OO Theory: ADT, Object and Class

서울대학교 컴퓨터공학부
Internet Data Base LAB
교수 김 형 주

Table of Contents

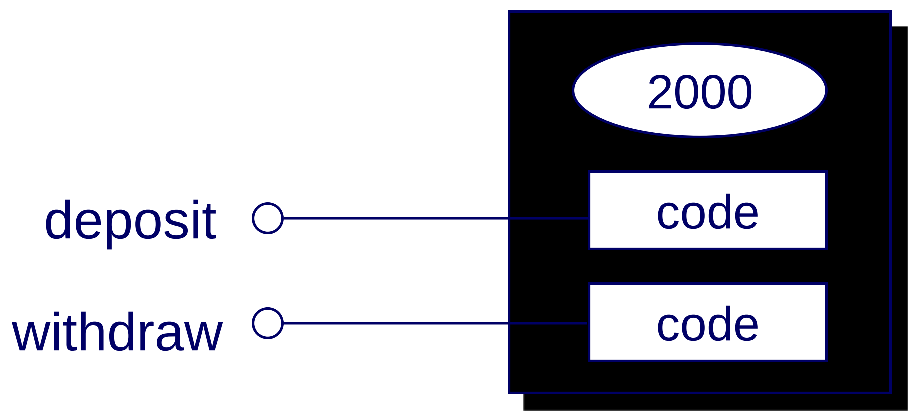
Forget Java & C++ at the moment!!!

SNU Internet DataBase Lab.

Object

An encapsulated software structure which usually models an application domain entity

Object = **Data + Operations**



Bank Account object

Related Terms

Instance variables

The variables(data) contained in an object are called *instance variables.*

Method

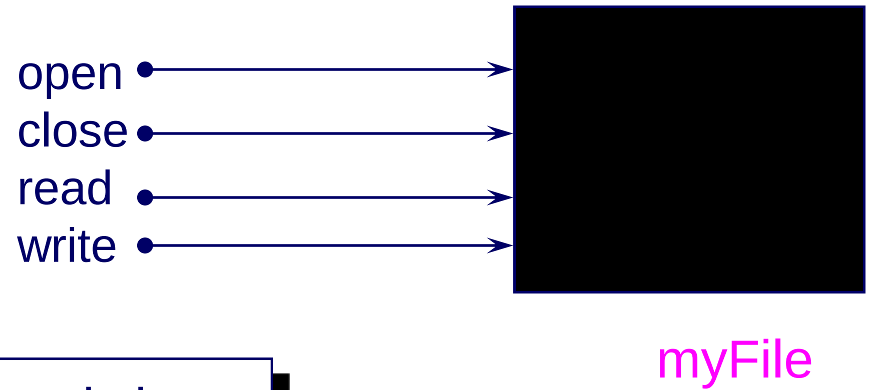
An operations of an object is called *method*.

Message

A request to invoke an operation is called *message.*

SNU Internet DataBase Lab.

Example: File Object



Object-oriented view

myFile open() : myFile, please open yourself.

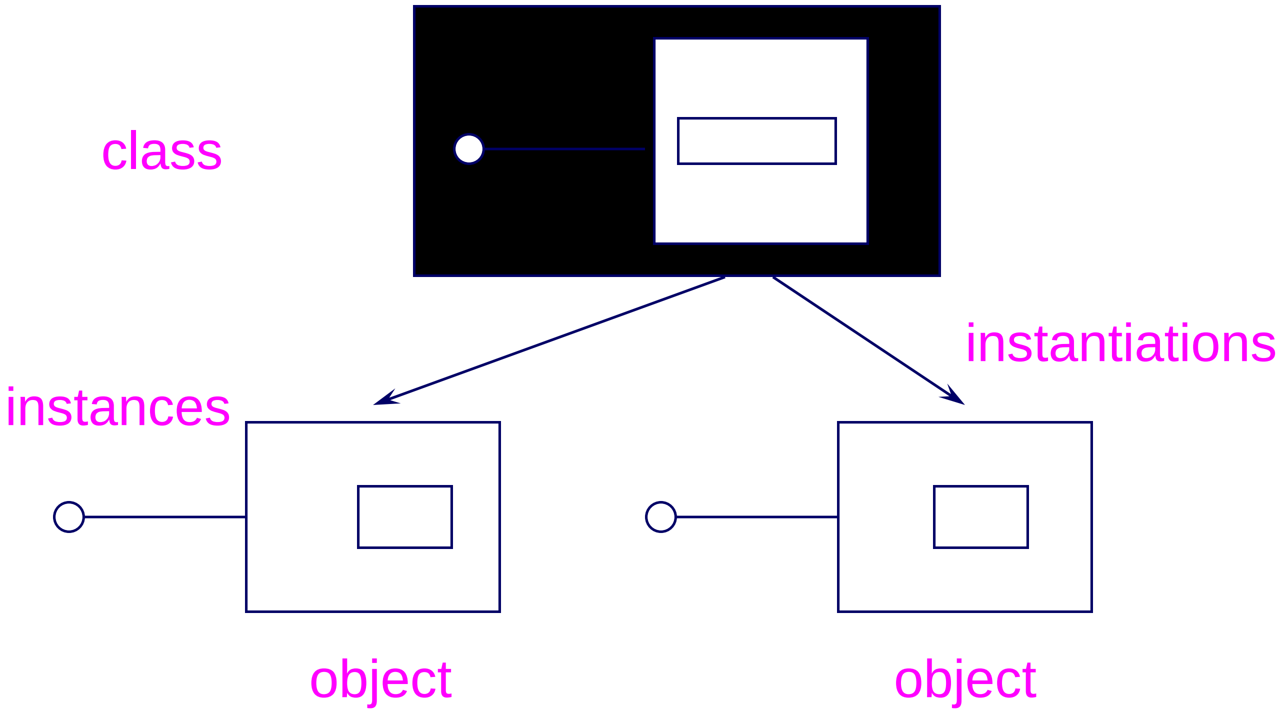
myFile read(ch) : myFile, please give me the next char.

myFile close() : myFile, close yourself.

