Programming Language Design

2015

Week #10: reflection and meta-programming

Instructor: Hidehiko Masuhara

Quiz (1/1) What do you say to Bob? (5 min.)

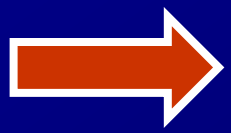
I'm designing a new language for my programs, which use only integer values. For truth values, I use 1 for true, 0 for false. Then I noticed this user function will work as an if-then-else statement.

```
int myif(int cond, int thenVal, int elseVal){
  return cond*thenVal + (1-cond)*elseVal;
}
```

Now I can make my language without the if-then-else statement, which is much simpler!

Review: templates

- template <class T>
  class Stack {
    T items[MAX];
    int top;
    T pop() {...}
    void push(T a) { ...}
  }
- Stack<int> s;
  s.push(123);
  printf("%d", s.pop());



- class Stack$1 {
    int items[MAX];
    int top;
    int pop() {...}
    void push(int a) { ...}
  }
- Stack$1 s;
  s.push(123);
  printf("%d", s.pop());

Template expansion as program execution

```
template <class T>
class Stack {
  T items[MAX];
  int top;
  T pop() {...}
  void push(T a) { ...}
}
Stack<int> s;
s.push(123);
printf("%d", s.pop());
```

a template program

program execution

```
class Stack$1 {
  int items[MAX];
  int top;
  int pop() {...}
  void push(int a) { ...}
}
Stack$1 s;
s.push(123);
printf("%d", s.pop());
```

output

Templates as a programming language

- Templates: functions from type names to code
- Purpose: to generate definitions for many types
- Input: type names
  - (some template mechanisms can receive constant values)
- Output: code (classes and functions)
- Operations:
  - fill type names into holes (hence "templates")
  - use other templates
- Timing: before compilation

Macros: generalized templates

- Macros: functions from expressions to expressions
- Purpose: shorter descriptions, defining control structures and data structures
  - also used inside of expressions
- Input: expressions
- Output: expressions or definitions
- Operations: varying from simple to powerful ones
  - filling holes+α (CPP)
  - all LISP functions (LISP)
- Timing: before compiling / before evaluating

expressive power of macro definition language matters

C preprocessor (CPP)

- Description language
  - filling holes (by referencing variables)
  - +α : concatenating identifiers
- What's hard in : evaluation order, number of evaluations

why or is not a function?

why (x) rather than x?

eg.

```
#define or(x,y) ((x)!=NULL?(x):(y))
… open(or(file, opt[opt_i++])) …
```

yellow: expressions as data

LISP Macro

- Description language: LISP itself (powerful!)
  - with support for template like descriptions
    - S-expressions: LISP's programs use the same representation as LISP's basic data structures (ie lists)
    - Quasi-quotes: embed computation into expression as data
- eg:

```
(defmacro dotimes (spec body)
  `(let ((max ,(cadr spec))
         (,(car spec) 0))
     (while (< ,(car spec) max)
       ,body
       (setq ,(car spec) (+ ,(car spec) 1)))))
```

problems?

```
…(dotimes (i n)
    (setq sum (+ i sum))) …
```

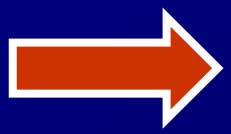
↓

```
…(let ((max n) (i 0))
    (while (< i max)
      (setq sum (+ i sum))
      (setq i (+ i 1))))…
```

A problem: variable capturing

- a macro-generated variable name may overlap with the one already used in the expression where the macro is used

```
(def max-element (a)
  (let ((max (- INFINITY)))
    (dotimes (i (length a))
      (if (< max (aref a i))
        (setq max (aref a i))))
    max)))
```



```
(def max-element (a)
  (let ((max (- INFINITY)))
    (let ((max (length a))
          (i 0))
      (while (< i max)
        (if (< max (aref a i))
          (setq max (aref a i)))
        (setq i (+ i 1)))
      max)))
```

Hygienic Macros [Kohlbecker+86]

- Safe macro mechanism wrt variable names
  - variable names in templates will be automatically renamed

