IOOR

Lecture 4

Inheritance vs. delegation, Actor-based languages,

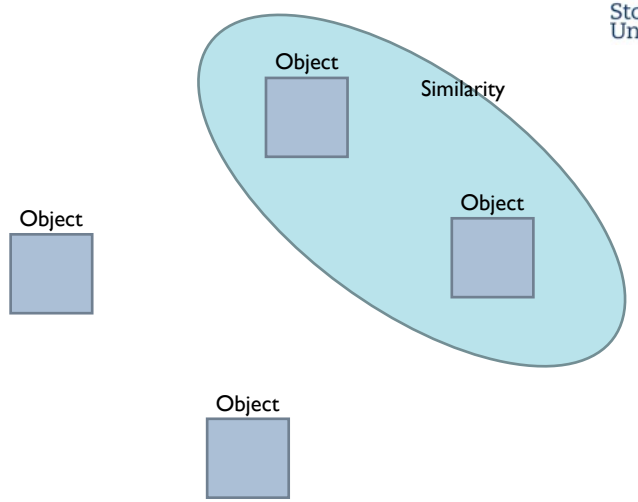
Methods vs messages,

Course council!

Prototype-based PLs

- Invented after class-based languages in the 70'ies
- Replaces class instantiation with copying existing objects
- Replaces inheritance with more flexible delegation
- Cloned objects can can change invariantly of each other
- Also called:
  - Instance-based, Prototype-Oriented, Class-less
- Examples of languages:
  - Self, Cecil, JavaScript, Io

Prototype-based



JavaScript

- JavaScript is THE scripting language of the Web
- JavaScript is used in millions of Web pages to add functionality, validate forms, detect browsers, and much more
- But:
  - JavaScript has no direct relationship to Java
  - JavaScript can be used for other things than scripting browsers

JavaScript Syntax

Comments:	`// single line comment`
	`/* multi line`
	`comment */`

Identifiers:	First character must be a letter, _, or $; subsequent characters can be digits: `i, v17, $str, __proto__`

Basic literals:	`'a string', "another string", "that's also a string"`
	`17, 6.02e-32`
	`true, false, null, undefined`

Object literals:	`var point = { x:1, y:2 }`
	empty: `{}`
	nested: `var rect = {`
	`upperLeft: { x:1, y:2 },`
	`lowerRight: { x:4, y:5 } }`

Function literals:	`var square =` `function(x) { return x*x; }`
Array literals:	`[1,2,3]` `[]`
Operators:	assignement: = equal: == ! ! strict equal: ===

Object Properties

Reading properties	`var book = { title:'JavaScript' };` `book.title; //=>'JavaScript'`
Adding new properties (at runtime)	`book.author = 'J. Doe';` `'author' in book; //=>true`

Inspecting objects	`var result = '';`
	`for (var name in book) {`
	`result += name + '=';`
	`result += book[name] + ' ';`
	`};`
	
	`//=>title=JavaScript author=J. Doe`

Deleting properties	`delete book.title;` `'title' in book; //=>false`

Slots in PBLs

- Slots are simply storage locations located in objects
- Slots can be divided into two types:
  - Data slots, holding data items
  - Method slots, holding methods
- Methods are stored in exactly the same way as data items

Methods

- At runtime the keyword `this` is bound to the object of the method

```
var obj = { counter:1 };
obj.increment = function(amount) {
  this.counter += amount;
};
obj.increment(16);
obj.counter; //=> 17
```

- Accessing (vs. executing) methods

```
var f = obj.increment; typeof f; //=>
'function'
```

