OO Theory:

Class Hierarchy and Inheritance

서울대학교 컴퓨터공학부

Internet Database LAB

교수 김 형 주

Remember?

Recent Programming Languages의 문제점

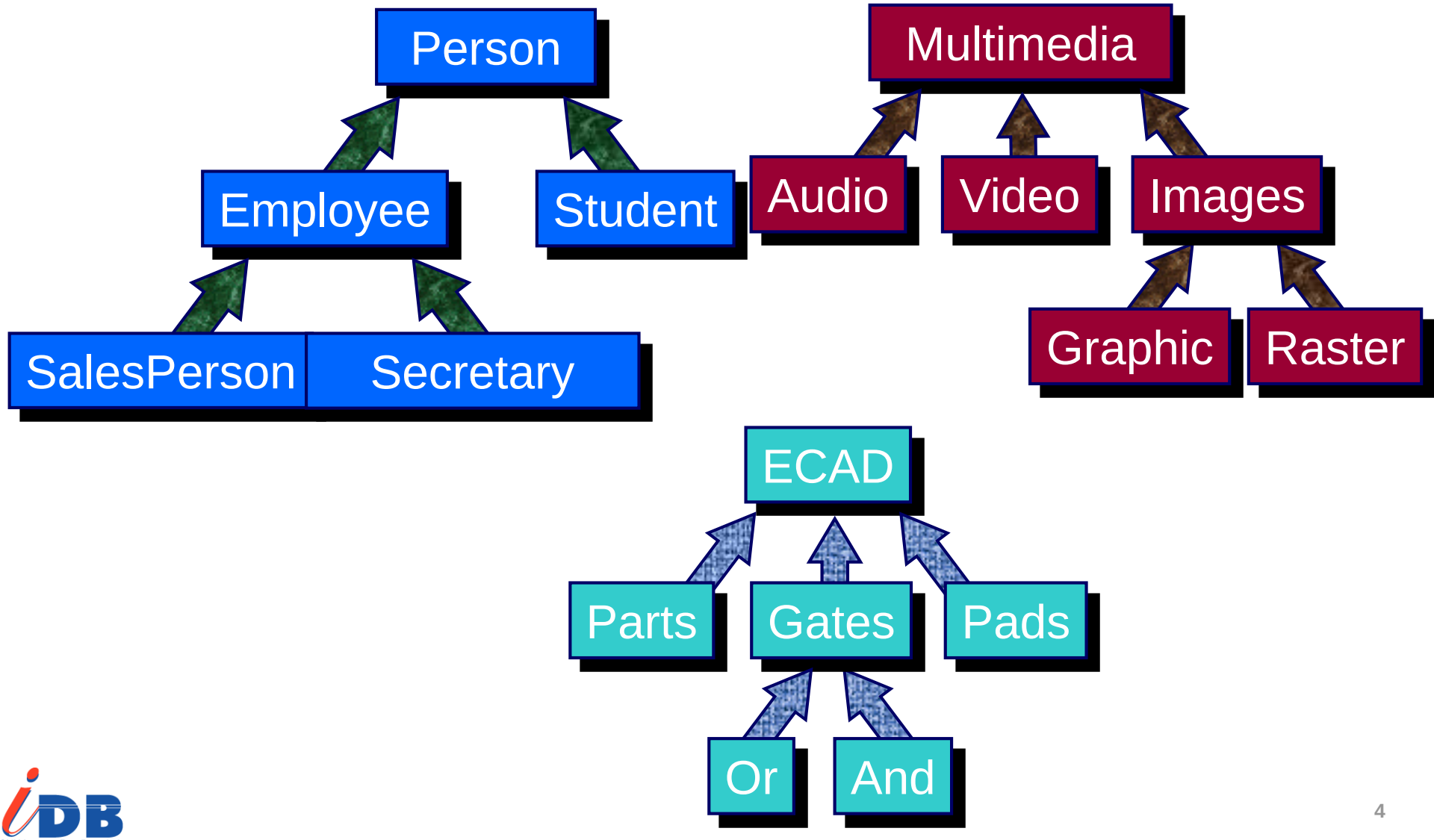
- Class, Subclass, Inheritance에 대해 가볍게 대처
- Always taking an easy and simple way
- Multi Paradigm Programming Language를 주장

- Programming Language Theory를 대폭 무시
  - Type에 대한 이론들에 무지
  - Correctness of Program의 중요성을 간과
  - Abstract Data Type의 기본 취지의 퇴색

Contents

Forget Java & C++ at the moment!!!

Examples of Inheritance Hierarchies

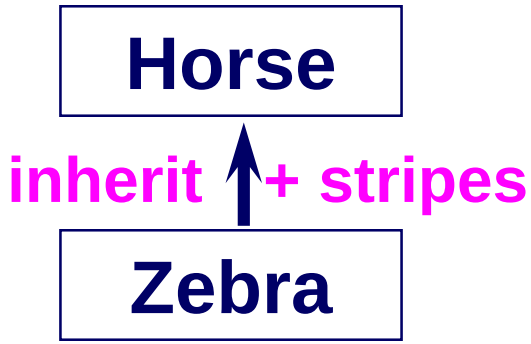


Inheritance with additional representation

A mechanism which allows a new class to be incrementally defined from an existing class.

Problem **What is a Zebra ?**

"A Zebra is like a horse but has stripes"

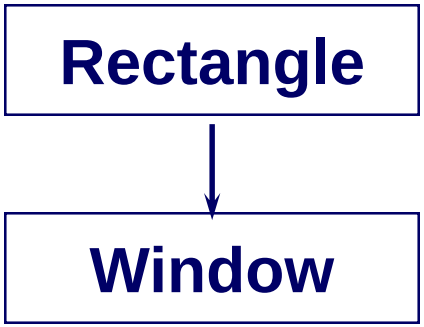


Inheritance avoid repetition and confusion!

Inheritance with additional behavior

Problem

Define a new class called Window, which is a rectangle, but also resizable.



add “resize” method

```
Class Window
    inherit Rectangle
    add operation
            resize(int h1, w1);
            { h=h1; w=w1; display(); }
```

Inheritance in OOPLs

The common operations supported by most OOPLs during inheritance are :

- *Addition* of new instance variables and methods
- *Redefinition*(*overriding*) of inherited methods

There are still other possibilities

- renaming
- *exclusion* of inherited methods
- redefinition of inherited attributes

Polymorphism from Inheritance

An object or methods may have multiple types.

Subtyping

An object has its own type and also supertypes. This is the most significant contribution of OOPLs toward polymorphism.

Parametric Polymorphism

Type definition may have type parameters.
eg. Stack[Int], Stack[Real], Stack[Process]

Overloading

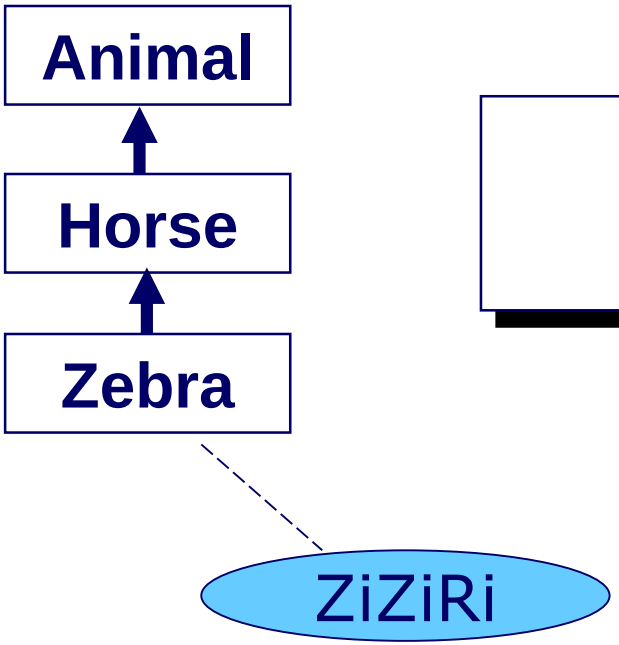
A single operation name may denote several different operation.
eg. Use of “+” for integer addition, array addition, and string con’n.

Coercion

Implicit type conversion inserted by the translator.
eg. Use of Integer for Real in FORTRAN

Subtyping (Subtype Polymorphism)

Subtyping allows an object to have not only its own type but also its supertypes.



A Horse is an Animal.
A Zebra is a Horse.
A Zebra is an Animal.

