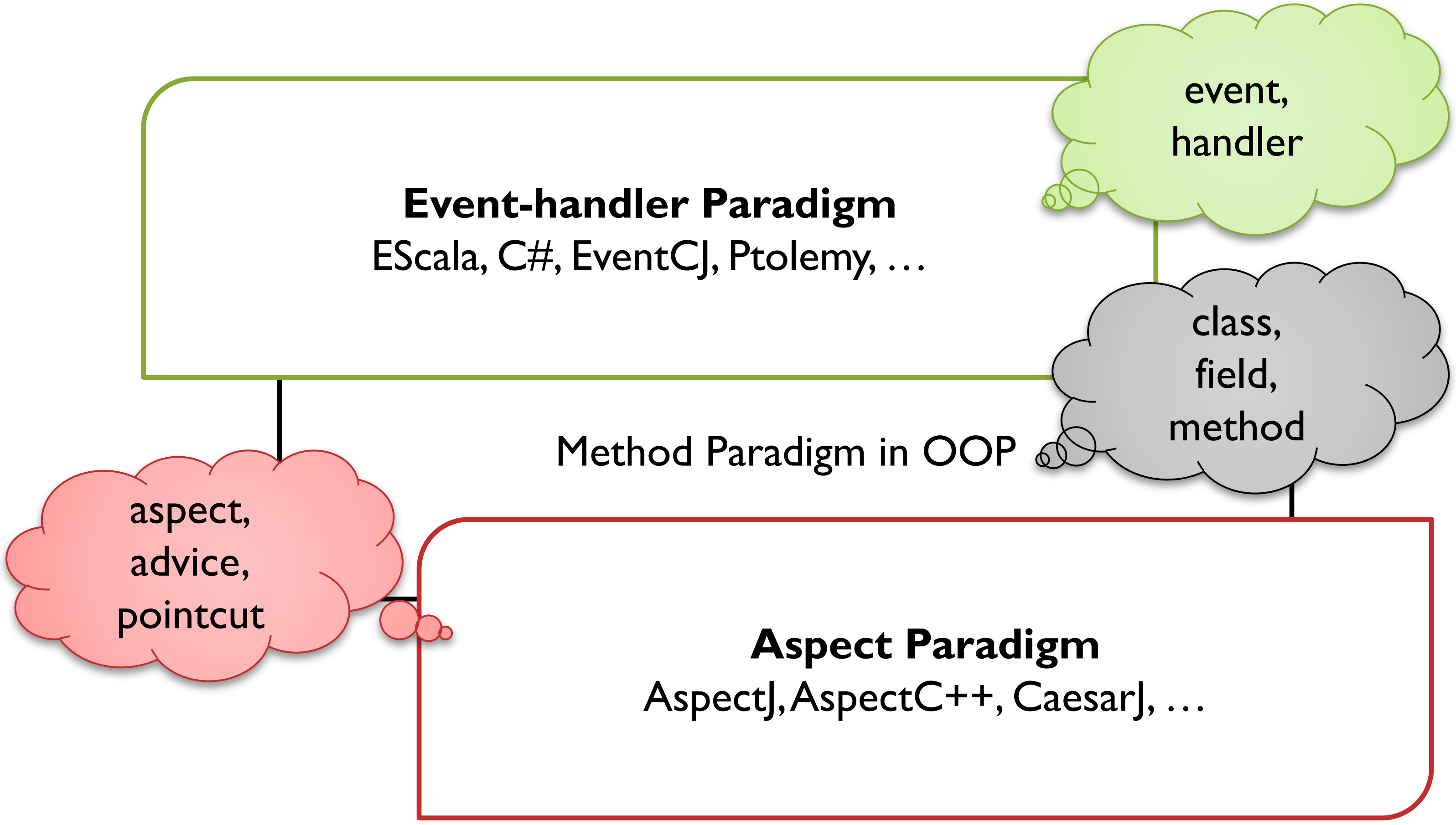
Method Slots: Supporting Methods, Events, and Advices by a Single Language Construct

YungYu Zhuang and Shigeru Chiba

The University of Tokyo

More and more paradigms are supported by dedicated constructs



What we want to learn are paradigms, Not constructs!

