Programming Language Design

2015

Week #3: Object-oriented programming (OOP) (1)

Instructor: Hidehiko Masuhara

Quiz (15 min.)

1. Define the meaning of "orientation/-oriented" in the context of OOP
2. List the language features that characterize OOP

   if each of such features is missing, it can no longer be called OOP
3. List the common language features in OOP and abstract datatypes

cf. there are PLs that are called "*-oriented programming" other than OOP

“***-Orientation"

- Definition: to consider things by centering ***
- Example: OOP = to program by centering objects in the problem domain

Note: we say "functional programming", but not "function-oriented prog."

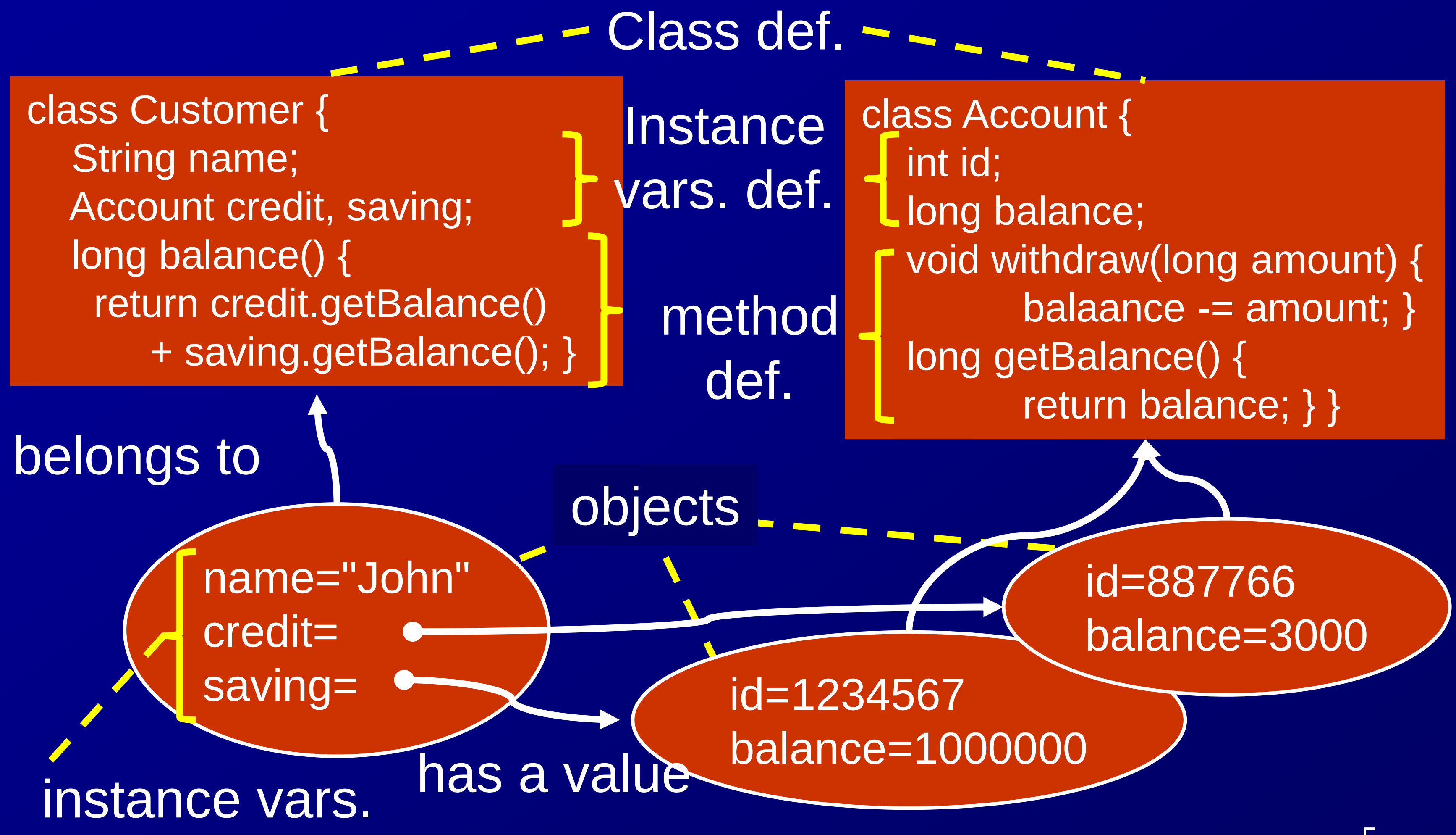
Note: not only for programming; e.g., object-oriented design

