CSCI-GA.3033.003 Scripting Languages

10/07/2013

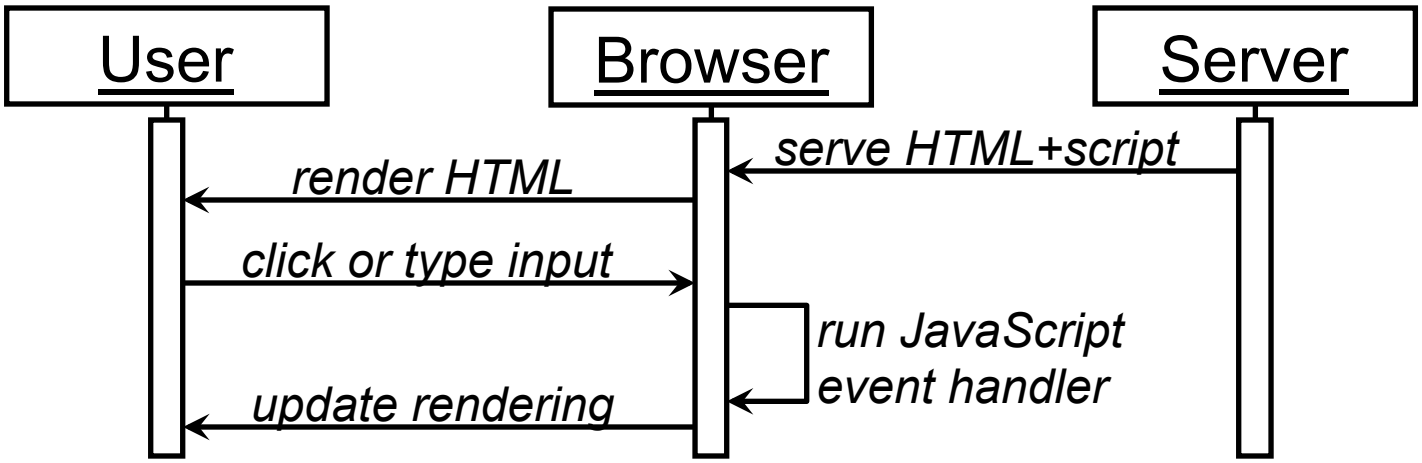
Client-Side Scripting (JavaScript)

Outline

- JavaScript Basics
- Prototypes

About JavaScript

- Prototype-based object oriented language
  - There are no classes (unlike Java, C++, etc.)
  - Object inherits from prototype object
  - Similar to Self programming language
- Event-driven: callbacks for browser actions
- Mostly used for client-side scripting



Related Languages

- JavaScript is not related to Java
  - Originally LiveScript, by Brendan Eich at Mozilla
  - Name change is Netscape/Sun marketing ploy
- 1999: ECMAScript v.3 = JavaScript 1.5
- Abandoned: ECMAScript v.4 = JavaScript 2.0
  - Would have had classes and gradual typing
- Current: ECMAScript v.5 = JavaScript 1.8.5
- Dialects: JScript (IE); ActionScript (Flash)

Lexical Peculiarities

- Embedded in HTML **`<script>`**…**`</script>`**
- Case sensitive
- No sigils, no interpolation
- Semicolon optional at end of line, that can cause bug when prefix valid statement
- Single-line comments: **`//`**…, or at start **`<!--`**…
- Multi-line comments: **`/*`**…**`*/`**
- Literals: **`"s"`**, **`'s'`**, **`true`**, **`null`**, RegExp **`/`**…**`/`**, Object **`{x:1, y:2}`**, Array **`[1,2,3]`**