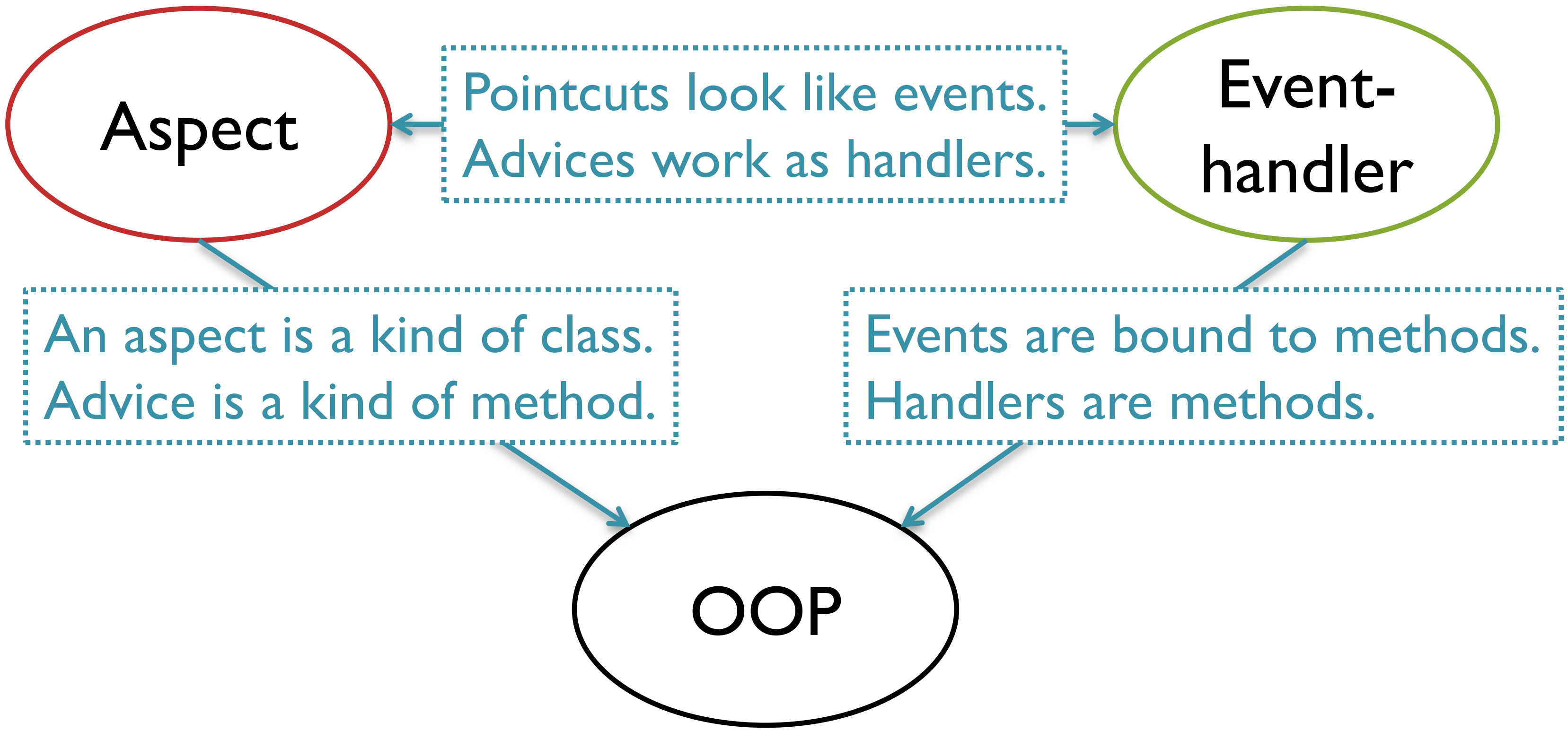
- Supporting by new constructs is a trend
  - Even for existing paradigms like event-handler
  - e.g. C# and EScala

- However, not all constructs are easy to learn!
  - e.g. AspectJ

→ How about reusing constructs?

How about integrating the constructs in the three paradigms

- Their constructs and implementation are very similar
  - Although the problems they address are quite different


Aspect
Pointcuts look like events.
Advices work as handlers.
Event-handler
An aspect is a kind of class.
Advice is a kind of method.
Events are bound to methods.
Handlers are methods.
OOP

Goal

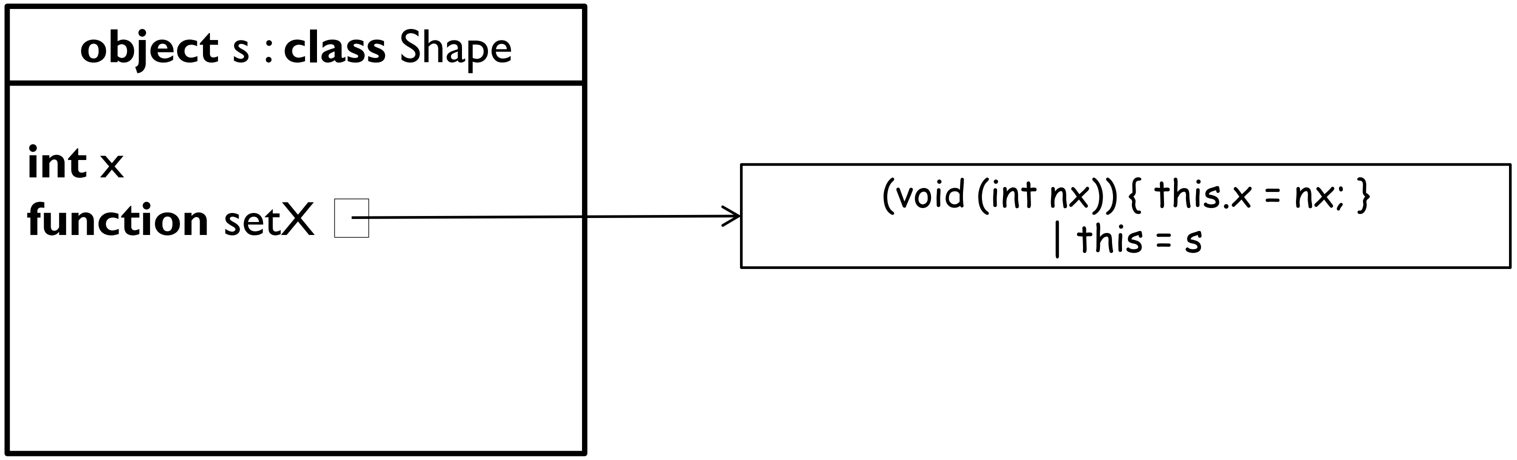
- Develop a new language supporting
  - Event-handler paradigm
  - Aspect paradigm

