Object Oriented Programming

1. Reflection

2. Java Interfaces

3. OO Application Development

4. UML Object and Class Diagrams

Methods in the `Object` class

- *Object* defines default versions of the following methods:
  - `toString()` – returns a string (readable representation of the object)
  - `equals(Object obj)` – must be overridden for content equality
  - `hashCode()` – returns the hash-code value for the object; values are different for different objects
  - `getClass()` – returns an object of type Class; there is an object of type Class for each class of an application
  - `notify()`, `notifyAll()`, `wait()`, `wait(long timeout)`, `wait(long timeout, int nanos)` – used with multithreading
  - `clone()` – creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object.
  - `finalize()` – intended to perform cleanup actions before the object is irrevocably discarded.

Equality

- There are two different kinds of equality:
  - *Identity equality* means that two expressions have the same identity. (I.e., they represent the same object.)
  - *Content equality* means that two expressions represent objects with the same value/content .

The == symbol tests for identity equality, when applied to reference data.

- Example:

```java
AOval ov1, ov2;
ov1 = new AOval(0, 0, 100, 100);
ov2 = new AOval(0, 0, 100, 100);
if (ov1 == ov2){ System.out.println("identity equality");}
 else {System.out.println("content equality");}
```

- *The equals* method from `Object` can be used to provide a content equality check.

The class `Class`

- The class Class is defined as

  public final class Class extends Object implements Serializable, …

- Instances of the class `Class` represent classes and interfaces in a running Java application.

- An object of type `Class` contains information about the class whose instance is the calling object

- It does not have a self constructor

- `Class` objects are constructed at run-time by the JVM

- Two ways of constructing objects of this type:
  - `getClass()` from class `Object`
  - `forName()` from class `Class` (static method)

The class **Class**

- Methods:
  - **public String getName()**
    - returns a String representing the name of the entity represented by the **Class** object **this**
    - the entity can be: class, interface, array, primitive type, void
  - **public static Class forName(String className) throws ClassNotFoundException**
    - returns an object of type **Class** that contains info about the class of the call object
  - **public Class[] getClasses()**
    - returns an array of objects of type **Class**;
    - all classes and interfaces, public members of the class represented by this **Class** object

The class `Class`

- ## Methods (cont'd)
  - ### `Field[] getFields`
    - returns an array containing `Field` objects reflecting all the accessible public fields of the class or interface represented by this `Class` object
  - ### `Method[]getMethods()`
    - returns an array containing `Method` objects reflecting all the public *member* methods of the class or interface represented by this `Class` object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
  - ### `Constructor[]getConstructors()`
    - returns an array containing `Constructor` objects reflecting all the public constructors of the class represented by this `Class` object.

The `instanceof` Operator

- The `instanceof` operator checks if an object is of the type given as its second argument

  `Object instanceof ClassName`

  - This will return `true` if `Object` is of type `ClassName`, and otherwise return `false`
  - Note that this means it will return `true` if `Object` is the type of *any descendent class* of `ClassName`

The `getClass()` Method

- Every object inherits the same `getClass()` method from the `Object` class
  - This method is marked `final`, so it cannot be overridden
- An invocation of `getClass()` on an object returns a representation *only* of the class that was used with `new` to create the object
  - The results of any two such invocations can be compared with `==` or `!=` to determine whether or not they represent the exact same class

    ```
    (object1.getClass() == object2.getClass())
    ```

`instanceof` and **`getClass()`**

- Both the **`instanceof`** operator and the **`getClass()`** method can be used to check the class of an object

- However, the **`getClass()`** method is more exact
  - The **`instanceof`** operator simply tests the class of an object
  - The **`getClass()`** method used in a test with **`==`** or **`!=`** tests if two objects *were created with* the same class

Examples

- Print a class name using a **Class** object

```java
void printClassName(Object obj) {
    System.out.println(obj + " is of class " +
    obj.getClass().getName());
}
```

- More examples

```java
Circle c = new Circle(5);
printClassName(c);
Class c1 = c.getClass();
System.out.println(c1.getName()); // prints "Circle"
Triangle t = new Triangle(7);
printClassName(t);
try {
    Class c2 = Class.forName("Triangle");
    System.out.println(c2.getName()); // prints "Triangle"
}
catch (ClassNotFoundException e) {
    System.err.println("No class for \"Triangle\"" +
  e.getMessage());
}
```