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes (v n) body)
     (let ((max n) (v 0))
       (while (< v max)
         body
         (setq v (+ v 1)))))))
```

pattern

template (max will be renamed)

```
(def max-element (a)
  (let ((max (- INFINITY)))
    (let ((max$1 (length a))
          (i 0))
      (while (< i max$1)
        (if (< max (aref a i))
            (setq max (aref a i)))
        (setq i (+ i 1)))
      max)))
```

eval: run time macros

- eval: a function from expressions to results
- Input: expressions (often as strings)
- Output: values in the language
- Purpose: program generation by using runtime information, eg:
  - computation by using interactively obtained inputs
  - loading function definitions in a file
- Timing: during program execution

example of eval: program specialization

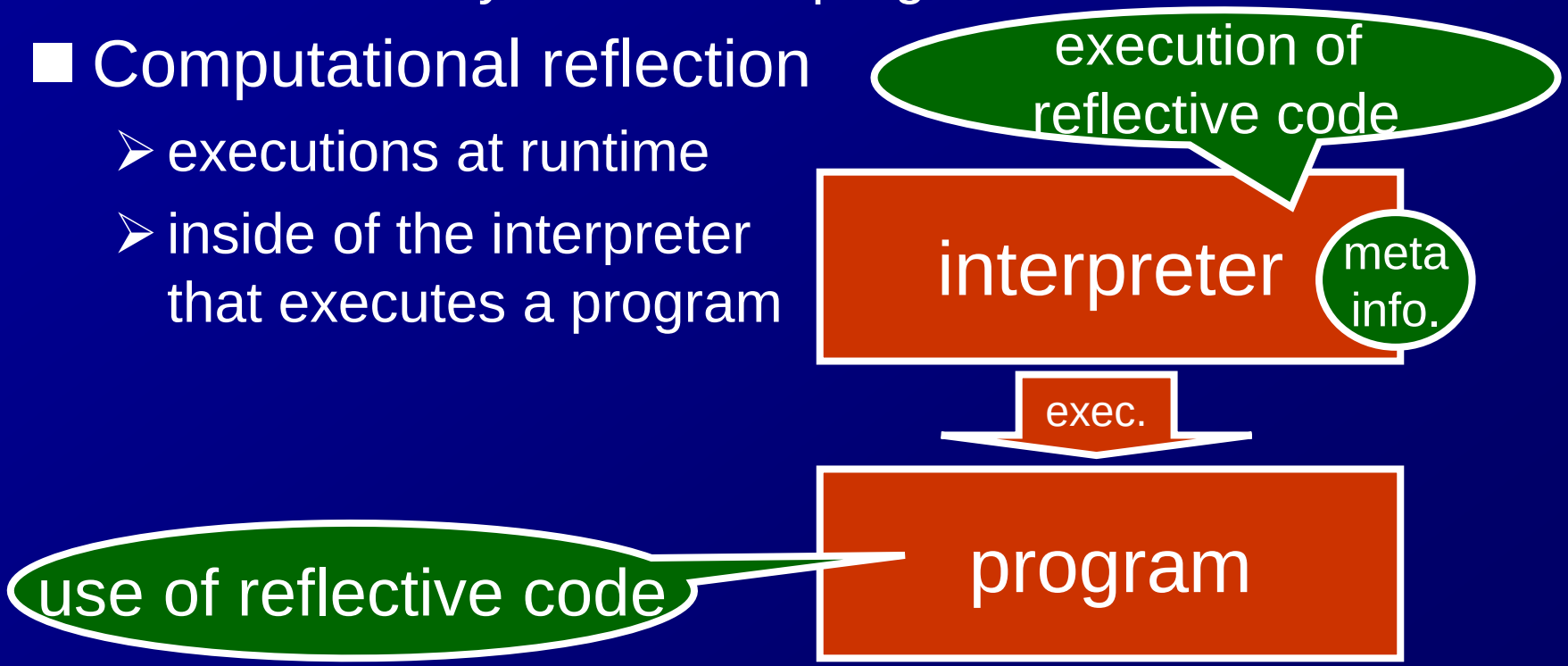
```
> def dp(n)
>  "def power"+n.to_s+"(x)"+("x*"*n)+"1; end"
> end
> dp(10)
"def power10(x)x*x*x*x*x*x*x*x*x*x*1; end"
> eval(dp(10))
> power10(2)
1024
```

Problems of eval