  <u>By a single construct!</u>

- Extend the most basic one
  - Method paradigm
    (a method in JavaScript)

You know the methods in JavaScript…

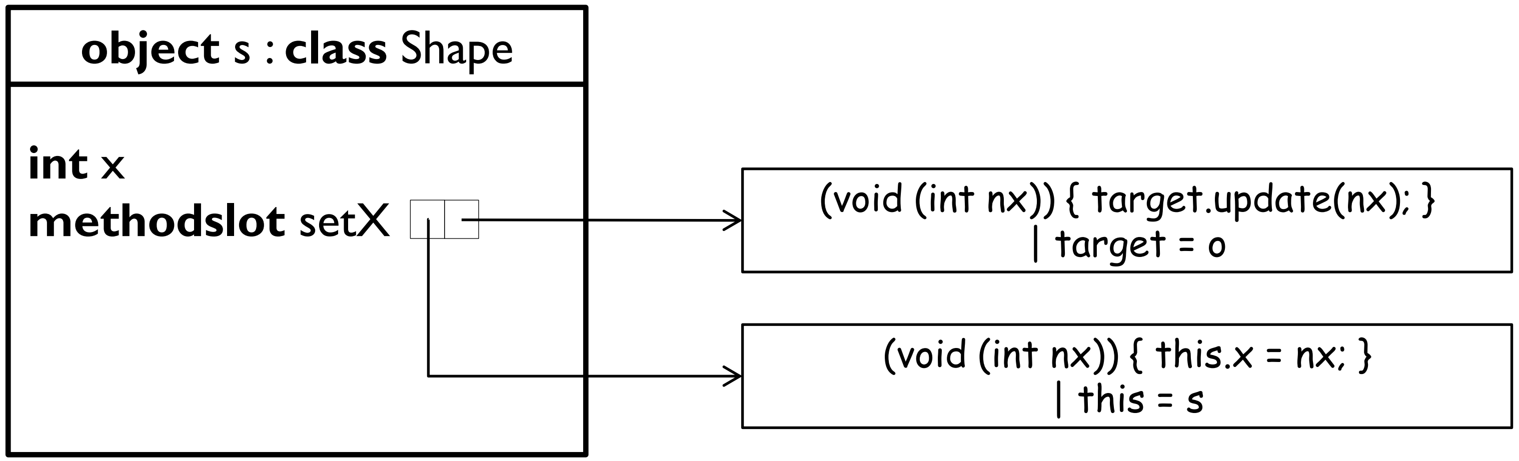
- Methods (function closures) can also be held in fields
  - setX = function(int nx) { this.x = nx; }   // assign the method
  - setX   // return the method
  - setX(10)   // call the method



object s : **class** Shape
int x
function setX
(void (int nx)) { this.x = nx; }
| this = s

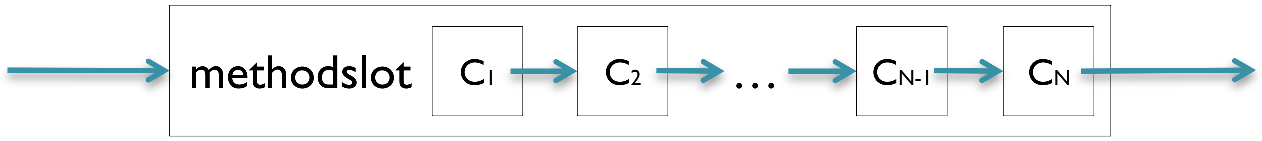
Our Proposal: Method Slots

- Extend the Method paradigm
  - A "field" holds an array of function closures rather than a function closure


object s : **class** Shape
int x
methodslot setX
(void (int nx)) { target.update(nx); }
| target = o
(void (int nx)) { this.x = nx; }
| this = s

The behavior of a method slot

- When a method slot is called
  - All closures in it are executed in order
    - With the arguments given to the method slot
- If its return type is not void
  - The return value is returned by the last closure
    - Every closure can get the return value of the previous closure by a keyword $retval
    - A default value (0/false/null) is given to the first closure



- No closures in it? Just returns the default value

DominoJ: introduce method slots into Java

- No methods, only method slots
- No closures in Java!

  → Give 4 operators to handle closures in a method slot

  - ***<expr>.<methodslot> <op> <expr>.<methodslot>;***
    - Method slots at both sides share the same type (return type and parameter types)
    - Create a closure calling the right one, and add or remove to/from the left one

+=	append to the end of the array
^=	insert at the beginning of the array
-=	remove such closures from the array
=	add and remove the others from the array

  - For example, `s.setX += o.update;`
    - Create a closure `{ o.update(…); }` and append it to `s.setX`

