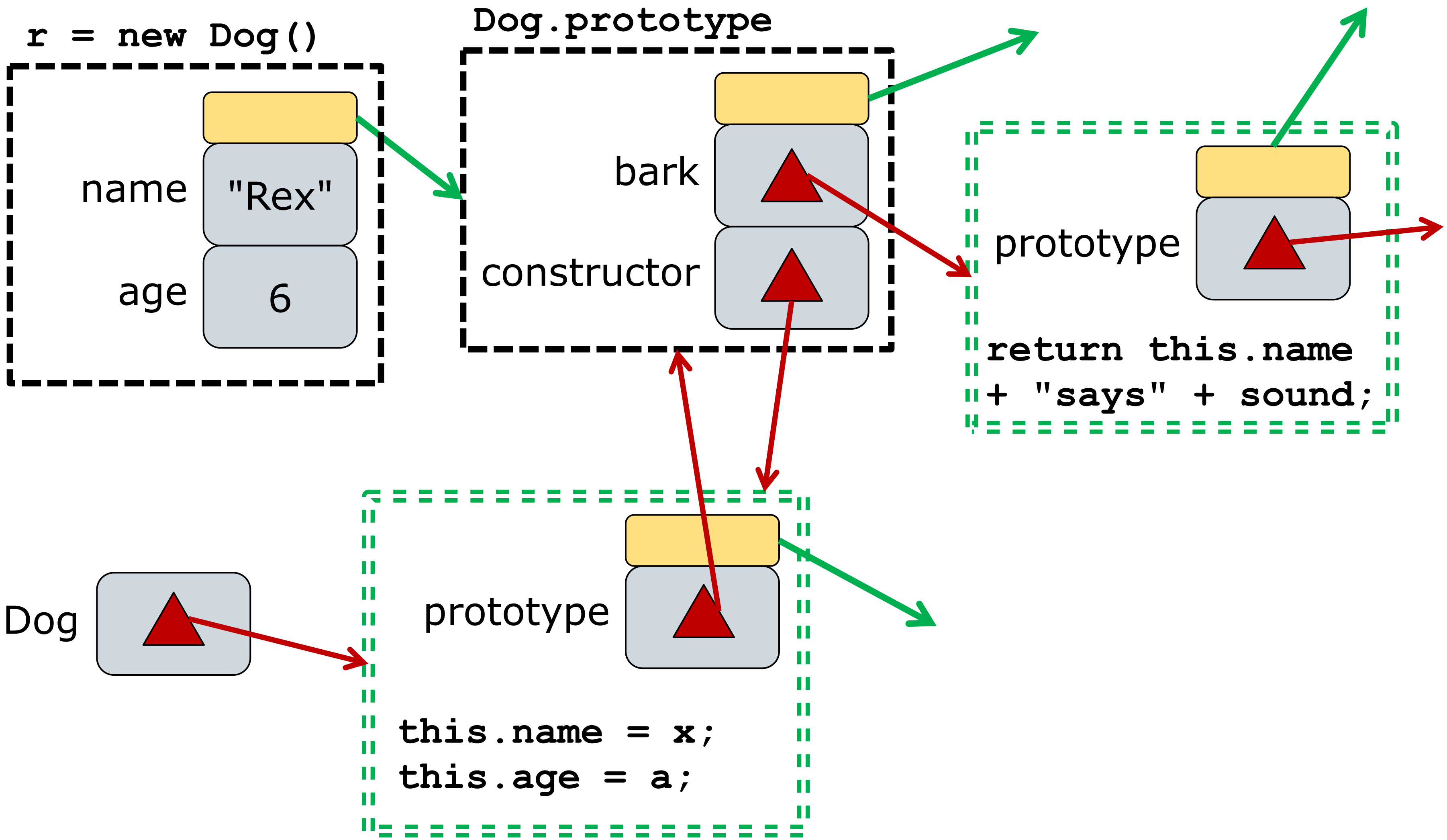
Idiom: Methods in Prototype

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

