Programming in Java

OOP, Object, Class

蔡文能
交通大學資訊工程學系
tsaiwn@csie.nctu.edu.tw

Outline

- OOP programming Paradigm
- Object, Class
- Encapsulation and Information Hiding
- Inheritance (extension) vs. Contain
- Object-Oriented Technology: CRC, UML, . . .
- The Life Cycle of an Object
- Abstract Classes
- Interface
- Access Modifiers
- Nested Class
- Upcasting and Polymorphism

The Object-Oriented (OO) Programming Paradigm

- Object-Oriented Programming (OOP) is one of the programming paradigms (典範) in computer science
- OOP is the dominant moden programming paradigm
- Object-Oriented Programming is well-known in the business world by the name of *'Object Technology'*

Programming Paradigms

- Imperative Programming (FORTRAN, C, Pascal, ...)
  - The most common programming paradigm
- **Functional Programming (LISP, ...)**
- **Logic Programming (Prolog)**

  (Declarative programming language; 宣告式語言)
- **Object-Oriented Programming**

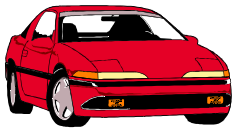
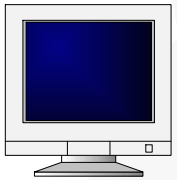
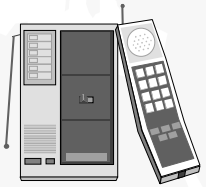
  (Smalltalk, C++, **Java**, ...)
- **Simply using C++/Java constructs does not automatically lead to well-organized Object-Oriented Programs.**

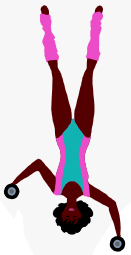
Why OO Programming?

- Better concepts and tools to model and represent the real world as closely as possible (including concurrency, e.g., in Windows GUI)
  => model of reality
  => behavior modeling
- Better **reusability** & extensibility (inheritance)
  => reduce the time/cost of development
- Enhanced maintainability & improved reliability – "Encapsulation" and "Information Hiding"
  - Object only accessible through the external interface
  - Internal implementation details are not visible outside
  - Localized changes to implementation of classes
  - Completeness: all required operations are defined
  - Independent testing and maintenance
- Help writing good program more easily

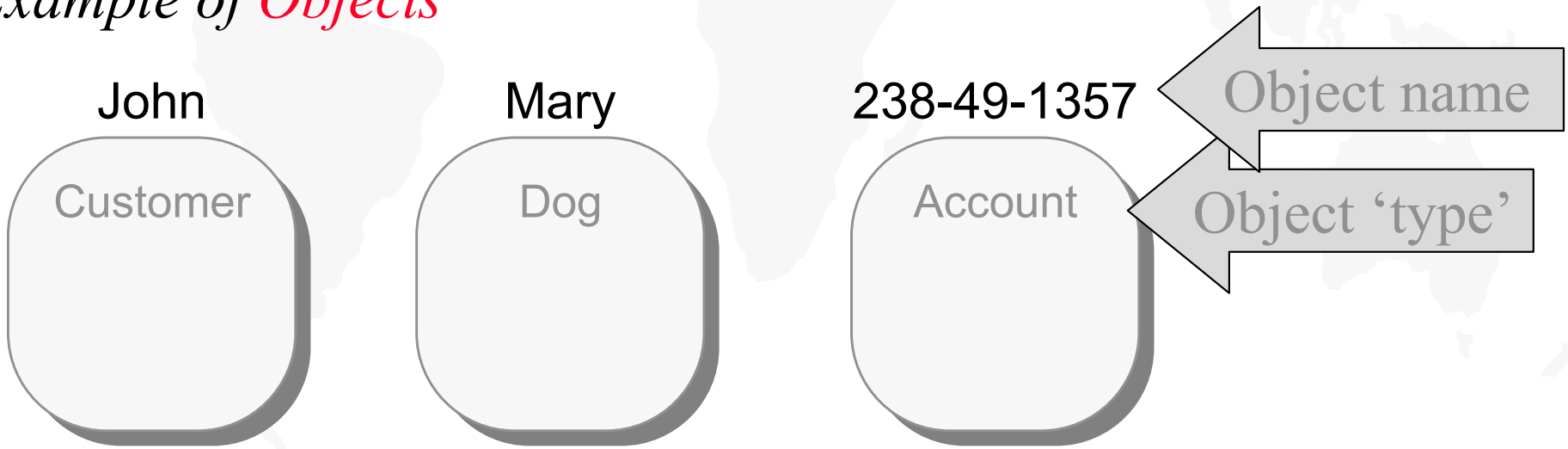
Objects

- An *object* is a thing.



Example of Objects

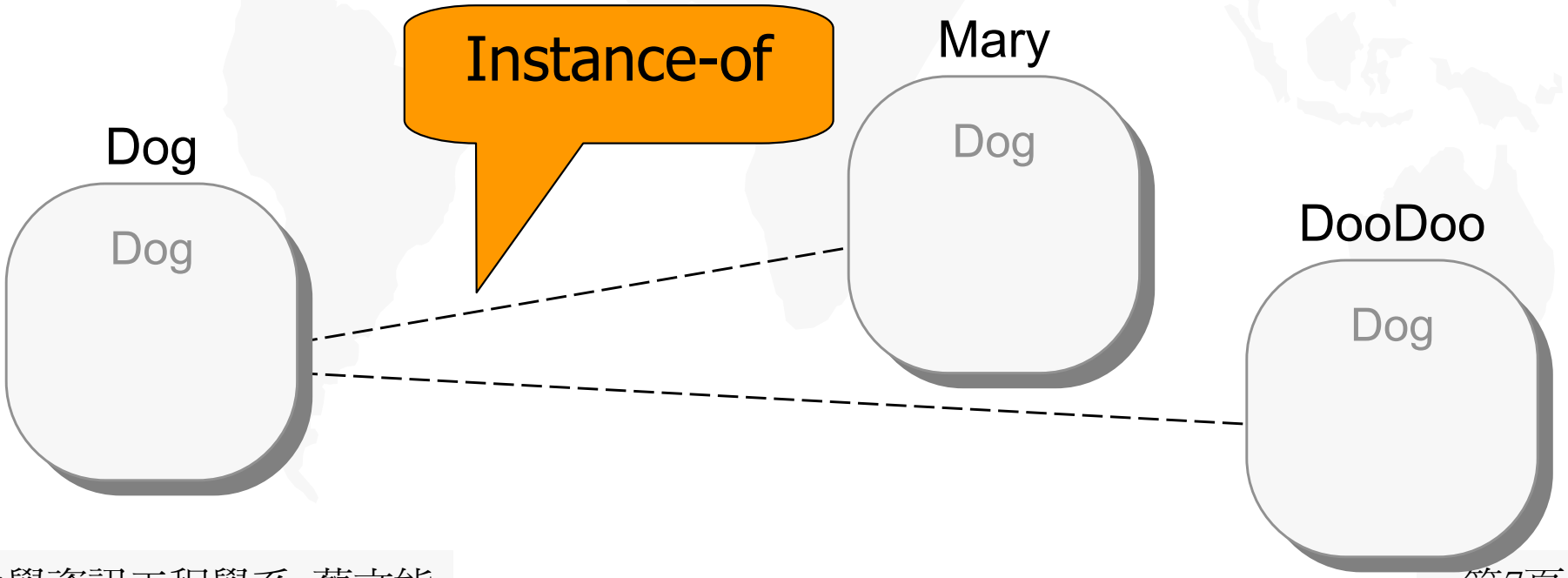


Classes

- A *class* (*e.g.*, Dog) is a kind of mold or template to create objects *(e.g.*, Mary and DooDoo)

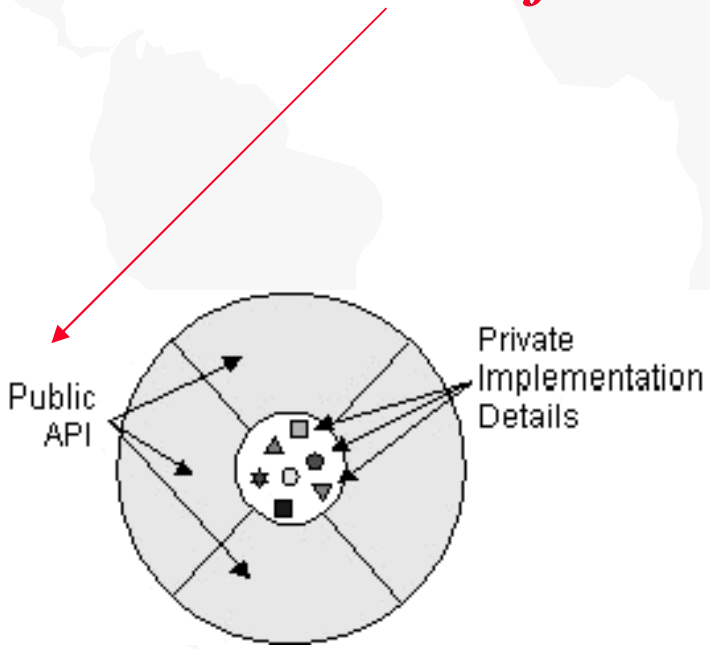
mold = 模型 = mould

- An object is an *instance* of a class. The object *belongs to* that class



What Is an Object?

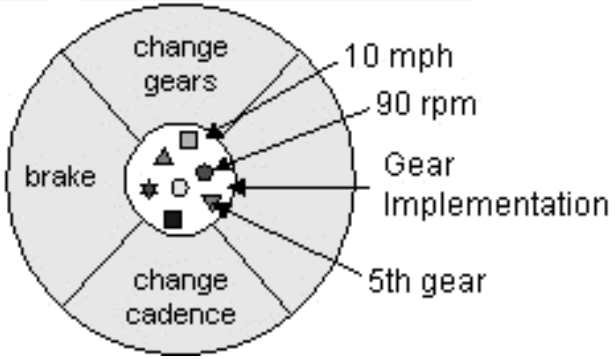
- These real-world objects all have *states* and *behaviors*.
- Definition: An object is a software bundle of ***variables*** and related ***methods*** (***function***).



A common visual representation of a software object.

Example: Bicycle

- The software bicycle's current state:
  - its speed is *10 mph*,
  - its pedal cadence is *90 rpm*, and
  - its current gear is the *5th gear*.



- The software bicycle would also have methods to
  - *brake,*
  - *change the pedal cadence, and*
  - *change gears.*

What Are Classes?

- Definition:
  - A class is a blueprint or prototype
    - Defines the variables and methods common to all objects of a certain kind.
- The Benefits of Classes
  - Reusability -- Software programmers use the same class, and the same code to create many objects.

Objects vs. Classes

- Objects:
  - E.g., a real car.
- Classes: or types
  - E.g., BMW-500

A Class is a blueprint or template of some objects

Object is an **Instance** of some class.

Object (物件, 個體) -- 就是以前的變數 (Variable)

再談資料型別

- char, short, int, long, **long long**
- float, double, long double

```
double  ya = 1234567.2, yb=1234567.8;
float xa =ya, xb =yb;
#include <stdio.h>
main( ) {
    printf(" y: %13.4f, 13.4f\n", ya, yb);
    printf(" x: %13.4f, 13.4f\n", xa, xb);
}
```

在Turbo C++中,
打入 char 或 short 或…
然後敲入 Control_F1

結構與型別 (如何自定資料結構?)
solution: struct

- 只能用前述基本 **data type** 嗎?
- **User defined data type?**
- 考慮寫程式**處理全班成績資料**, 包括加總平均並排序(Sort)然後印出一份照名次排序的以及一份照學號排序的全班資料
  - 如何做 Sort (排序)?
  - Sort 時**兩學生資料要對調, 要如何對調?**
  - 有很多 array ? 存學號的 array, 存姓名的 array? 存成績的 array? **...**

Bubble sort, Insertion sort, Selection sort

So, Why struct?

