16—Object-Oriented Programming

CS1101S: Programming Methodology

Martin Henz

October 17, 2012

Generated on Wednesday 17 October, 2012, 14:14

The Truth About JediScript Lists

```
// pair constructs a pair using a two-element array
// LOW-LEVEL FUNCTION, NOT JEDISCRIPT
function pair(x, xs) {
    return [x, xs];
}
```

Terminology

We call these things "vectors". We are often impose limitations on ourselves when using "vectors". This is called "array discipline".

Vectors in JavaScript

Definition

Vectors in JavaScript are tables that enjoy special syntactic support.

Creating an empty vector

```
var my_vector = []; // creating an empty vector
```

Adding Fields

Syntax for adding fields

We can add fields using the following syntax:

```
// creating an empty vector
var my_vector = [];

// adding a field with key 42
my_vector [ 42 ] = "some string";

// accessing the field
alert( my_vector [ 42 ] );
```

What if the key has not been added yet?

Access using non-existing keys

```
// creating an empty vector
var my_vector = [];

// accessing using non-existent key
alert( my_vector [ 88 ] );

// shows "undefined"
```

What can be used as key?

Anything!

We can use any data structure as a keys

```
// creating an empty vector
var my_vector = [];

// adding a field with key "fortytwo"
my_vector [ "fortytwo" ] = "some string";

// accessing the field
alert( my_vector [ "fortytwo" ] );
```

Array Convention for Vectors

Array Convention

If the keys of a vector are integers ranging from 0 to a given number, the vector is called an *array*.

JediScript Week 10

All vectors in JediScript are arrays.

JavaScript follows this convention

The built-in functions on vectors assume that vectors are arrays. For example, `my_vector.length` ignores fields that are not non-negative integers.

Back to pair

```
function pair(x, xs) {
    return [x, xs];
}
```

Literal arrays

JavaScript supports the creation of a literal array, without a need to list the keys.

```
var my_array = [ "somestring", "someotherstring" ];
...my_array[0]...
```

What if we want keys that are not non-negative integers?

JavaScript support for strings as keys

JavaScript supports strings as keys in *objects*.

Example

If my_object is an object that has a field with key "my_field", you can use it like this:

```
// field access expression
... my_object["my_field"]...

// field assignment statement
my_object["my_other_field"] = 42;
```

Creating Objects

Syntax of creating objects

An empty object is created using the syntax {}.

Example Session

```
var my_object = {};
my_object["a"] = 13;
alert( my_object["a"] );
my_object["b"] = 42;
alert( my_object["b"] );
alert( my_object["c"] );
```

Syntactical Support for Objects

Objects will appear *a lot* in your programs. We need a syntax that avoids the quotation marks around the names of fields.

Syntactic Convention 1

If the key of a field is a string that looks like a JavaScript identifier, you can write

```
... my_object . my_field ...
```

instead of

```
... my_object [ "my_field" ] ...
```

Syntactical Support for Objects (cont'd)

Syntactic Convention 2

If the key of a field is a string that looks like a JavaScript identifier, you can write

```
my_object . my_field = ...;
```

instead of

```
my_object [ "my_field" ] = ...;
```

Syntactical Support for Objects (cont'd)

Syntactic Convention 3

We can construct objects *literally*, using the syntax `{...}`, by listing the key-value pairs, separated by ":" and ",".

Example

```
var my_object = { my_field : 42,
                  my_other_field : 88 };
```

Objects as pairs

```
function make_stack() {
    var stack = pair("stack", []);
    return stack;
}
function push(stack, x) {
    set_tail(stack, pair(x, tail(stack)));
}
// other functions
var my_stack = make_stack();
push(my_stack, 4);
```

Objects as functions

```
function make_account(initial) {
  var balance = initial;
  return function(message, amount) {
           if (message === "withdraw") {
             balance = balance - amount;
             return balance;
           } else { // message === "deposit"
             balance = balance + amount;
             return balance;
           }
}
var my_account = make_account(100);
my_account("withdraw", 50);
```

Objects as functions, a variant

```
function make_account(initial) {
  var balance = initial;
  return function(message) {
           if (message === "withdraw") {
             return function(amount){
                      balance = balance-amount;
                      return balance;
                    };
           } else { // message === "deposit"
             return ...;
           };
}
var my_account = make_account(100);
(my_account("withdraw"))(50);
```

Knowledge Representation View of Objects

- Aggregation
- Classification
- Specialization

Aggregation

```
var myVehicle = {max_speed: 85};
```

Classification

```
function new_vehicle(ms){
    return { max_speed: ms};
}

var my_vehicle = new_vehicle(85);
```

Specialization

```
function new_car(mp) {
    var c = new_vehicle(95);
    c . max_passengers = mp;
    return c;
}

var my_car = new_car(5);
```

Specialization

Usually, the concept of inheritance (class extension) achieves specialization in object-oriented languages.