- Slow: due to interpreter execution (or runtime compilation)
  - (then, can we use it for optimizations?)
- Dangerous:
  - No safety guarantee (eg type safety)
  - Major source of security holes
  - All static analyses become invalid just one use of eval

Computational reflection [Smith84]
macros that can use runtime information

- Macros' limitation: no runtime information
  - because they run ***before*** program execution
- Computational reflection
  - executions at runtime
  - inside of the interpreter that executes a program

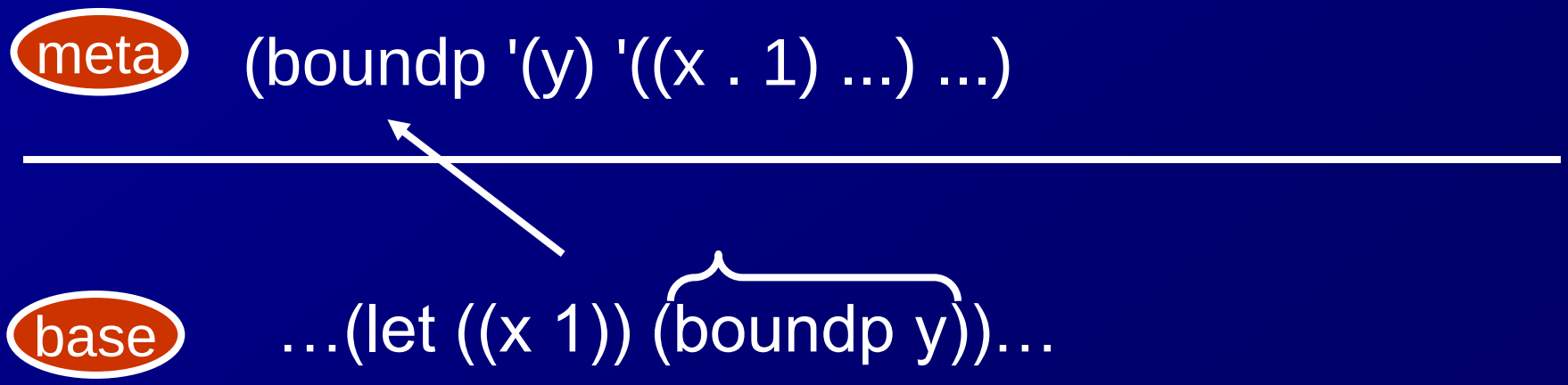


Computational reflection

- User-defined computation that runs inside of an interpreter
  - can access meta information (eg: variable environment, continuation)
- allows extensions of the interpreter from inside of a user-program
  - "is this variable name already defined?"
  - "delete this variable definition"
  - can realize embedded "meta-computations" as user-defined functions (eg) boundp, throw/catch

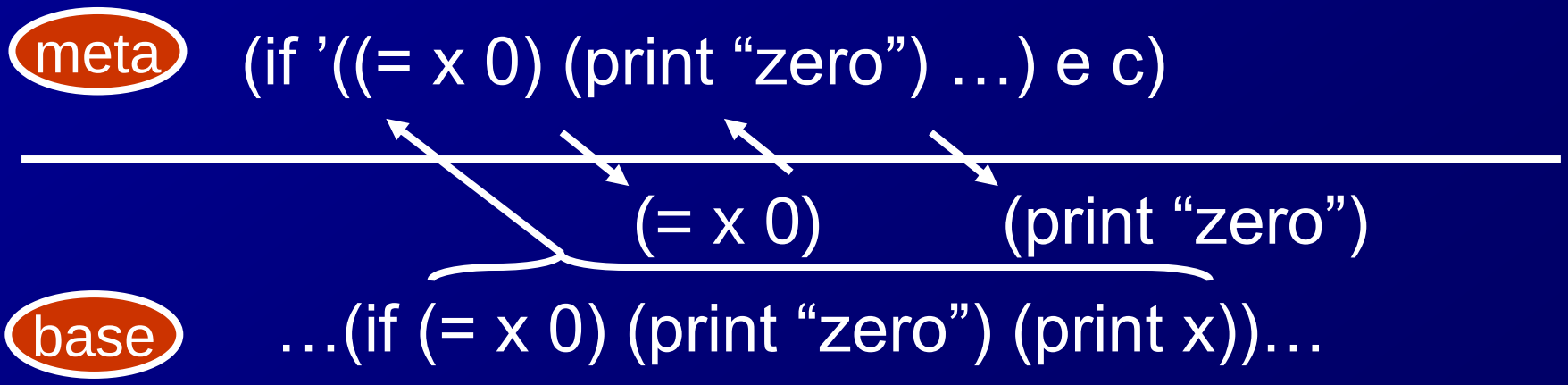
Examples of computational reflection