An object is anything that has a fixed shape or form, that you can touch or see, and that is not alive. [Cobuild]

Example of an OOPL
Use case: Bank account & customer

- Objects
  - (Bank： customers and accounts)
  - Customer: name, checking account, saving account
  - Account: balance, id
- Behavior
  - calculate total balance of a customer
  - withdraw money from an account
  - calculate a balance of an account

Example of an OOPL



Example of an OOPL classes and objects

- Class: a definition that bundles
  (= written as a program)
  - method defs.: determine object behaviors
  - instance var. defs. : determine object state
- Object: (constructed at runtime)
  - is a value (stored in a variable)
  - belongs to a class
  - has instance variables

Example of an OOPL method calls

```
class Customer {
    String name;
    Account credit, saving;
    long balance() {
      return credit.getBalance()
          + saving.getBalance(); } }
```

```
class Account {
    int id;
    long balance;
    void withdraw(long amount) {
            balaance -= amount; }
    long getBalance() {
            return balance; } }
```

john.balance() => 1003000

name="John"
credit=
saving=

id=887766
balance=3000

id=1234567
balance=1000000

Method call:

- search body in a receiver's class
- executes as a function
  - where instance vars. are visible

Example of an OOPL method calls

- Method call: john.balance()
  - similar to a function call
  - "receiver": 0th argument
- Method dispatching:
  - identifies a class of the receiver, then
  - finds a method def. in the class def.
- Variable environment of method execution:
  - instance variables are visible
  - a pseudo variable to indicate "this"/"self"

Example of an OOPL: inheritance

```
class FixedDeposit extends Account {
    Date maturity;
    void withdraw(long amount) {
      if (maturity.isExpired())
        super.withdraw(amount);
      else
        error; } }
```

inheritance

```
class Account {
    int id;
    long balance;
    void withdraw(long amount) {
            balaance -= amount; }
    long getBalance() {
            return balance; } }
```

saving

id=1234567
balance=1000000

credit

id=887766
balance=3000

credit.getBalance()=>3000
saving.getBalance()=>1000000

credit.withdraw(1000) => OK
saving.withdraw(1000) => error!

Example of an OOPL: inheritance

```
class FixedDeposit extends Account {
    Date maturity;
    void withdraw(long amount) {
      if (maturity.isExpired())
          super.withdraw(amount);
      else
          error; } }
```

```
class Account {
    int id;
    long balance;
    void withdraw(long amount) {
        balaance -= amount; }
    long getBalance() {
        return balance; } }
```

- Inheritance 継承
  - creates a class by adding elements to and/or modifying some elements in another class
  - super/sub-classes
- Method dispatching looks in
  - a belonging class,
  - if not found, look superclasses
- super-call
  - executes method defs in a superclass

saving: id=1234567, balance=1000000

credit: id=887766, balance=3000

credit.balance()=>3000 credit.withdraw(1000) => OK

saving.balance()=>1000000 saving.withdraw(1000) => error!

Language features and "orientation"

- OO = think objects first

$\leftrightarrow$

- OOP features: class, method, instance vars., method call, inheritance, overriding, ...

How the features make OO possible?

- Are they essential to OO?

Characteristics of OOPLs: Encapsulation

- When we focus on one object
  - we distinguish the focused object and others: to what extent the object "itself"?
    → "object" as one value is a natural unit called "encapsulation"
  - we ignore other objects:
    other objects can only be accessed through method calls (cf. ADT)
    vs. instance vars in "self" can directly be accessed

Characteristics of OOPLs: Encapsulation

- When we focus o
  - we distinguish th
    others: to what e
    → "object" as on
    called "encapsulation"
  - we ignore other objects:
    other objects can only be accessed through method calls (cf. ADT)
    vs. instance vars in "self" can directly be accessed

```
class Point {
  int x, y;
  boolean equals(Point other) {
    return this.x == other.x
         && this.y == other.y; } }
```

are we focusing on one "object"?