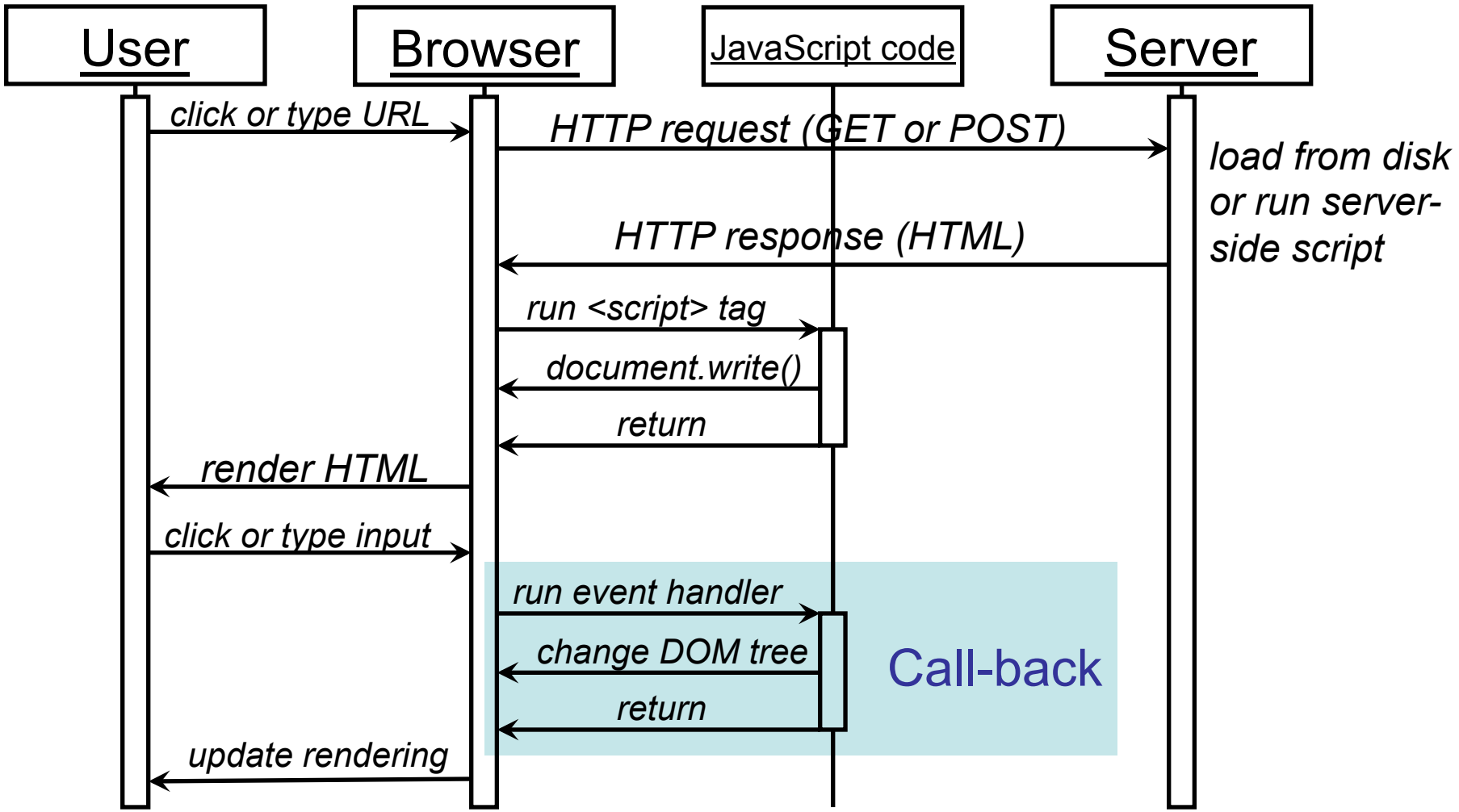
How to Write + Run Code

- Put in local file, then File→Open in web browser:

```
<html><head>
  <meta http-equiv="Content-Script-Type" content="text/javascript" />
  <script>
    function greet() {
      var who = document.question.who.value;
      var answer = document.getElementById("answer");
      answer.innerHTML = "Hi, " + who + ".";
    }
  </script>
</head><body>
  <span class="result" id="answer"></span>
  <form name="question">
    Who shall be greeted:
    <input type="text" name="who" onchange="greet();">
    <input type="button" value="Greet" onclick="greet();">
  </form>
</body></html>
```