SNU Internet DataBase Lab.

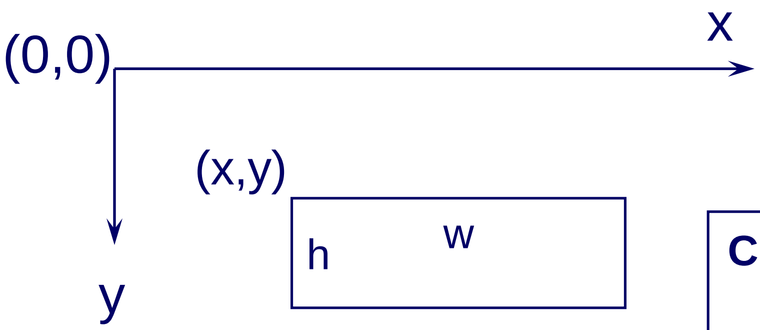
Class

An abstract data type which define
the representation and behavior of objects.



Example: Rectangle

Define a Rectangle Class.




```
Class Rectangle
   data
            int x, y, h, w;
   method
            create (int x1, y1, h1, w1)
              { x=x1;   y=y1;   h=h1;   w=w1; }

            moveTo (int x1, y1)
              { x=x1;   y=y1;   self.display(); }

            display ()
              { drawRectangle(x, y, h, w); }
End Class
```

SNU Internet DataBase Lab.

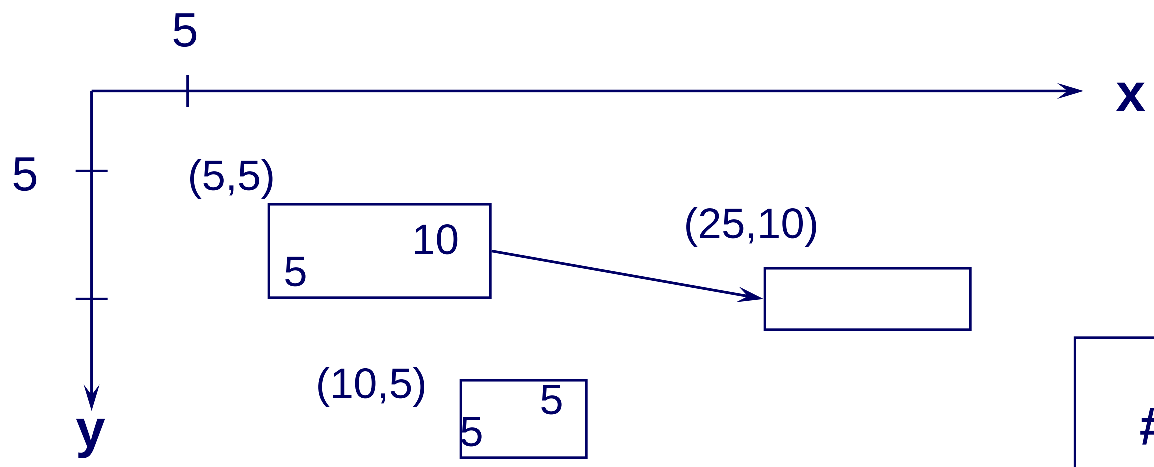
Interface of Rectangle

```
class Rectangle {
   operations
         create(int x1, y1, h1, w1);

         moveTo(int x1, y1);

         display();
}
```

Example: Using Rectangle




```
#Import Rectangle

Rectangle  r1, r2;
r1.create(5, 5, 10, 5);
r1.display();
r1.moveTo(25,10);

r2.create(10, 15, 5, 5);
r2.display();
```

SNU Internet DataBase Lab.

Related Terms

Behavior

The set of methods exported by an object is called its *behavior* or *interface*.

Encapsulation

The data of an object can only be accessed via the methods of the object.

Data Abstraction

Definition of an abstract type.
Encapsulation is need.

Table of Contents

SNU Internet DataBase Lab.

1. Introduction

- ADT(Abstract Data Types)
  - represented and implemented through classes
- Class
  - basic implementation and representation unit
  - a class is basically a type

SNU Internet DataBase Lab.