```
(define boundp
  (lambda reflect ((vname) env cont)
    (assq vname env)))
```



Examples of computational reflection

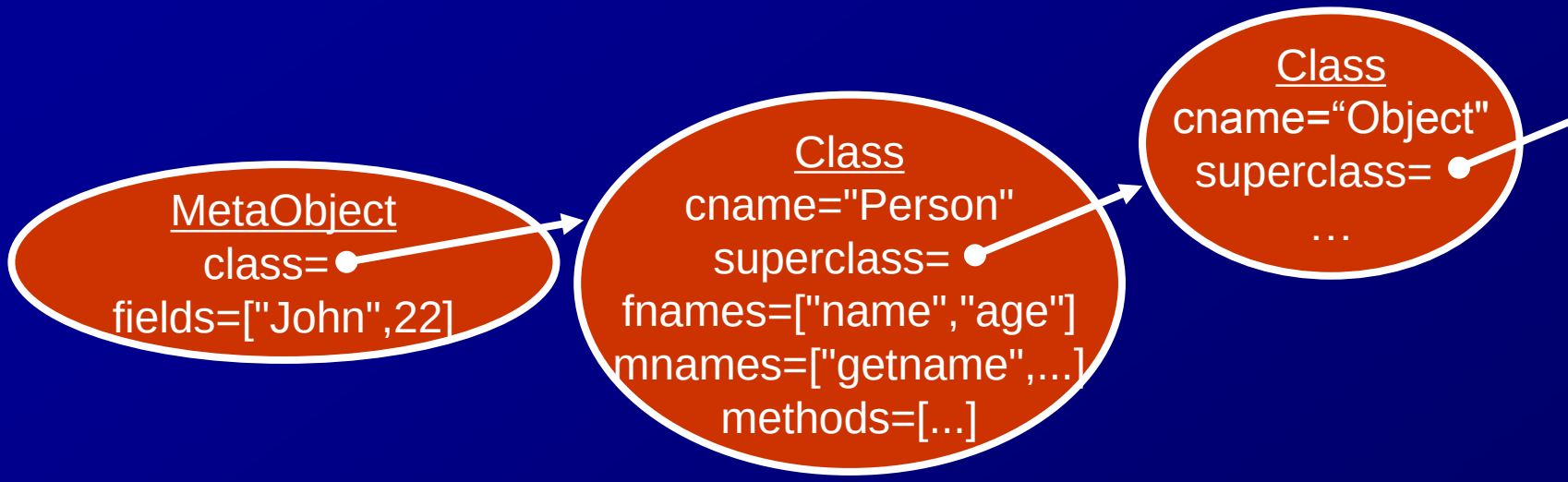
```
(define if
  (lambda reflect ((cond then else) env cont)
    (eval cond env
      (lambda (c) (eval (ef c then else) env cont)))))
;; ef is applicative conditional branch
```



Meta-level computation in OOPL

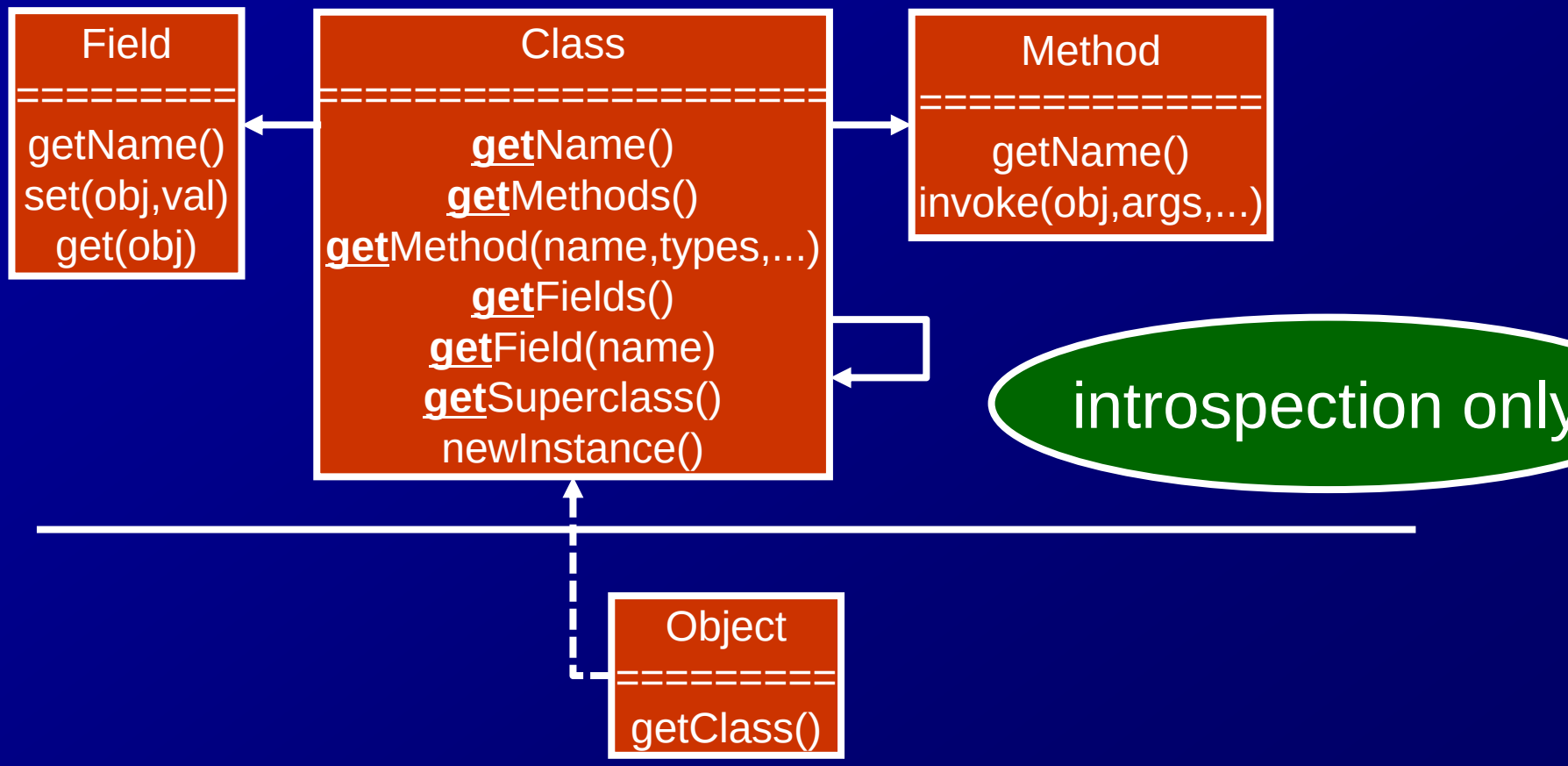
- Examples
  - to know a class of an object