Subtyping makes programs more flexible and reusable

Function Overloading

Solution without polymorphism & inheritance

** Suppose we have types “Triangle” and “Rectangle”

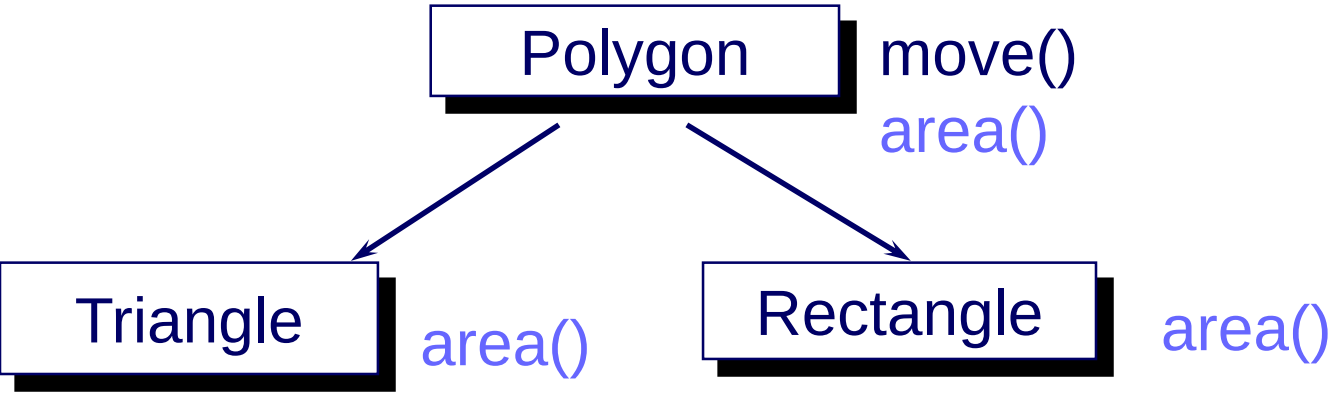
```
function compute_area(p: Polygon) : Real;

        case p.polygonType of
                Triangle:       return(p.w * P.h);
                Rectangle:    return(p.w * P.h/2);
        end case;

end: /* compute_area */
```

Function Overloading

Solution with Polymorphism & Inheritance




```
function compute_area(p: Polygon): real;
        return(p.area());
end;
```

** Consider a new type “Hexagon”

Terminology

- Inheritance → Reusability, Extensibility
- Inheriting behavior → code sharing
- Inheriting representation → structure sharing

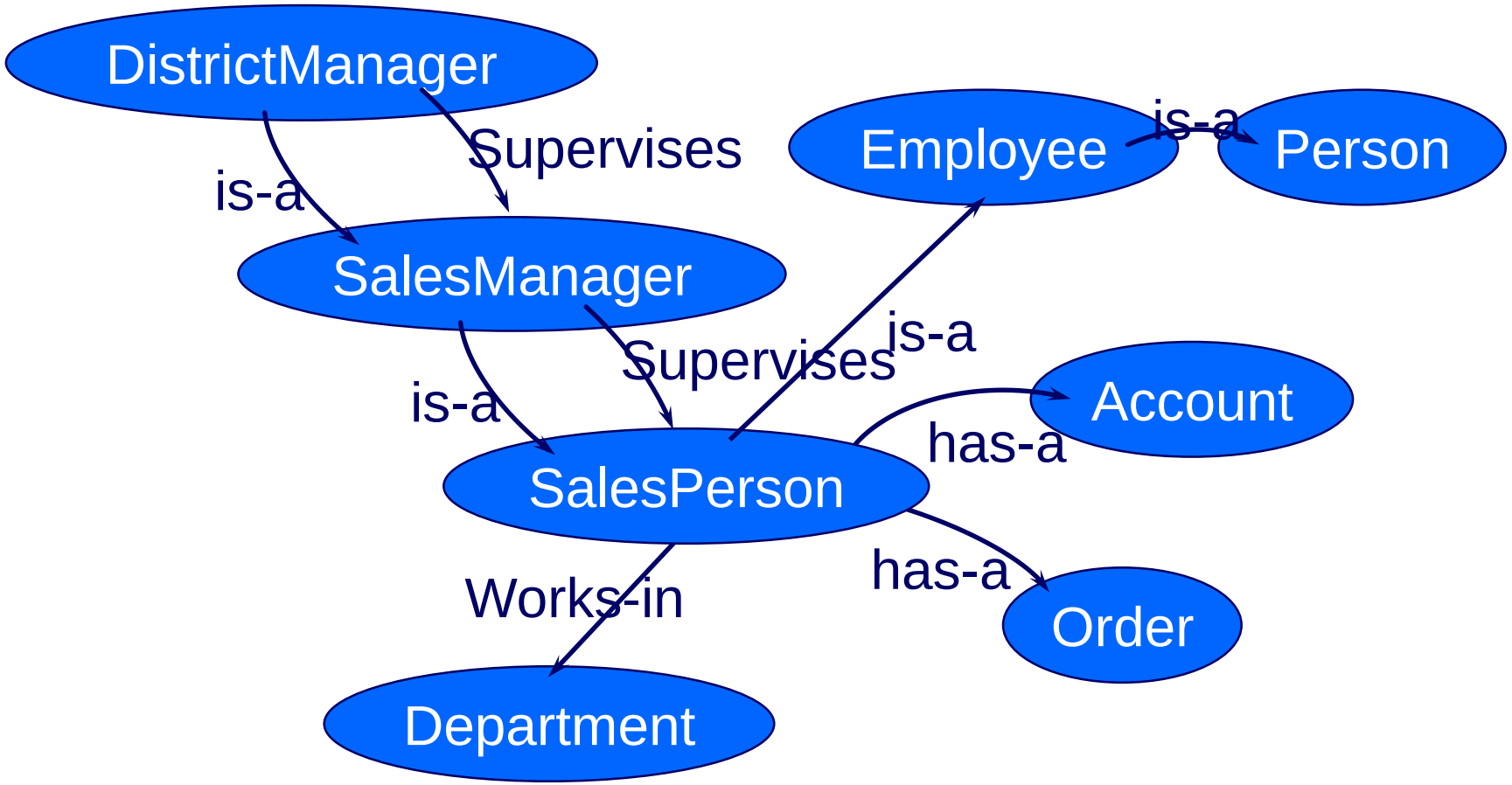
- A natural mechanism for organizing information
  - There are many similar things in the real world
  - SW developers tend to similar SWs in a certain domain

- Primary Concern
  - Taxonomizing objects into well-defined inheritance hierarchy

Origin of Inheritance

- Semantic network – *Quillian (1968)*
  - "Node-and-link" model
  - *Node*: concepts(objects)
  - *Link*: relationships among concepts
  - *Label*
    - 'IS-A': inheritance relationships
    - 'HAS-A': attributes of concepts
- Frames – *Minsky (1975)*
  - Record-like structure: *Slot*
  - New concepts from previously defined frames

AI Semantic Network for SalesPerson



Contents

2. Inheritance and Subtyping

The concepts of inheritance and subtyping are often confused!

- Inheritance
  - Implementation side
  - Implementational hierarchy
    - Inheriting instance variables
    - Inheriting methods
- Subtyping
  - Semantic relationship among the types of objects
  - Interface side
  - behavioral hierarchy

2.1 Subtyping

- A type T_1 is a *subtype* of type T_2 IF every instance of T_1 is also an instance of T_2

  ex) prime number vs. integer
- *Principle of substitutability*
  - If T1 is a subtype of T2 then an instance of T_1 can be used whenever an operation expects an instance of type T_2
- *3 kinds of subtyping*
  - *Subsets* / *Subtyping of Structured Types (tuples)* / *Subtyping of functions*
- Subtyping relation: a partial order (Reflexive, Transitive, Anti-symmetric)
- Inheritance hierarchy
  - Single inheritance: Tree structure
  - Multiple inheritance: DAG structure

2.1.1 Subsets as Subtype

- We can view "Subset" as Subtype, but not always correct!
- Example: the type Integer "I and the subsets of integers
  - R = the integers in the range 1...100
  - P = the prime numbers
  - E = the set of even integers

- R is a subtype of I ? / P is a subtype of I ? / E is a subtype of I ?
  - Algebraic Property of "I" (API) : associativity, distributivity
  - Closure Property of "I" (CPI): the sum or product of two integers are also an integer
  - R,P,E preserve API, but Only E preserves CPI

- Complete subtype - T1 to be a complete subtype of T2
  - Operators of T2 should behave compatibly with arguments from T1

2.1.2 Subtyping of Structured Types [1/2]

**** Inclusion semantics of subtype can be extended to tuples or arrays**

Type Person

[Name: Character String

Age: Integer

Address: Character String]

(Intstance 1)

[Name:"Mary Beth"

Age: 21

Address:"202 Spring St., MW76503"]

(Instance 2)

[Name:"Jonh Smith"

Age: 20

Address:"101 Spring St CA, 94563"

Salary: $25,000

Major:"Music"]

** Instance 1 and 2 are both members of type Person

2.1.2 Subtyping of Structured Types [2/2]

- Subtyping relationship between tuples (<=)

SalesPerson <= *Employee*

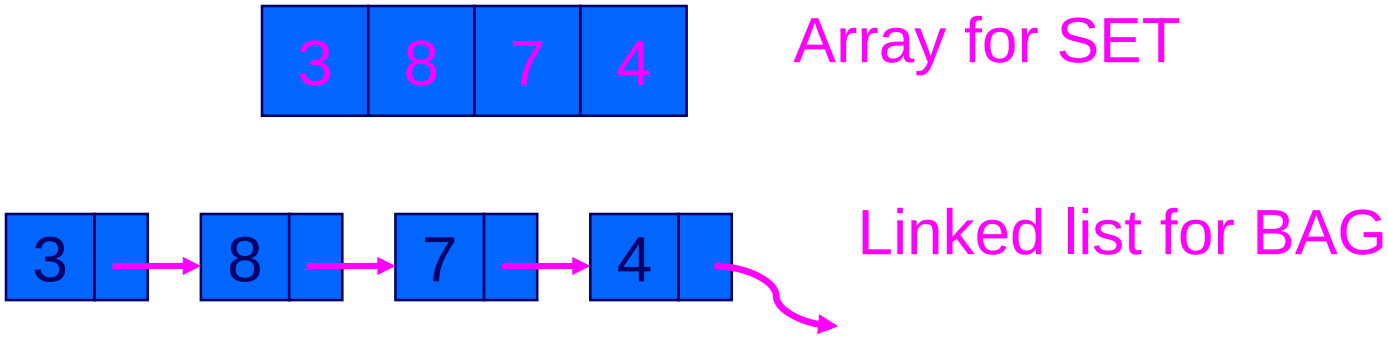
[Name: Character String
Age: 1...21
Address: Character String
Salary: Dollar
Major: Character String]