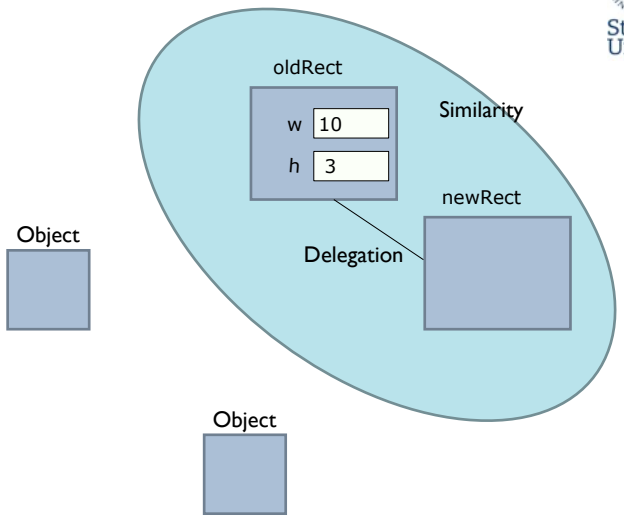

Delegation

- When an object receives a message it looks for a matching slot, if not found, the look-up continues its search in other known objects
- Typically, the search is done in the object's "parent", in its "parent's" "parent" and so on
- In JavaScript, an object delegates to its prototype object (the Mozilla interpreter allows one to access the prototype through the property __proto__)

Delegation, cont'd

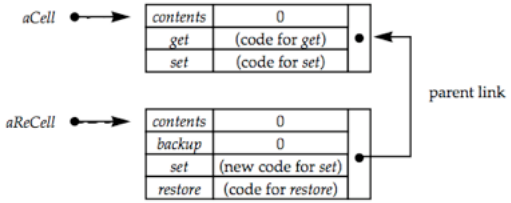
```
var oldRect = { width:10, height:3 };
var newRect = {};
newRect.__proto__ = oldRect;


"width" in newRect; //=>true
newRect.hasOwnProperty("width"); //=>false


newRect.width; //=>10
newRect.foo; //=>undefined
```


Prototype-based



Delegation

- As opposed to inheritance, delegation can be manipulated dynamically
- The method of the delegate will be executed in the scope of the original receiver
- Depending on the language, the number of possible delegates may differ

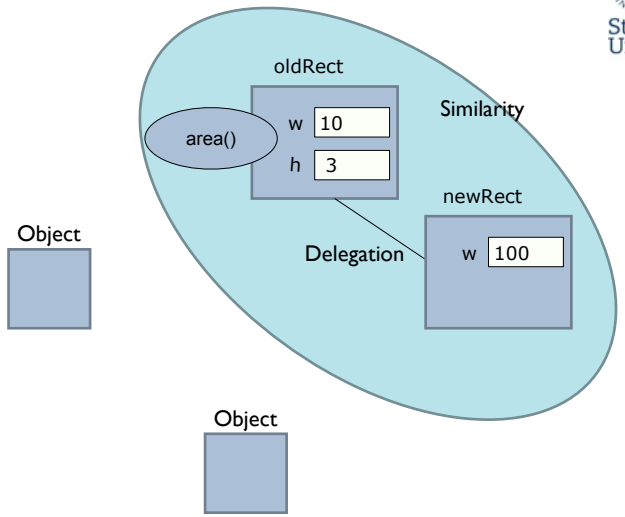


(Single-parent) Delegation

Delegation, cont'd

```
newRect.width = 100;

oldRect.area = function() {
   return this.width * this.height;
};

newRect.area(); //=>300
```


Prototype-based



Use of delegation

- Delegation — executing a method of some other object but in the context of self
- A lot more powerful than mere forwarding
- Delegation can be used to implement inheritance but not vice versa
- Very powerful — delegates are not known statically as in inheritance and can change whenever

Constructor Functions

- Constructors are functions that are used with the `new` operator to create objects

```
function Rectangle(w, h) {
   this.width = w;
   this.height = h;
   this.area = function() {
      return this.width * this.height;
   };
};

rect = new Rectangle(3,4);
rect.area(); //=>12
```

- The operator `new` creates an object and binds it to this in the constructor. By default the return value is the new object.