Philosophy of Objects in JavaScript/JediScript

- Minimality: Introduce only three constructs (**new**, **this**, and Inherits)
- Flexibility: Supports many object-oriented programming techniques, not just the traditional class/object scheme

Constructor Functions

Constructors are ordinary functions

Any function in JavaScript can serve to create an object.

```
function Student() {
}
```

Creating Objects with "new"

Constructors are ordinary functions

The keyword **new** creates an object, referred to by "**this**" in the function, and allows method invocation (see later)

```
function Student() {
}
var peter = new Student();
```

Example using "this"

```
function Student(number) {
    this.matric_number = number;
    this.year_of_study = 1;
}
var peter = new Student("A0014234X");
```

Functions operating on objects

```
function set_year(student, y) {
    student.year_of_study = y;
}
var peter = new Student("A0014234X");
set_year(peter, 2);
```

Alternative: Methods

What are methods?

Methods are functions that are defined with the intention of being applied to objects through object invocation

```
function Student(number) {
    this.matric_number = number;
    this.year_of_study = 1;
}
function set_year(y) {
    this.year = y;
}
var peter = new Student("A001342X");
peter.set_year(2); // will this work?
```

Connecting methods to objects

Prototype field of Constructors

The prototype field of constructors are used for object invocation.

```
function set_year(y) {
    this.year = y;
}
Student.prototype.set_year = set_year;

Student.prototype.set_year =
    function(y) {
        this.year = y;
    };
```

Invoking Methods

```
var peter = new Student("A001342X");
peter.set_year(y);
```

Invocation mechanism

This special notation calls the function Student.prototype.set_year using the argument 2 as y, and the implicit argument peter as "this".

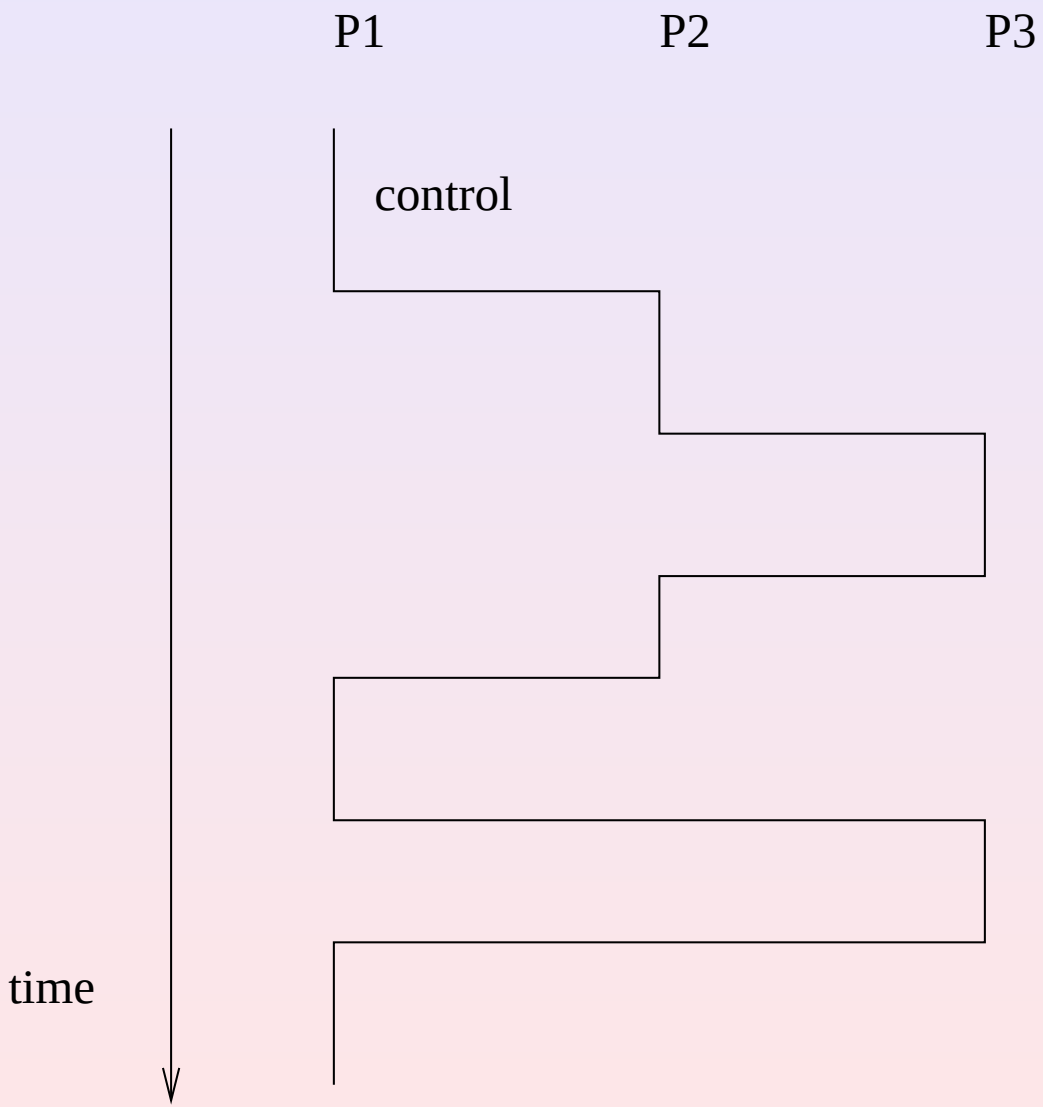
Inheritance

```
function GraduateStudent(s,n) {
    this.supervisor = s;
    Student.call(this,n);
}
GraduateStudent.Inherits(Student);
var paul = new GraduateStudent("Prof Leong",
                               "A0014234X");
```