[Name: Character String
Age: Integer
Address: Character String]

* *Extend this concept to array, stack, tables…..*

2.2 Contrasting Inheritance with Subtyping

- Ex) Set vs. Bag (multiset)
  - Instances of *Set* could be implemented as *arrays*
  - Instances of *Bag* could be implemented as *linked lists*
- Implementations are different, but Set is a subtype of Bag

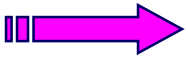


2.2.1 Implicit Subtyping vs. Explicit Inheritance

- Declaring subtyping relationships in OOPLs
  - Explicitly by naming a type to be a subtype of another type
    - Most conventional PLs
    - Through "superclass/subtype" clause
  - Implicitly by inferring the subtype relationship from the properties of the types
    - Type inferrencing in some OO languages such as Eiffel
  - Through hybrid schemes

2.2.2 Subtyping and Dynamic Binding

- T' is a subtype of T

Y : T';
X : T;
X := Y; 

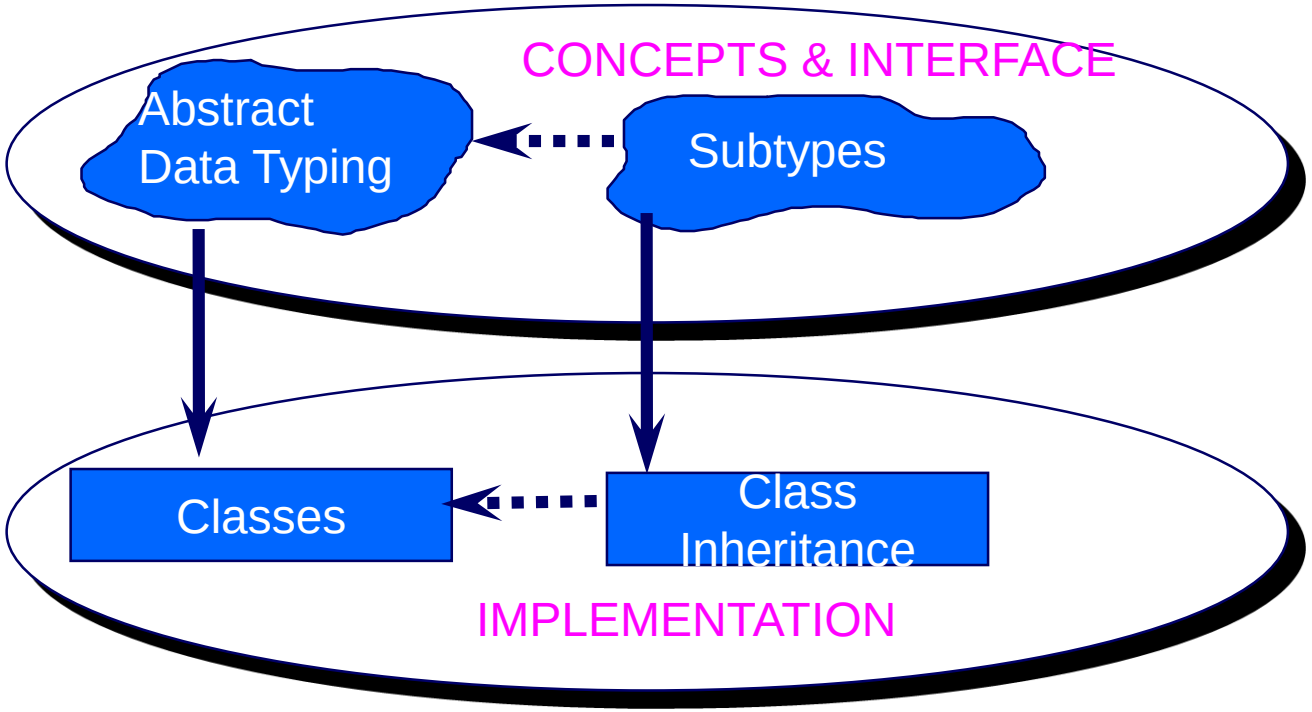
is *dynamically binding* X to an object of a different type (different from its static type)

- Dynamic binding also can apply to methods

(SalesPerson is a subtype of Employee)

```
Mary:   Employee;
Jill:     SalesPerson;
Mary.EvaluateBonus(…);
Mary := Jill;
Mary.EvaluateBonus(…);
```

2.2.3 What Do Classes inherit?



The *interface* of the superclass (a set of messages)

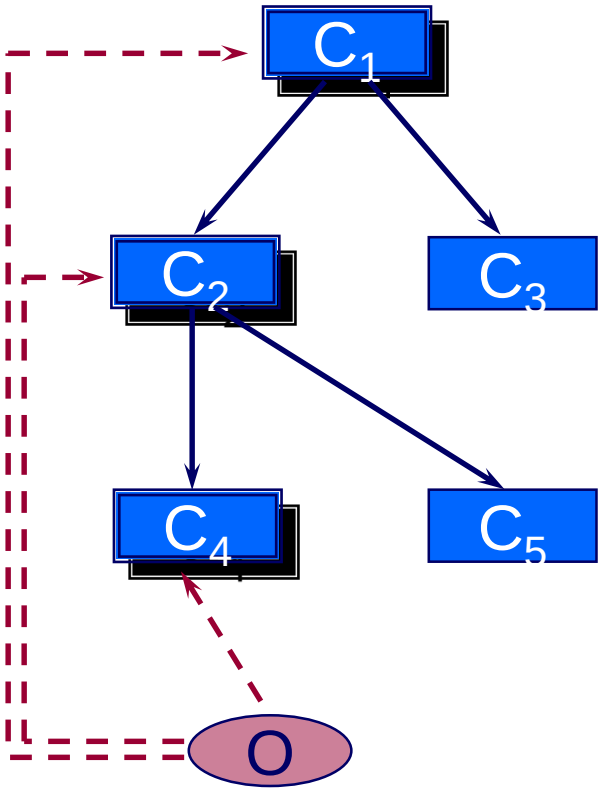
The *representation* of the superclass (a set of instance variables)

The *code* that implements the methods of the superclass (a set of methods)

Contents

3. Class Inheritance

- *O* := *new C*
  - *O* is a member of *C*
  - *O* is a member of every superclass of *C*
- *O* is a member of C_1, C_2, and C_4.



O := new C_4

Several Overriding Options

- For Instance Variables
  - No redefinition
  - Arbitrary redefinition
  - Constrained redefinition
  - Hidden definition
- For Methods
  - No redefinition
  - Arbitrary redefinition
  - Constrained redefinition

Overriding Flexibility vs. Typing

- Arbitrary overriding
  - Flexible & Cannot guarantee strong typing
- Constrained overriding
  - Less flexible & Guarantees strong typing & Type-safe

- To prevent a run-time type error in strongly typed language
    - All redefined methods in C' must conform with the corresponding methods in C
  - Allow only *dynamic binding* of variables to subclass instances

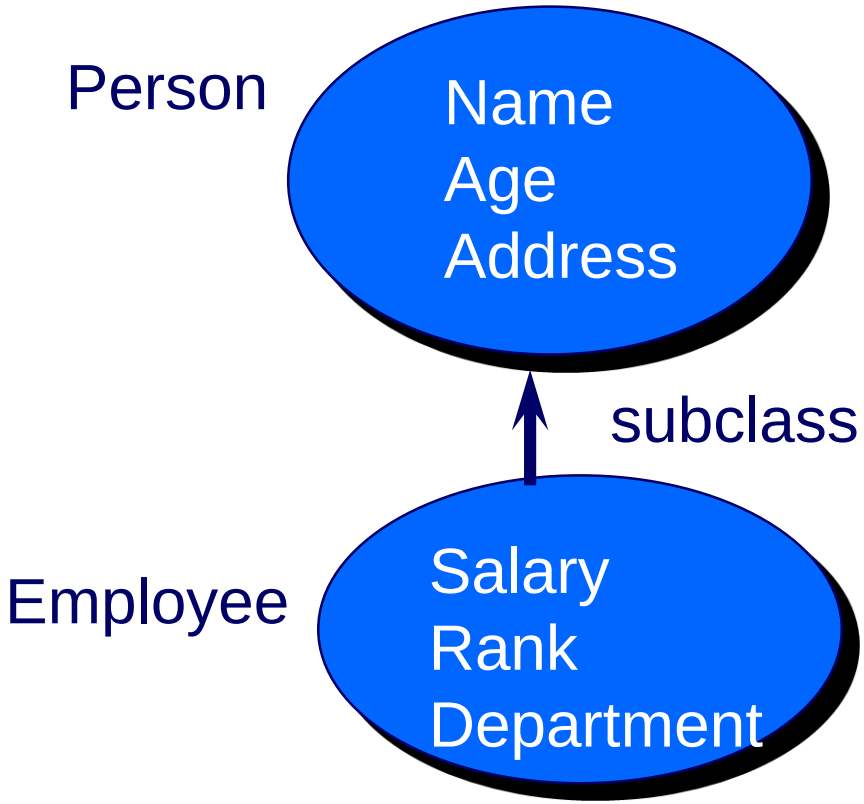
ex) Sp: SalesPerson

Sm: SalesManager

Sp := Sm

3.1 Inheriting Instance Variables

- Instances of a subclass must retain the same types of information as instances of their superclasses



State of an Employee

Name
Age
Address
Salary
Rank
Department

3.1.1 Redefining instance variables

- Arbitrary overriding
  - Problem: run-time type error
- Constrained overriding
  - The type of an inherited instance variable in the subclass must be a subtype of the type of the corresponding instance variable in the superclass
  - The type of an inherited & modified instance variable should be a subtype of the original instance variable!
- The trade-off between *arbitrary* and *constrained* overriding is between *flexibility* and *type-safe programming*
- Java에서는? Python에서는?