1.1 Data Types

- Informal Definition
  - representation + operations
    - representation = attributes or the structure of the type
    - operations = constructor operations + base operations

    - type construct operators: generic extraction operations
      - e.g.) field selection, arrays, lists, sets and sequences

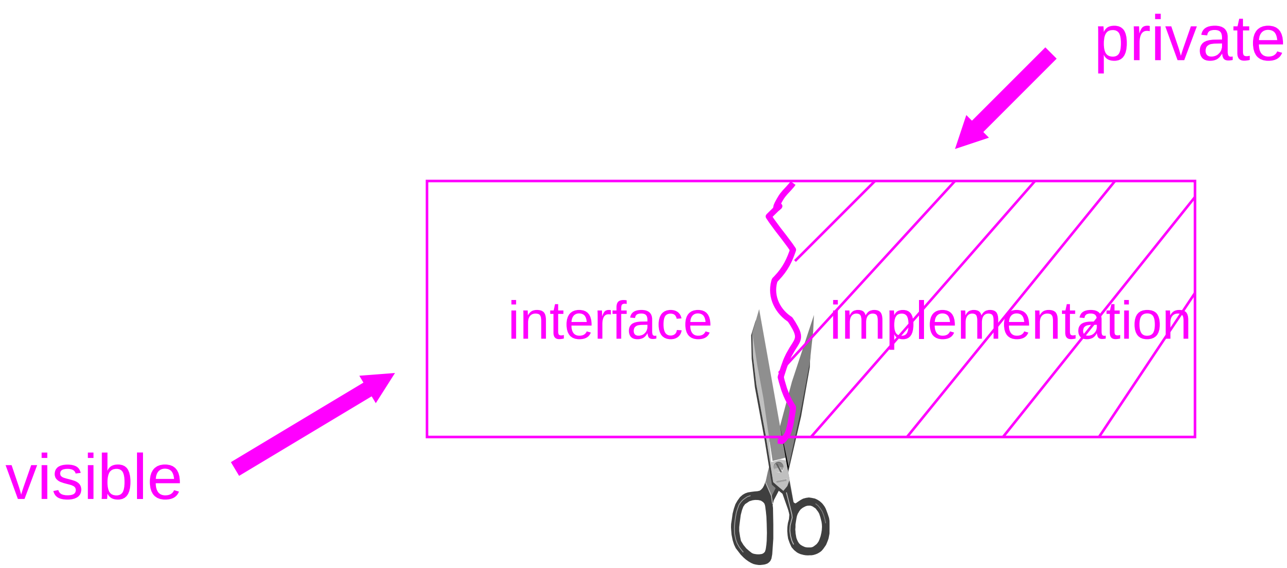
      - e.g.) ITM.Cost := ITM.Cost * 1.1;

constructor operation base operation

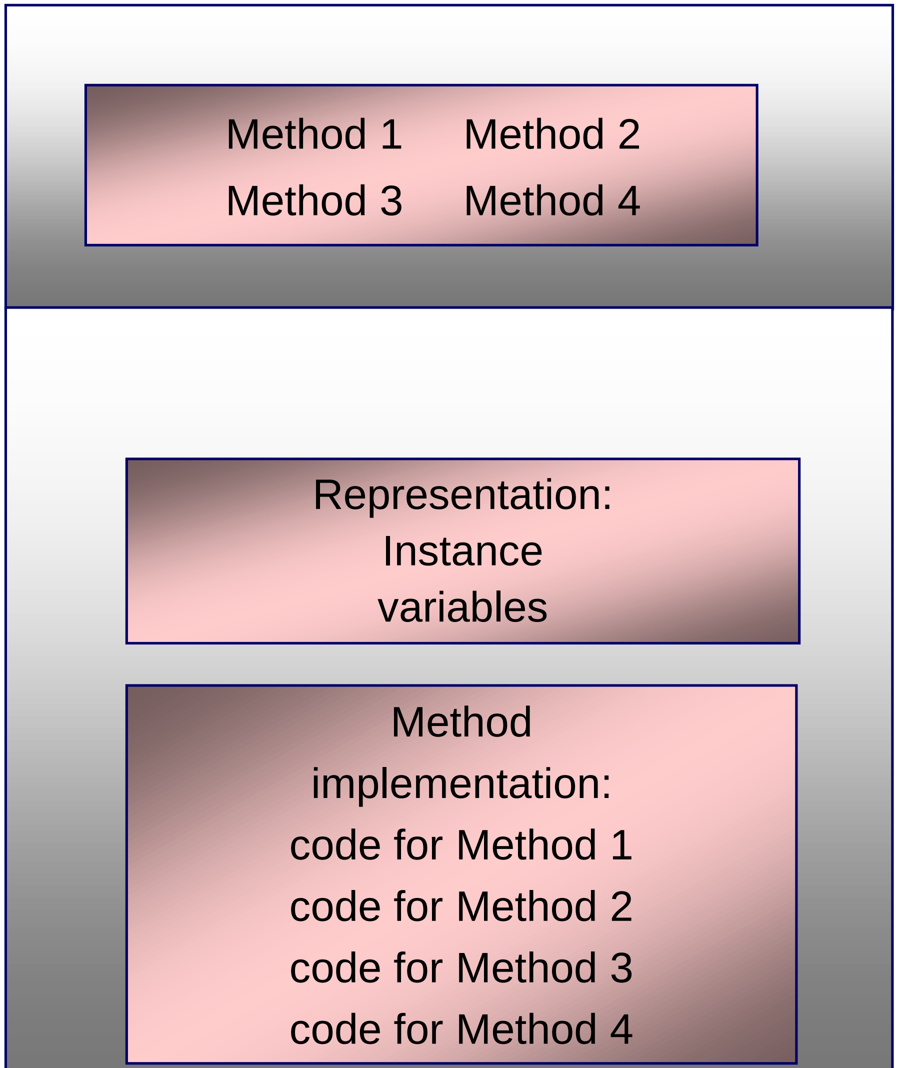
SNU Internet DataBase Lab.