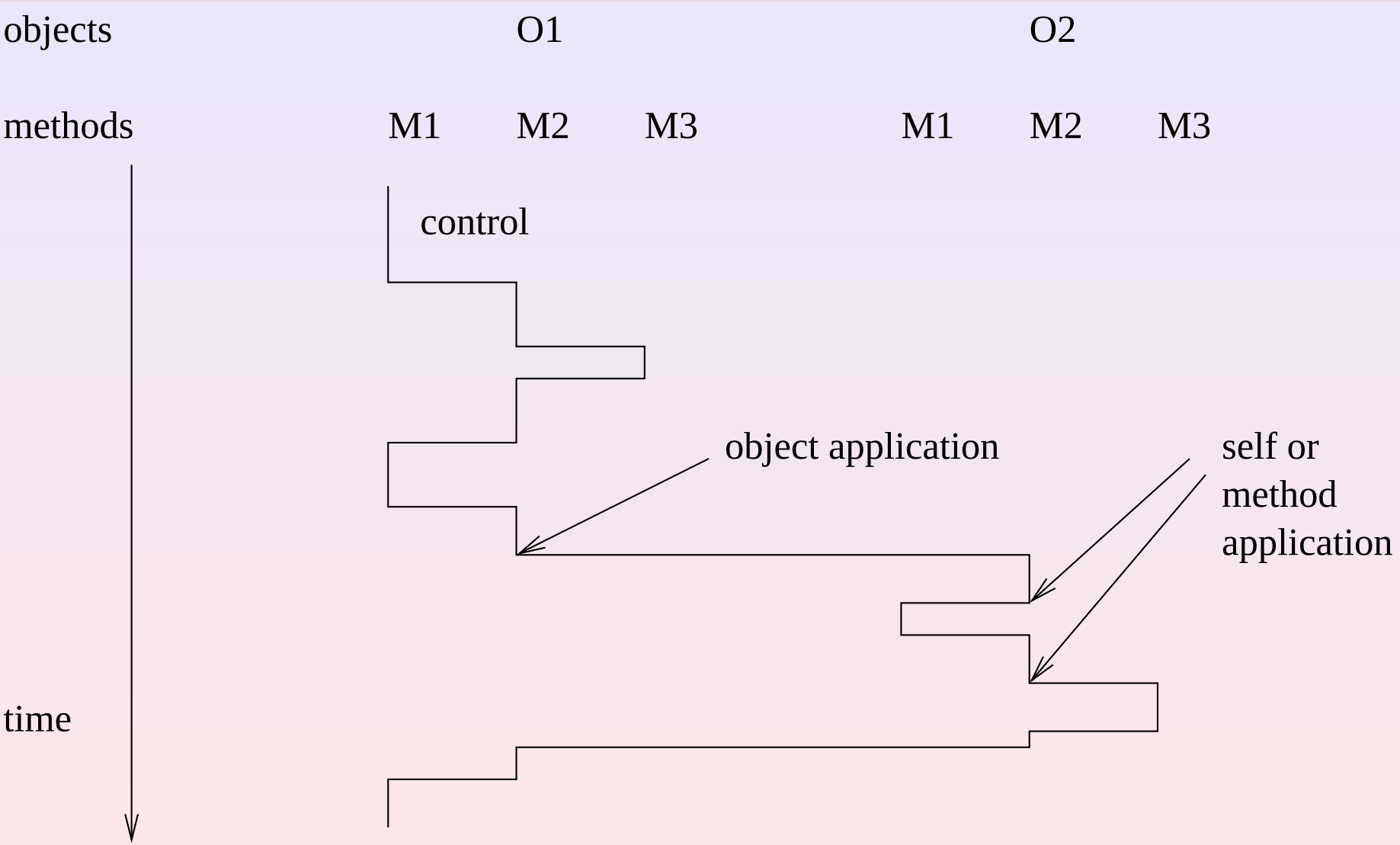
Overriding Methods

```
GraduateStudent.prototype.set_year =
    function(y) {
        if (y < 5) {
            alert("a very junior grad student");
        } else {
            Student.prototype.set_year.call(this,n);
        };
    }
}
```