3.1.2 Hiding instance variables [1/3]

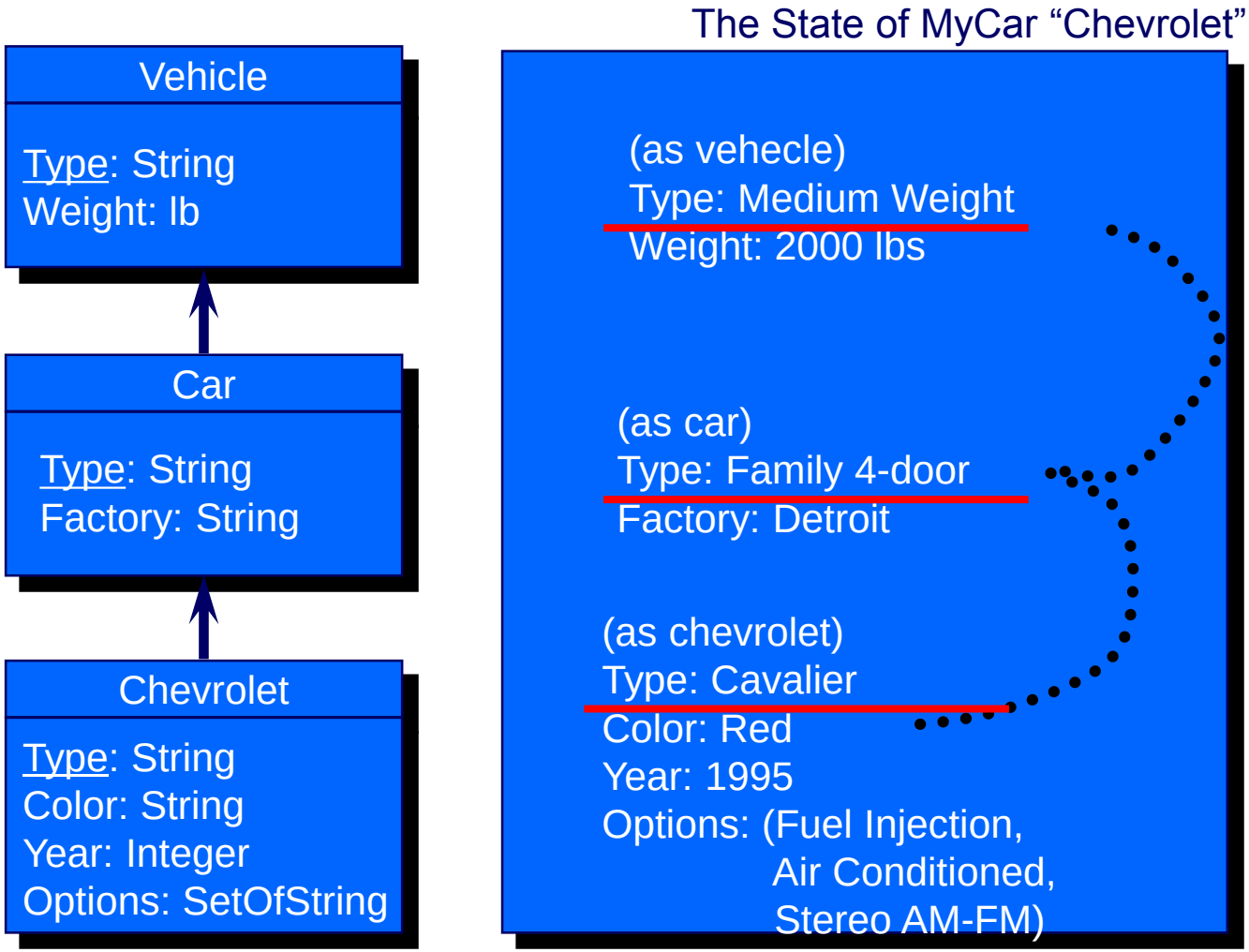
- The superclass can hide its variables from the subclass
- Controversial, sometimes dangerous!
- SalesPerson superclass and SalesManager subclass,
  - Suppose Account variable in SalesPerson is hidden!
  - Methods of Salesmanager can access Account (only through the methods of superclass)
  - An instance of SalesPerson can access Account directly

3.1.2 Hiding instance variables [2/3]

- *CommonObjects (Snyder, 1986)*
  - Supports independent definitions of instance variables in the inheritance class hierarchy
  - The inheriting clients (subclasses) cannot directly access or manipulate the instance variables of their superclasses
  - The instance variables of the superclass must be accessed and/or updated through the methods of the superclass
  - All "private" instance variables

** CommonObjects distinguished inherited variables and local variables with the same name.

** No overriding and all private!



3.1.2 Hiding instance variables [3/3]

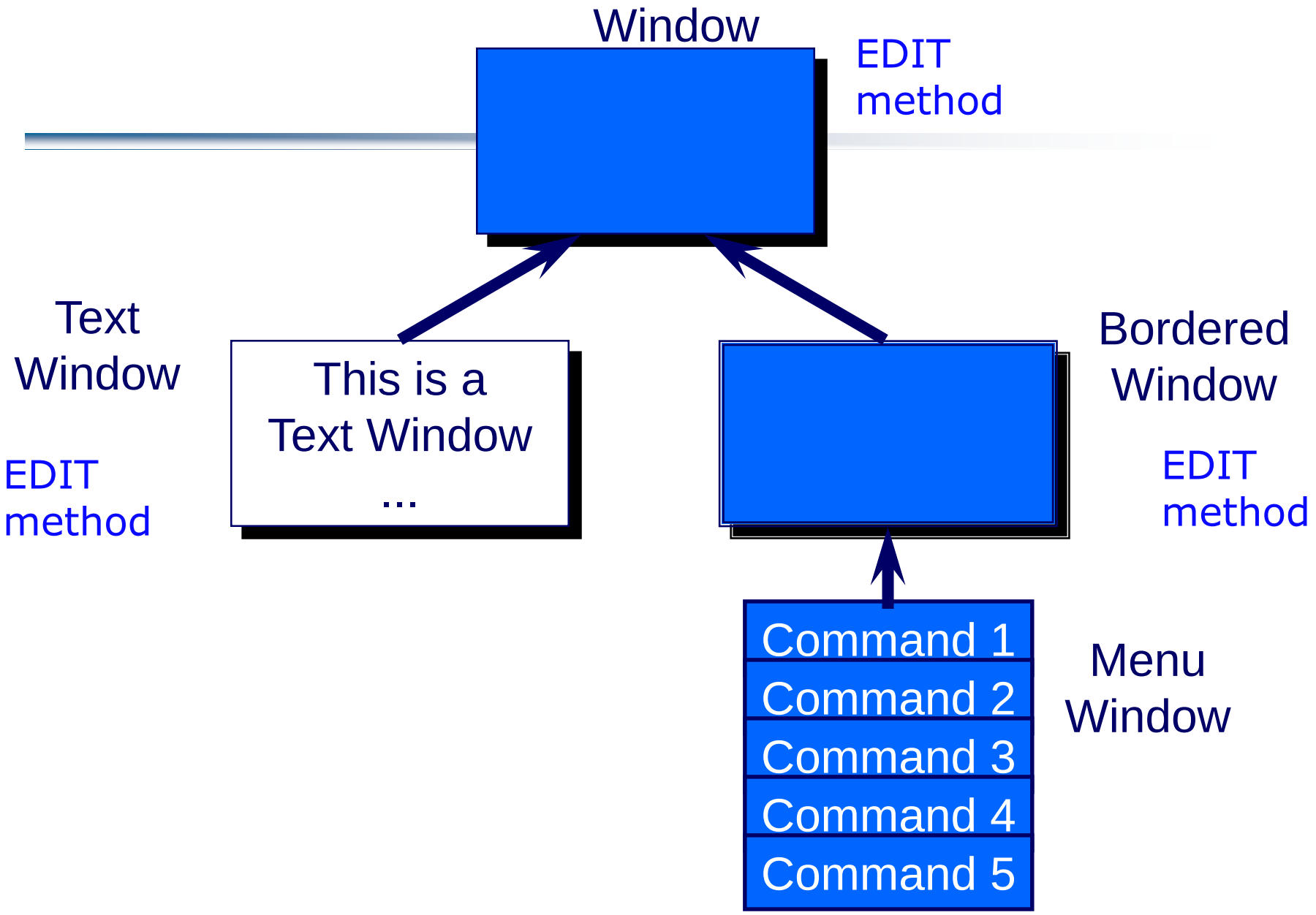
- The most general approach that combines efficiency and flexibility
  - *Public (open to any client)*
  - *Private (open to no client)*
  - *Subclass Visible (open to inheriting clients)*

    ex) "protected" (in C++)

- Java에서는? Python에서는?