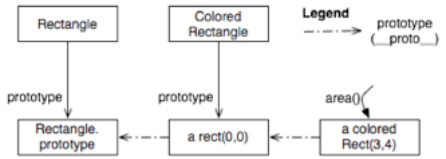
Constructor.prototype

- Each constructor has a prototype property (which is automatically initialised when defining the function)
- All objects created with a constructor share the same prototype

```
function Rectangle(w, h) {
   this.width = w;
   this.height = h;
};

Rectangle.prototype.area = function() {
   return this.width * this.height;
};
```


Constructor.prototype

```
   ...

function ColoredRectangle(w, h, c) {
   this.width = w;
   this.height = h;
   this.color = c;
};

ColoredRectangle.prototype = new Rectangle(0,0);

coloredRect = new ColoredRectangle(3,4,'red');

coloredRect.area();
```



Predefined Objects

- Global functions: Array, Boolean, Date, Error, Function, Number, Object, String,... eval, parseInt, ...
- Global objects: Math

Extending Predefined Objects

- Extending all objects:

```
Object.prototype.inspect = function() {
   alert(this);
};

'a string'.inspect();
true.inspect();
(new Date()).inspect();
```

- The last object in the prototype chain of every object is `Object.prototype`

The arguments object

```
function concat(separator) {
   var result = "";
   for (var i = 1; i < arguments.length; i++)
      result += arguments[i] + separator;
   return result;
};

concat(";", "red", "orange", "blue");
// =>"red;orange;blue;"
```


Other Prototype-based Languages

- Basic mechanisms
  - Object creation:ex nihilo, cloning, extension
  - Object representation(slots in JavaScript, Self, Io vs. attributes and methods in Agora, Kevo)
- Delegation
  - Double delegation in Io/NewtonScript
  - Multiple prototypes(aka.parents) in Self
  - Can prototype link be changed at runtime?
- Organization of programs (prototypical instance, traits, ...)

Benefits of prototypes

- Simple model, simpler than the class-based
- No use for special "inheritance" relations in the language
- Very flexible and expressive
- Changing prototypes to reflect state is a powerful concept
- Delegation is very powerful
- Handles special cases very well

Performance

- Sharing data and copy-on-write Method caches
- Inheritance (at least in static cases) costs memory in many slots
- Locality of reference if the methods are actually in the object

Prototypes vs. Classes

- Classes are static—requirements are not
- Unless you can predict all future requirements up front, class hierarchies will evolve
- Evolution of base classes is tricky and might break subclasses
- Eventually, refactoring or redesign is needed
- It is not uncommon to design a class that is only to be instantiated once. [Liebermann86]

Concurrent Programming

Threads

- Threads are a seemingly straightforward adaptation of the dominant sequential model of computation to concurrent systems.
- Languages require little or no syntactic changes to support threads, and operating systems and architectures have evolved to efficiently support them.

Lost Update Problem

Process 1	Process 2
`a = acc.get()`	
`a = a + 100`	`b = acc.get()`
	`b = b + 50`
	`acc.set(b)`
`acc.set(a)`	

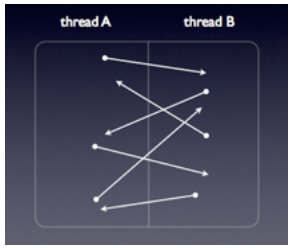
Deadlock Problem

Process 1	Process 2
`lock(A)`	`lock(B)`
`lock(B)`	`lock(A)`

```
... Deadlock! ...
```


The Problem With Threads

- Although threads seem to be a small step from sequential computation, in fact, they represent a huge step
- They discard essential and appealing properties of sequential computation:
  - understandability
  - predictability
  - determinism



Actor-based Languages

Actors

- Hewitt et al in the early 1970's
- Actor methodology was developed as an attempt to understand complex systems -- AI systems, parallel or distributed systems
- Languages:
  - io, Erlang, Scala, ...