The Need for Specifications

- A program is assembled from a collection of classes that must "work together" or "fit together"
  - What does it mean for two classes fit together?
- Example:
  - A portable radio – requires batteries to operate. What batteries?
  - "Two AA batteries" – a specification
  - Purposes for this specification:
    - For the user: tells which component must be fitted to the device to operate
    - For the radio manufacturer: what size for battery chamber, and what voltage and amperage to use for device electronics
    - For the battery manufacturer: size, voltage and amperage for batteries so that others can use

Specifications and Java

- The Java language and compiler can help us:
  - Write specifications of classes, and
  - Check that a class correctly matches (implements) its specification
- We will study:
  - the `interface` construction – lets us code in Java the information we specify. e.g. in a class diagram
  - the `extends` construction – lets us code a class by adding methods to a class that exists
  - the `abstract class` construction – lets us code an incomplete class that can be finished by another class

An Example

- Two people work on the same project, simultaneously:
  - one models a bank account
  - the other writes a monthly payment class for the account
- To accomplish this task, the two must agree on the *interface,* e.g.

```java
/** BankAccountSpecification specifies the behavior of a bank account. */
public interface BankAccountSpecification
{
   /** deposit adds money to the account
   * @param amount - the amount of the deposit, a nonnegative integer */
   public void deposit(int amount);
   /** withdraw deducts money from the account, if possible
   * @param amount - the amount of the withdrawal, a nonnegative integer
   * @return true, if the the withdrawal was successful;
   * return false, otherwise. */
   public boolean withdraw(int amount);
}
```

What is an Interface?

- The *interface* states that, whatever class is written to implement a **BankAccountSpecification**, the class must contain two methods, **deposit** and **withdraw**, which behave as stated

- In general, an *interface* is a device or a system that unrelated entities use to interact. Examples:
  - A remote control is an interface between you and a television set,
  - The English language is an interface between two people
  - The protocol of behavior enforced in the military is the interface between individuals of different ranks.

What is an Interface?

- Java: an *interface* is a type, just as a class is a type.

  - *Like* a class, an interface *defines methods*.

  - *Unlike* a class, an interface *never implements methods*;

  - Instead, classes that implement the interface implement the methods defined by the interface.

  - A class can implement multiple interfaces.

  - Use an interface to define a *protocol of behavior* that can be implemented by any class anywhere in the class hierarchy.

Definition. Interface Usefulness

- Definition: *An interface is a named collection of method definitions, without implementations*.

- An interface is not a class but a set of *requirements* for classes that want to conform to the interface.

- Interfaces are useful for the following:

  - *Capturing similarities* among *unrelated classes* without artificially forcing a class relationship

  - *Declaring methods* that one or more classes are expected to implement

  - Revealing an object's programming interface without revealing its class

  - Modeling multiple inheritance, a feature of some object-oriented languages (Java can do this) that allows a class to have more than one superclass

Defining an Interface

- An interface definition has two components: the interface declaration and the interface body

    - The interface *declaration* defines various attributes of the interface, such as its name and whether it extends other interfaces.

    - The interface *body* contains the constant and the method declarations for that interface.

A Class that References a Java Interface

```
/** MortgagePaymentCalculator makes mortgage payments */
public class MortgagePaymentCalculator
{
    private BankAccountSpecification bankAccount; // holds the address of
    // an object that implements the BankAccountSpecification
    /** Constructor MortgagePaymentCalculator initializes the calculator.
    * @param account - the address of the bank account from which we
    * make deposits and withdrawals */
    public MortgagePaymentCalculator(BankAccountSpecification account)
    {   bankAccount = account; }
    /** makeMortgagePayment makes a mortgage payment from the bank account.
    * @param amount - the amount of the mortgage payment */
    public void makeMortgagePayment(int amount)
    {
        boolean ok = bankAccount.withdraw(amount);
        if ( ok )
        {  System.out.println("Payment made: " + amount); }
        else { ... error ... }
    }
    ...
}
```