3.2 Inheriting Methods

- A method defined for a class is inherited by its subclasses
- The inherited methods are part of the *interface* manipulating the instances of the subclass
- Sometimes, methods with same name in the sibling classes may be totally unrelated
  - Edit in Text-Window class is different from Edit in Bordered-Window class


Window
EDIT
method
Text
Window
EDIT
method
This is a
Text Window
...
Bordered
Window
EDIT
method
Command 1
Command 2
Command 3
Command 4
Command 5
Menu
Window

3.2.1 Method Overriding and Invocation

- Method Invocation Algorithm (bottom up to root)
  - Initially set *cC* (the "current" class) to *C*.
  - If a method *S* is declared in the definition of *cC*, then it is *M*.
    Stop searching and execute (invoke) it.
  - If *cC* is the root of the class hiearchy, generate an error and stop; the method is undefined.
  - Else set *cC* to its parent and go to step 2.

- Example (SalesPerson ← SalesManager)
  - *Mary* is an instance of *SalesManger*
  - *AddNewAccount* is defined only in class *SalesPerson*
  - *Mary AddNewAccount: NewAccount*

3.2.2 Invoking Superclass Methods [1/2]

- It is sometimes useful to call within a method of the subclass an overridden method of the superclass
- Use the pseudo parameter "super"

Examples: Employee ← SalesPerson ← SalesManager

StandardEmployeeBonus = (# defined in "Employee")
*Rank * 1000 + NumberOfYears * 100*

SalesPersonBonus = (# defined in "SalesPerson')
*StandardEmployeeBonus + TotalSalesAmount * 0.01*

SalesManagerBonus = (# defined in "SalesManager")
SalesPersonBonus + TotalSalesForceSales 0.005*

3.2.2 Invoking Superclass Methods [2/2]

- Invoking overriden superclass methods
  - When a message is sent to *super*, the search for the method starts with the superclass
  - Qualify the method with the class name: class.method
- Examples in *Smalltalk*
  - *EvaluateBonus* ← *defined in 'Employee'*

    *^((rank * 1000) + numberOfYears * 100))*
  - *EvaluateBonus* ← *defined in 'SalesPeson'*

    *^((super EvaluateBonus) + ((self TotalSalesAmount) * 0.01))*
  - *EvaluateBonus* ← *defined in 'SalesManger'*

    *^((super EvaluateBonus) + ((self TotalSalesForceSales) * 0.005))*

3.2.3 Constrained Overriding of Methods [1/5]

- Signature: the specification of the types of the input and output parameters of a method
- When does the signature of a function conform to the signature of another function?
- When is a signature $T_1 \rightarrow T_2$ a subtype of a signature $T_3 \rightarrow T_4$?

- *Covariant rule (intuitive, but not correct, but ok for most cases!)*
  - the arguments and the result of the method in the subclass be subtype of the arguments and the result of the corresponding method in the superclass
  - *(input parameter)* T_1 should be a subtype of T_3

    (output parameter) T_2 should be a subtype of T_4

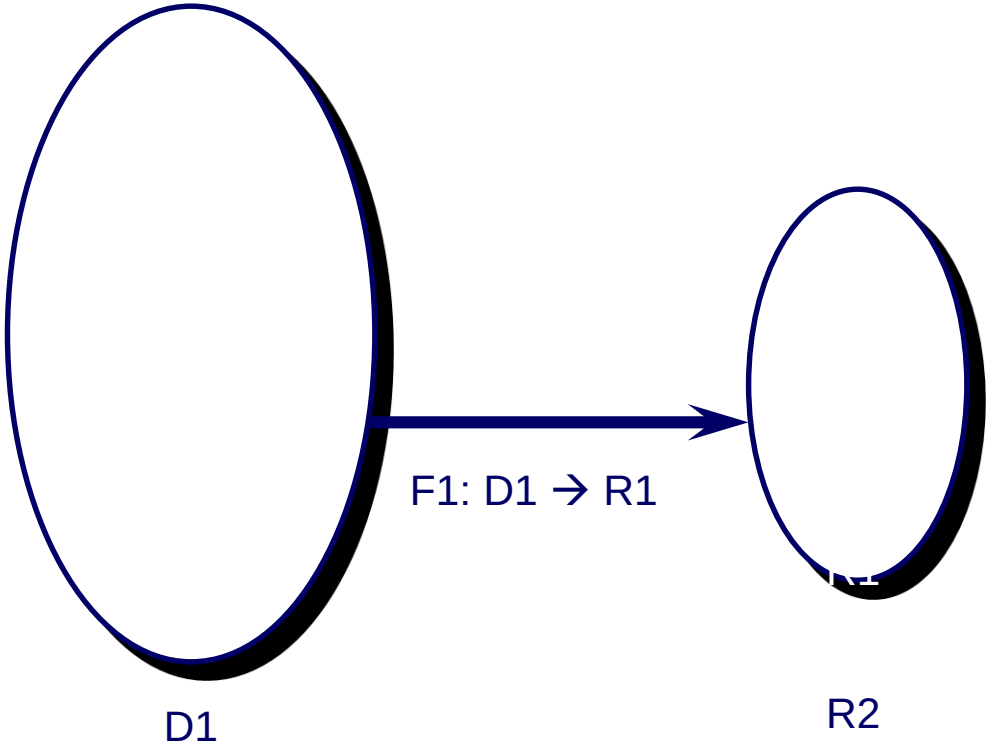
3.2.3 Constrained Overriding of Methods [2/5]

- *Contravariance rule*
  - Guarantee strong type checking and avoid run-time type errors
  - Contravariance applies to the *arguments* of the message or function
  - The input parameters of the modified method should be more special than those of the corresponding method
  - The output parameters of the modified method should be more general than those of the corresponding method

  - Not intuitive for programmers

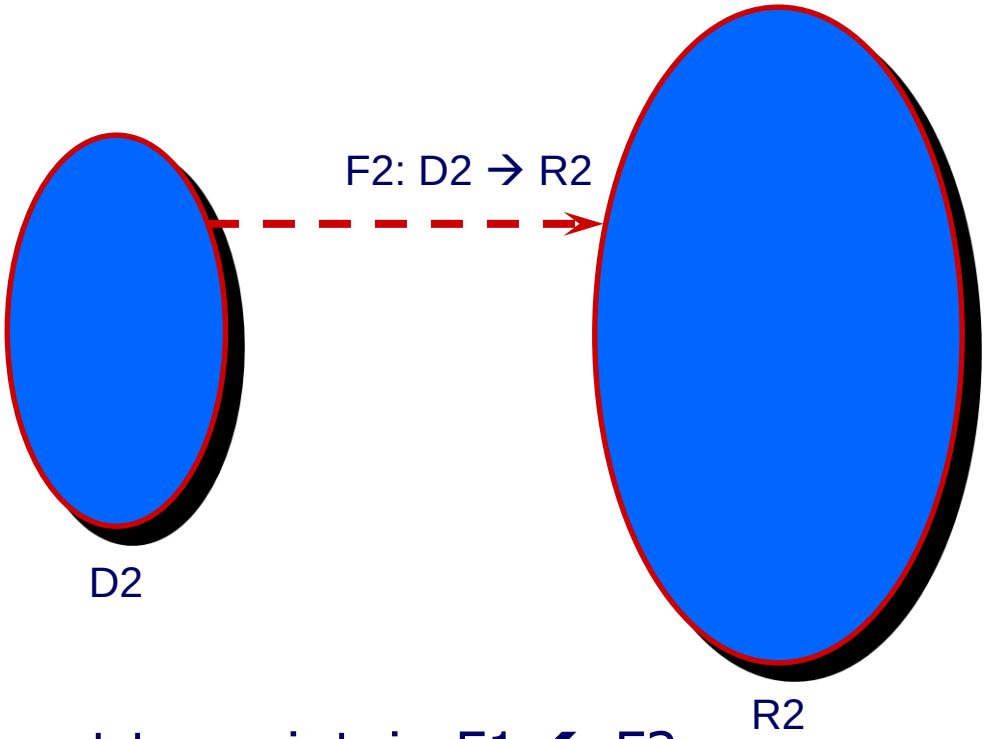
3.2.3 Constrained Overriding of Methods [3/5]