Characteristics of OOPLs: polymorphism

- each object in real world behaves differently
  - same action can result in different responses depending on objects --- called "polymorphism"
    → realized as ***method dispatching***
  - eg: saving.withdraw(10000) => error
    credit.withdraw(10000) => OK

Note: polymorphism (broader sense): ability to apply different types of data to the same program

  - "Polymorphism" in functional languages: both lists of integers and lists of strings cat be applied to List.length

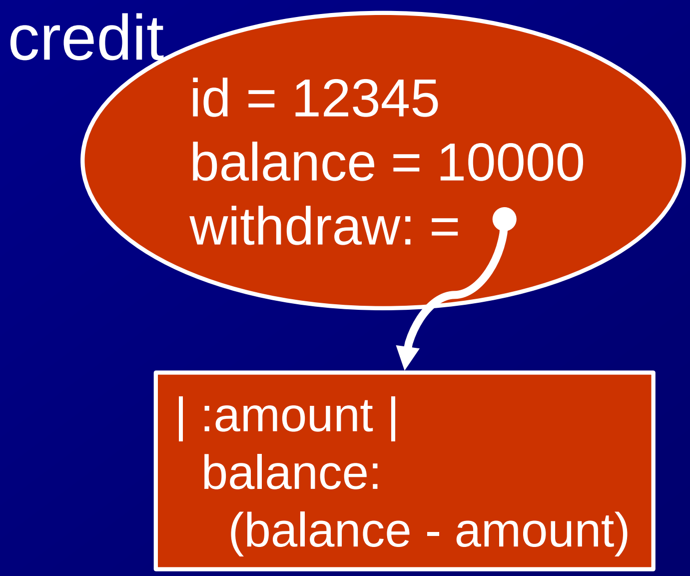
Characteristics of OOPLs: inheritance

- treat ***similar*** objects as one kind of value
  - ➢ single definition for objects with the similar properties
  - ➢ define "difference" for objects with slightly different properties

  → realized by ***class + inheritance***
- akin to "hierarchal categorization", "frame" (for knoweldge representaiton)
  eg. "a penguin is a bird yet cannot fly"

However, it is possible to "think by centering objects" by means of different bundling mechanisms

Classless OOPL: SELF [US87]



credit withdraw: 3000.

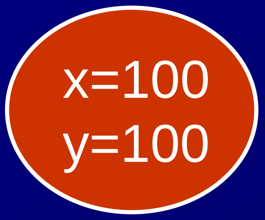
Instance-based OOP
(vs. class-based OOP)

- Object = set of instance vars.
- Methods are also values
  (called "blocks")
  - can be stored in instance vars.
- Method call
  = obtain a block in an instance var.
  + execution of the block
  (with binding self)

SELF: Object construction

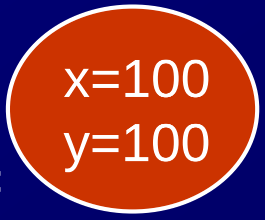
- to create a set of instance variables
  aPoint <- (| x = 100. y = 100. |)

aPoint=



- to copy an existing object
  bPoint <- aPoint copy.