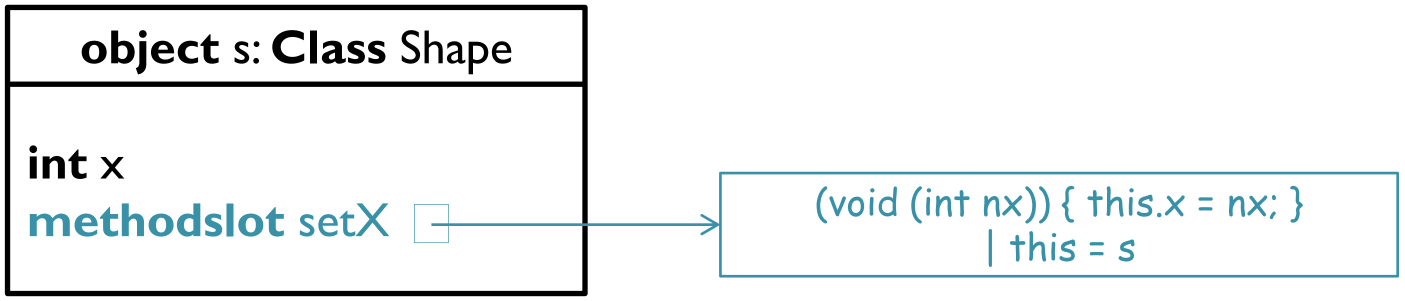
Unlike JavaScript, Java has class declaration and inheritance!

- A method slot is an object's property
  - Static method slots are kept on the class objects
  - Cannot be declared as local variables
- Declare the same method slot in subclasses
  - Overrides the one in the superclass
  - The overridden one can be called through `super` (it only contains the default closure)
  - The overriding one is selected according to the actual type of the object

DominoJ code at a glance

```
public class Shape {
    public int x;
    public void setX(int nx) {
        // default closure
        this.x = nx;
    }
}
```

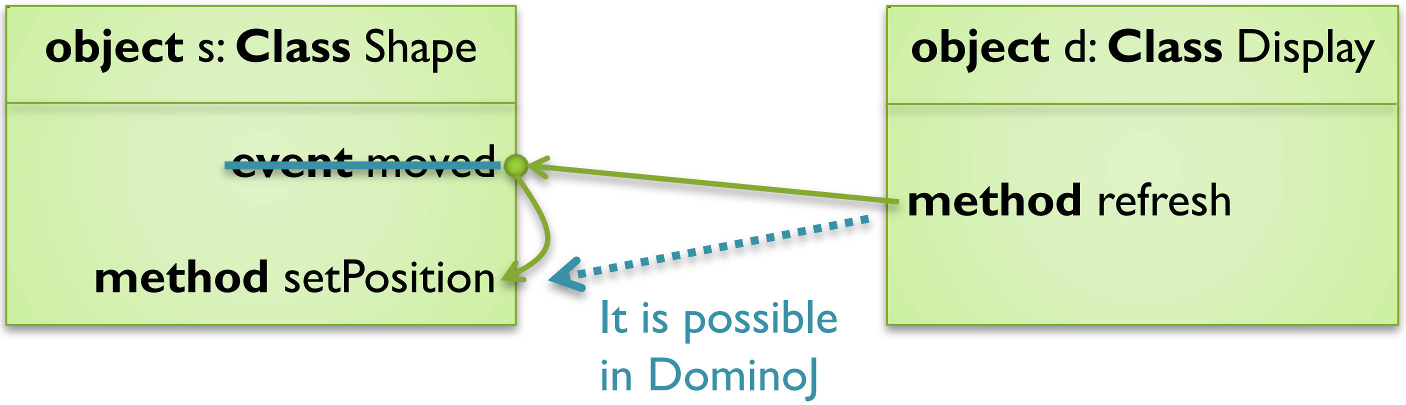
- The declaration looks like a method declaration
  - The body is the default closure (optional)



object s: **Class** Shape
int x
methodslot setX
(void (int nx)) { this.x = nx; }
| this = s