A Class that Implements a Java Interface

```java
/** BankAccount manages a single bank account; as stated in its
* header line, it implements the BankAccountSpecification: */
public class BankAccount implements BankAccountSpecification
{
    private int balance; // the account's balance
    /** Constructor BankAccount initializes the account */
    public BankAccount() { balance = 0; }
    // notice that methods deposit and withdraw match the same-named
    // methods in interface BankAccountSpecification:
    public void deposit(int amount) { balance = balance + amount; }
    public boolean withdraw(int amount) {
        boolean result = false;
        if ( amount <= balance ) {
                balance = balance - amount;
                result = true;
        }
        return result;
    }
    /** getBalance reports the current account balance
    * @return the balance */
    public int getBalance() { return balance; }
}
```
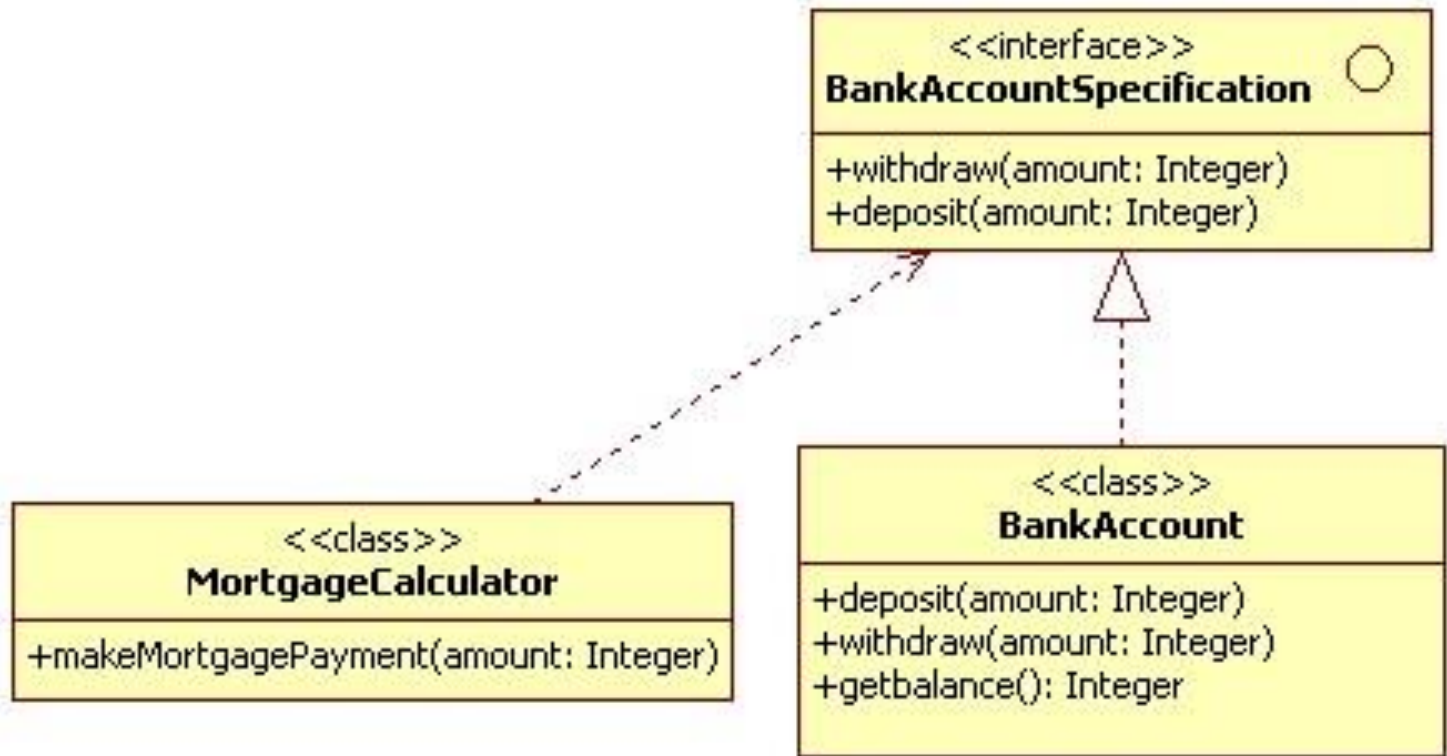
■To connect together the two classes – write a start-up method like this:

```
BankAccount myAccount = new BankAccount();
MortgageCalculator calc = new MortgageCalculator(myAccount);
. . .
calc.makeMortgagePayment(500);
```

Restrictions with Interfaces

- **Restrictions with interfaces:**
  - All its methods must be **abstract** instance methods, *no static methods allowed*.
  - Yet, all the variables defined in an interface must be **static final**, i.e. *constants*.
    - Values can be evaluated at compile time or delayed till class load time. (You can do some computation at class load time to compute the values)
    - Variables can be any type.
  - No static initializer blocks.
    - You must do your initializations in one line per variable.
    - No static initializer helper methods defined in the interface. If you need them, them must be defined outside the interface.