bPoint=

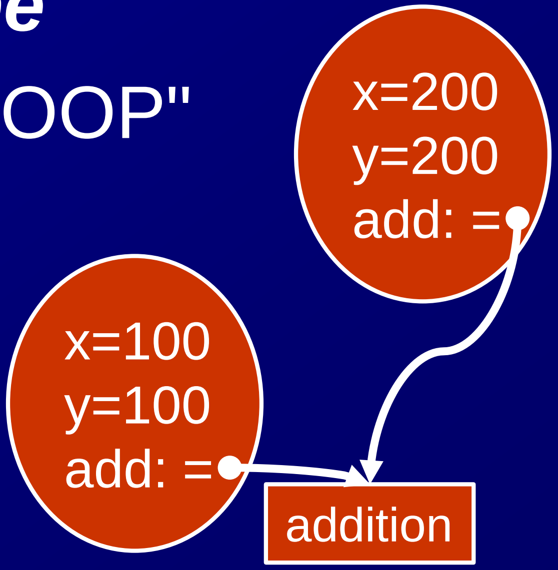


SELF: bundling similar objects together

- by constructing a reference object, and copying from that object each time
  - reference obj. = a ***prototype***
  - so called "prototype-based OOP"
- cPoint <- (| x = 100. y = 200.

```
  add: other = (
     (copy x: (x + other x))
              y: (y + other y)).
|)
```

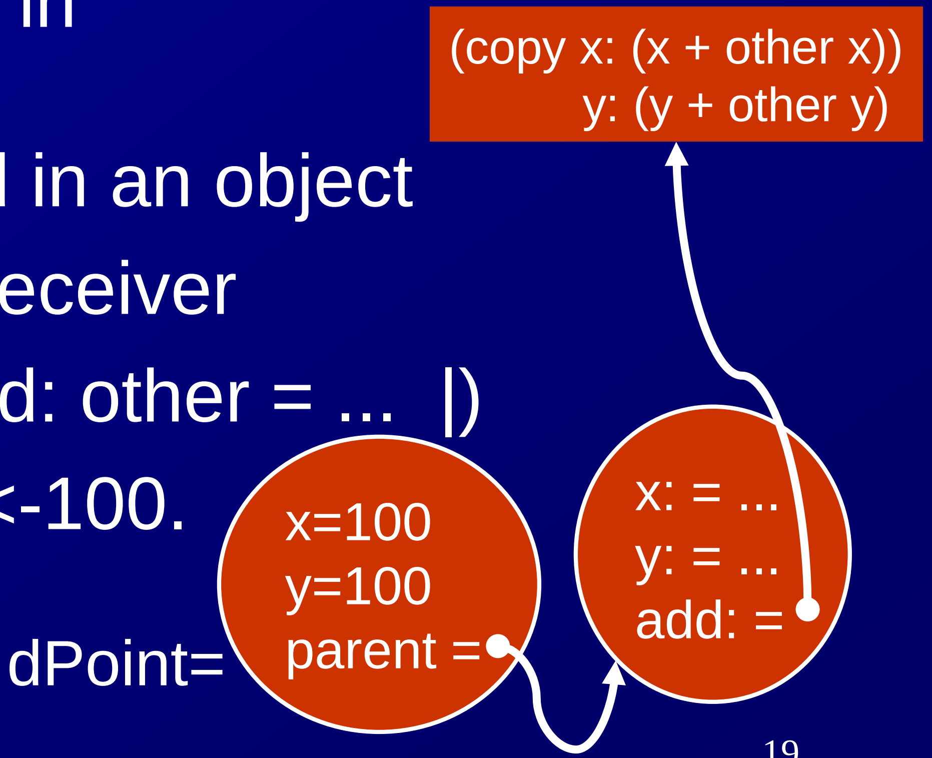
cPoint=



- cPoint add: cPoint.