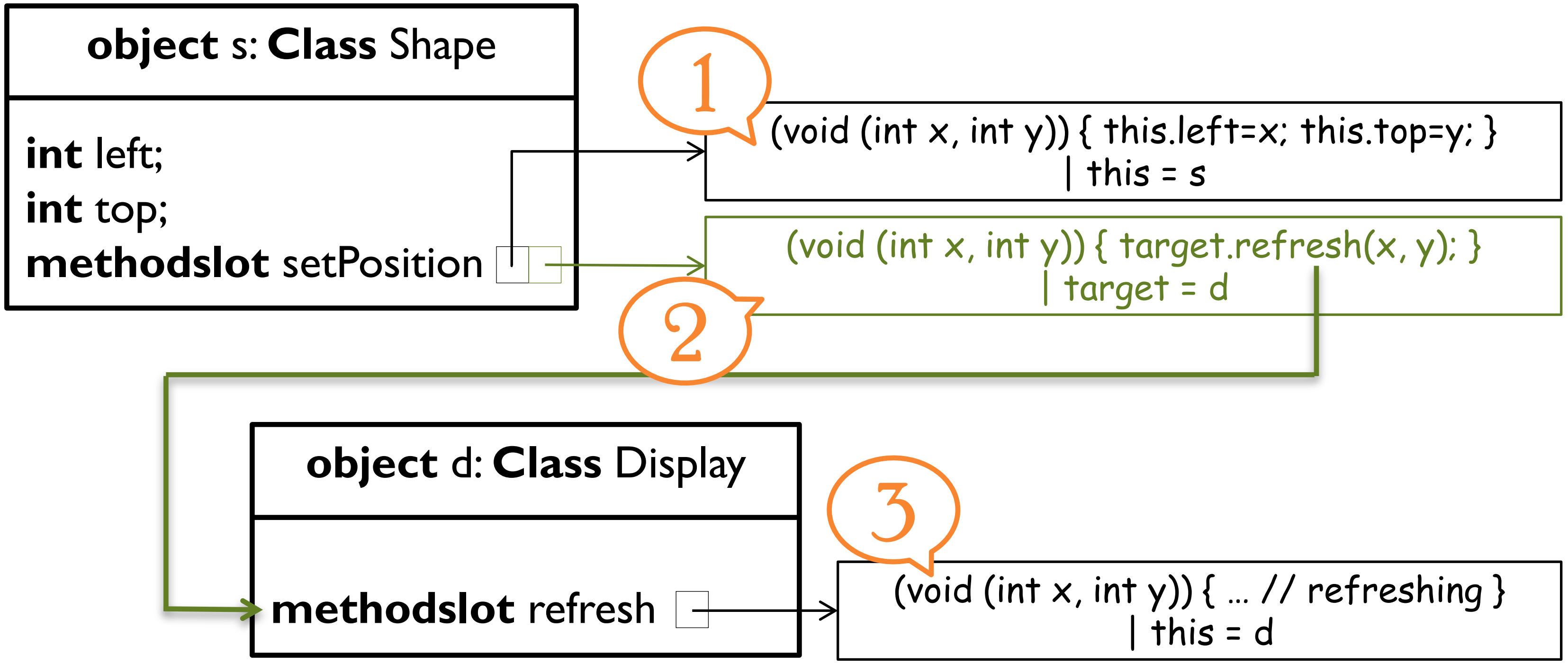
An example of using Event-handler in typical event mechanisms

- Suppose the *Display* object should be refreshed after the position of *Shape* objects are set
- The typical way in an event mechanism like EScala or C#
  - Expose an event moved for setPosition in s
  - Bind d.refresh to moved


object s: **Class** Shape
~~**event** moved~~
method setPosition
object d: **Class** Display
method refresh
It is possible in DominoJ

Use DominoJ to write the Event-handler example

```
s.setPosition += d.refresh;          // Add a closure calling d.refresh
s.setPosition(0, 0);                 // d.refresh will be called
```



object s: Class Shape
int left;
int top;
methodslot setPosition
1
(void (int x, int y)) { this.left=x; this.top=y; }
| this = s
(void (int x, int y)) { target.refresh(x, y); }
| target = d
2
object d: Class Display
methodslot refresh
3
(void (int x, int y)) { … // refreshing }
| this = d

Compare the code for this example in EScala and in DominoJ

- The event declaration can be omitted.
  - Any public method slots are regarded as events.

- In EScala (based on Scala)

```
class Display() {
  def refresh() {
    System.out.println("display is refreshed.")
  }
}
class Shape(d: Display) {
  var left = 0; var top = 0
  def setPosition(x: Int, y: Int) {
    left = x; top = y
  }
  evt moved[Unit] = afterExec(setPosition)
  moved += d.refresh
}
```

- In DominoJ (based on Java)

```
public class Display {
  public void refresh(int x, int y) {
    System.out.println("display is refreshed.");
  }
}
public class Shape {
  private int left = 0; private int top = 0;
  public void setPosition(int x, int y) {
    left = x; top = y;
  }
  public Shape(Display d) {
    this.setPosition += d.refresh;
  }
}
```

"Any method slots can be events."

This break the encapsulation? No!

- Follow the visibility in OOP
  - Rely on the visibility of method slots
  - A public method slot is always visible as an event to other objects