Concepts

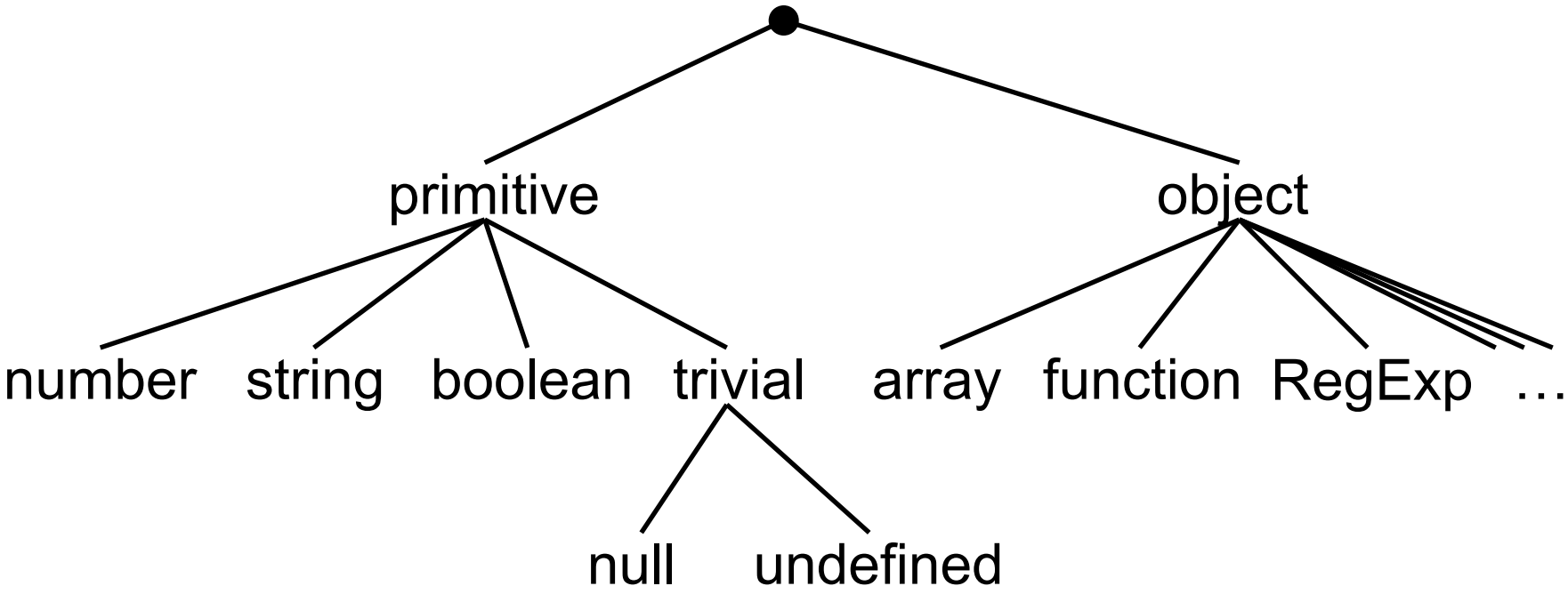
Call-backs



Input and Output

- Input:
  - Call-backs triggered by events,
    when user interacts with HTML **`<form>`** etc.
  - Reading information from DOM tree
- Output
  - **`document.write("…")`**
    - At HTML parse time: in-place insert
    - Later from event handler: overwrite document(!)
  - DOM tree manipulation:
    `document.getElementById("…").innerHTML = "…"`
  - Browser interaction, e.g., pop-up: **`alert("…")`**

Types


primitive
object
number
string
boolean
trivial
array
function
RegExp
…
null
undefined