- Struct 可以把相關資料 group 在一起

```
struct student x[99], tmp;
/* … */
tmp = x[i];  x[i] = x[k];  x[k] = tmp;
```

- 增加程式可讀性
- 程式更容易維護
- **其實即使都不用 struct 也能寫出所有程式!**
- **How to define / use struct ?**
  - **Next slides**

Struct – 自訂資料型別(1/5)

考慮寫個程式處理學生的成績資料, 如何表示一個學生的資料? 想一想. . .

```
#include <stdio.h>
struct Student {
    long  sid;
    char name[9];   /*可存四個Big5中文 */
    float score[13];   /*每人最多修13 科☺ */
};      /*注意struct與class之 分號不能省掉*/
 int main( ) {
    struct Student x;     /* C++ 和 C99 不用寫 struct */
    x.sid = 123;   /* dot notation */
    strcpy(x.name, "張大千") ;  /*注意字串不能= */
    /* 用 loop 把成績讀入 x.score[?] */
}
```

習慣上第一個字母大寫

//C 的字串不能直接用 = 複製! 也可用 memcpy( )

struct --- Structure (2/5)

```
struct Date {
    int day, month, year;
};

void main ( ) {
  struct Date birthday = { 14, 2, 1999 };
  struct Date today;
  today.day = 28;
  today.month = 2;
  today.year = 2003;
  printf("This year is %d", today.year);
  …
}
```

Struct 的常數要用{ };夾住

逐項填入也可以

birthday

birthday.day	14
birthday.month	2
birthday.year	1999

today

today.day	28
today.month	2
today.year	2003

Structure (3/5)

- Group related fields into a structure

```
struct date {
    int day, month, year;
}; /* 注意分號 */
```

```
struct date today;
```

C99 和 C++則只要寫

```
date today;
```

```
typedef struct date Date;
```

```
Date today;
```

想不寫 struct, 先在 typedef 寫一次, 然後

Structure (4/5)

- 兩種定義一次寫完

```
typedef struct date {
    int day, month, year;
} Date;     /* Date爲type*/
```

```
struct date today;
  or
Date today;
都一樣意思
```

注意以下例子意思不同!

```
struct date {
    int day, month, year;
} Date;  /*Date爲變數*/
```

Structure (5/5)

```
typedef
struct  student {
    long sid;
    char name[9];
    float height;
    double wet;   /*weight*/
} Student;
```

- **Student** 是 **type**

- What about this?

```
struct  student {
    long sid;
    char name[9];
    float height;
    double wet;
} Student, stmp, x[99];
```

- Student 是 變數

ADT --- Data Abstraction

- An **Abstract Data Type** (**ADT**) is a user-defined data type that satisfies the following two conditions: (**Encapsulation** + **Information Hiding**)
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit; also, other program units can create objects of the type.
  - The **representation of objects of the type is hidden from the program units that use these objects**, so the only operations (methods) possible are those provided in the type's definition which are known as interfaces.

ADT: 抽象資料型態 = 把資料以及對資料做處理的操作(function)一起封藏在一個程式單元

Encapsulation (封藏)

- Hide the object's nucleus from other objects in the program.
  - The implementation details can change at any time without affecting other parts of the program.
- It is an ideal representation of an object, but
  - For implementation or efficiency reasons, an object may wish to *expose some of its variables* or *hide some of its methods*.
- Benefits:
  - **Modularity**
    - The source code for an object can be written and maintained independently of the source code for other objects.
  - **Information hiding**
    - An object has a public interface that other objects can use to communicate with it.

問題與思考 (How?)

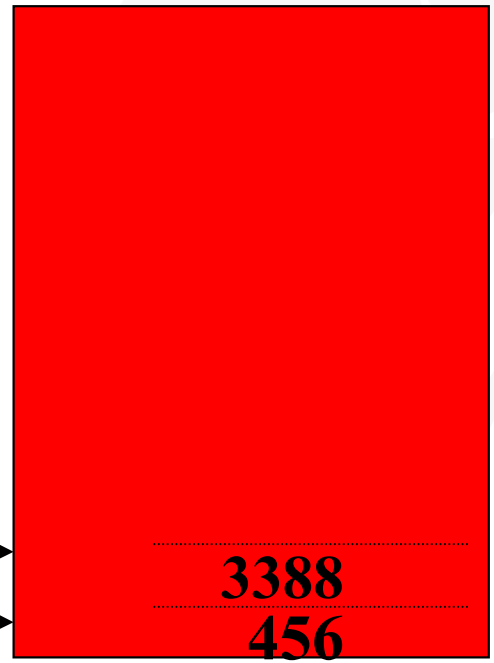
- 如何表示 **Linked List? (struct + pointer)**
- 如何表示**Stack? Queue?**
  - **Stack** 與 **Queue** 的應用
- 如何表示**Tree?**
  - 如何儲存 **tree ? --- Representation of Tree** **Binary Tree**
- **Tree** 的應用?
  - **Tree** 的 **traversal**
  - **Expression tree vs. parsing tree**

用 array 來做 STACK (1/2)

Initialization:

sptr = -1; /*empty*/

int x[99];



sptr

Stack Pointer

int sptr;

```
void push (int y) {
    ++sptr;
    x[sptr] = y;
}
```

```
push (456) ;
```

```
push (3388) ;
```

```
int ans = pop( );
```

```
int pop ( ) {
    int tmp=x[sptr];
    --sptr;
    return tmp;
}
```

需要一個array 和一個整數

用 array 來做 STACK (2/2)

```
static int x[99];   /* 別的檔案看不到這*/
static int sptr = -1;
void push(int y) {
      ++sptr;   x[sptr] = y;
 }
 int pop( ) {    /* C++ 程式庫的pop是 void*/
      return x[sptr--];
 }
/*  其它相關 function例如 isempty( ) */
```

其實這就是 **information hiding**

使用 STACK

```
extern void push(int);   /*宣告*/
extern int pop( );
#include <stdio.h>
int main( ) {
      push(543);
      push(881);
      printf("%d\n",  pop( ) );
 }
 /* 若需要兩個 Stack 呢? */
```

More about STACK (1/3)

/* 若要兩個 Stack 呢? */

解法之一

==> 使用兩個 array, 並把 array 當參數

push(ary2, x); ans = top(ary2);

但是這樣該些 array 必須 開放讓使用者知道

➔ 要讓宣告 extern就看得到, 不可再用 static Global

➔ 違反不必讓使用者知道push到哪去的 Information Hiding 原則 !

(或是該些 array 要定義在使用者程式裡面)

More about STACK (2/3)

/* 若要兩個 Stack 呢? */

解法之二

==> 整個檔案複製到另一個 file, 並把 各相 關函數名稱 都改掉, 例如 push2, pop2, …

==> array 與變數名稱不用改! Why?

(因爲 static Global)

但是這樣 也不是很方便, 函數名稱一大堆

➔ 若要三個 Stack 呢? 四個呢? …

(不過這比前面使用不同 array 的方法好!)

More about STACK (3/3)

```
/* 若要兩個 Stack 呢? */
/* 若要三個 Stack 呢?  四個? 五個 . . . */
/* 有沒有更方便直接的方法? */
```

solution ==> using C++/Java Class

to define a **Stack** as a **Software Component**
(**軟體元件或零件**)

```
Stack  x;
Stack  y;
x.push(13579);
y.push(258); /* 怎樣做才能這樣用? */
y.push( x.pop( ) );
```

把堆疊當物件(object)

一個堆疊就是一個**變數**

軟體 IC ?

Using C++ STL : #include <stack> (這樣就可使用 stack 與 queue 以及...)

Using Java util package: import java.util.*; (Queue 是 Interface)

Classes in Java

- A **class** -- is a template that describes the **data** and **behavior** associated with ***instances*** of that class.
- An *object* -- instantiate a class you create an ***object***.
- An *object* is an instance of some class.
- The data associated with a class or object is stored in *variables*.
- The behavior associated with a class or object is implemented with *methods*.
  - Methods are similar to the functions or procedures in C.

Class **Bicycle** in Java

Not allowed in C++ Language

```
public class Bicycle {
    private int speed = 10;
    private int pedalCadence = 90;
    private int currentGear = 5;
    // a constructor!
    public void brake() { /* ... */ }
    public void changePedalCadence(int x) { /* */ }
    public void changeGear(int x) { /* ... */ }
}
```

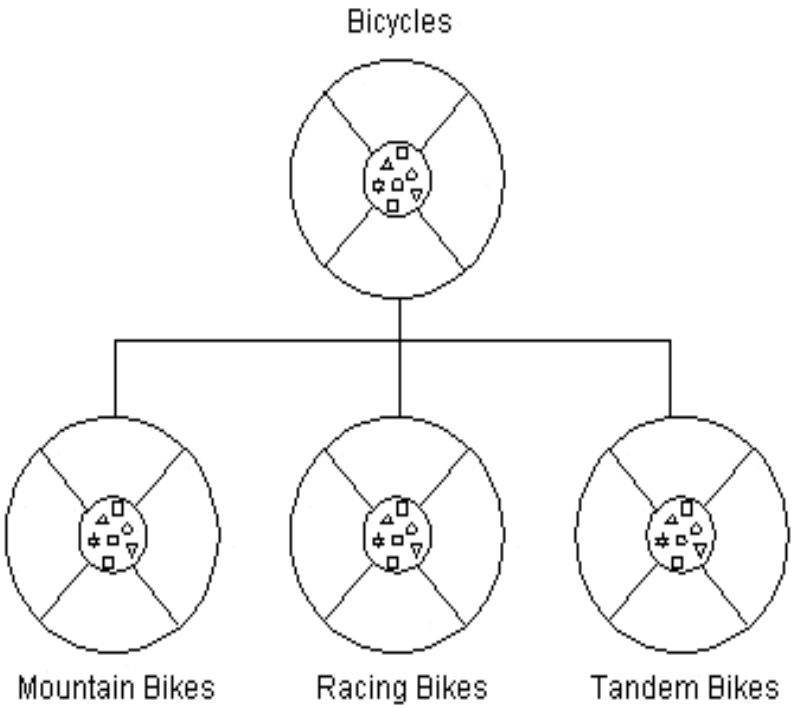
Class HelloWorld (Applet version)

inherit **Applet**

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloWorld extends Applet {
        public void paint(Graphics g) {
            g.drawString("Hello world!", 50, 25);
        }
}
```

What Is Inheritance? (extends)

- Subclasses ***inherit*** **variables** and **methods** from superclasses.
  - States:
    - Gear & Speed
  - Behaviors:
    - Brake & Accelerate
- Subclasses can
  - Add **variables** and **methods**.
  - **Override** inherited **methods**.
- Benefits:
  - Provide specialized behaviors
  - Reuse the code in the superclass
  - *Abstract classes* -- define "generic" behaviors.



Mankind extends Animal

Should be in different files

```
public class Animal {
    private int height = 0;
    private int weight = 0;
    public void talk( ) { System.out.println("Arhh");}
}

public class Mankind extends Animal {
    private int iq = 120;
    public void talk( ) { System.out.println("Hello");}
}
```

One Java file can only have one public class

Mankind is-a Animal

Mankind inherits Animal (C++)

```
class Animal {
    int height = 0;
    int weight = 0;
    public:
        void talk( ) { System.out.println("Won");}
};

class Mankind :public Animal {
    private: int iq = 120;
    public: void talk( ) { System.out.println("Hello");}
} ;
```

Classes can be in same file

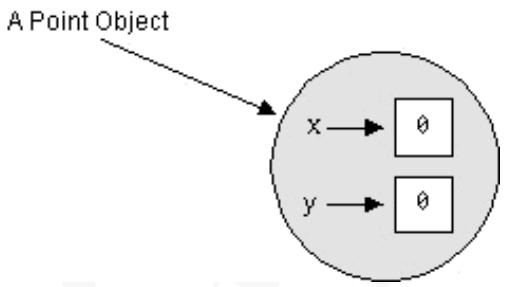
In C++, ";" is required to terminate a class

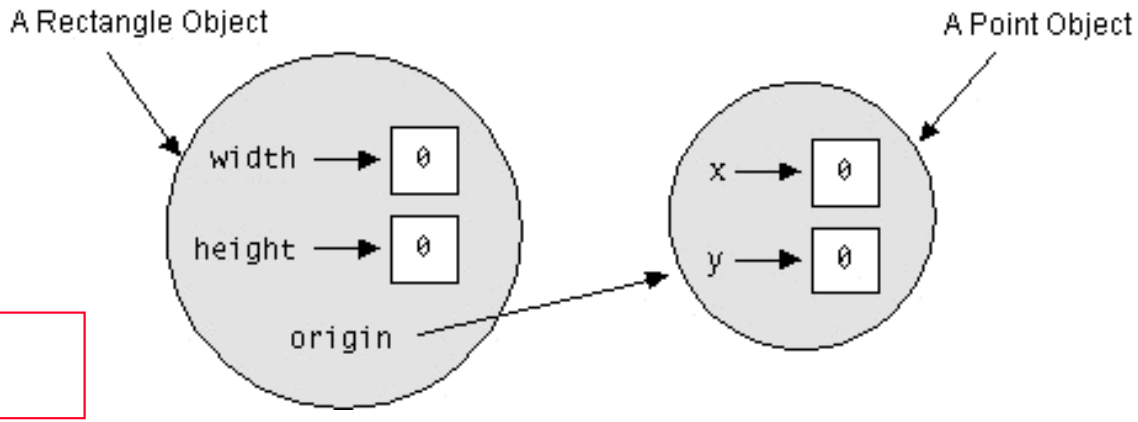
Rectangle Contains Point

Should be in different files because one file can NOT have 2 public classes

```
public class Point {
    private int x = 0;
    private int y = 0;
}
```

```
public class Rectangle {
    private int width = 0;
    private int height = 0;
    private Point origin = new Point();
}
```





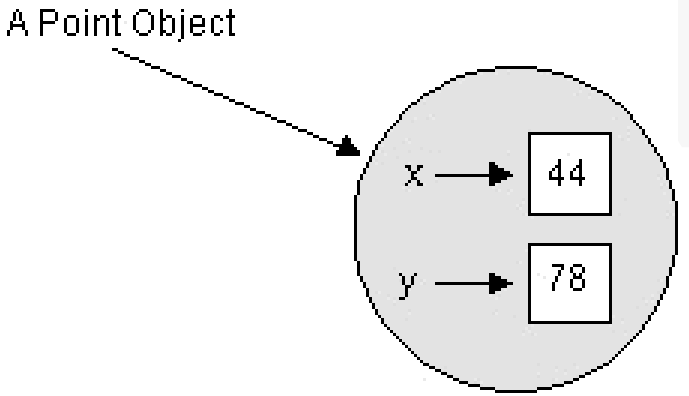
Car **has-a** Wheel

Point and it's constructor