1.2 From Data Types to ADTs

- Clear distinction between implementation and interface
  - interfaces are *public*
  - representations and implementations of interfaces are *private*
  - "encapsulation"



The overall structure of an abstract data type (Class)



Interface
Public

Implementation
Private

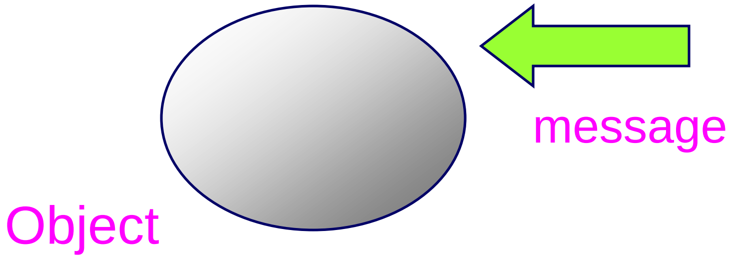
From Data Types to ADTs

- Suppose we have ADT: a set of integers
  - A linked list of records vs. An array of integers
  - The interfaces of two implementations are same to users' view
  - The implementation of associated operations like "Find an element" would be different

SNU Internet DataBase Lab.

1.3 The Object / Message Paradigm (1)

- The data is active computational entity
  - Messages comprise the object's *interfaces*
  - Messages can change/retrieve the object's internal states



SNU Internet DataBase Lab.

1.3 The Object / Message Paradigm (2)

- Procedural Model
  - function call to return data values
  - function call to update input data parameter
- Object / Message model
  - send messages to perform computation and return a value
  - send messages to ask an object to change the object's content
- In terms of computational power
  - The procedural model and the object/message model are same
- In terms of modeling, software development, software extensibility, reusability
  - The object/message model is far superior

SNU Internet DataBase Lab.

1.4 Modularization

- Naturally provide a divide-and-conquer strategy
- In procedural model → through procedures
- In OO model → through *abstract data typing*
  - advantages
    - models the real world
    - autonomous
    - generates correct applications
    - reusable

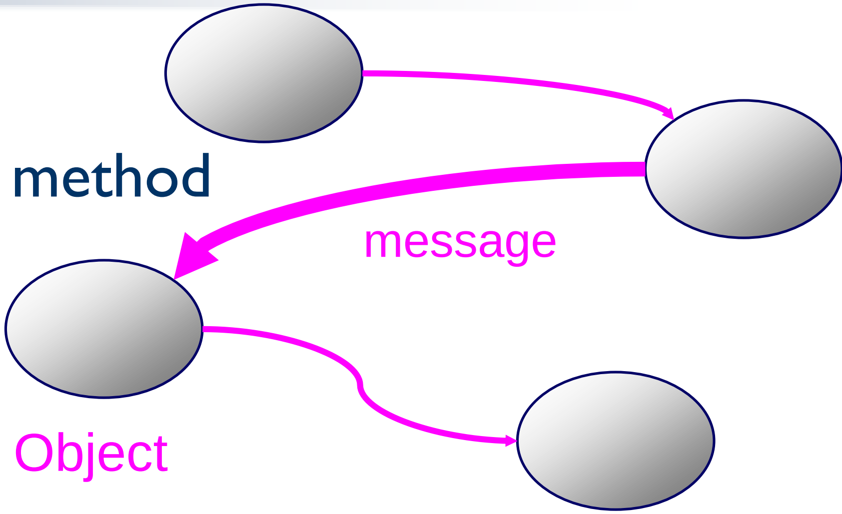
SNU Internet DataBase Lab.

1.4.1 Modeling of the Real World

- Data-based vs. Procedure-based
- OO → Data-based
  - more direct representation of the real world
  - improved clarity, robustness, debugging and maintenance of programs
  - encapsulation

1.4.2 Autonomy

- Object is an autonomous agent
  - messages are the only interacting method



- Nodes: Mail service, Secretary, Salesperson, Salesmanager
- Procedural Abstraction: spanning many nodes and many things
- ADT Abstraction: autonomous
  - define messages
  - code correpondant method locally

SNU Internet DataBase Lab.

1.4.3 Generation of Correct Applications

- OO Modularization Can develop more robust code bases
  - Encapsulated ADT can easily be *stubbed*
    - actual code that implements an operation is commented out and replaced by a diagnostic message.
  - Helps isolate errors within ADT
    - Exception-checking and error-handling mechanism can be incorporated easily

1.4.4 Reusability

- Conventional library mechanism has several drawbacks
  - no parametric polymorphism and overloading
    - replications of function and excessive use of case statement
- ADT and OO provide overloading and polymorphism
  - Naturally guarantee reusability

1.5 Benefits of ADT (Summary)

- Provides better conceptualization and modeling
- Enhances robustness
- Enhances performance
- Better captures the semantics of the type
- Separates between implementation and specification
- Allows extensibility of the system

Table of Contents

- Object: Background
- Introduction
- **Classes**
- Overloading
- Dynamic binding
- Parametric polymorphism
- Constraints
- Summary

2.2 Classes

- Language construct to define ADT in OO PLs
- Class definition includes
  - name of the class
  - external operations (interfaces)
  - internal representation
  - internal implementation of the interface

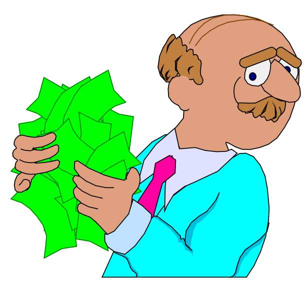
2.1 Instance Variables

- Internal representation of a class
  - hold the state of objects that are instances of a class
  - take on different values for each instance
  - may specify type constraints

- Strongly typed OO languages: Eiffel
  - successful compilation guarantees no return errors
- Weakly types OO languages: JavaScript, Python
- Typeless OO languages: Smalltalk
  - flxibility, rapid prototyping

Instance variables

Instance John



Name: “John Chan”
Age: 32
Salary: $35,000

SNU Internet DataBase Lab.