Control Flow in Procedural Languages (time)

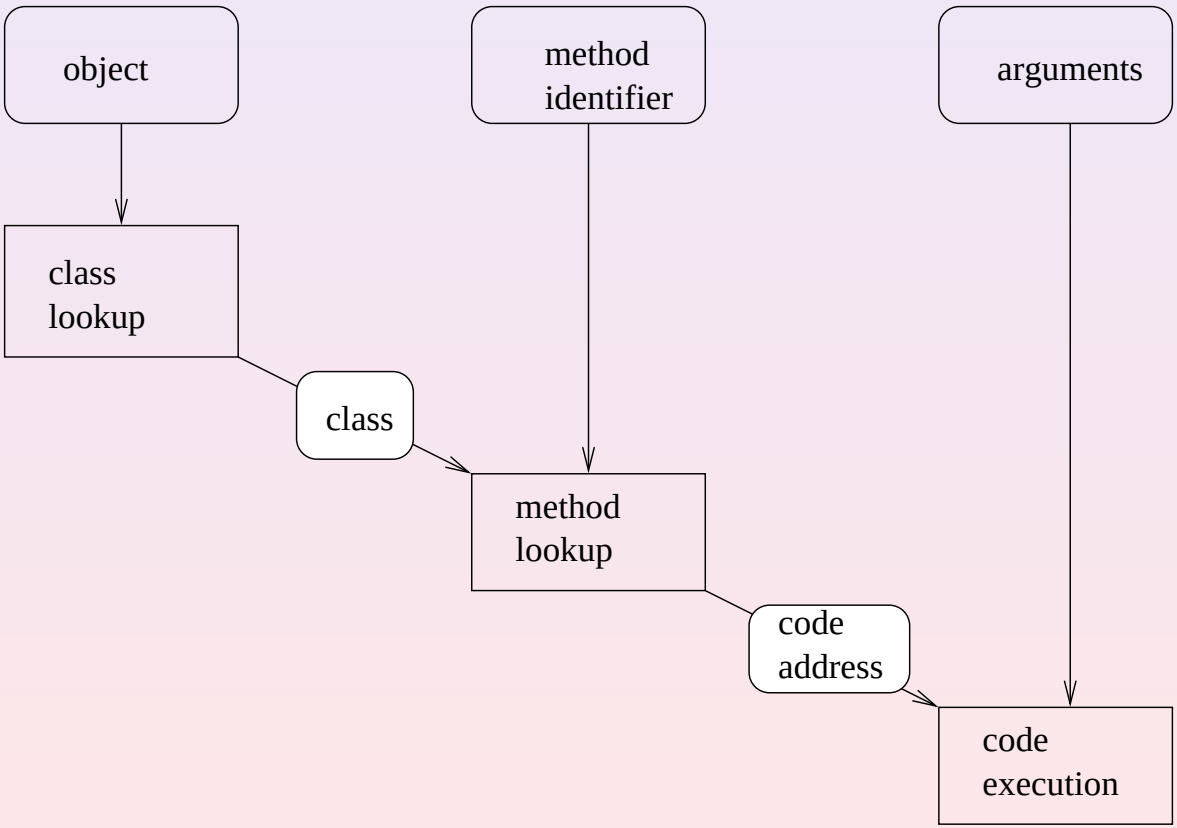


Control Flow in Object-oriented Languages (time)