```
public class Point {
    private int x = 0;
    private int y = 0;
    // a constructor!
    public Point(int xxx, int y) {
        x = xxx;
        this.y = y;
    }
}
. . .
Point p = new Point(44,78);
```

this.x

this.y



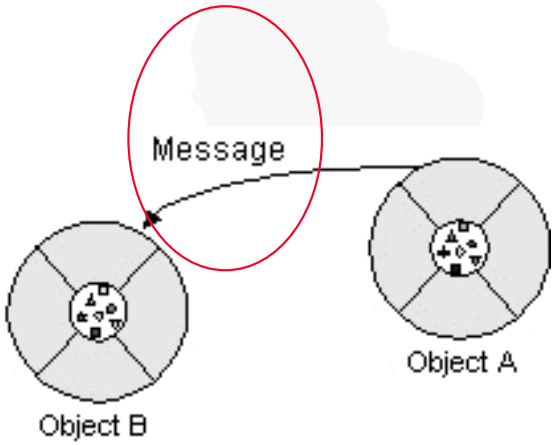
Class may have many Constructors

```
public class Rectangle {
    int width = 0;
    int height = 0;
    Point origin;
    // four constructors
    public Rectangle() { origin = new Point(0, 0); }
    public Rectangle(Point p) { origin = p; }
    public Rectangle(int w, int h) { this(new Point(0, 0), w, h); }
    public Rectangle(Point p, int w, int h)
        { origin = p; width = w; height = h; }