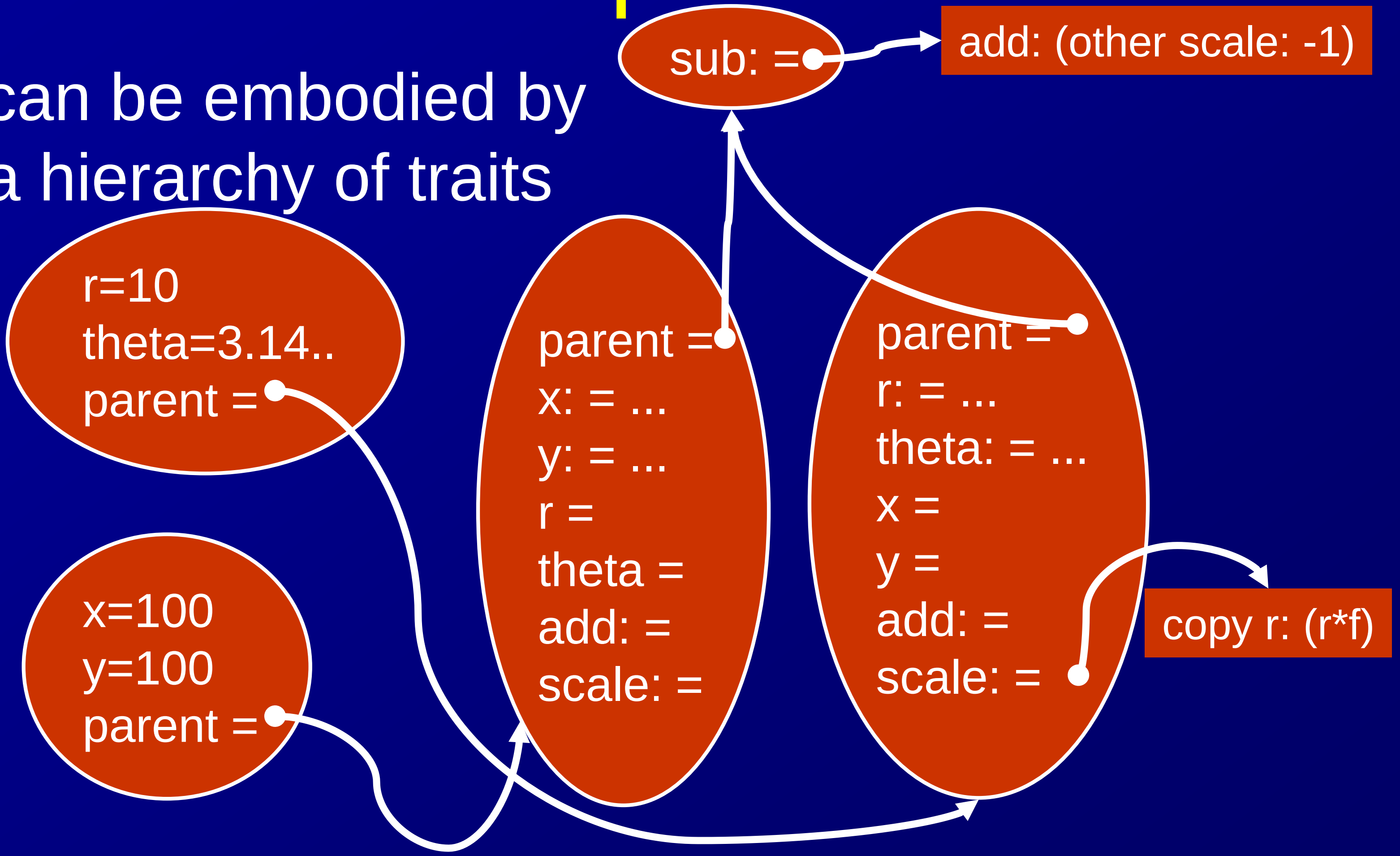
SELF: bundling similar objects together

- Delegation
  - searches a method in other objects when no definitions found in an object
  - "self" refers to the receiver
- pointTrait <- (| x. y. add: other = ... |)
- dPoint <- (| x<-100. y<-100. parent*=pointTrait |)
- dPoint add: dPoint.



SELF: hierarchical decomposition

- can be embodied by a hierarchy of traits



Why instance-based?

- any object can be a unit of grouping
  - no need to create a class for one exception
- less language constructs
  - only object, instance vars, and delegation
- invention of faster implementation tenchiques
  - Polymorphic inline cache (PIC), dynamic compilation, etc.

other than SELF: Simula, Javascript

References

- [所93] 所 真理雄, "オブジェクト指向計算", in オブジェクト指向コンピューティング, pp.1-56, 岩波書店, 1993.
- [US87] David Ungar and Randall Smith, “Self: The power of simplicity”, in *Proceedings of OOPSLA*, 1987.