  - to get a list of methods/fields in a class
  - to know a superclass of a class
  - to create an object of a named class
- Example usage
  - to store object states in a file, and reconstruct later

Meta-objects:
objects that offer meta-level operations



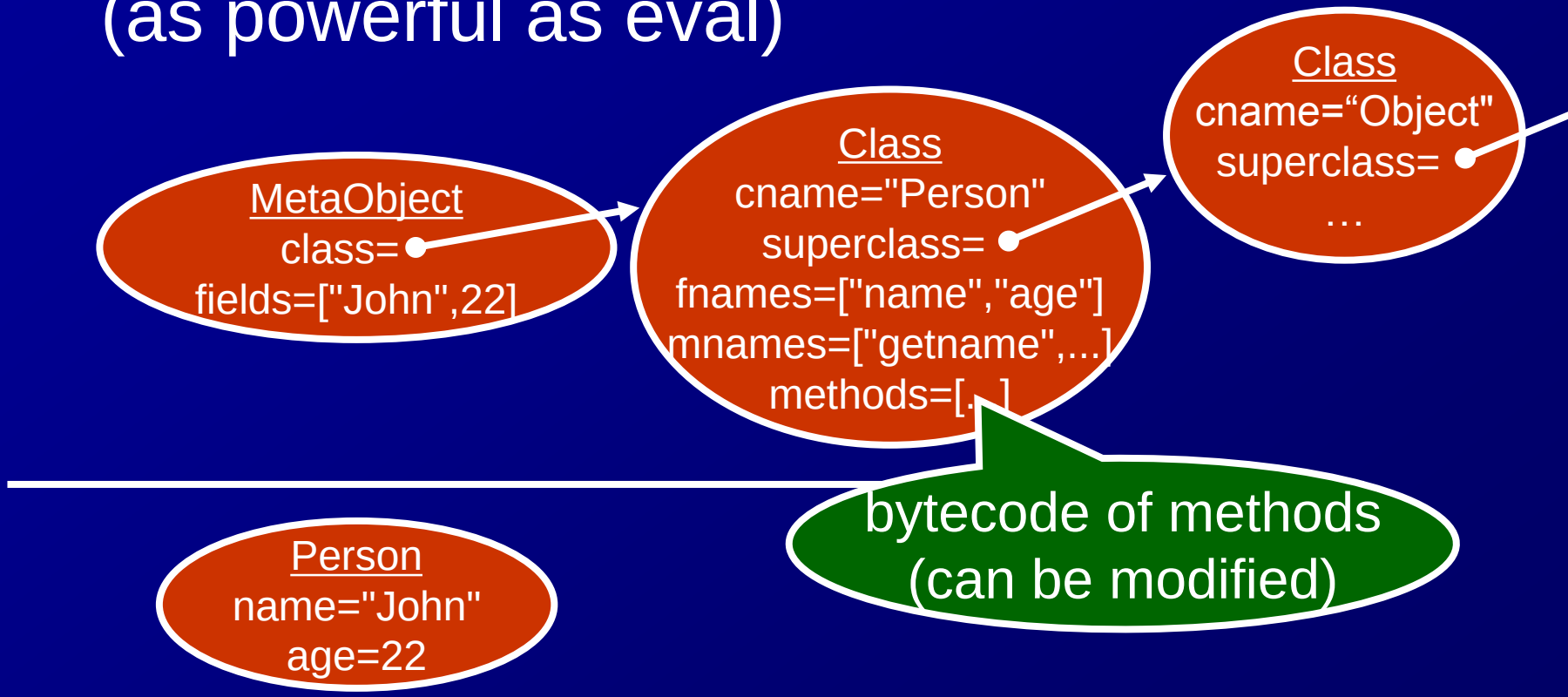
- to know a class of an object
- to get a list of methods/fields in a class
- to know a superclass of a class
- to create an object of a named class

Java Reflection



Smalltalk meta-object