    // a method for moving the rectangle
    public void move(int x, int y) { origin.x = x; origin.y = y; }
    // a method for computing the area of the rectangle
    public int area() { return width * height; }
}
```

What Are Messages?

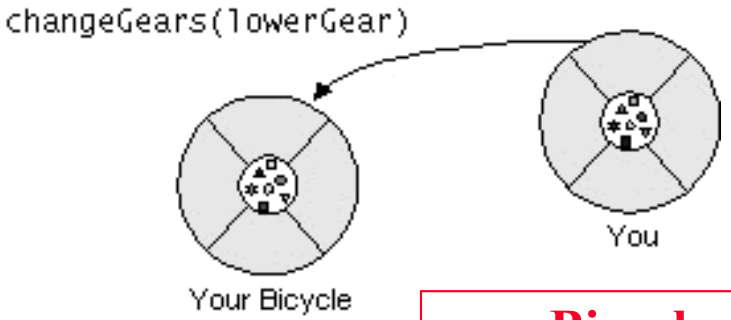
- Message in an object.
- Software objects interact and communicate with each other **by sending messages to each other**.



When object A wants object B to perform one of B's methods, object A sends a message to object B.

Messaging

- Three components comprise a message:
  1. The object to whom the message is addressed (Your Bicycle).
  2. The name of the method to perform (changeGears).
  3. Any parameters needed by the method (lowerGear).



yourBicycle.changeGears(lowerGear)

- Benefits:
  - Message passing supports all interactions between objects.
  - Messages can be sent and received between processes in the **same** machine or in **different** machines.

Object-Oriented Technology

- Object-Oriented Analysis (OOA)
  - Goal: Understand the domain
- Object-Oriented Design (OOD)
  - Goal: Design a solution, a model of the domain in which the desired activities occur
- Object-Oriented Programming (OOP)
  - Goal: Implement the solution
- Note: A Good Design is 2/3 Before You Hit the Keyboard

It's a process, not a **waterfall**

- Design is an iterative activity
  - Start in OOA, go to OOD, forget something in OOA
  - Get to OOP, realize you didn't understand something and go back
- The stages are there to identify **emphases**
  - For example, OOA is TOTALLY Language-Independent

Waterfall model 是早期軟工強調的**軟體發展過程**

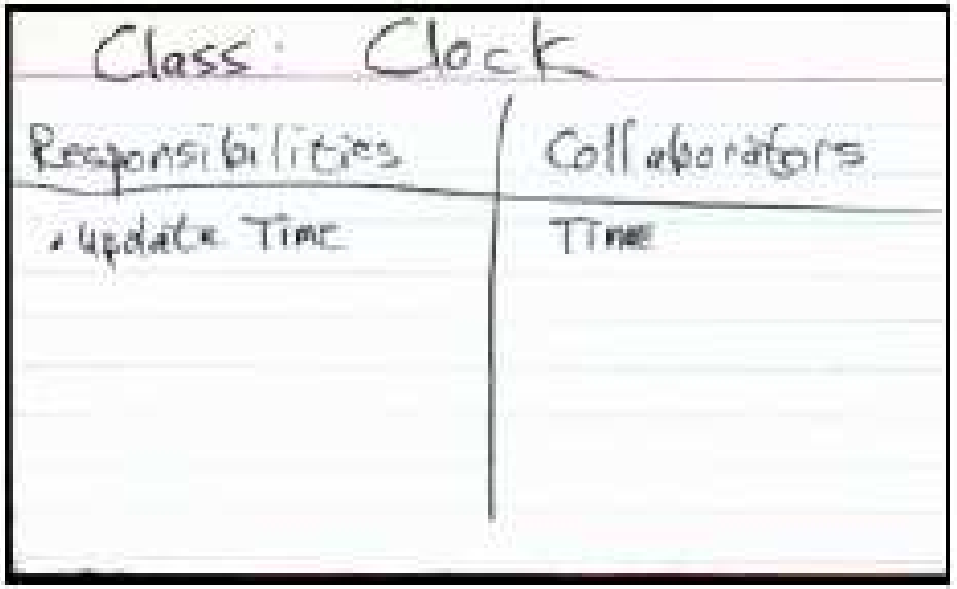
emphases = 重點 = emphasis 的複數

Object-Oriented Analysis

- Step one: Brainstorming Candidate Classes
  - Write down all the objects that related
    - Focus on the **nouns**
    - Good objects have ***attributes*** and ***services***
  - Now, filter the candidates
    - Deal with the interface *later* (Not part of the domain)
    - Are some candidates *attributes* of others?
    - Are some *subclasses* of others? (Inheritance)
    - Are some *instances* of others?

OOA: CRC Cards

- Step two: Write CRC cards and work through scenarios
  - **Class-Responsibility-Collaborator** Cards (Cunningham and Beck)
  - Just 3x5 cards



Although CRC is not part of UML, they add some very useful insights throughout a development.

Data fields (attributes) 寫在背面

How to use CRC Cards

- Make one for each candidate class
- Invent scenarios: What should these objects do?
- Play the cards
  - Lay down the card that starts the scenario
  - Write down its responsibility
  - Add collaborator objects to help with that responsibility
  - Pick up cards as they leave the scenario

Why CRC Cards?

- Help you identify objects and their responsibilities
- Help you understand interactions between objects
- Cards form a useful record of early design activity
- Cards work well in group situations and are understandable by non-technical stakeholders

Object-Oriented Design

- Step one: Create a UML class diagram of your objects
- Step two: Create a detailed description of the services to be performed
  - Peter Coad's "I am a Count. I know how to increment…"
  - Activity, sequence, or collaboration UML diagrams

MDA-Model Driven Architecture?

- A New Way to Specify and Build Systems
  - ***2001年由OMG制定的新開發架構 (http://www.omg.org)***
  - ***以UML Model(塑模)爲基礎***
  - 支援完整開發週期: Analysis, Design, Implementation, Deployment, Maintenance, Evolution & Integration with later systems (改進與後續整合)
  - 內建協同運作性及跨平台性
  - 降低開發初期成本及提高ROI (投資報酬)
  - 可套用至你所使用的任何環境:
    - Programming language • Network
    - Operating System • Middleware

Unified Modeling Language

http://www.omg.org

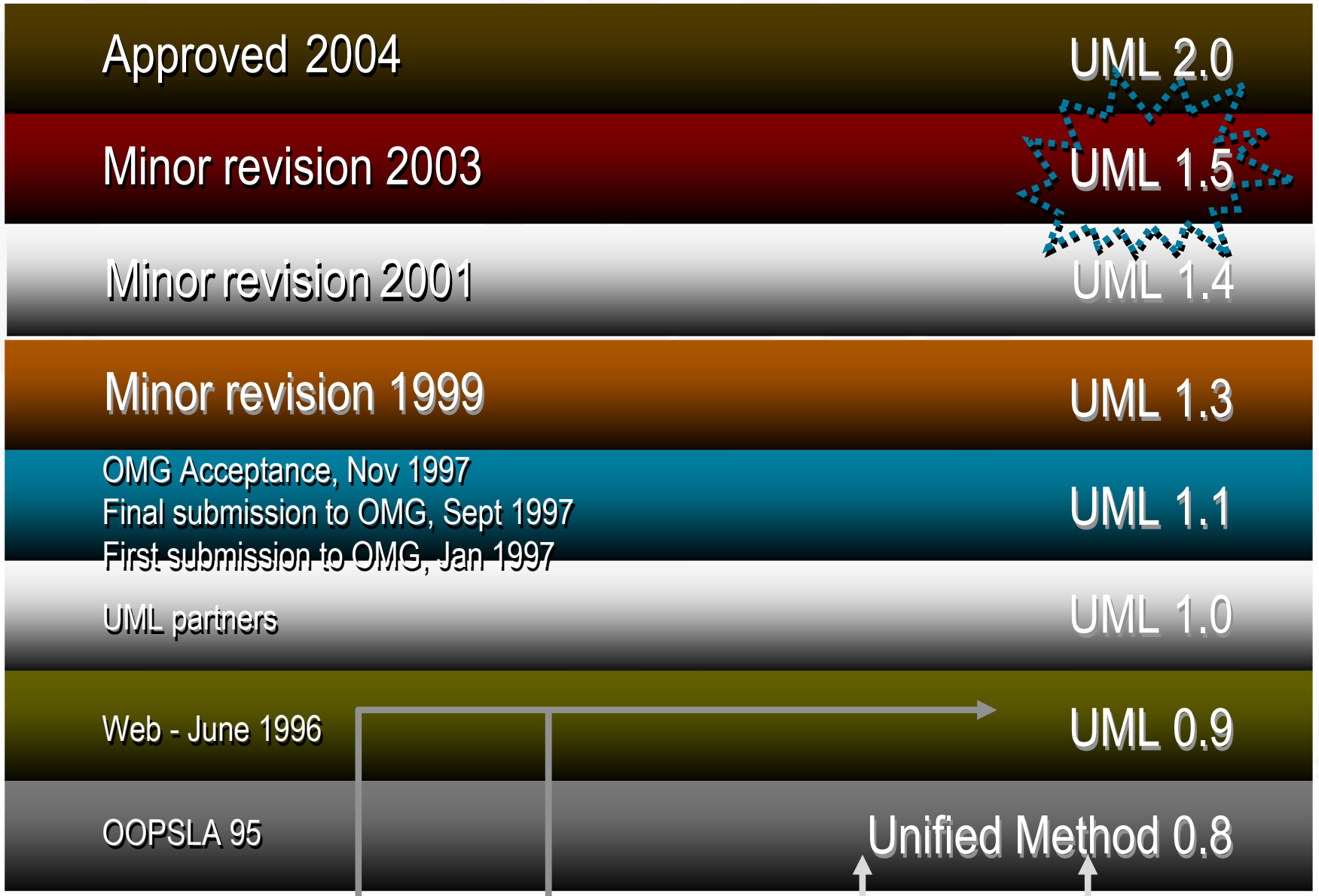
http://www.UML.org

- There have been O-O gurus for many years
- Three of them worked together to define UML ("Three amigos": **Booch**, **Rumbaugh**, **Jacobson**)