Variable Declarations

Implicit	`b = 5;`	Reading an undeclared variable is a runtime error
Explicit	`var x;` `var i=0, msg="hi";`	Declaration without assignment is **`undefined`** value

- Scope of implicit declaration is global
- Scope of explicit declaration is local to function (there is no block scope)

JavaScript

Type Conversions

Value		Boolean	Number	String	Object
`undefined`		**`false`**	**`NaN`**	`"undefined"`	Error
`null`		**`false`**	`0`	**`"null"`**	Error
Boolean	**`false`**	Identity	`0`	`"false"`	Boolean object
	`true`		`1`	**`"true"`**	
Number	`0`	**`false`**	Identity	`"0"`	Number object
	`NaN`	**`false`**		**`"NaN"`**	
	Other	**`true`**		String	
String	`""`	**`false`**	Number or **`NaN`**	Identity	String object
	Other	**`true`**			
Object		**`true`**	**`valueOf()`**, **`toString()`**, or **`NaN`**	**`toString()`**	Identity

JavaScript

Operators

`.`, `[]`, `()`, **`new`**			Member/array/call access, creation
`++`, `--`, `+`, `-`, `~`, `!`, **`delete`**, **`typeof`**, **`void`**	1		**`delete`**: remove property from object; **`typeof`**: return type; **`void`**: discard value
`*`, `/`, `%`	2	L	Multiplicative
`+`, `-`	2	L	Additive; `+`: add numbers/concat strings
`<<`, `>>`, `>>>`	2	L	Bitwise shift
`<`, `<=`, `>`, `>=`, **`instanceof`**, **`in`**	2	L	Comparison; **`in`**: check membership
`==`, `!=`, `===`, `!==`	2	L	Identity; `===`, `!==`: equal value+type
`&`	2	L	Bitwise and
`^`	2	L	Bitwise xor
`|`	2	L	Bitwise or
`&&`	2	L	Logical and
`||`	2	L	Logical or
`?:`	3	R	Conditional
`=`, `+=`, `-=`, ...	2	R	Assignment
`,`	2	L	Multiple evaluation

JavaScript

Control Statements

Conditional	**`if (`***expr***`)`** … [**`else`** …] **`switch(`***expr***`){case`** *expr***`:`**… … **`default:`**…**`}`**
Fixed-iter. loops	**`for (`***var* **`in`** *expr***`)`** …
Indefinite loops	**`for (`***expr***`;`** *expr***`;`** *expr***`)`** … **`while (`***expr***`)`** … **`do`** … **`while(`***expr***`);`**
Unstructured control	**`break`** [*labelName*]**`;`** **`continue`** [*labelName*]**`;`** **`return`** [*expr*]**`;`** **`throw`** *expr***`;`**
Other	**`try {`**…**`}`** [**`catch(`***id***`){`**…**`}`**] [**`finally{`**…**`}`**] **`with (`***expr***`)`** …

Writing Subroutines

- Declaration: **`function`** *id* **`(`***arg*`*`**`)`** **`{`**`...`**`}`**
  - Declaration creates named function at compile time
  - *id* mandatory; visible externally in enclosing scope
- Arguments: *arg* ::= *id*
  - User may pass more or fewer than declared
  - Missing arguments have value **`undefined`**
  - All (declared and surplus) arguments are in **`arguments`** array-like object
- Literal: **`function`** [*id*] **`(`***arg*`*`**`)`** **`{`**`...`**`}`**
  - Expression creates anonymous function at runtime
  - *id* optional; only visible internally for recursion

JavaScript

Anonymous Functions

- Literal: **`function`** [*id*] **`(`***arg**`) {...}`**
- Example script:

```
<html><head>
  <meta http-equiv="Content-Script-Type" content="text/javascript"/>
  <script>
    function callFn(fn, x) { fn(x); }
    function printIt(it) { document.write("printIt " + it + "\n<br/>"); }
    //pass reference to named subroutine; prints "printIt Hello"
    callFn(printIt, "Hello");
    //pass reference to anonymous subroutine; prints "lambda Hi"
    callFn(function(it) { document.write("lambda " + it + "\n<br/>"); }, "Hi");
  </script>
</head><body>
</body></html>
```