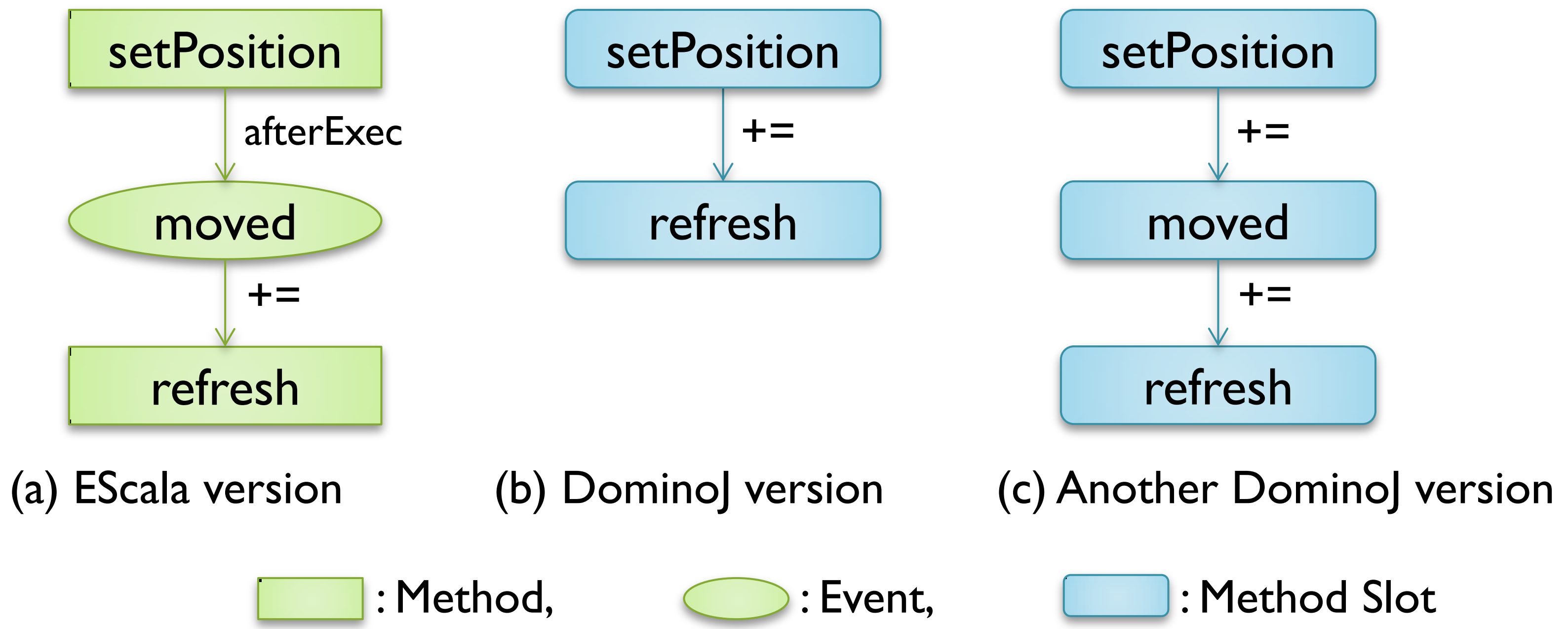
- Simpler but limited
  - Cannot separate the event from a method

→ Declare a higher-level event?

Higher-level events are also possible

- Declare an empty method slot,
  and let it be triggered by another one

```
public void moved();
setPosition += moved;
```



Compare DominoJ with EScala

	Type	EScala	DominoJ
role	*Event*	field (`evt`)	method slot
	Handler	method	
binding	*Event-to-Handler*	`+=`	`+=`
		`-=`	`-=`
	Event-to-Event	`||`	`+=, ^=`

		`&&`
		`\`
		`filter`
		`map`
		`empty`
		`any`
			use Java expression in the default closure of method slots

	Handler-to-Event	`afterExec`	`+=`
		`beforeExec`	`^=`
		`imperative`	explicit trigger is possible

Check the example from the viewpoint of Aspect

- Suppose we have
  - *Display* class and *Shape* class
  - A crosscutting concern: when to refresh
- In AspectJ, we can write such an aspect

```
public aspect UpdateDisplay {
  after() returning:
   execution(
    void Shape.setPosition(int, int)) {
      Display.refresh();
  }
}
```

In DominoJ, classes can be aspects, method slots can be advices

- Class-based behaviors?
  - Emulate by binding method slots in constructors

```
public Shape() {  this.setPosition += Display.refresh;  }
```

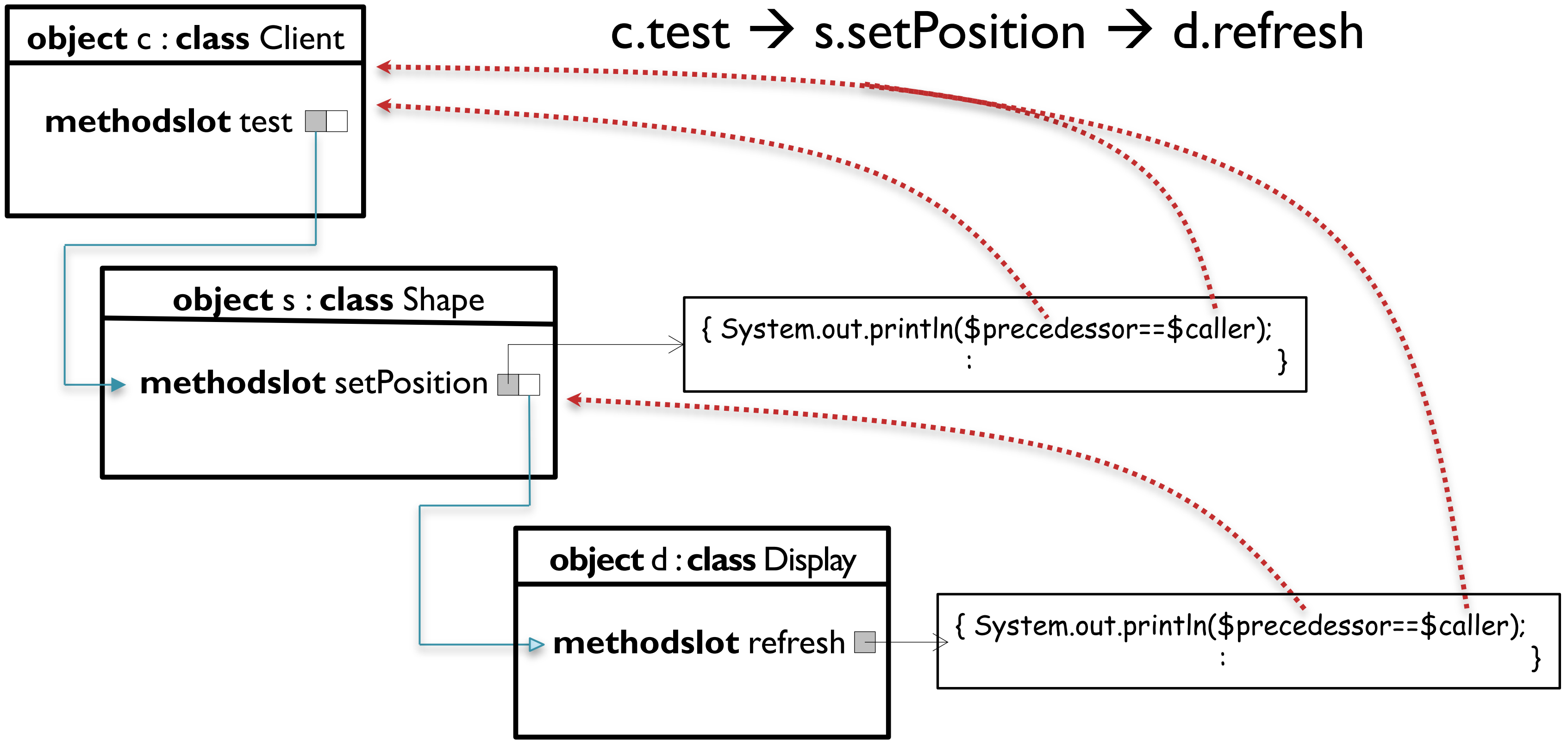
- Obliviousness?
  - Attach to public method slots (including constructors)
- No complicated instantiation models
  - Need to manage objects manually

```
public class UpdateDisplay {
  public static void init() {
    ((Shape)$predecessor).setPosition += Display.refresh;
  }
  static { Shape.constructor += UpdateDisplay.init; }
}
```