Suppose we have F1:D1 → R1 in a superclass SUP



3.2.3 Constrained Overriding of Methods [4/5]

Suppose we want to modify F1 into F2 in a subclass SUB



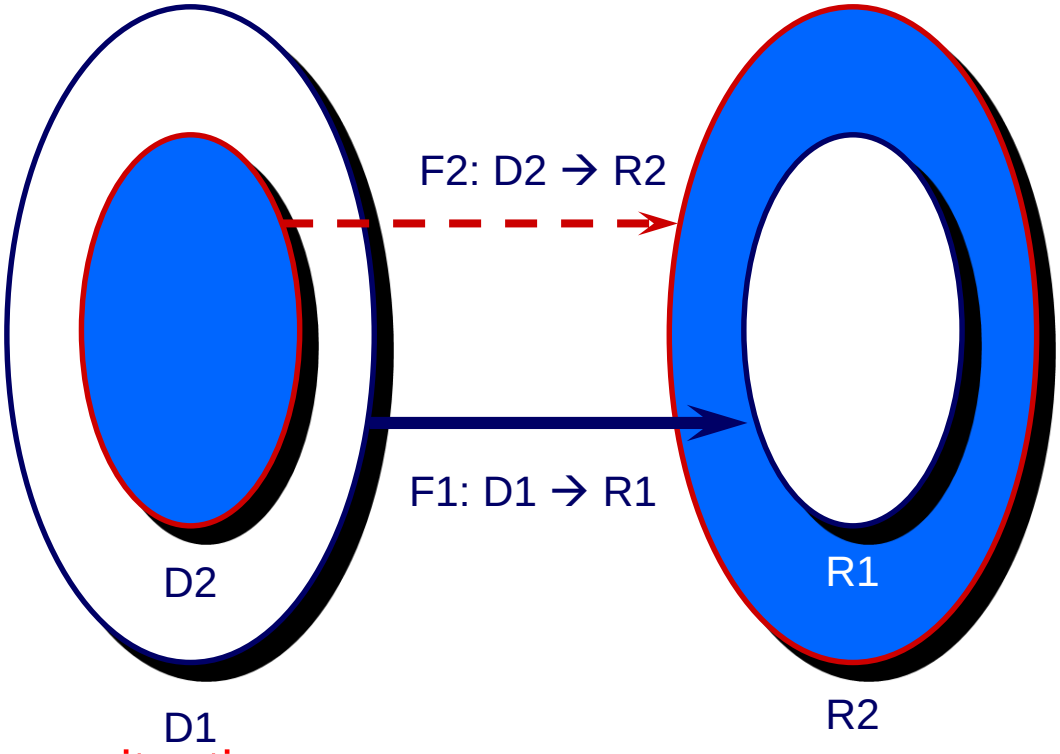
Of course, we want to maintain F1 ← F2

We want F2 can safely override F1

Yes, we want to keep SUP ← SUB relations

3.2.3 Constrained Overriding of Methods [5/5]

The function F2 with domain D2 and the range R2 is a function F1 with domain D1 and the range R1



** Under the above situation

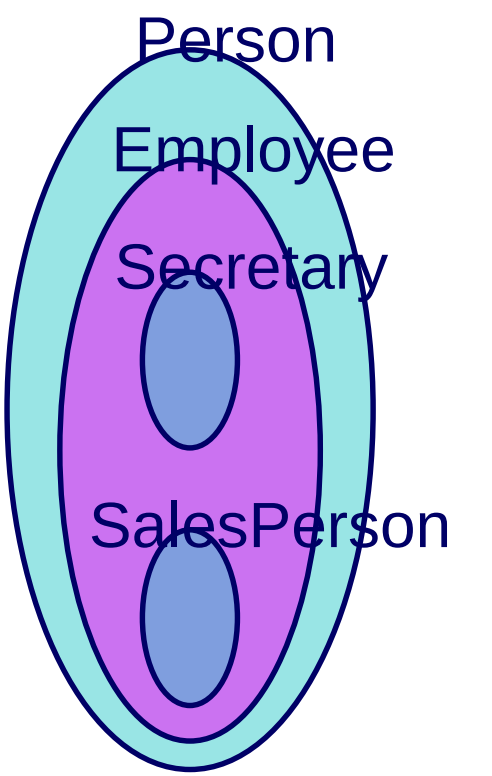
For all x in D2, the result (R1) of F1 is-a the result (R2) of F2

** F1:D1 → R1 ← F2:D2 → R2 IFF D2 <= D1 and R1 <= R2

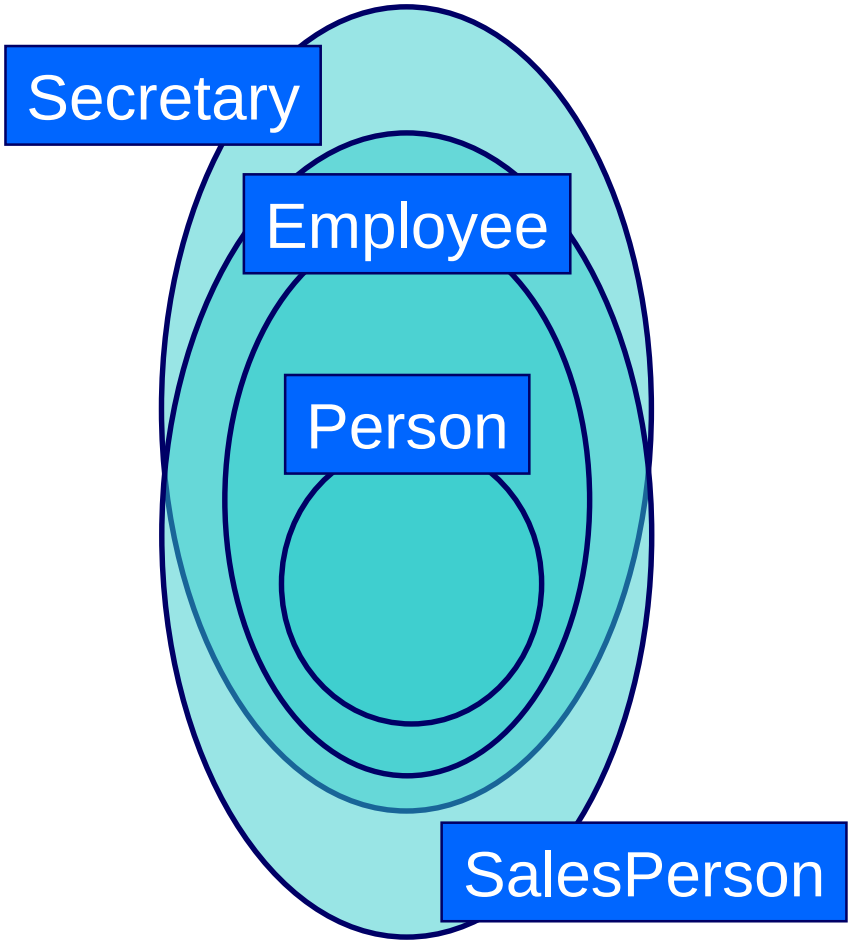
** F1 can safely override F2

3.2.4 Inheriting the Interface

- A class C_1 inherits from class C_2 but the *interface* of C_1 is a *superset* of the interface of C_2.
- Apart from specialization, inheritance also can be viewed as an *extension*.



Inclusion of classes

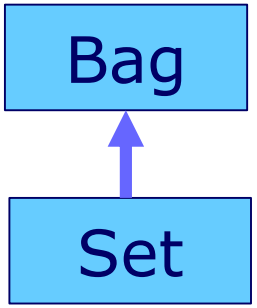


Inclusion of interface and/or representation

3.2.5 Excluding Superclass Methods

- Methods defined for the class *Bag*

  Insert *Difference*
  Delete *CartesianProduct*
  Intersect *NumberOfOccurrences*
  Union



- Create the subclass *Set*
- Exclude *NumberOfOccurrences* from the *interface* of the class *Set*
- 2 basic ways for excluding inherited methods
  - Override the method and send a diagnostic message when it is invoked on an instance of a subclass
  - Specifying implicitly or explicitly that the inherited method should not be inherited

- Java에서는? Python에서는?

Contents

4. Metaclasses

- *Possible questions*
  - *Is a class an object?*
  - *What is the class of a class?*