Outline

- JavaScript Basics
- Prototypes

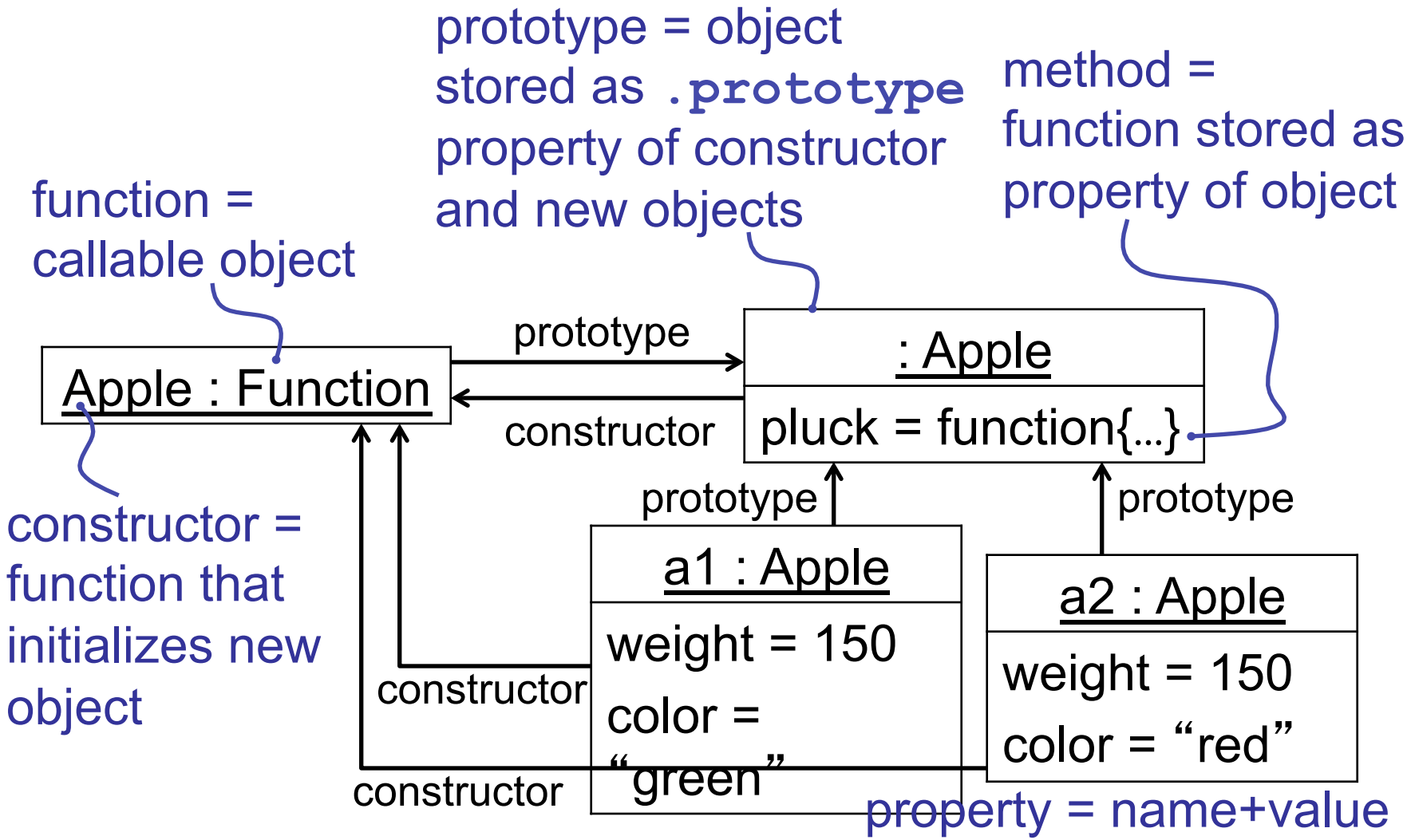
JavaScript

Using Objects

`a1 = new Apple(150, "green");` `a2 = new Apple(150, "green");`	Constructor calls
`a2.color = "red";`	Property access
`document.write(` `  a1.prepare("slice") + " \n");` `document.write(` `  a2.prepare("squeeze")+" \n");`	Method calls