Using the keywords $predecessor and $caller to get preceding objects in a call sequence

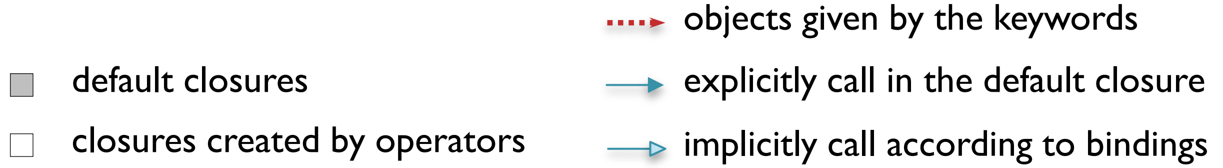
- Suppose s.setPosition is called in c.test where c is an object of class *Client*

c.test → s.setPosition → d.refresh



```
{ System.out.println($predecessor==$caller);
                  :                        }
```

```
{ System.out.println($predecessor==$caller);
                  :                        }
```


default closures
closures created by operators
objects given by the keywords
explicitly call in the default closure
implicitly call according to bindings

Rewrite AspectJ code by DominoJ

- Obliviousness and class-based behaviors are possible

- In AspectJ

```
public class Display {
  public static void refresh() {
    System.out.println("display is refreshed.");
  }
}
public class Shape {
  private int left = 0; private int top = 0;
  public void setPosition(int x, int y) {
    left = x; top = y;
  }
}
public aspect UpdateDisplay {
  after() returning:
   execution(
    void Shape.setPosition(int, int)) {
      Display.refresh();
  }
}
```

- In DominoJ

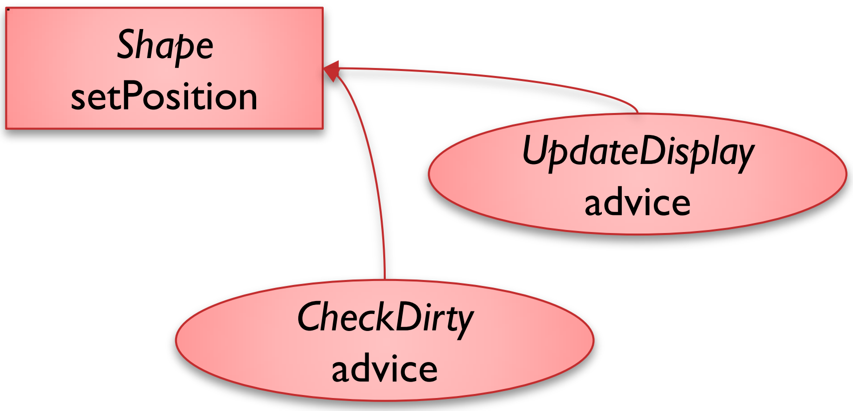
```
public class Display {
  public static void refresh(int x, int y) {
    System.out.println("display is refreshed.");
  }
}
public class Shape {
  private int left = 0; private int top = 0;
  public void setPosition(int x, int y) {
    left = x; top = y;
  }
}
public class UpdateDisplay {
  public static void init() {
    ((Shape)$predecessor).setPosition
                                   += Display.refresh;
  }
  static { Shape.constructor += UpdateDisplay.init; }
}
```