2.2 Methods and Messages

- Methods define the behavior of instances of a class
- Sending messages (or invoking methods) involves
  - target object
  - *selector* (name of the method)
  - arguments of the operator
- Example
  - *C initializeReal : 3 Imaginary : 2*
    - C : target object
    - initializeReal, Imaginary : selector
    - 3, 2 : arguments
- Protocol
  - set of messages that the object can respond

SNU Internet DataBase Lab.

2.2.1 The Method Body

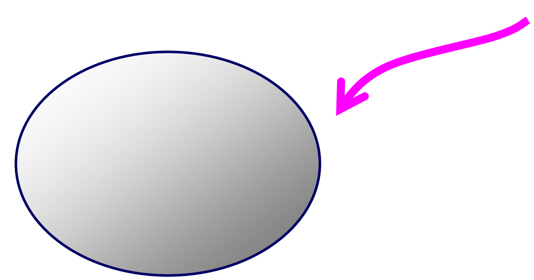
- Code that implements the method
- Possibly several method bodies: Dynamic Binding
  - actual code of the method will be determined at run time

- Smalltalk: A method (within the method body) can access the instance variables of the class and its superclasses

SNU Internet DataBase Lab.

2.2.2 Implicit Parameters and The Pseudovariable "Self"

- How is the target object identified in the method body
  - Implicitly by the pseudovariable "self" or "this"
- Smalltalk "self" and C++ "this"
- Inside the method body: "self"
  - "self" is an implicit formal parameter
- "Super" indicates its superclass



2.2.2 The implicit pseudovariable

- Purpose of pseudovariable
  - As a return variable of a method
  - To invoke other methods on the target object
  - To distinguish between the object as an instance of the class and the object as an instance of a superclass

SNU Internet DataBase Lab.

2.2.3 Accessor and Update Methods

- Generic operations
- Accessor methods: Retrieve operations of instance variables (rather than perform complex computations)
- Update methods: Update operations of instance variables (rather than perform complex computations)

- Automatically generated in some OO languages because they are so common
  - get_real, get_imaginary
  - put_real, put_imaginary

2.3.1 Creating Objects (1)

- In conventional languages (Pascal, C)
  1. Explicit declaration and definition
  2. Allocate space from the memory heap and declare pointer to it

```
Type
ComplexPtr = ^Complex

Complex = RECORD
    Real: real;
    Imaginary: real;
END;
```

```
VAR
  C1, C2: Complex;

C1.Real := 1.0;
C1.Imaginary := 2.0;
```

```
VAR
  pC1, pC2; ComplexPtr;

new(pC1);
pC1^.Real := 1.0;
pC1^.Imaginary := 2.0;
```

2.3.1 Creating Objects (2)

- In OO languages: Ask the class to stamp out an instance
  - Suppose we have "Complex" class
  - Use special object-creation class method to initialize instance variables
    - new Complex Real: Imaginary:
    - new Complex Real: 3 Imaginary: 2
  - Use generic new method with uninitialized instance variables
    - C := new Complex
    - C initializeReal: 3   initializeImaginary: 2

SNU Internet DataBase Lab.

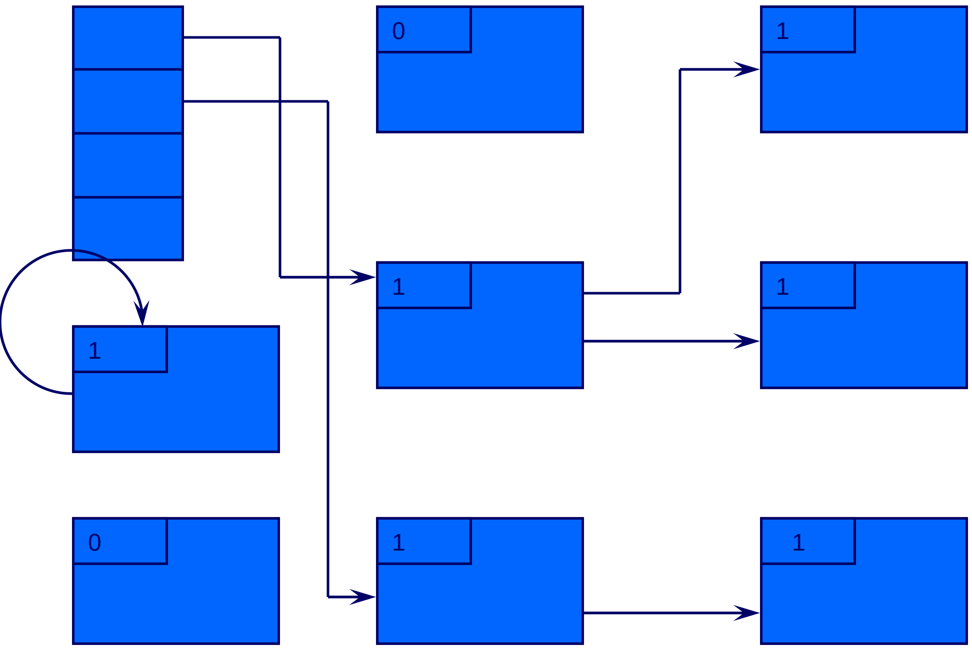
2.3.2 Destroying Objects and Garbage Collection

- Two strategies
  - Explicit "dispose" or "delete" operation (Pascal,C)
    - little overhead
    - problem of "dangling" references
  - Implicit disposal or reclamation of the object when it is no longer reachable (Smalltalk, Lisp)
    - garbage collection
    - no dangling reference
    - run-time overhead

SNU Internet DataBase Lab.