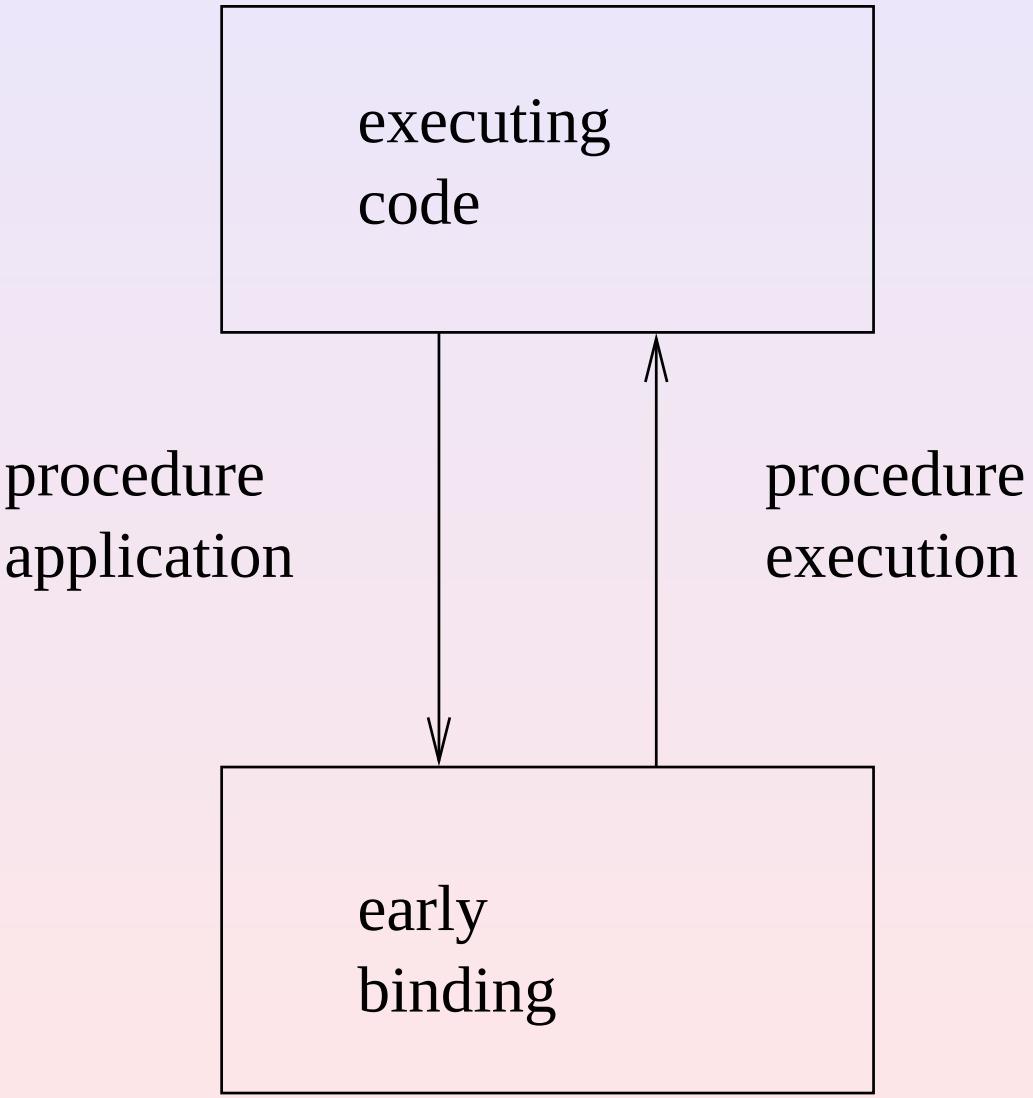


Execution of Object Application

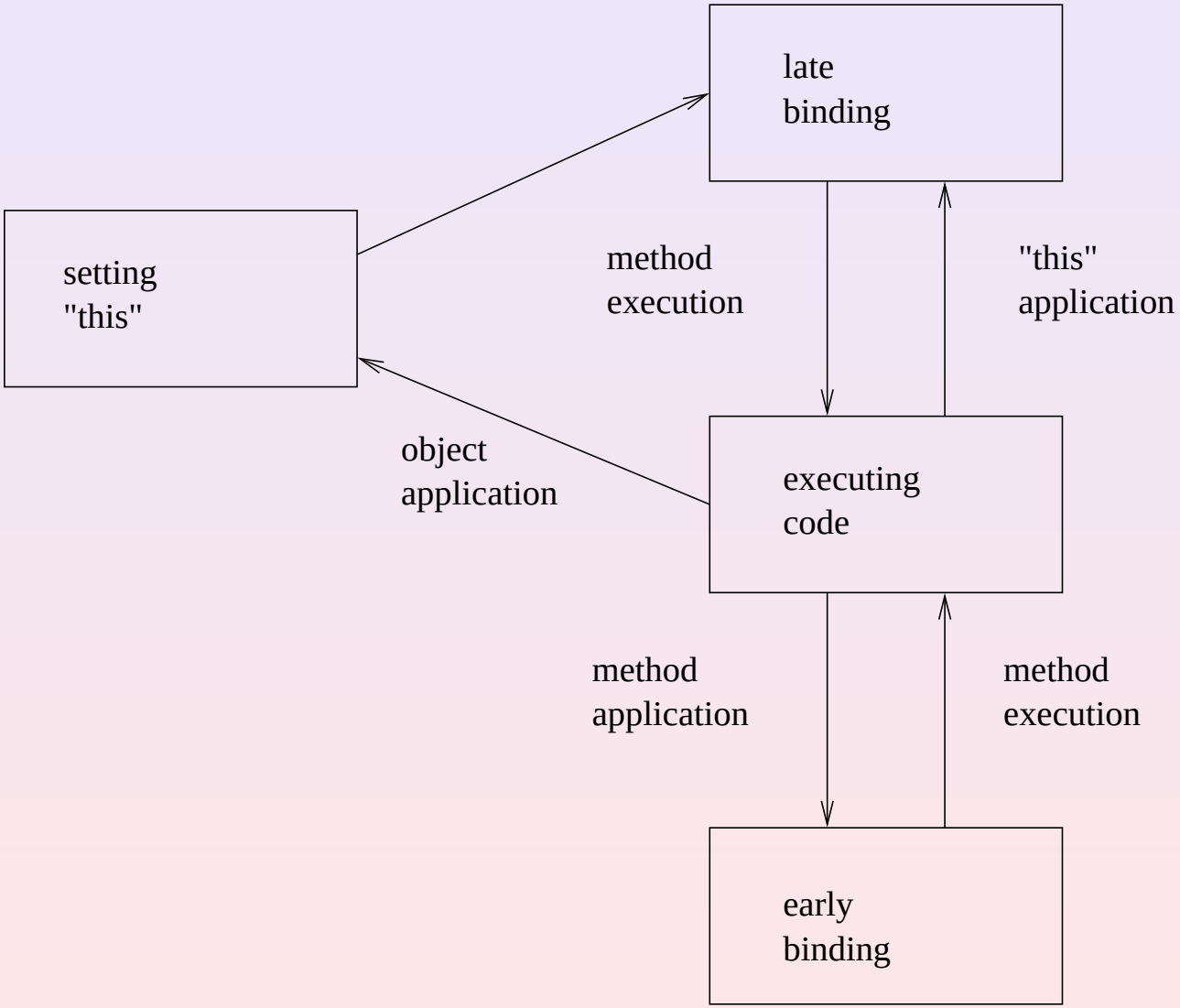
```
window . move ( 100 , 50 )
```



Early Binding in Procedural Languages



Late Binding in Object-oriented Languages



Summary

- Objects are possible and widely used already in JediScript Week 8
- In JediScript Week 10, objects enjoy special support (new, this, Inherits)