Concepts

Prototypes and Constructors



JavaScript

Defining ~~Classes~~ Constructors

- All functions f are objects with a prototype property
- f.`prototype` initially points to an almost empty object
- f.`prototype.constructor` points back to f

```
function Apple(weight, color) {
  this.weight = weight;
  this.color = color;
}
```

```
Apple.prototype.pluck = function() {
  return this.color + " apple";
}
```

Convention:

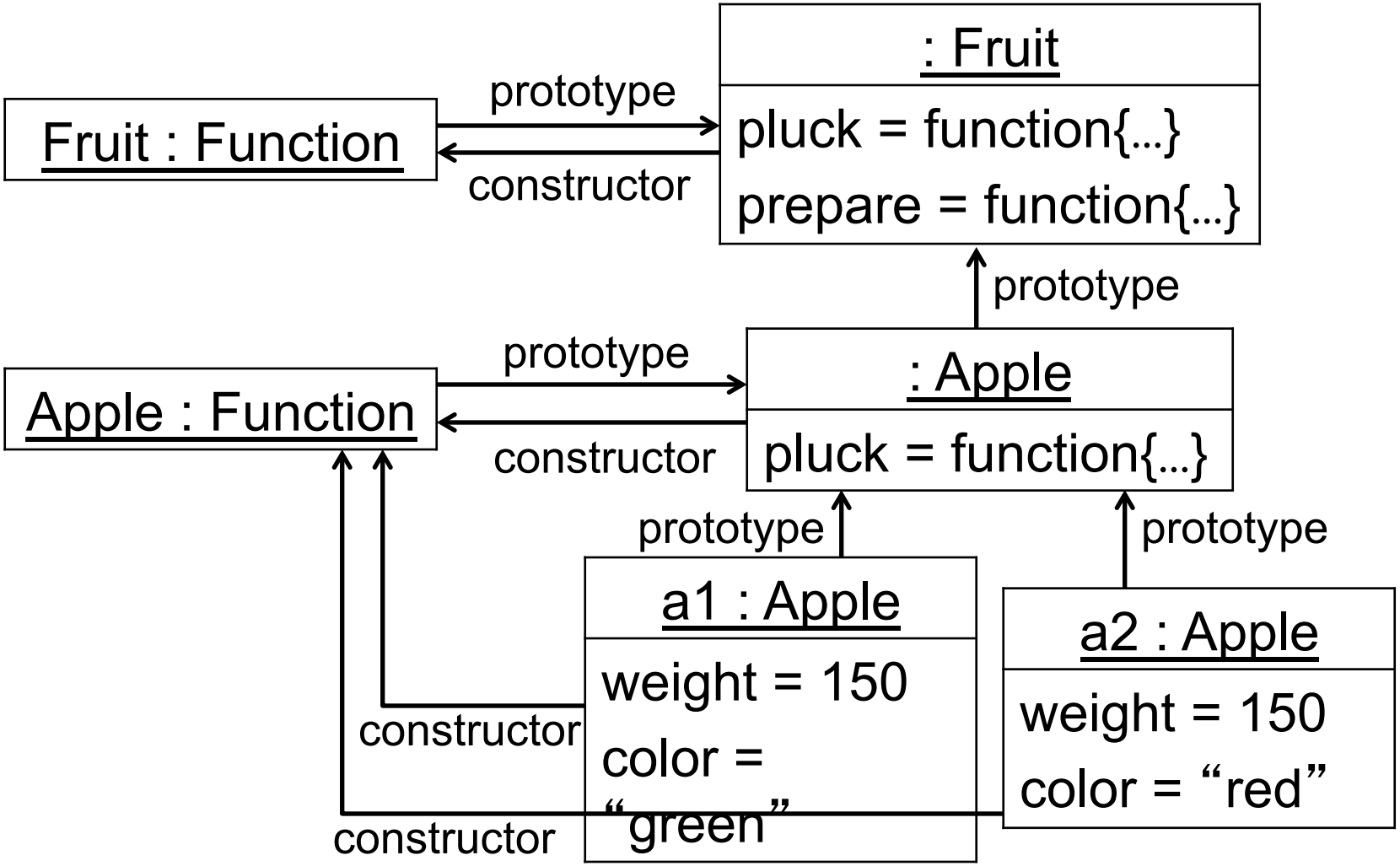
- Assign object properties (fields) in constructor
- Assign prototype properties (methods) before any calls to constructor