Actors -- Fundamental Concepts

- Every object is an Actor -- has a mail address and a behaviour
- Messages can be exchanged between actors, which will be buffered in the mailbox
- When receiving a message an Actor can:
  - send messages to other actors (an actor may send messages to itself)
  - create new actors
  - designate the behaviour to be used for the next message received

Actors -- Fundamental Concepts, cont'd

- Communication with other Actors occur asynchronously
  - sender does not wait for a message to be received upon sending it
  - no guarantees in which order messages will be received by the recipient
- All communication is handled through messages, no shared state

Actors in io

- Any object can be sent an asynchronous message by placing a @ before the message name
- This returns a future object which will become the return value "when it is ready"
- If a future is accessed before the result is ready, the accessor will be put to wait until the result is ready
- When an object receives an asynchronous message it puts the message in its queue and starts to process the queue

Actors in io, cont'd

- An object processing a message queue is called an "actor".
- Queued messages are processed sequentially in a first-in-first-out order
- Control can be yielded to other actors by calling yield -- It's also possible to pause and resume an actor
- Blocking operations such as reading on a socket will automatically unschedule the caller until the data is ready or a timeout or error has occurred.

Example Using io

```
o1 := Object clone
o1 name := "One"
o1 test := method(for(n, 1, 3, write( name, " ", n, " \n") yield))
o2 := o1 clone
o2 name = "Two"

// @ means send an asynchronous message
o1 @test; o2 @test
// wait for the messages to get processed
while(Scheduler waitForCorosToComplete, yield)
```

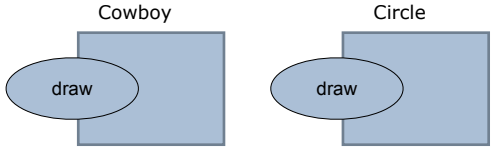

Example Using Erlang

```
1. -module(counter).
2. -export([run/0, counter/1]).
3.
4. run() ->
5.     S = spawn(counter, counter, [0]),
6.     send_msgs(S, 100000),
7.     S.
8.
9. counter(Sum) ->
10.    receive
11.        value -> io:fwrite("Value is ~w~n", [Sum]);
12.        {inc, Amount} -> counter(Sum+Amount)
13.    end.
14.
15.send_msgs(_, 0) -> true;
16.send_msgs(S, Count) ->
17.    S ! {inc, 1},
18.    send_msgs(S, Count-1).
19.
20.% Usage:
21.%    1> c(counter).
22.%    2> S = counter:run().
23.%       ... Wait a bit until all children have run ...
24.%    3> S ! value.
25.%    Value is 100000
```


Why is the Actor Model Important Now?

- The importance of concurrency is growing with the growing number of multi-processor machines
- The Actor model faces issues including the following:
  - scalability -- the challenge of scaling up concurrency both locally and non-locally
  - transparency -- bridging the gap between local and non-local concurrency
  - inconsistency -- inconsistency is the norm because all very large knowledge systems about human information system interactions are inconsistent

message ≠ method ≠ function




```
c draw
```


Messages

- Objects send and receive messages
- The response to a message is executing a method
- Which method to use is determined by the receiver at run-time.

References

- Iain Craig, "The Interpretation of Object-Oriented Programming Languages", 2nd edition, Springer Verlag, 2002.
- Gul Agha, "An Overview of Actor Languages"
- Edward A. Lee , "The Problem with Threads", 2006.
- Martín Abadi and Luca Cardelli, "A Theory of Objects", Springer Verlag, 1996.
- io: http://www.iolanguage.com
- Anton Eilëns, Principles of Object-Oriented Software Development, 2nd edition. Addison-Wesley, 2000.
- Kim Bruce, Foundations of Object-Oriented Languages: types and semantics, MIT Press, 2002.

Stockholm University

References

- Timothy Budd,"An Introduction to Object- Oriented Programming", 2nd edition. Addison-Wesley, 2000.
- Ian Joyner, "Objects Unencapsulated", Prentice-Hall, 1999.
- Henry Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems", 1986.
- James Noble and Brian Foote, "Attack of the Clones", Proceedings of the 2002 conference on Pattern languages of programs.
- D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, 1972.