2.3.2.1 Garbage Collection (1)

- Reference counting

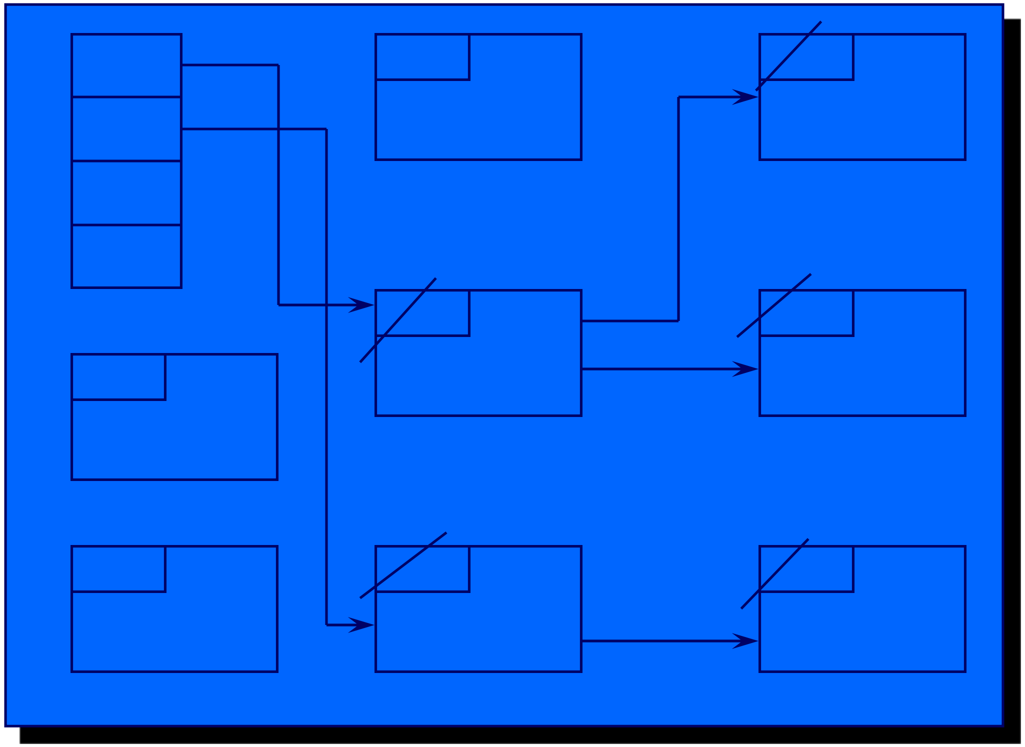


- no stop-and-reclaim phase
- dangling reference finding
- space utilization is enhanced
- "circular reference" problem
- overhead of incrementing and decrementing the reference

SNU Internet DataBase Lab.

2.3.2.1 Garbage Collection (2)

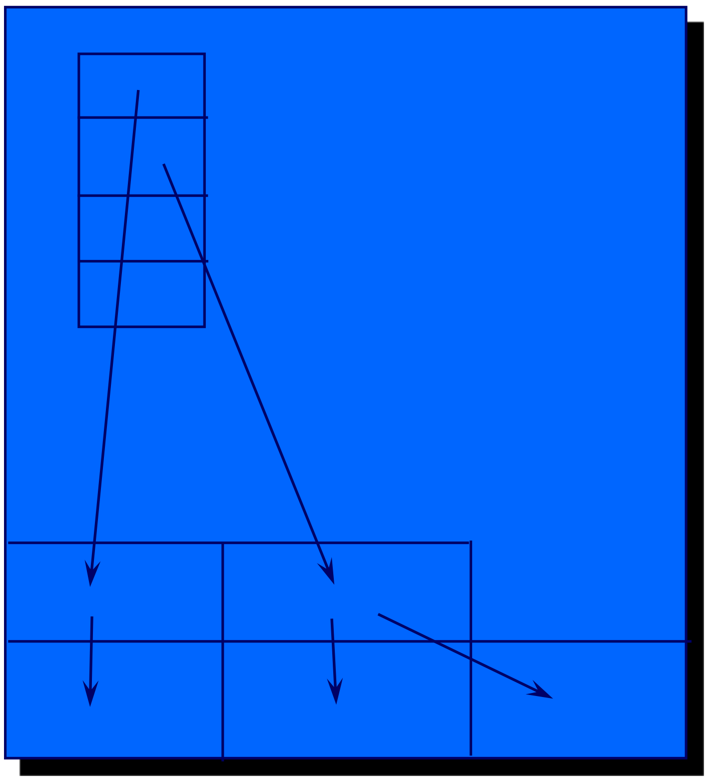
- Mark-and-sweep



- mark phase

2.3.2.1 Garbage Collection (3)

- Mark-and-sweep (cont'd)



-sweep phase

The mark-and-sweep scheme
-- use global information to reclaim storage stop-and-sweep overhead
-- no "circular reference" problem
-- memory compaction is possible
-- system-stop overhead

SNU Internet DataBase Lab.