JavaScript

Operators on Objects

new *C*(…)	• Create new empty object *o* • Set *o*.**prototype** = *C*.**prototype** • Set *o*.**constructor** = *C*.**prototype.constructor** • Call *C*(…), pass *o* as value for **this** • Return result of *C*(…) or *o* if none
o.*p*	• If object *o* has property *p*, return it • Otherwise, look in *o*.**prototype**
o.*p* = *expr*	• If object *o* has property *p*, assign it • Otherwise, create it and assign it
o **instanceof** *C*	• If *o*.**constructor** is *C*, return **true** • Otherwise, look in *o*.**prototype**

Prototype Inheritance



JavaScript

Inheritance in JavaScript

```
function Fruit(weight) {
  this.weight = weight;
}
Fruit.prototype.pluck = function() {
  return "fruit(" + this.weight + "g)";
}
Fruit.prototype.prepare = function(how) {
  return how + "d " + this.pluck();
}

function Apple(weight, color) {
  this.weight = weight;
  this.color = color;
}
Apple.prototype = new Fruit();
delete Apple.prototype.weight;
Apple.prototype.constructor = Apple;
Apple.prototype.pluck = function() {
  return this.color + " apple";
}
```

- Constructor assigns fields
- Top-level assigns methods
- Inherit from Fruit.prototype
- Remove spurious property
- Enable instanceof checks

Receiver Object

`y = o.f(x)`	During `f`, `this` == `o`
`y = f.call(o, x)`	
`y = f.apply(o,[x])`	
`y = new f(x)`	During `f`, `this` == new object
Top-level code (outside any function)	`this` == global object
`y = f(x)`	During `f`, `this` == global object

Soap-box

Evaluating Prototypes

Strengths

- Orthogonality
  - No class/object duality
- Flexibility
  - Can emulate classes
  - But don't have to
  - E.g., can borrow (copy) method instead of inheriting

Weaknesses

- Lack of familiarity
- Lack of static guarantees
  - Less error checking
  - Harder to optimize (but, Self pioneered many optimizations later used for Java)

More Operators on Objects

o`["`*p*`"]`	• If object *o* has property *p*, return it • Otherwise, look in *o*`.prototype`
o`["`*p*`"] =` *expr*	• If object *o* has property *p*, assign it • Otherwise, create it and assign it
`"`*p*`" in` *o*	• If object *o* has property *p*, return `true` • Otherwise, look in *o*`.prototype`
`typeof` *o*	• String describing JavaScript type, e.g., `typeof new Fruit == "object"`
`delete` *o*`.`*p*	• If object *o* has property *p*, set its value to `undefined` • `for(`*x* `in` *o*`)` loop omits `undefined` *p*

Last Slide

- Nothing to announce
- Today's lecture
  - Client-side scripting
  - JavaScript
  - Prototypes
  - Closures
- Next lecture
  - JavaScript
  - Closures