- Has now been approved as a standard by the Object Management Group (OMG)
- Very powerful, many forms of notation
  - It's even provable! (with OCL) **(Object Constrain Language)**

amigos = friends

History of the UML(http://www.uml.org/)

Public feedback



UML 常用的 Diagrams

- **Use case** diagrams
  - **Functional behavior of the system as seen by the user.**
- **Class** diagrams
  - **Static structure of the system: Objects, Attributes, and Associations.**
- Activity diagrams
  - **Dynamic behavior of a system, in particular the workflow, i.e. a flowchart.**
- Sequence diagrams
  - **Dynamic behavior between actors and system objects.**
- Statechart diagrams
  - **Dynamic behavior of an individual object as FSM (有限狀態機).**

UML 12 Diagrams

- Behavior :
  - Use Case
  - Sequence
  - Collaboration
  - State Chart
  - Activity
- Structural:
  - Class
  - Component
  - Deployment
  - Object
- Model Management:
  - Packages (class diagram contains packages)
  - Subsystems (class diagram contains subsystems)
  - Models (class diagram contains models)

So, What is UML?

軟體工程師共通的語言

- UML is a Unified Modeling Language
- UML is a set of notations, not a single methodology
- Modeling is a way of thinking about the problems using models organized around the real world ideas.
- Resulted from the convergence of notations from three leading Object-Oriented methods:
  - **Booch** method (by Grady Booch)
  - OMT (by James Rumbaugh)
  - OOSE (by Ivar Jacobson)
- You can model 80% of most problems by using about 20% of the UML

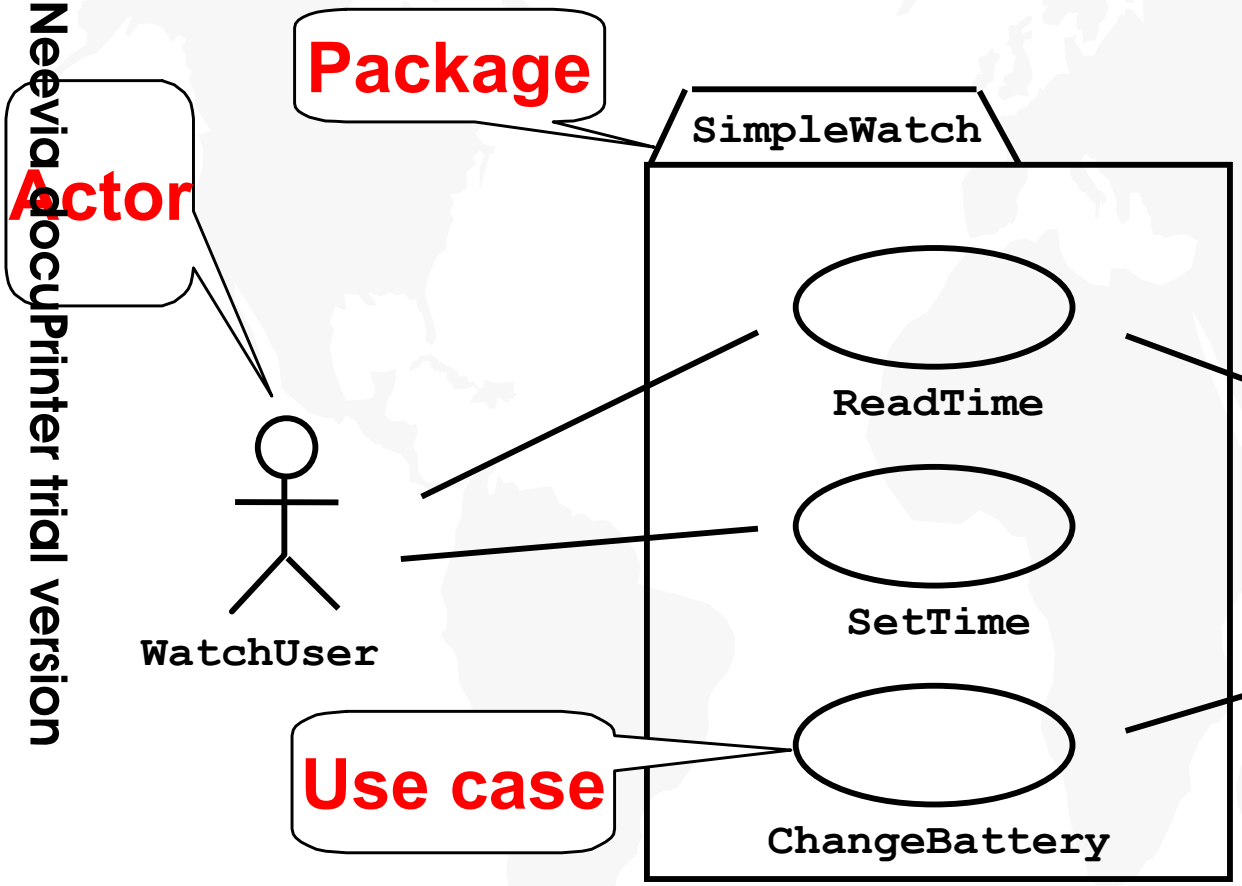
UML is a "blueprint" for building complex software

UML Core Conventions

- Rectangles are classes or instances
- Ovals are functions or **use cases**
- Types are denoted with non-underlined names
  - ✓ `SimpleWatch`
  - ✓ `Firefighter`
- Instances are denoted with an underlined names
  - ✓ `myWatch:SimpleWatch`
  - ✓ `Joe:Firefighter`
- Diagrams are higraphs
  - Nodes are entities (e.g. classes, states)
  - Arcs are relationships among entities (e.g. sender/receiver)
  - Containment represents belonging (e.g. use cases in package)

Higraphs are an extension to the familiar Directed Graph structure where nodes are connected by edges to other nodes. Nodes represent entities in some domain (in our case, classes, packages, methods, etc.).

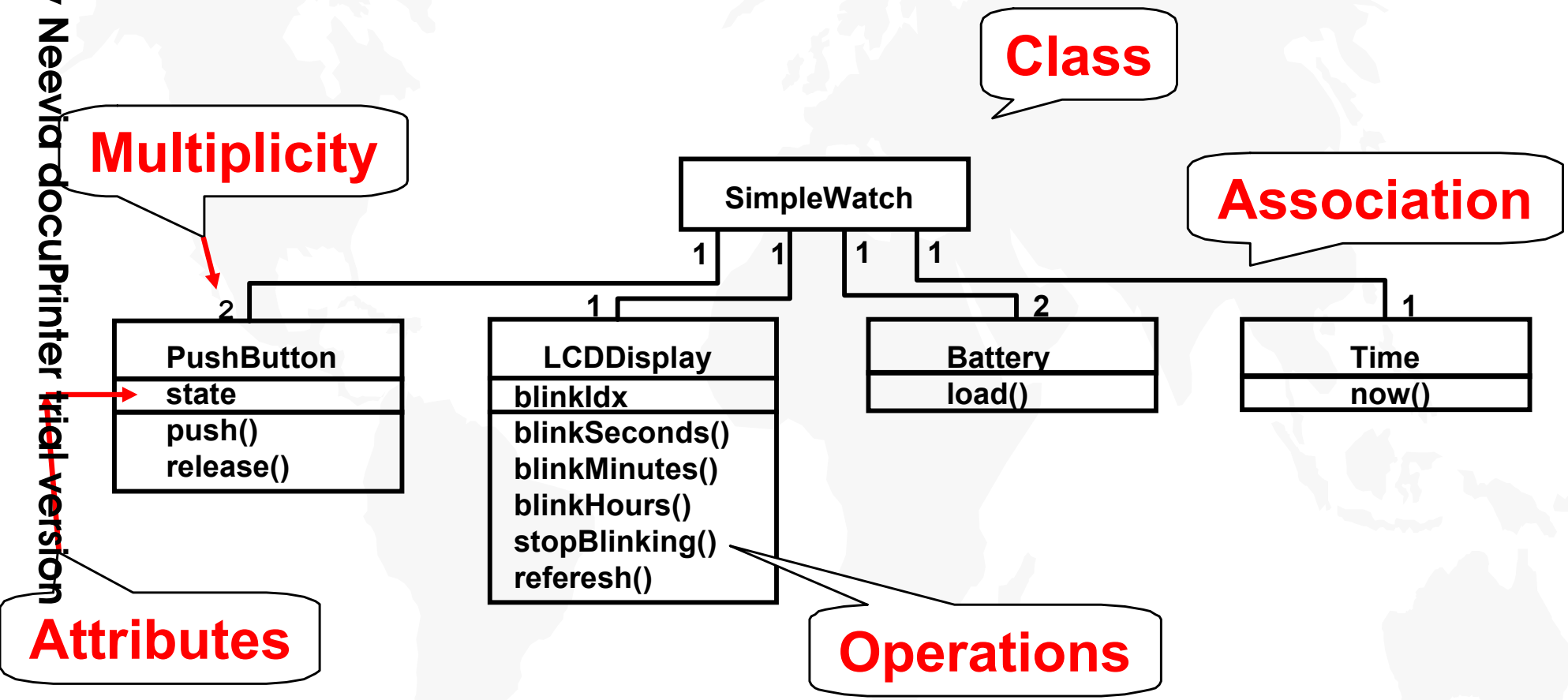
Use Case Diagram examples



A use case documents the interaction between the system user and the system. It is highly detailed in describing what is required but is free of most implementation details and constraints.

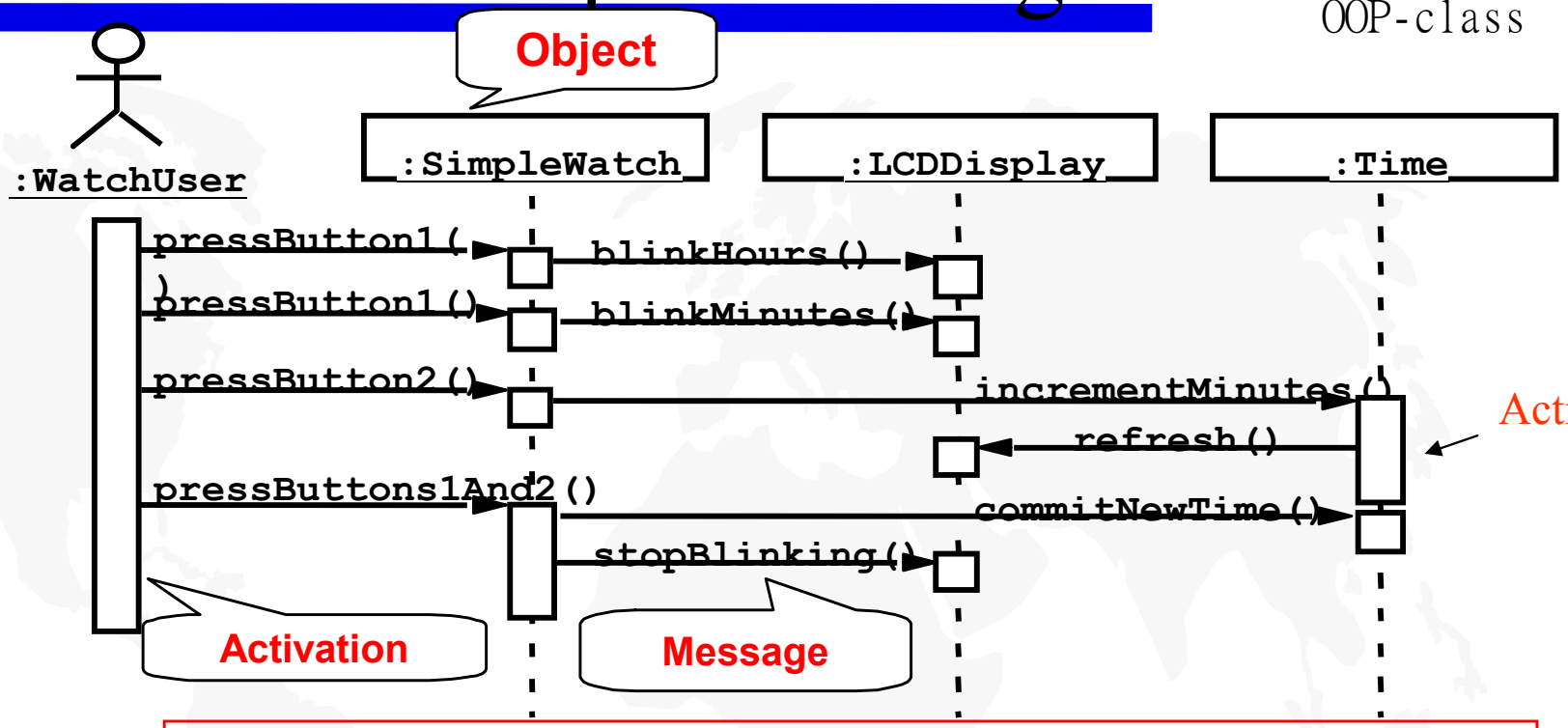
Use case diagrams represent the functionality of the system from user's point of view. (強調 what, 但暫不管 how)

Class Diagram : a simple Watch

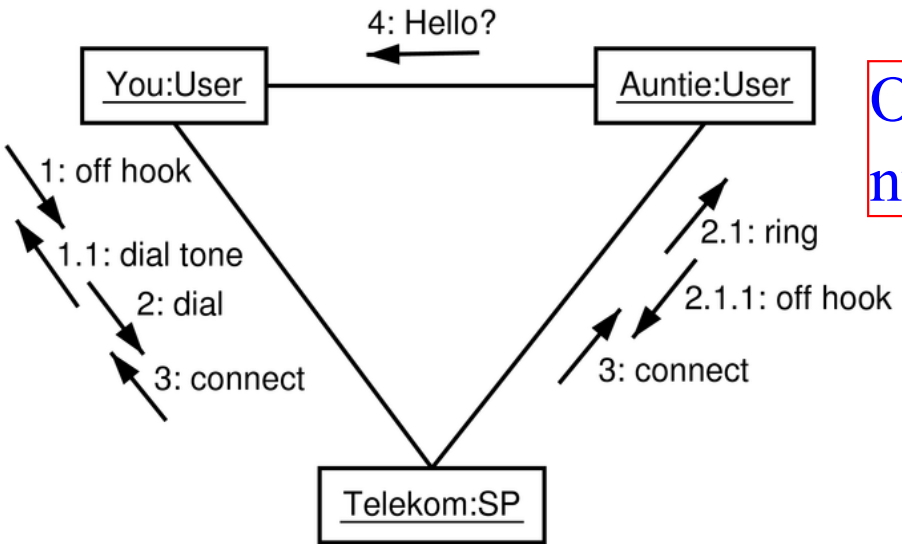


Class diagrams represent the structure of the domain or system

Sequence Diagram



Sequence diagrams represent the behavior as interactions
It shows sequence of events for a particular use case

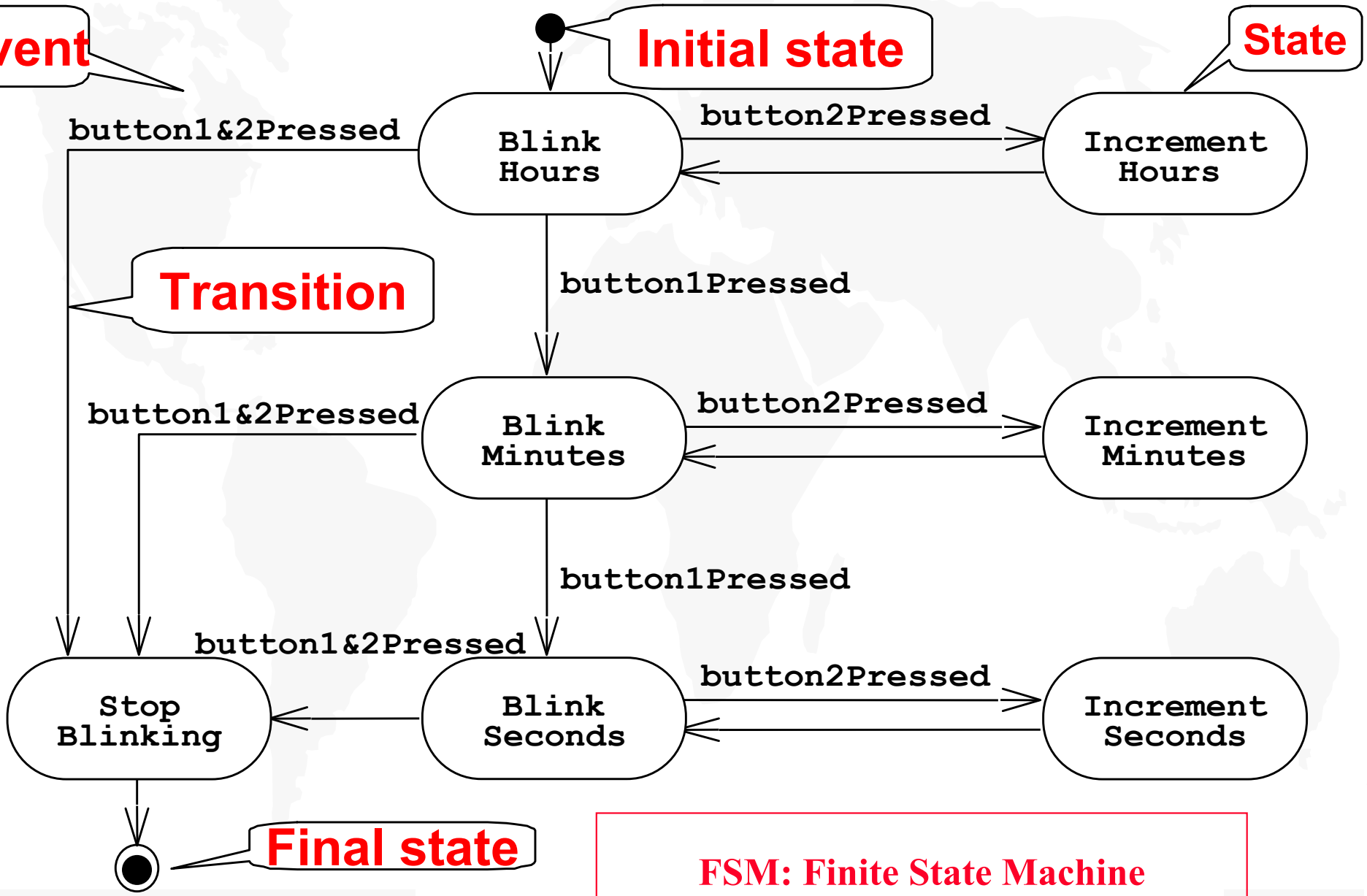


Object diagram with numbered messages

Sequence numbers of messages are nested by procedure call

Collaboration Diagram

State chart Diagrams for the watch



FSM: Finite State Machine

Activity Diagrams

- An activity diagram shows flow control within a system

- An activity diagram is a special case of a **state chart diagram** in which states are activities ("functions")
- Two types of states:
  - *Action state:*
    - Cannot be decomposed any further
    - Happens "instantaneously" with respect to the level of abstraction used in the model
  - *Activity state:*
    - Can be decomposed further
    - The activity is modeled by another activity diagram

描述Business process或use case的操作流程; 像流程圖

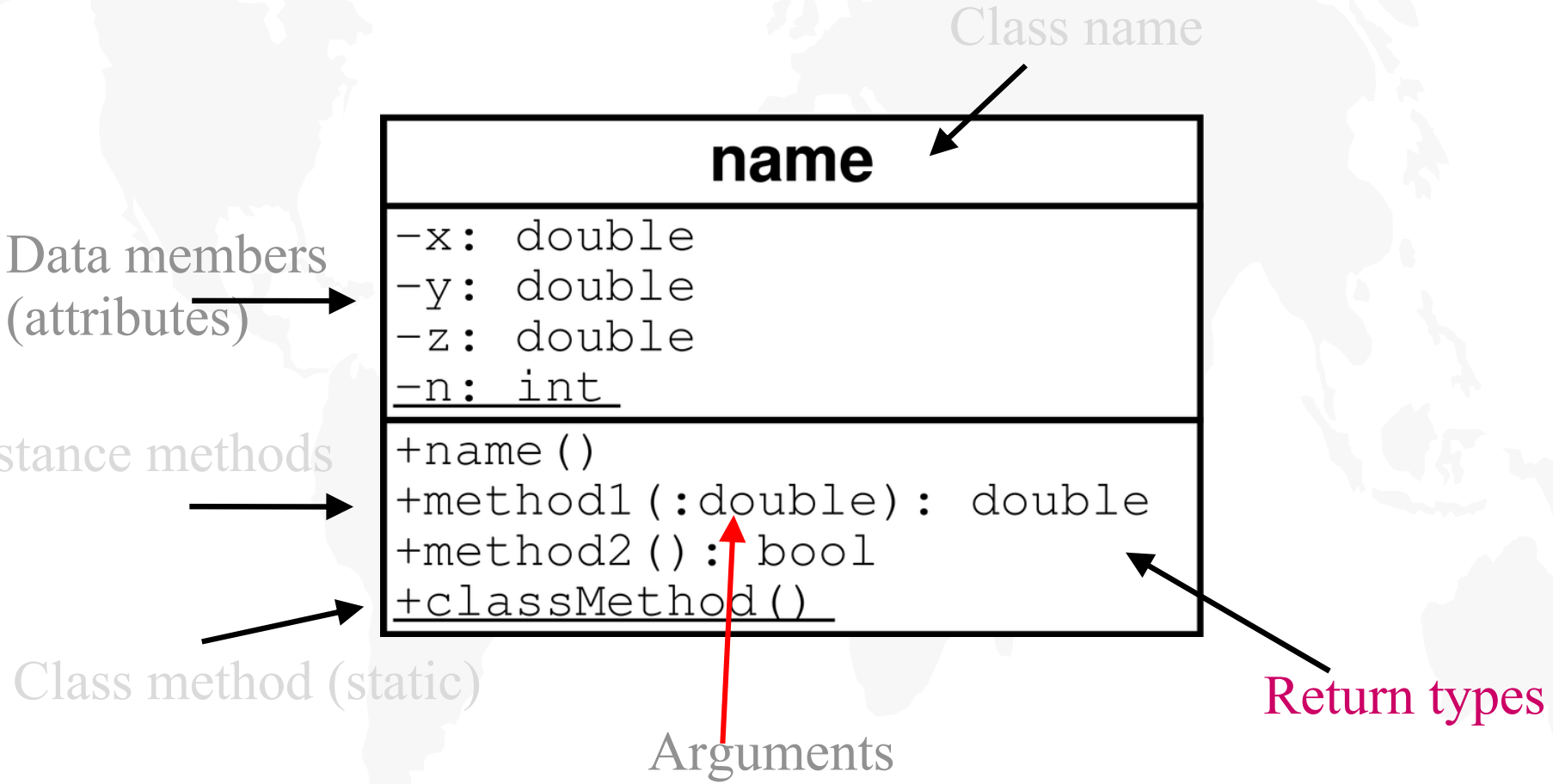
Classes in UML

- **Classes** describe objects
  - Behaviour (member function signature / implementation)
  - Properties (attributes and associations)