- provides internal information ***directly*** (as powerful as eval)



CLOS Metaobject Protocol [Kiczales91]

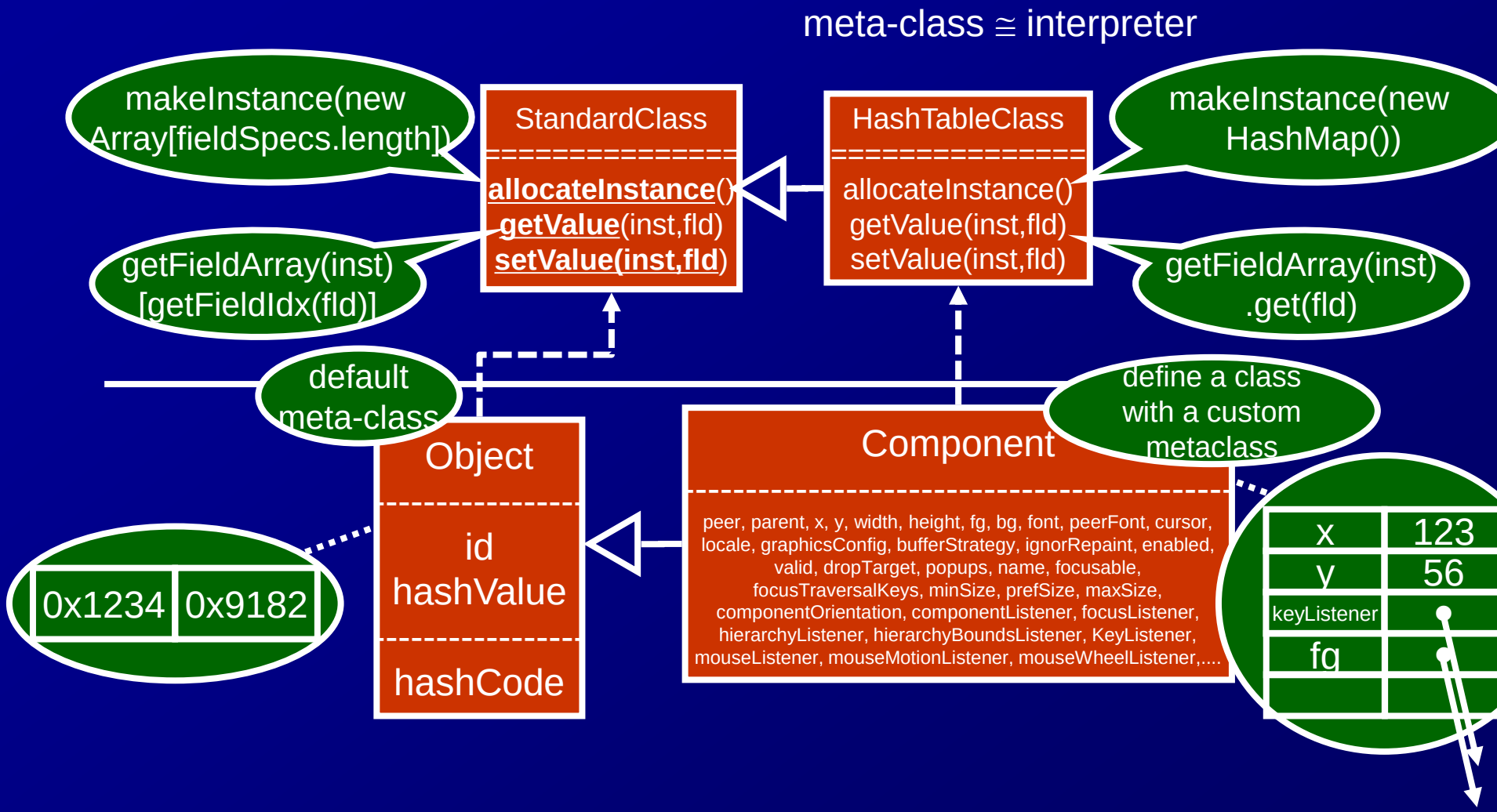
CLOS= Common Lisp Object System

Customizable meta-objects

Usages: to customize

- field representation of objects (cf. sparse objects)
  - default rep.: array (r/w O(1))
  - customized: hash table, database sync'd, ...
- method selection algorithm (multiple inheritance)
  - default: sorting by classes of arguments
  - customized: change method precedence by runtime condition

Reflection by CLOS MOP



References

[Kohlbecker+86] Kohlbecker, Eugene, et al. "Hygienic macro expansion." Proceedings of the 1986 ACM conference on LISP and functional programming. ACM, 1986.

[Smith84] Smith, Brian Cantwell. "Reflection and semantics in Lisp." Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, 1984.

[Kiczales91] Kiczales, Gregor. The art of the metaobject protocol. MIT press, 1991.