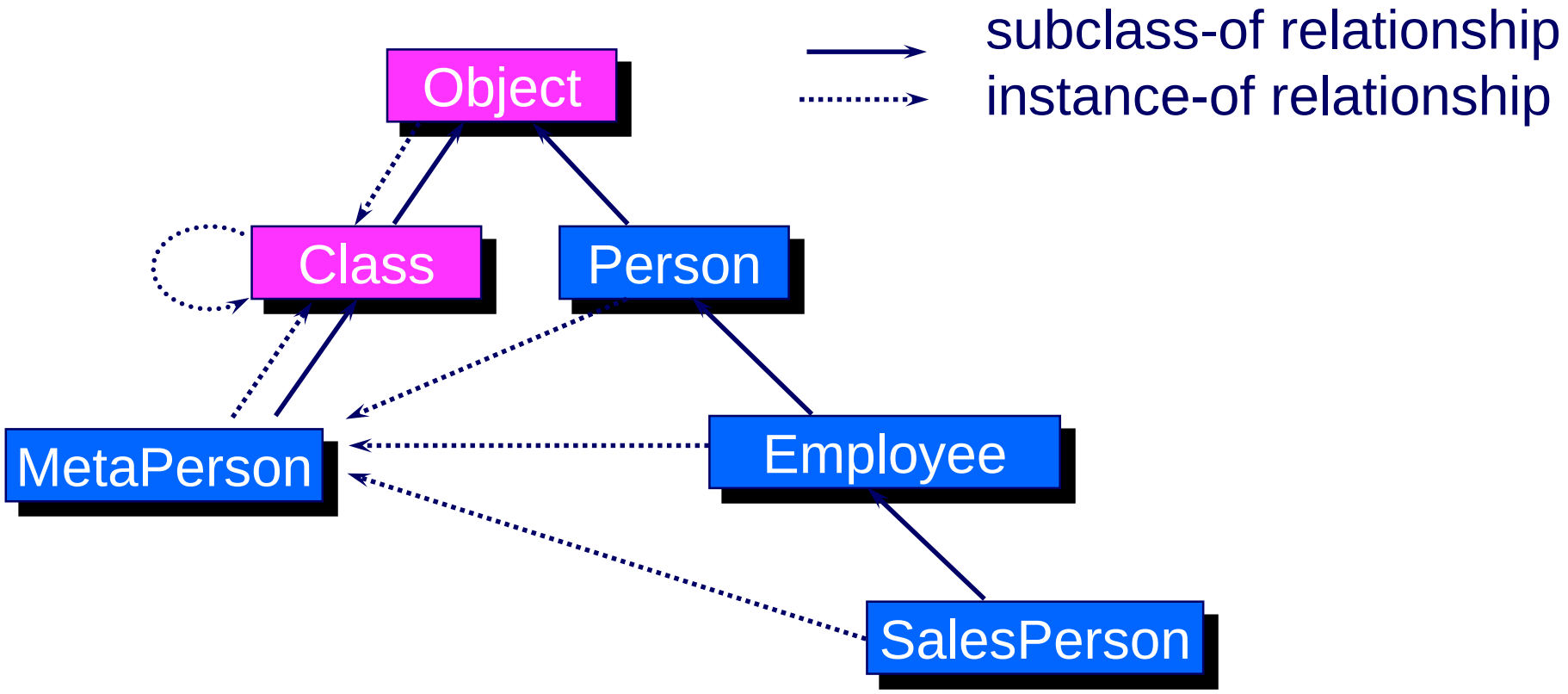
- *Metaclasses* are classes whose instances are also classes.
- Two advantages in treating classes as objects
  - Provide the storage of class variables and class methods
  - Conceptually convenient in creating a new class

- Class를 type으로 보면 MetaClass 고민이 없어짐

4.1 Explicit Support of MetaClass [1/3]

- *ObjVlisp* treats objects, classes, and metaclasses uniformly as objects
- Two built-in system classes
  - *Class*
    - its own instance, and a subclass of *Object*
    - metaclass: a subclass and an instance of *Class*
  - *Object*
    - the root of all classes (the most general class)
    - an instance of *Class*

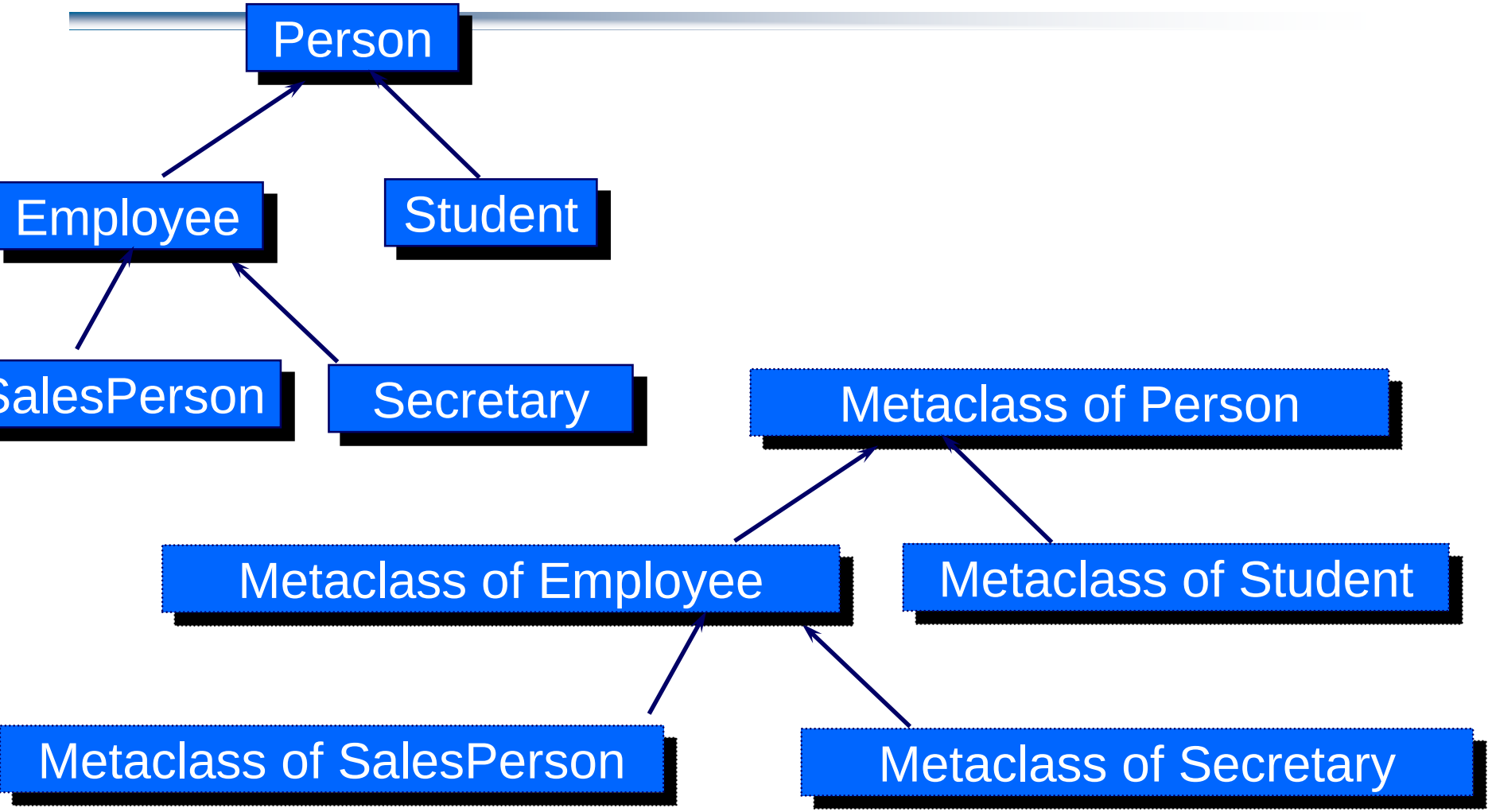
4.1 Explicit Support of Metaclass [2/3]



4.2 Implicit or Hidden Metaclasses

- In *Smalltalk*, Mataclasses cannot be declared and created explicitly
- Each metaclass has exactly one instance, which is its class.
- *Class methods*: the methods declared in a metaclass
- *Class variables*: the variables of the class in a metaclass
- Built-in classes
  - *Object*
    - Every object is an instance of *Object*
    - The *Object* class does not have a metaclass
  - *Class*
    - Every metaclass is a subclass of Class
    - The metaclass *Class* is a subclass of the *Object* class
  - *Metaclass*
    - All metaclasses are instances of the *Metaclass* class
    - The metaclass of Metaclass is *Metaclass class*

Implicit Metaclass:
Parallel “class hierarchy” and “metaclass hierarchy”



4.3 Various Viewpoints about Metaclass

- "Classes as objects"
  - *Smalltalk*: hidden metaclass approach
  - *ObjVlisp*: explicit metaclass approach
  - not satisfactory

- "Classes as types"
  - *C++, Simula, Eiffel*
  - a clear and mature technology

- What about Java? What about Python?

Contents

5. Object Inheritance

- An object instance *O* inherits the state of an object instance *O'*
  - if the value of an instance variable *i'* defined in *O' determines* the value of the same instance variable in *O*
- Instance inheritance & delegation

person.John

Name: John Smith
Age: 32
Address: 1212 Main St.

inherits

inherits

employee.John

Salary: $32,000
Rank: 15
Department: Hardware

student.John

GPA: 3.8
Advisor: Dr. Bill Jacob
Major: Business

inherits

salesman.John

Accounts: {11, 101, 235}
Orders: {55, 72, 113}
Quota: 5
Commission: 5%

John Smith

5.1 Prototype Systems and Delegation

- Prototype system
  - The distinctions between instance objects and class objects are removed
  - create concepts first and then ...
  - generalizing concepts by saying what aspects of the concepts are allowed to vary
- Delegation
  - The mechanism used to implement prototypes