  - Association, aggregation, dependency, and inheritance relationships
  - Multiplicity and navigation indicators
  - Role names
- Objects described by classes collaborate
  - Class relations → object relations
  - Dependencies between classes

UML Class

Visibility shown as

+	public
-	private
#	protected



Data members, arguments and methods are specified by
visibility name : type

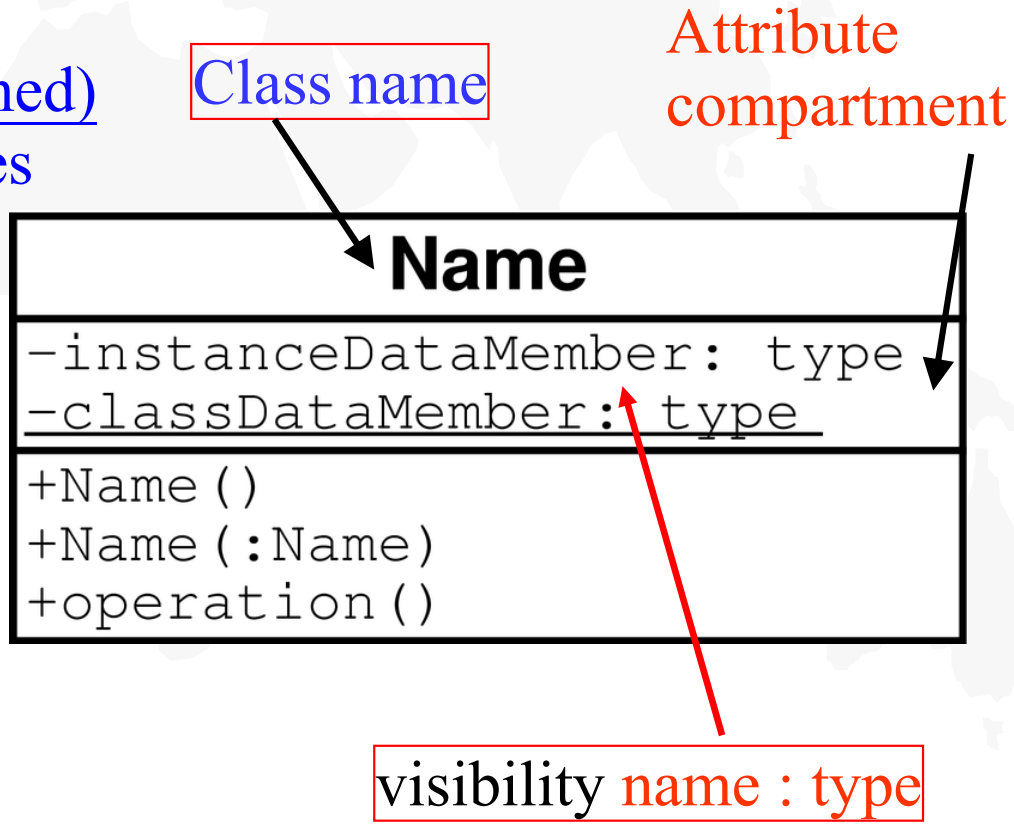
Class Attributes

Attributes are the instance data members and class data members

Class data members (underlined) are shared between all instances (objects) of a given class

Data types shown after ":"

Visibility shown as
- \+ public
- \- private
- # protected

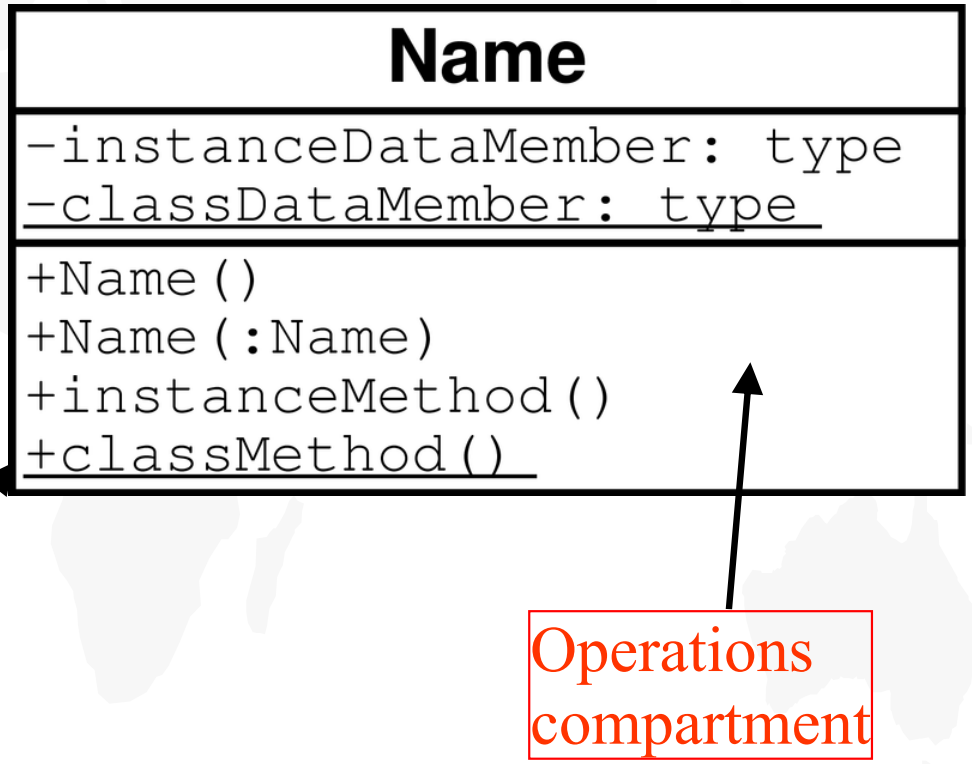


Class Operations (Interface)

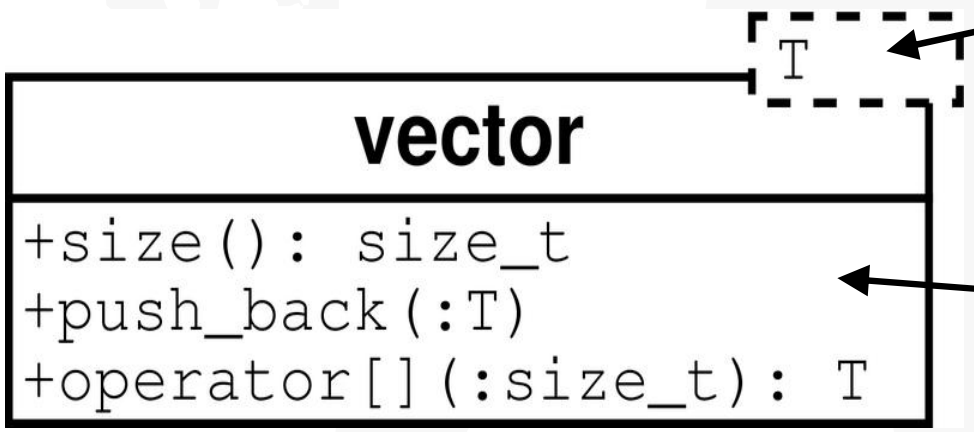
Operations are the class methods with their argument and return types

Public (+) operations define the class interface

Class methods (underlined) can only access to class data members, no need for a class instance (object)



Template Classes

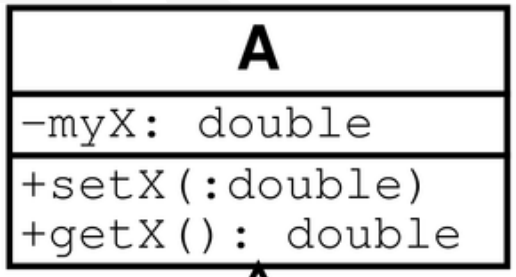


Type parameter(s)

Operations compartment as usual, but may have type parameter instead of concrete type

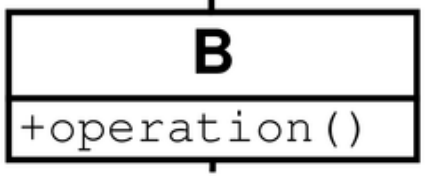
Generic classes depending on parametrised types

Class Inheritance

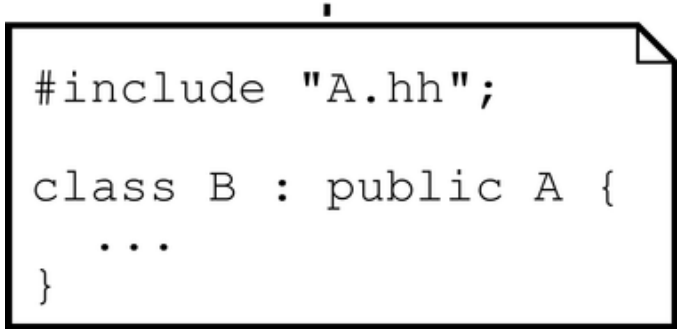


Base class or super class

Arrow shows direction of dependency (B inherits A)



Derived class or subclass

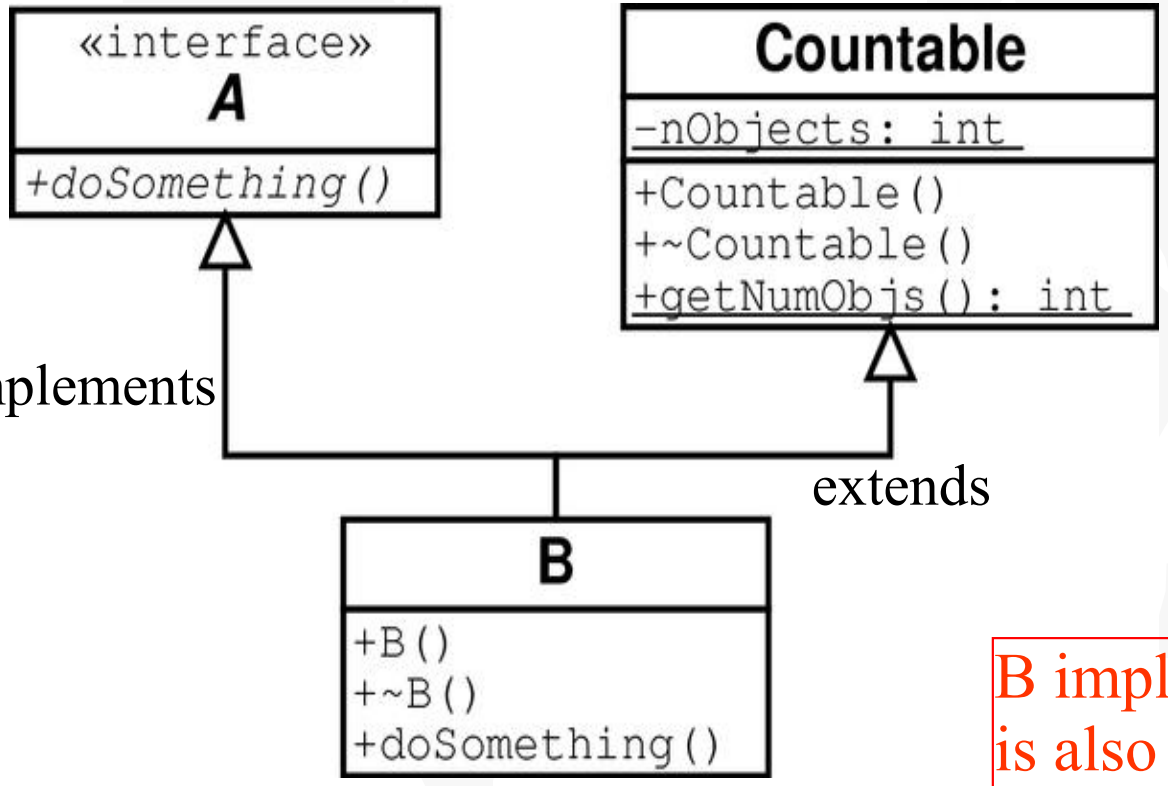


→ B inherits A's interface, behaviour and data members

→ B can extend A, i.e. add new data members or member functions

→ B depends on A, A knows nothing about B

Multiple Inheritance (Java)



The derived class inherits interface, behaviour and data members of all its base classes

Extension and overriding works as before

B implements the interface A and is also a "countable" class since it inherits class Countable

class B extends Countable implements A { /*…*/ }

How can two classes be related?

- *Generalization-specialization* or **IsA**
  - NamedBox IsA Box
  - Diagram: Triangle on the relationship line
- *Association* or **HasA**
  - Box HasA Pen
  - Diagram: Just a relationship line
  - *Aggregation* is a *part-whole* relationship
    - Diagram: Diamond on the line
- *Dependency* or **TalksTo**
  - Dependency is sort of temporary HasA
    - Diagram: Dashed line in UML

Recommended Book: UML Distilled

- Serious O-O designers ***DO*** use UML
- UML Distilled by **Martin Fowler** is a great *practical* introduction to UML
- Official UML book series published by Addison-Wesley

http://www.omg.org

UML Tools

- Most complete tool: Rational Rose, http://www.rational.com
- Lots of others
  - Together by Object International, http://www.oi.com
  - BOOST (Basic Object-Oriented Support Tool) by Noel Rappin, available on CD/CoWeb
  - Argo-UML, ObjectPlant, Posiden, etc.

O O Features

- Encapsulation
- Information Hiding (Data Hiding)

Object Based

- **Inheritance**
- **Polymorphism**

For Software Reuse

Object **Oriented**
(物件導向)
(個體導向)

OO features

- ✓ Encapsulation
- ✓ Information Hiding

抽象化的概念以前
就有: 函數/副程式

The concept of abstraction is fundamental in programming

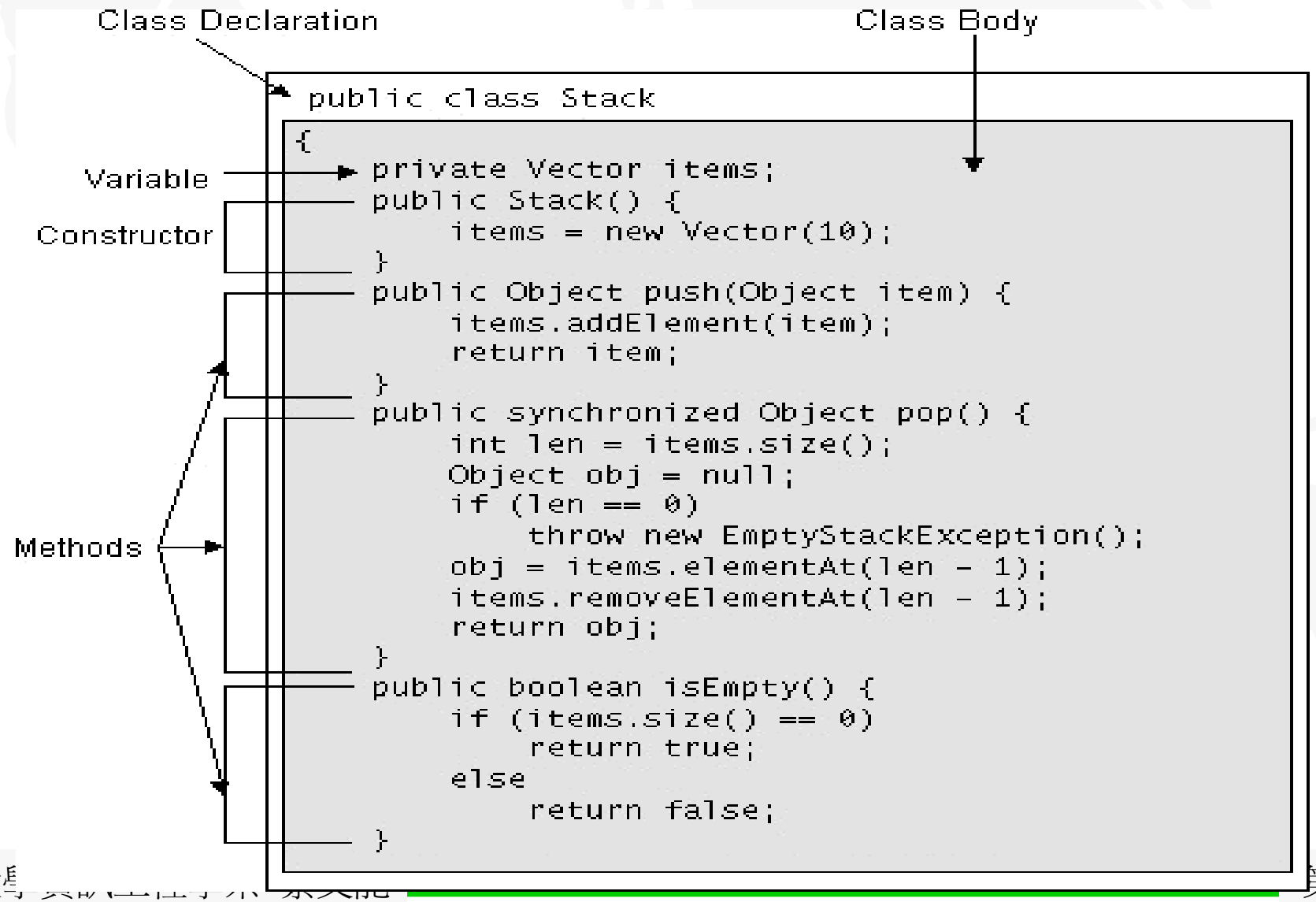
- ◆Subprogram / function
  - ➢ **process abstraction**
- ◆ADT
  - ➢ **Data abstraction**

增加:

- ✓ Inheritance
- ✓ Polymorphism

Software Reuse

Class elements



Constructors

For purpose of initialization objects have special methods called *constructors.*

To write a constructor you must adhere to two rules :

1. The method name must exactly match the class name.
2. There must not be a return type declared for the method.

Java has NO destructor.

The Default Constructor

Every class has a constructor.

If you don't write one then Java provides one for you.
This constructor takes no arguments and has an empty body.

? What If you write one constructor ?

Struct – 自訂資料型別 in C/C++

Why using struct ?

Java has no struct