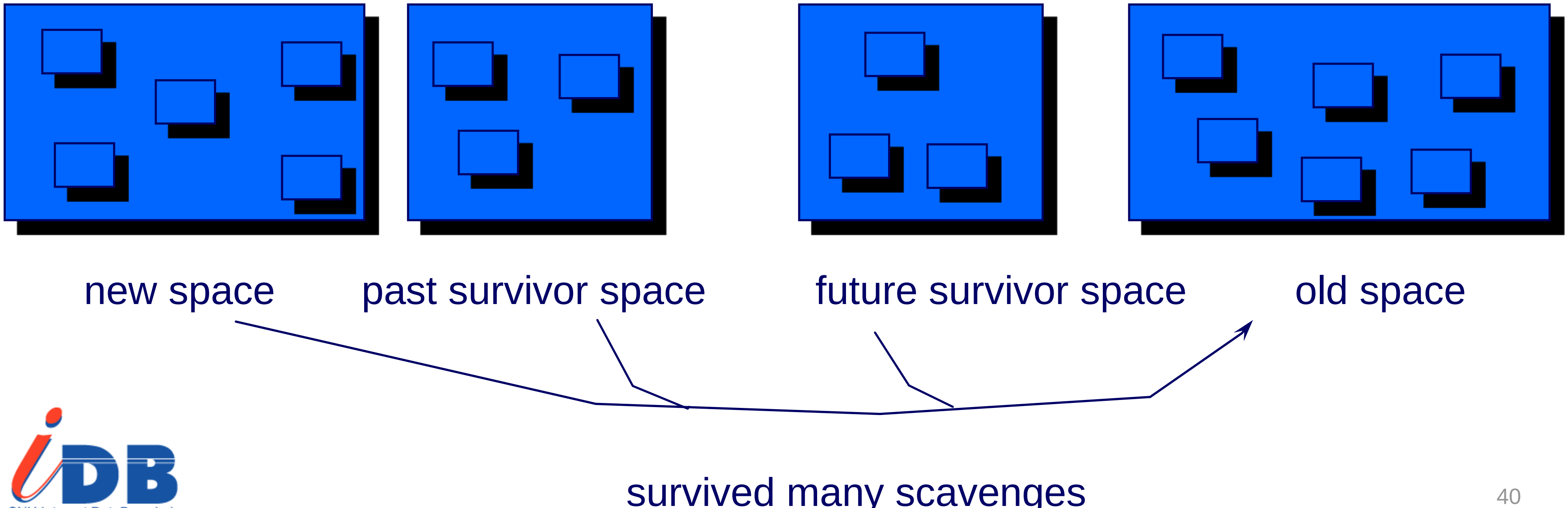
2.3.2.1 Garbage Collection (4)

- Reference counting: circular reference problem and counting overhead
- Mark and sweep: stop-and-sweep overhead when memory runs out
- Copy and Swap algorithm
  - Half the space H1 and H2
  - Allocate objects from H1
  - If H1 is filled up, copy the live objects of H1 into H2
  - H1 and H2 are reverse

SNU Internet DataBase Lab.

2.3.2.1 Garbage Collection (5)

- Copy and swap: space overhead
- Generation Scavenging (scavenge: 청소하다, 배기하다)
- Assumption: 새로이 생성된 객체일수록 삭제되기가 쉽다



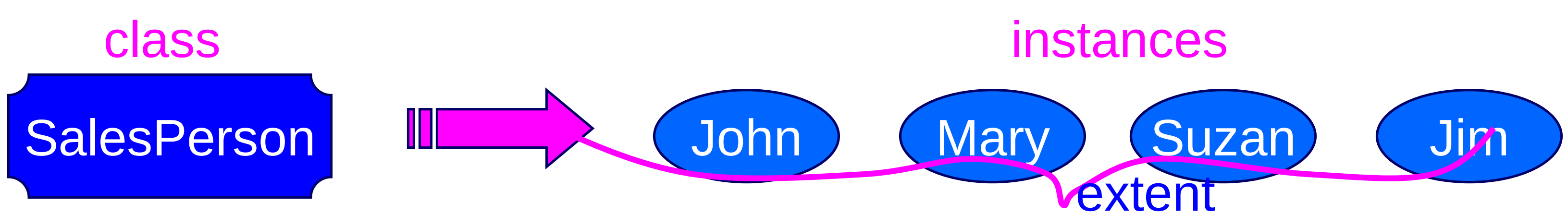
2.3.2.1 Garbage Collection (6)

- Generation Scavenging
  - new space: creating new objects
  - past survivor space: survivors from the previous round
  - future survivor space: space for scavenging
  - old space: “die-hard” area for old objects

- past s-space and future s-space are reversed after scavenging
- reduces the overhead of the stop-and-copy
- there can be copying and age-tracking and space overhead

2.4 Class Extensions (1)

- Meaning: actual, existing instances of a class
- How can we facilitate processing large numbers of objects of the same type?
- Conventional PL (Pascal): A type represents the set of all possible objects
- DBMS: A table type represents the set of all records
- In OOPLs, we already used the name of class when we want to create an instance of a class, then How?