Compare DominoJ with AspectJ

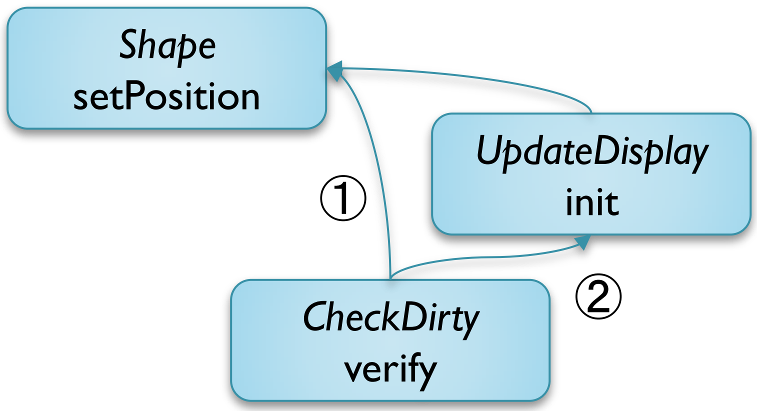
Construct	AspectJ	DominoJ
grouping	`aspect`	`class`
code piece	advice body	method slot body (default closure)
pointcut and advice declaration	`after returning` **and** `execution`	`+=` **and** `$retval`
	`before` **and** `execution`	`^=`
	`around`	`=`
	`this`	`$caller`
	`target`	`$predecessor`
	`args`	by parameters

Advices for advices are possible

- If you think attaching CheckDirty to UpdateDisplay is more meaningful…
  - Yes, you can do it in DominoJ!



(a) AspectJ version



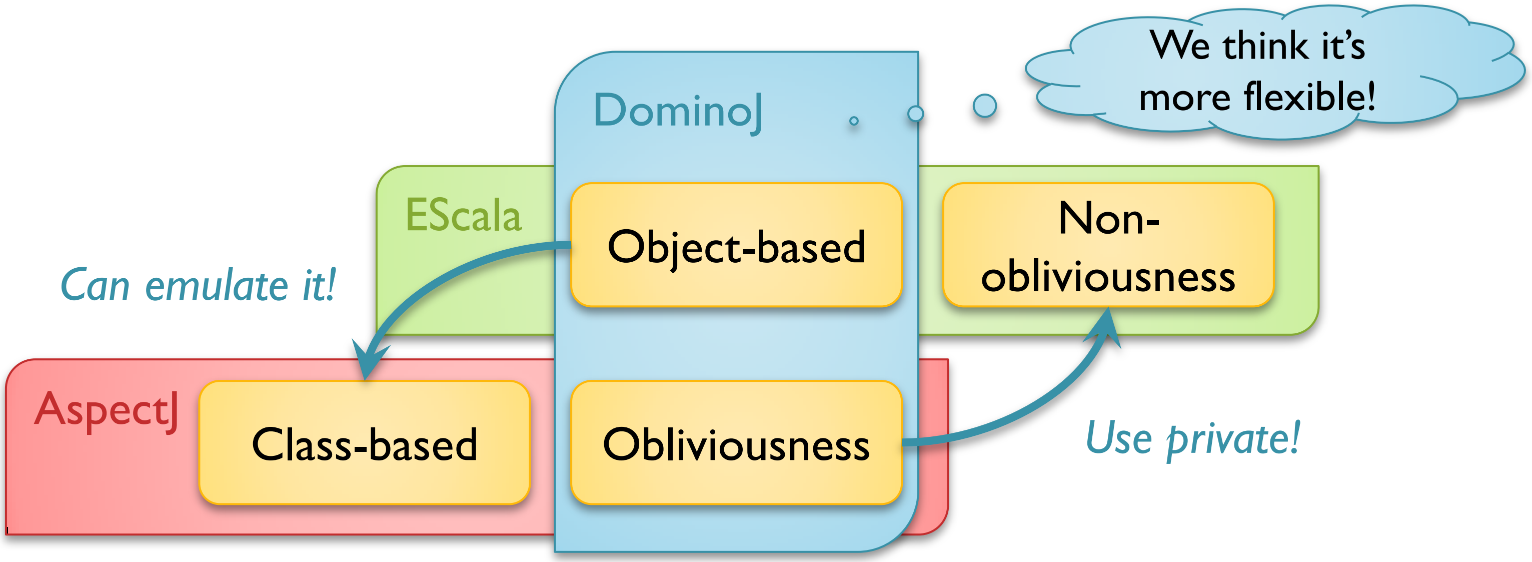
(b) DominoJ version

🟥 : Method, ⬭ : Advice, 🟦 : Method Slot

Event-handler vs. Aspect

- In my opinion, they are the same except
  - Object-based or Class-based?
  - Non-obliviousness or Obliviousness?

  → *Impossible to support contradictory things at the same time unless giving both constructs*
- DominoJ want to make all <u>available</u> by one construct, and let programmers decide how to use
  - Different from Object-based AOP languages? → Simpler



Related Work

- The delegation in C#
  - A delegate is similar to a method slot
  - Events and methods are separate constructs
- Delegation-based AOP
  - Supports the mechanisms in OOP and AOP
  - A proxy delegates messages to an object
- Ptolemy
  - Treat the execution of any expression as an event
  - Events are global, class-based

Conclusion

- We proposed a <u>simple</u> and generic construct ---*Method slots*
  - Covering <u>most</u> functionality of
    - Event-handler paradigm
      - Lack of rich event expression
    - Aspect paradigm
      - No inter-type declaration and reflection

- Future work
  - Supporting more paradigms
  - Case study