```
#include <stdio.h>
struct Student {
    long  sid;
    char name[9];   /* 四個Big5中文 */
    float score[13];   /* 最多修13 科☺ */
};      /*注意 分號 */
int main( ) {
   struct Student x;     /* C++ 和 C99 不用寫 struct */
   x.sid = 123;
   strcpy(x.name, "張大千") ;  /*注意 */
   /* 用 loop 把成績讀入 x.score[?] */
}
```

Struct vs. Class in C++ (1/2)

```
#include <iostream.h>
class Student {       // 若改爲 struct Student { …?
    long  sid;
};
int main( ) {
    Student x;
    x.sid = 123;  // compile錯, access not allowed
    cout << "== " << x.sid << endl;  // compile錯
    return 0;
}
```

Struct vs. Class in C++ (2/2)

```
#include <iostream.h>
class Student {      //  private:
    long  sid;
 public:  showNumber( ) { return sid; }
          setId( long xx) { sid = xx; }
};
int main( ) {
   Student x, y;
   x.setId(123);   // OK, 透過 public function setId( )
   cout << " == " << x.showNumber( ) << endl;  // OK
   return 0;
}
```

Class(Object) Constructor 1/4 (C++)

```
#include <iostream.h>
class Student {      //  private:
     long  sid;
  public:  showNumber( ) { return sid; }
           setId( long xx) { sid = xx; }

};
int main( ) {
    Student x, m, n;
    Student y(456); // Error
    x.setId(123);
    return 0;
}
```

使用 default constructor
沒有參數

Student y(456) ; 有問題!
因爲 default constructor
就是不能有參數

Class(Object) Constructor 2/4 (C++)

```
#include <iostream.h>
class Student {      //  private:
     long  sid;
 public:  showNumber( ) { return sid; }
          setId( long xx) { sid = xx; }
          Student(long  x) { sid = x; }
};
int main( ) {
    Student x;   // Error
    Student y(456); // OK now
    x.setId(123);
    return 0;
}
```

加入這constructor

Student x; 有問題!
因爲 default constructor
不能用了 –
有兩種解決方法:

Student y(456) ; 現在OK了

Class(Object) Constructor 3/4 (C++)

```
#include <iostream.h>
class Student {      //  private:
     long  sid;
 public:  showNumber( ) { return sid; }
          setId( long x) { sid = x; }
          Student(long  x=0) { sid = x; }
};
int main( ) {
    Student x, y(456); // OK
    x.setId(123);
    return 0;
}
```

Student x; 有問題
解決方法一
使用Default 參數

Class(Object) Constructor 4/4 (C++)

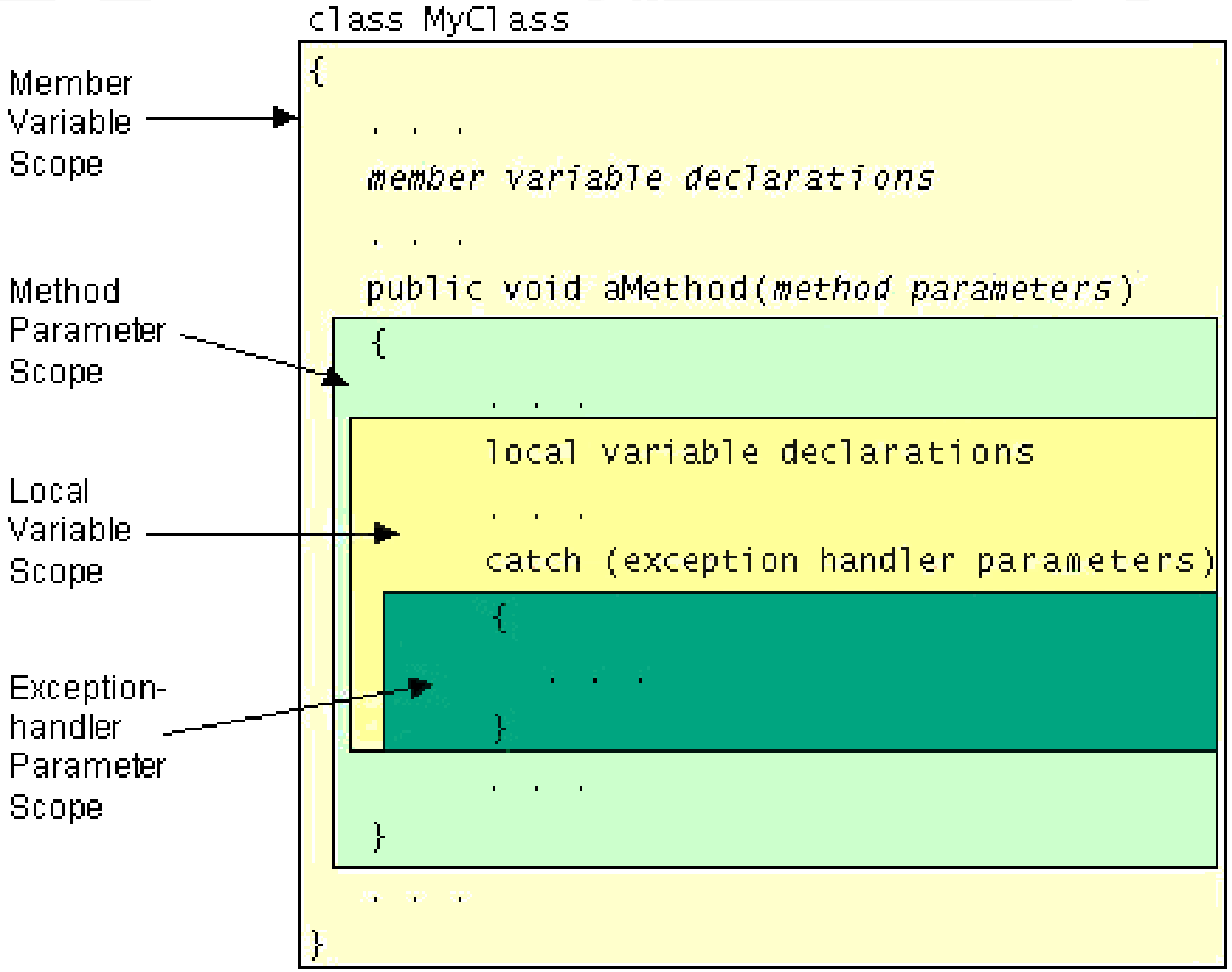
```
#include <iostream.h>
class Student {      //  private:
    long  sid;
 public:  showNumber( ) { return sid; }
          setId( long xx) { sid = xx; }
          Student(long  x) { sid = x;  cout << "Hehehe\n"; }
          Student( ) { sid=0; cout << "Haha\n"; }
};
int main( ) {
   Student x, y(456); // OK
   x.setId(123);
   return 0;
}
```

Student x; 有問題
解決方法之二
再加一個沒參數的constructor

More about Classes

- Properties:
  - Single implementation inheritance - *extends*
    - The inherited class is called "superclass".
  - Multiple interface inheritance - *implements*
- All classes inherit from java.lang.**Object**
- Scope: (next slide)

Scope



The Life Cycle of an Object – Creation (1/2)

- Creating Objects

```
Rectangle rect = new Rectangle();
```

  - Declaring an Object
    - Object declarations can also appear alone :

```
Rectangle rect;
```

    - In Java, classes and interfaces can be used as data types.
    - **Declarations do not create new objects.**
  - Instantiating an Object
    - The *new* operator allocates memory for a new object.
    - The *new* operator requires a argument: a call to a constructor.
    - The *new* operator creates the object, and the constructor initializes it.

```
new Rectangle(100, 200);
```

    - The *new* operator returns a reference to the newly created object.

```
Rectangle rect = new Rectangle(100,200);
```

    - After this statement, `rect` refers to a `Rectangle` object.

The Life Cycle of an Object -- Creation (2/2)

- Initializing an Object
  - Constructors **have the same name as the class** and **have no return type**.

```
public Rectangle(Point p)
public Rectangle(int w, int h)
public Rectangle(Point p, int w, int h)
public Rectangle()
```

Example:

```
Rectangle rect = new Rectangle(100, 200);
Rectangle rect = new Rectangle(
    new Point(44,78));
```

  - A constructor can takes no arguments, is called a *no-argument constructor*.
  - If a class does not define any constructors, Java provides a no-argument constructor, called the ***default constructor.***

The Life Cycle of an Object -- Access

- Using Objects
  - Referencing an Object's Variables

```
objectReference.variable
```

Example:

```
int areaOfRectangle = rect.height * rect.width;
```

  - Calling an Object's Methods

```
objectReference.methodName();   or
objectReference.methodName(argumentList);
```

Example:

```
rect.move(15, 37);
int areaOfRectangle =
     new Rectangle(100, 50).area();
```

The Life Cycle of an Object – Cleanup

- Cleaning Up Unused Objects
  - An object is eligible for garbage collection when there are no more references to that object.
  - The Garbage Collector
    - Automatically frees an object, if not referenced.
    - prevents memory leaks
    - claimed delay ~200ms for 4MB
    - Mark-sweep garbage collector
    - The garbage collector runs in a low-priority thread.
      - When the system runs out of memory.
      - In response to a request from a Java program.
      - When the system is idle.

Finalization

- When freeing an object, the system will call this first.
- The `finalize` method is a member of the `Object` class.
- Implementing a `finalize` method to **release resources** that aren't under the control of the **garbage collector**, such as native peers.
- Scenario: For a **file** object, when freed, need to **close** it.
- Problem: Make the internal JVM implementation for memory management very complicated.
- Example

```
public class Rectangle {
    public Point origin;
. . .
    protected void finalize() throws Throwable {
        origin = null;
        super.finalize();
    }
}
```

Example for Overriding (1/2)

```
class Point {
   protected int x,y;
   Point(int x, int y) {this.x = x; this.y = y;}
   void draw(){
     System.out.println("Point at"+x+","+y);
   };
}
```

Subclassing:

```
class Cpoint extends Point{
   private Color c;
   Cpoint(int i, int j, Color c){
     super(i,j);
     this.c =c;
     };
   void draw(){
     System.out.println("Color Point at " + i + "," + j + "," + c);
   };
}
```

Example for Overriding (2/2)

```
class PointApp{
      public static void main(){
        Point p = new Point(5,5);
        Cpoint cp = new Cpoint(1,1,Red);
        p.draw();
        cp.draw();
        p = cp;
        p.draw();
             // call Point's draw() or Cpoint's draw()?
      }
}
```

Overriding(蓋掉) 不是 overloading !

Overloading (重載)

- A Method's Name (function name)
  - Java supports method name *overloading* so that multiple methods can share the same name.