In conventional PLs : Pascal

```
TYPE
   Complex =
         RECORD
            Real : real;
            Imaginary : real;
         END;
```

denotes

{ (Real, Imaginary) |
Real is real;
Imaginary is real }

Set of all complex numbers

Complex type ← like template

OO Concepts

Class

extent

Set of objects

Extent : It is useful in DBMS

Process <u>large numbers of objects of the same type</u>

```
CREATE TABLE SalesPerson
   (Name          CHAR(20),
    Address       CHAR(20),
    Tel_No        INTEGER,
    Salary        FLOAT,
    …)
```

extent

instances

John Mary Suzan Jim

extent

```
Query : SELECT Name, Address
        FROM SalesPerson
        WHERE Salary > 50000
```

<u>** SQL has the language construct for EXTENT: relation name</u>

2.4 Class Extensions (2)

- Two strategies to access existing instances
  - Through the class extensions (if the language supports extension).
    - Keyword "extent" or "extension"
  - Through a collection object (almost all OOPLs support this)
    - keyword: "collection", "set", "bag", "array"

2.4.1 Collections

- Most OO PLs support "collection" class
- Similar effect to extensions
- Subclasses of collection class
  - Sets, Arrays, or Bags (Sets with duplicates)
- Smalltalk
  - SalesPeople := Set new
  - SalesPeople add: John
- Eiffel
  - SalesPeople: SET[SalesPerson]

Table of Contents

- Object: Background
- Introduction
- Classes
- <u>**Overloading**</u>
- Dynamic binding
- Parametric polymorphism
- Constraints
- Summary

SNU Internet DataBase Lab.

3 Overloading

- Operations of the *same name* but *different semantics* can be invoked for objects of different type
- Conventional PLs support basic overloading on built-in basic types such as int, real and character
- OOPLs take one step further for any operation of any object type
  - If S1 and S2 are sets, S1 + S2 would be set union

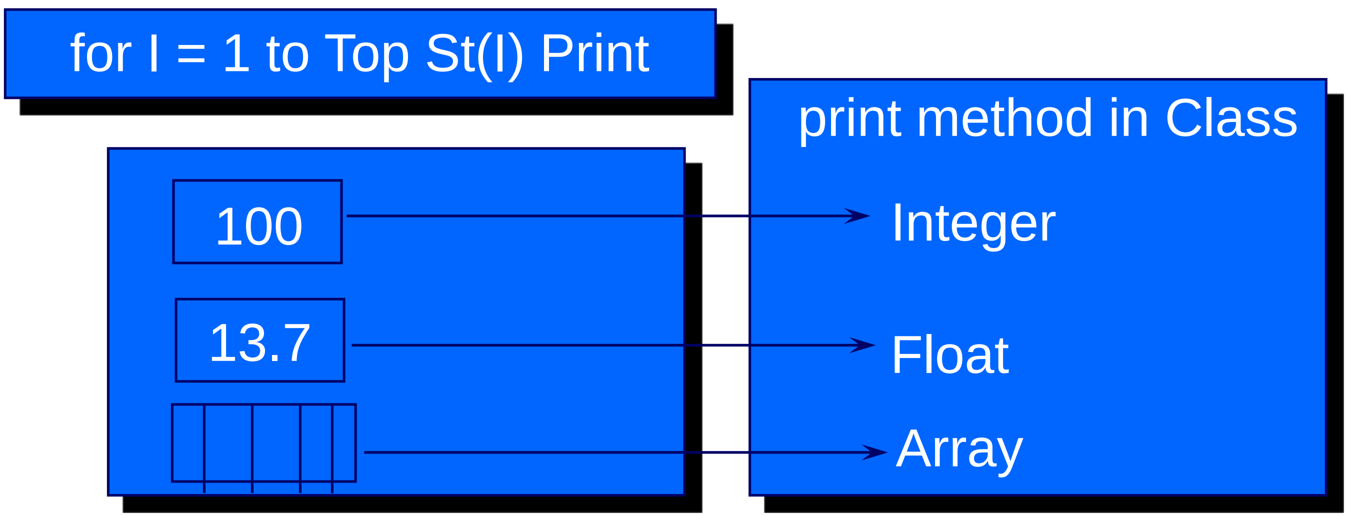
SNU Internet DataBase Lab.

Table of Contents

SNU Internet DataBase Lab.

4 Dynamic Binding (1)

- Supports operator overloading
- In typeless language, variable's type will be known *at run time*
- Provide extensibility, compact code, clarity!



Dynamically binding "*Print*"

4 Dynamic Binding (2)

- Manipulation collections of objects of different types implies "not strongly typed"
- The power of overloading and dynamic binding is best utilized by typeless language
- Advantages
  - Extensibility
  - Development of more compact code
  - Clarity
- Disadvantages: slow

Table of Contents

- Object: Background
- Introduction
- Classes
- Overloading
- Dynamic binding
- **Parametric polymorphism**
- Constraints
- Summary

SNU Internet DataBase Lab.

5 Parametric Polymorphism

- *"Anything goes"* makes it a form of polymorphism
- *Uses types as parameters* in generic type declarations or classes
  - e.g.) template in C++, parameterized type in CLU
- Genericity
- flexibility & advantage of code sharing with power of strong typing

Parameterized polymorphic types in CLU

List-sum = proc[t: type] (a: list[t])
    returns(t)
    requires the type t has a binary operator
        + : proctype(t, t) returns(t) that "adds" two objects of type t.
    Effects it returns the sum of the elements
        in the list using the binary '+'

List-sum[int](IL)

List-sum[real](RL)

Integer list

List of real numbers

SNU Internet DataBase Lab.

Table of Contents

SNU Internet DataBase Lab.

6 Constraints

- ADT needs to be correct and complete
- Test the correctness or completeness
  - constraints on objects and instance variables
  - pre- and postconditions of methods
- Help the programmer express the semantics of ADT as directly as possible

SNU Internet DataBase Lab.