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


// better: extend existing prototype
```

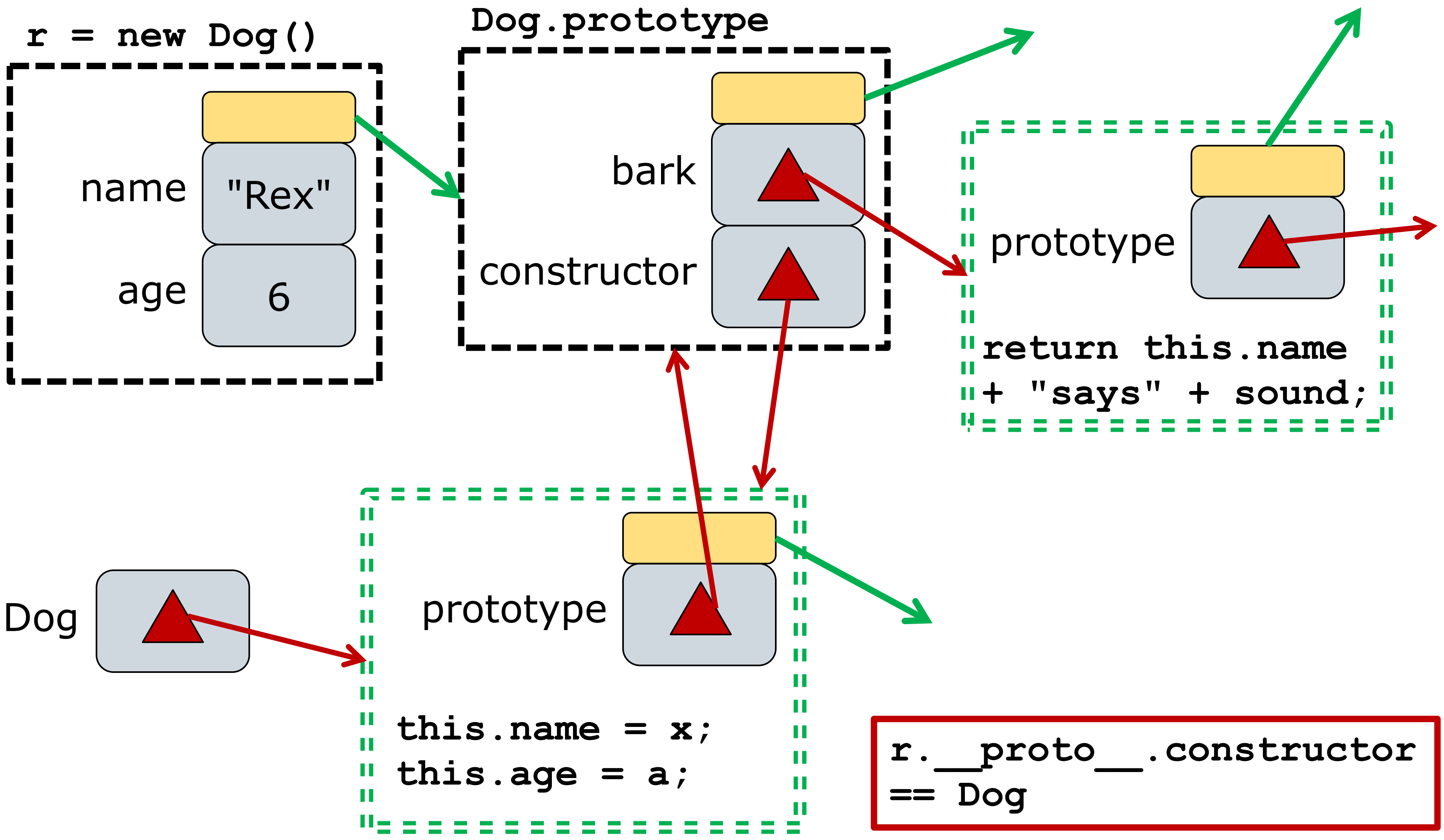
Idiom: Methods in Prototype