```
class DataRenderer {
    void draw(String s) { . . . }
    void draw(int i) { . . . }
    void draw(float f) { . . . }
}
```

  - **Overloaded** methods are differentiated by the **number and type of the arguments.**
  - A subclass may ***override*** a method in its superclass.
    - The overriding method must have the same name, return type, and parameter list as the method it overrides.

Java 不可以 operator overloading

Abstract Classes in Java

- Abstract classes created using the abstract keyword:

  public abstract class MotorVehicle { … }
- In an abstract class, several abstract methods are declared.
  - An abstract method is not implemented in the class, only declared. The body of the method is then implemented in subclass.
  - An abstract method is decorated with an extra "abstract" keyword.
- Abstract classes can not be instantiated! So the following is illegal:

  MotorVehicle m = **new MotorVehicle;** **// 錯 !**

  // 抽象類別不能拿來生出物件

  //因爲抽象的東西根本不存在☹

Abstract methods

- Abstract methods are declared but do not contain an implementation.
- For example, the MotorVehicle class may have an abstract method gas( ):

```
public abstract class MotorVehicle {
    private double speed, maxSpeed;
    void accToMax( ) { speed = maxSpeed;}  //這個不是抽象函數!
    public abstract void gas( );  // 抽象函數;在 C++ 是寫 =0; 的函數
    /* … */
}
```

- So MotorVehicle can NOT be instantiated
  (ie., can not be used to create an object.)

Example of Abstract Classes (1/2)

- Example:

```
abstract class Stack {
  abstract void push(Object o);
  abstract Object pop();
}

public class ArrayStack extends Stack {
  .... // declare elems[] and top;
  void push(Object o) {   elems[top++] = o; }
  Object pop() {    return elems[--top]; }
}

class LinkedStack extends Stack {
  .... // declare ...
  void push(Object o) {  .... }
  Object pop() {  ....  }
}
```

Example of Abstract Classes (2/2)

```
// ================================ Another file
...
static public int testStack(Stack s) { ....
    s.push(x);      s.push(y);
    o1 = s.pop();   o2 = s.pop();
    // check if o1 == y && o2 == x.
}
...
ArrayStack as = new ArrayStack();
testStack(as);
...
```

Interfaces

- Defines a set of methods that a class must implement
  - An *interface* is like a class with nothing but abstract methods and final and static fields (constants).
  - An *interface* can also have fields, but the fields must be declared as final and static.
  - All methods are abstract implicitly.

  - *Interface* can be added to a class that is already a subclass of another class. (implements)
  - To declare an interface:

```
public interface ImportTax {
        public double calculateTax( );
}
```

Example of *Interfaces* (1/2)

```
interface Stack {
  void push(Object o);
  Object pop();
}

public class ArrayStack implements Stack {
  .... // declare elems[] and top;
  void push(Object o) {   elems[top++] = o; }
  Object pop() {    return elems[--top]; }
}

class LinkedStack implements Stack {
  .... // declare ...
  void push(Object o) {  .... }
  Object pop() {  ....  }
}
```

Example of *Interfaces* (2/2)

```
//================================ Another file
...
static public int testStack(Stack s) { ....
    s.push(x);      s.push(y);
    o1 = s.pop();   o2 = s.pop();
    // check if o1 == y && o2 == x.
}
...
ArrayStack as = new ArrayStack();
testStack(as);
...
```

Difference between Abstract Class and *Interface*

- An abstract class is a class containing several abstract methods. Each abstract method is prefixed with the keyword “abstract”.
- An abstract class can not be instantiated but can be extended (subclassed).
- An *interface* contains only abstract methods and constants. Each abstract method is not prefixed with the keyword “abstract”.
- An *interface* can only be implemented.

抽象類別與 Interface（介面）用法不同!

How C++ Deals with *Interfaces* or Abstract Classes

- Use virtual functions:

```
class Stack {
  virtual void push(void* o) = NULL;
  virtual void* pop() = NULL;   // NULL 就是 0
}
```

// 注意 C++ 只有 **virtual**函數才可寫 = NULL; 或 = 0;
// 寫 = NULL; 或 =0; 的 **virtual** function叫做pure virtual function
// Java 則永遠自動 **virtual**

- What are the internal designs?
  - Key for the designs of COM/DCOM (MicroSoft)

Example for C++

```
class ArrayStack : Stack {
  .... // declare elems[] and top;
  void push(Object o) {   elems[top++] = o; }
  Object pop() {    return elems[--top]; }
}
class LinkedStack : Stack {
  .... // declare ...
  void push(Object o) {  .... }
  Object pop() {  ....  }
}
============================== Another file
...
int testStack(Stack s) { ....
    s.push(x);     s.push(y);
    o1 = s.pop();  o2 = s.pop();
    // check if o1 == y && o2 == x.
}
...
ArrayStack as = new ArrayStack();
testStack(as);
...
```

Packages and Library

- Organize class **name space**
- Hierarchical division must be reflected in directory structure, i.e.,

```
package cyc.sys;
import java.io.*;
class Login {  /* in fact, this is cyc.sys.Login */ }
```

- Default packages (library) -- JDK
  - java.lang - Object, Thread, Exception, String,...
  - java.io - InputStream, OutputStream, ...
  - java.net - Networking
    - support for Socket & URL-based objects
  - java.awt - Abstract window toolkit
    - defines a simple, restricted windowing API
  - java.util - Hashtable, Vector, BitSet, Regexp, ..
  - java.tools - Compilation, Debugging, Documentation
  - …

Naming a Package

- The name of the Rectangle class in the graphics package is really graphics.Rectangle
- The name of the Rectangle class in the java.awt package is really java.awt.Rectangle
- Convention
  - com.company.package
  - com.company.region.package

Java 並沒有 graphics.Rectangle

Access Modifiers

- (no keyword): class/package access (**different from C++**)
  I.e., **package available**
- **`public`**: world access (same as C++)
- **`private`**: class access (same as C++)
- **`protected`**: subclassing access(same as C++)

- **`static`**: only one copy for all instances
- **`abstract`**: left unimplemented
- **`final`**: not overridden
- **`native`**: methods implemented in native code.
- **`synchronized`**: described in Thread
- **`transient`**: data that is not persistent

Comparisons of Access Modifiers

- Controlling Access to Members of a Class

Specifier	Class	Subclass	Package	World
private	X			
protected	X	X	X	
public	X	X	X	X
package	X		X	

注意西方人是打 X 表示勾選!

Static member data

- static:
  - only one copy for the class and all instances (data)
  - Usually used as a global variable (even no instances)

```
class TestStatic {
  int x, y, z;
  int average() {return (x+y+z)/3; }

  static int j = 0;
  static int peek() {
      j += 2;
      x = 3;      // error!
  }
  static int k = peek();
  static { j = j*5; }   // done when loading class
  public static void main(String[] args) {
    ...
    TestStatic ts = new TestStatic();
    int ave = ts.average();
    ...
  }
}
```

Static method

Means You can use it even you have NO object.

Static variable j,k

Static block

Static method

Method 不論是否 static 都是只會有一份copy

Final

- Final variables:
  - The value of a final variable cannot change after it has been initialized.
    - Such variables are similar to constants in other programming languages.

```
final Color red = Color.red;
```

    - A final local variable that has been declared but not yet initialized is called a blank final.

```
final int blankfinal;
. . .
blankfinal = 0;
```

- Final methods: methods that cannot be overridden.
  - Prevent clients from overriding.
  - Allow for JIT to optimize.

Transient vs. Volatile

- **`transient`** : used in object serialization to mark member variables that should not be serialized.
- **`volatile`** : used to prevent the compiler from performing certain optimizations on a member.

Nested Class

- A nested class is a class that is a member of another class.

```
class EnclosingClass{
    . . .
    class ANestedClass {
        . . .
    }
}
```

- Define **a class within another class** when the nested class makes sense only in the context of its enclosing class.
  - For example, a text cursor makes sense only in the context of a particular text component.
- A nested class has a special privilege:
  - It has unlimited access to its enclosing class's members, even for private.
    - Just like a method that can access private.

Types of Nested Class

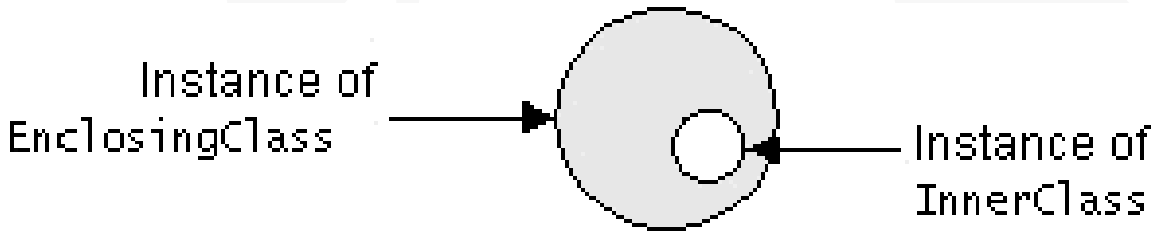
```
class EnclosingClass{
    . . .
    static class AStaticNestedClass {
        . . .
    }
    class InnerClass {
        . . . /* static is wrong in this inner class */
    }
}
```

- A **static nested class** is called: a static nested class.
  - A static nested class cannot refer directly to instance variables or methods defined in its enclosing class
- A **nonstatic** nested class is called an **inner class**.
  - An inner class is associated with an instance of its enclosing class and has direct access to that object's instance variables and methods.
  - Because an inner class is associated with an instance, it cannot define any static members itself.

Nested vs. Inner

- To help differentiate the terms nested class and inner class:
  - The term "nested class" reflects the syntactic relationship between two classes.
  - In contrast, the term "inner class" reflects the relationship between instances of the two classes.
    - An instance of InnerClass can exist only within an instance of EnclosingClass,
    - It has direct access to instance variables and methods of its enclosing instance.

```
class EnclosingClass {
    . . .
    class InnerClass {
        . . .
    }
}
```



Example: Adaptor

- Using an Inner Class to Implement an Adapter

```
import java.util.Enumeration;
import java.util.NoSuchElementException;
public class Stack {
    private java.util.Vector items;

    // ... code for Stack's methods and constructors not shown...

    public Enumeration enumerator() {
        return new StackEnum();
    }

    class StackEnum implements Enumeration {
        int currentItem = items.size() - 1;
        public boolean hasMoreElements() {
            return (currentItem >= 0);
        }
        public Object nextElement() {
            if (!hasMoreElements())
                throw new NoSuchElementException();
            else
                return items.elementAt(currentItem--);
        }
    }
}
```

Anonymous Class

- **You can declare an inner class without naming it.**
- **Example: (Rewrite the previous slide.)**

```
public class Stack {
    private Vector items;
    ...//code for Stack's methods and constructors not shown...
    public Enumeration enumerator() {
        return new Enumeration() {
            int currentItem = items.size() - 1;
            public boolean hasMoreElements() {
                return (currentItem >= 0);
            }
            public Object nextElement() {
                if (!hasMoreElements())
                    throw new NoSuchElementException();
                else
                    return items.elementAt(currentItem--);
            }
        };
    }
}
```

Upcasting and Polymorphism(1/4)

- Upcasting: Taking an object reference and treating it as a reference to its base type is called *upcasting*, because of the way inheritance trees are drawn with the base class at the top.

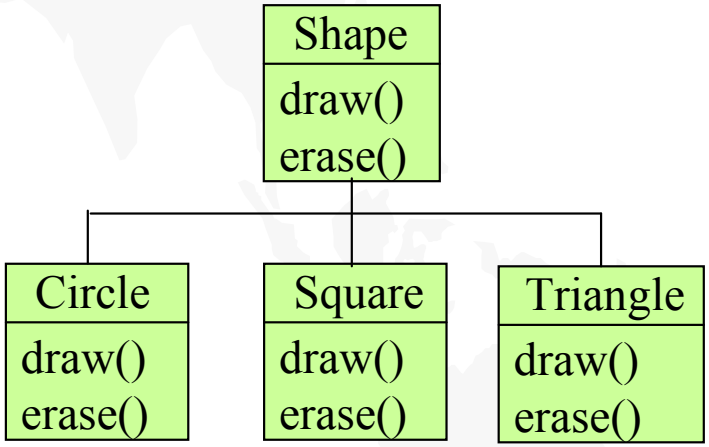
.

Upcasting and Polymorphism (2/4)

- Polymorphism: In Java, the principle that the actual type of the object determines the method to be called is called ***polymorphism***.

```
class Shape {
    void draw( ) {}
    void erase( ) {}
}

class Circle extends Shape {
    void draw( ) {
        System.out.println("Circle.draw( ) ");
    }
    void erase( ) {System.out.println("Circle.erase( )");}
}
```



Upcasting and Polymorphism (3/4)

```
class Square extends Shape {
    void draw( ) {
        System.out.println("Square.draw( ) ");
    }
    void erase( ) {System.out.println("Square.erase( )");}}

class Triangle extends Shape {
    void draw( ) {
        System.out.println("Triangle.draw( )");
    }
    void erase( )          {System.out.println("Triangle.erase( )");}
}
```

Upcasting and Polymorphism (4/4)

```
public class Shapes {
    public static Shape randShape() {
            switch((int) (Math.random()*3)) {
                case 0: return new Circle();
                case 1: return new Square();
                case 2: return new Triangle();
                default : return new Circle();}
    }
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        for (int i = 0; i < s.length; i++)
                s[i] = randShape();
         //Make polymorphism method calls:
        for (int i = 0; i < s.length; i++) s[i].draw();
    }
}
```

Polymorphism in C++ (1/2)

```
// polymo.cpp -- CopyWrong by tsaiwn@csie.nctu.edu.tw
#include <iostream.h>
class Shape{
   public:
     virtual   // 把這列去掉再 run 看看
       void draw( ) { cout << "drawing\n"; }
};
class Line: public Shape{
   public:
      void draw( ) { cout << "draw a line\n"; }
};
class Circle: public Shape{
   public:
      void draw( ) { cout << "here is a circle\n"; }
};
```

Polymorphism in C++ (2/2)

```
int main(){
    Circle * ppp;
    Shape * fig[9];
   // ppp = new Shape();   // error
    ppp = (Circle *)new Shape();
    ppp -> draw();
    ppp = (Circle *)new Line();      ppp -> draw();
    ppp = new Circle();        ppp -> draw();
    cout << "======" << endl;
    fig[0] = new Line();
    fig[1] = new Circle();
    fig[2] = new Circle();
    fig[3] = new Line();   fig[4] = new Circle();
    for(int k=0; k<5; k++){
            fig[k] -> draw();
    }
}
```

Java: OOP, Object, Class

謝謝捧場

http://www.csie.nctu.edu.tw/~tsaiwn/course/java/

蔡文能