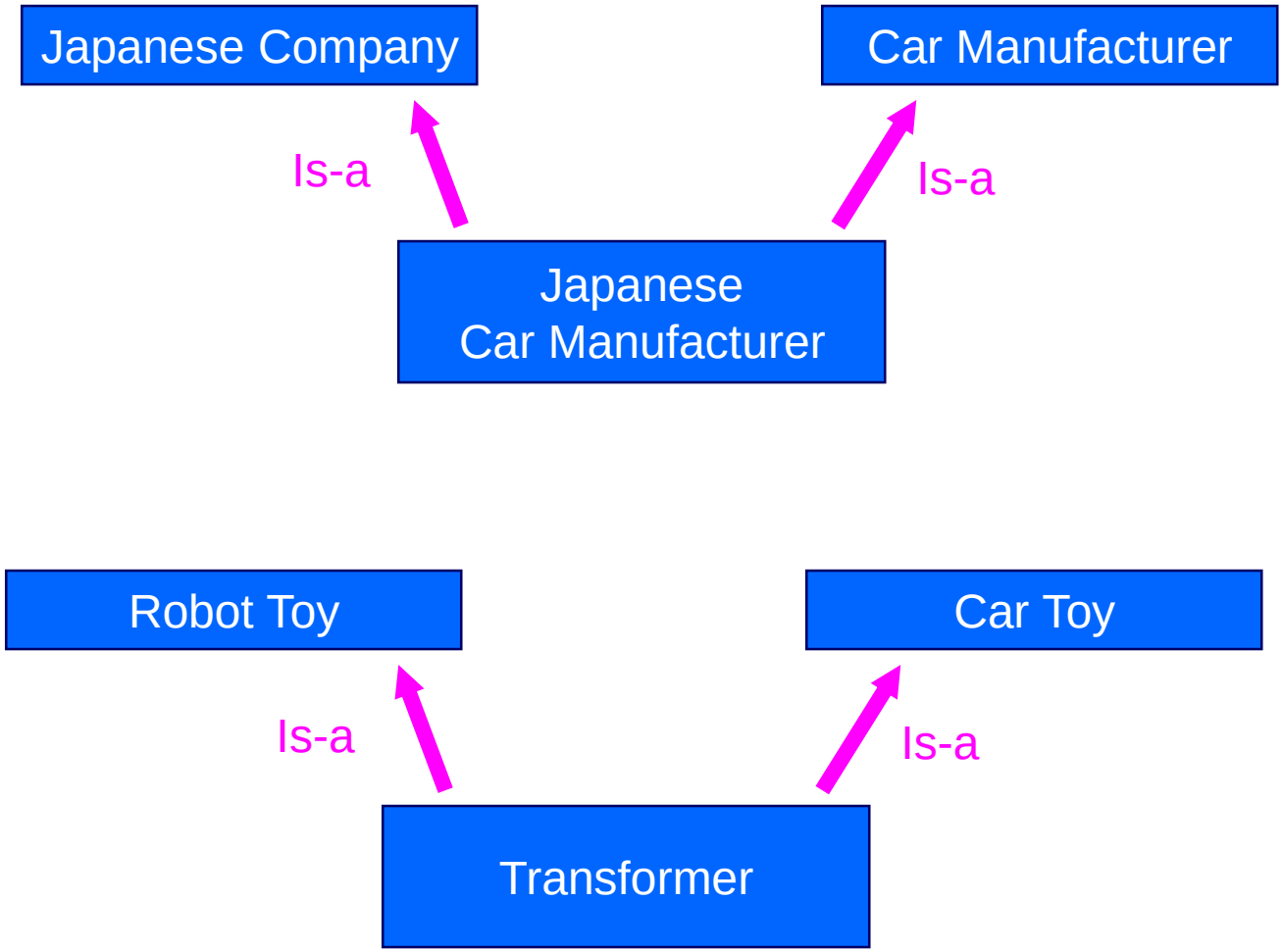
- 경우에 따라서는 유용할수 있는 feature
- Java에서는? Python에서는?

Contents

6. Multiple Inheritance

- The mechanism that allows a class to inherit from *more than one* immediate superclass.
- The class inheritance hierarchy becomes a *DAG* (Directed Acyclic Graph).
- "Conflict" arises when different methods or instance variables with the same name are defined by two or more superclasses.

Examples of multiple inheritance



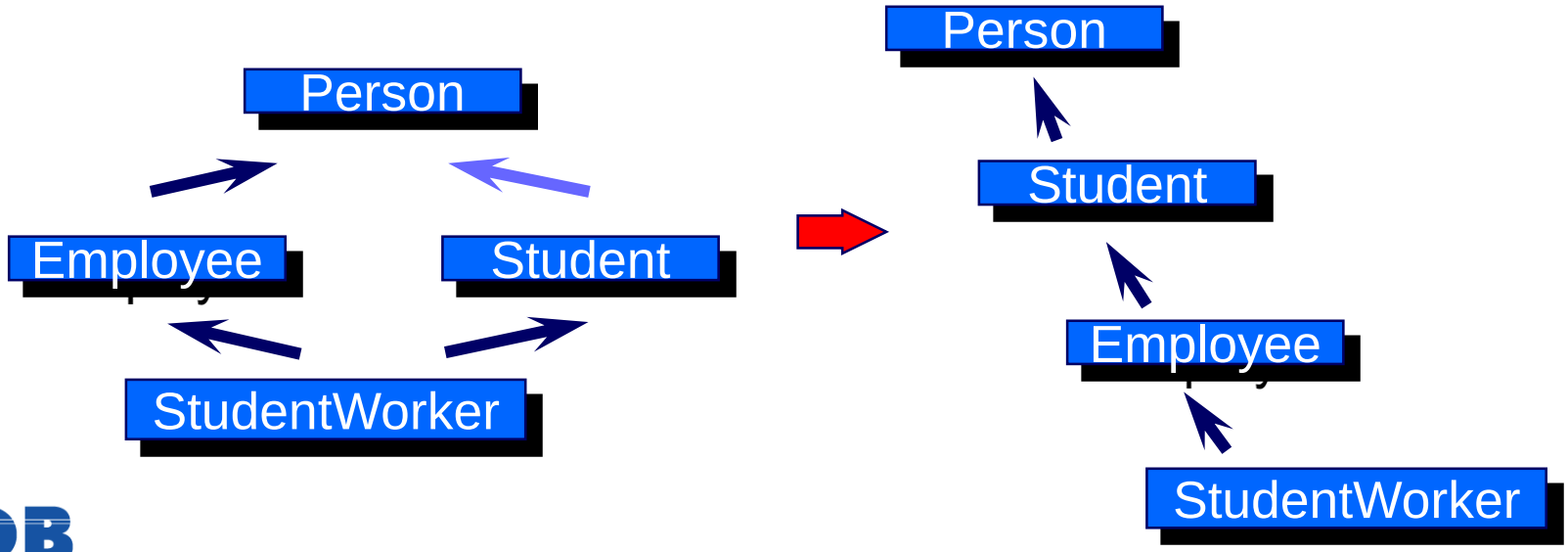
Conflict Resolution of Multiple Inheritance

- Two kinds of conflict
  - instance variables come from a common ancestor
  - instance variables are totally unrelated
- Conflict resolution strategies
  - Linearization
  - Forbidding conflicts and Renaming strategies
  - Qualifying instance variables and methods
  - The Meet operation for subtypes

6.1 Linearization

- Specify a linear, overall order of classes
- Mapping the DAG of the predecessors of a class into a linear order
- Problem
  - The ordering of superclasses in a class declaration has significant implications.
- Example

StudentWorker superclasses Employee, Student



6.2 Renaming Strategies

- The conflicting instance variables must be *renamed*
- Requiring renaming of conflicting instance variables or methods provides a lot of flexibility to the user
- Example(in *Eiffel* syntax)

class *TechnicalManager* inherit
Manager rename *Skill as ManagerSkill*
TechnicalConsultant rename *Skill as TechnicalSkill*

or

class *TechnicalManager* inherit
Manager rename *Skill as ManagerSkill*
TechnicalConsultant

6.3 Qualified Variables and Methods

- Qualification of variable or method names with the name of the class
  - *C++* uses this strategy
- Example

Manager::Skill ← *the Skill of Managers*

TechnicalConsultant::Skill ← *the Skill of TechnicalConsultants*

6.4 The Meet Operation

- Applies only to typed attributes

$T_1 = [a_1:t_1, a_2:t_2, a_3:t_3]$

$T_2 = [a_3:t_3', a_4:t_4]$

The meet of T_1 and T_2 is the greatest lower bound of T_1 and T_2 using the subtyping relationship.

$T_3 = T_1$ meet T_2

$= [a_1:t_1, a_2:t_2, a_3:t_3$ meet $t_3', a_4:t_4]$

YoungAdultAge = 20 -40
and
AdultAge = 30 -70

then the meet opreation:
YoungAdultAge meet AdultAge = 30 -40

6.5 Evaluating the Strategies

- Linearization
  - Hides the conflict resolution problem from the user
  - Introduces a superfluous ordering
- Renaming or qualifying
  - Puts the burden of conflict resolution on the user
  - Provide more *flexibility* to the user
- The meet strategy
  - Provides a clean semantics for multiple inheritance
  - Introduces certain limitations
- *Renaming and qualifying is the most promising strategies*
- Java에서는? Python에서는?

이런 OOP의 이론적 배경에도 불구하고...

- Fast Prototyping와 Easy & Flexible Coding를 강조하다보면 Class와 Inheritance의 부정확한 사용을 하게 될수있는데...

- What if! (가정의 시나리오)
- Inheritance의 이해가 서로 다른 여러명이 team projec을 한다면?
- 그렇게 탄생한 SW가 수년뒤 화성에 인간을 싫어나르는 우주선의 trajectory control SW라면?
- 그렇게 탄생한 SW가 전세계 stock marke들의 stock tradin을 동시에 가능하게 하는 SW라면?