```
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



Meaning of `r instanceof Dog`

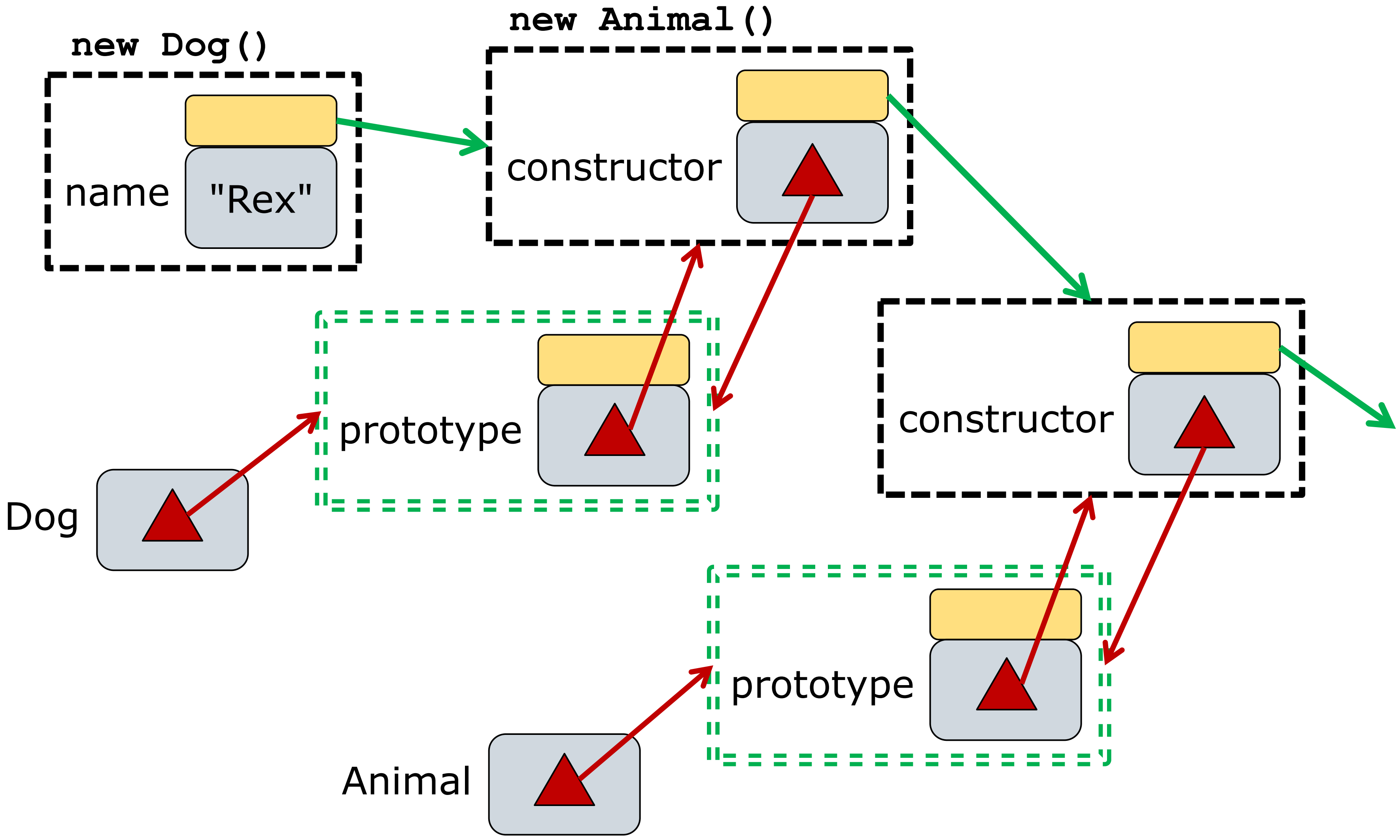


```
r.__proto__.constructor
== Dog
```

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


new Dog()
name
"Rex"
new Animal()
constructor
prototype
Dog
constructor
prototype
Animal