Instantiating

- Interface methods are always instance methods.
  - To use them, there must be some associated Object that implements the interface.
  - You can't instantiate an interface directly, but you can instantiate a class that *implements* an interface.
  - References to an Object can be done via the class name, via one of its superclass names, or one of its interface names.

What to Put in an Interface

- It is considered bad form to write an interface with nothing but static final constants it in.

- Usually, these qualifiers are left off. We just say:

```
interface MyConstants{
    double PI = 3.141592;
    double E = 1.7182818;
}
```

can be acessed as either MyConstants.PI or just PI by any class that implements the interface

- An interface should have at least one abstract method.

- If all you want is a collection of constants, use an ordinary class with `import static`

Another Example. A "database"

- Design a class named `Database`, to hold a collection of "record" objects, each of which possesses a unique "key" object to identify it
- Essential behaviors – an informal specification:
  - The `Database` holds a collection of `Record` objects, where each `Record` holds a `Key` object. The remaining structure of any `Record` is unimportant and unknown to the database
  - The `Database` will posses `insert, find`, and `delete` methods
  - `Record`s, regardless of their internal structure, will posses a `getKey` method that returns the `Record`'s `Key` object
  - `Key` objects will have an `equals` method that compares two keys for equality and returns true or false

Interfaces for Record and Key

```java
/** Record is a data item that can be stored in a database */
public interface Record
{
   /** getKey returns the key that uniquely identifies the record
   * @return the key */
   public Key getKey();
}
/** Key is an identification, or "key", value */
public interface Key
{
   /** equals compares itself to another key, m, for equality
   * @param m - the other key
   * @return true, if this key and m have the same key value;
   * return false, otherwise */
   public boolean equals(Key m);
}
```
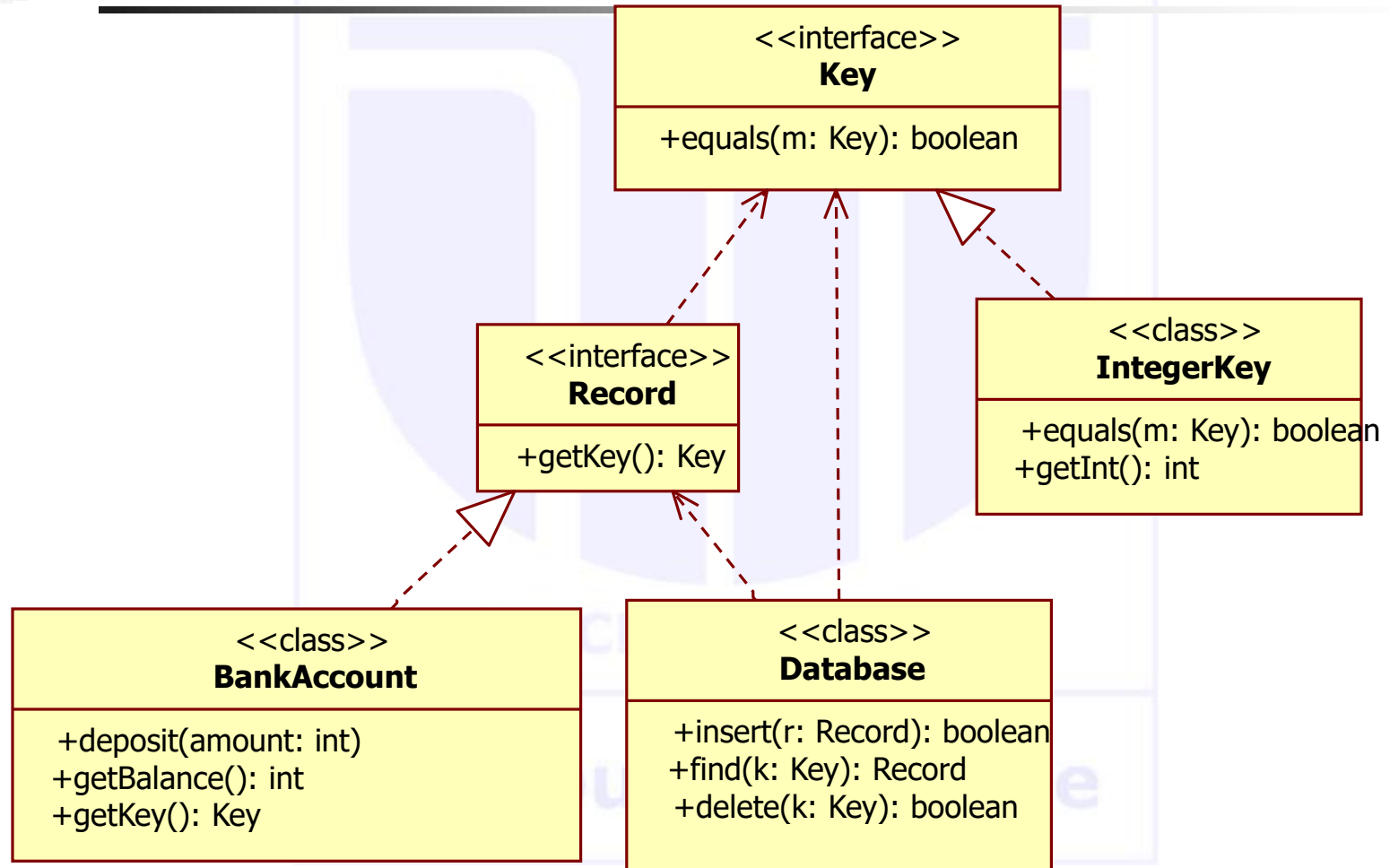
The Example in BlueJ
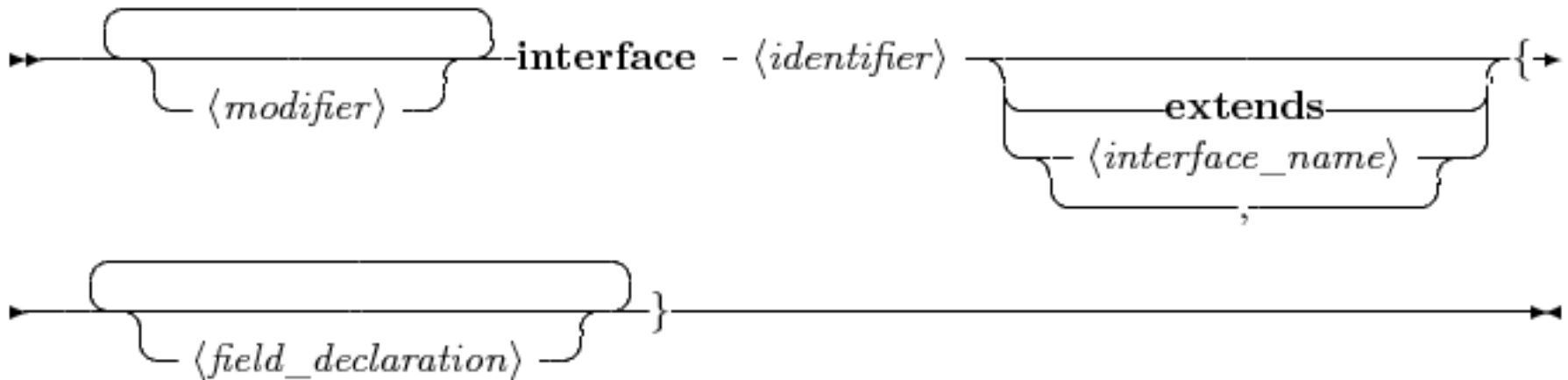
- BlueJ demo and discussion
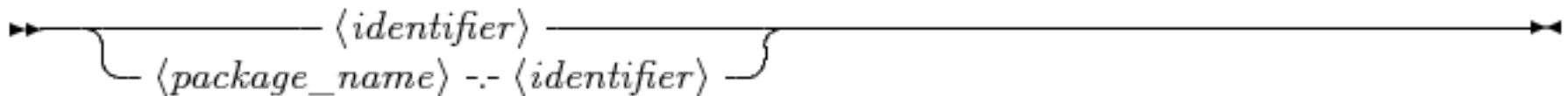
Class Diagram with Interfaces

```
        ┌──────────────────────────────┐
        │        <<interface>>         │
        │            Key               │
        ├──────────────────────────────┤
        │  +equals(m: Key): boolean    │
        └──────────────────────────────┘
```

<<interface>> Key
+equals(m: Key): boolean

<<interface>> Record
+getKey(): Key

<<class>> IntegerKey
+equals(m: Key): boolean
+getInt(): int

<<class>> BankAccount
+deposit(amount: int)
+getBalance(): int
+getKey(): Key

<<class>> Database
+insert(r: Record): boolean
+find(k: Key): Record
+delete(k: Key): boolean

Syntax Diagram for Interface Declaration

Interface_declaration

Interface_name

Superinterfaces

- If an `extends` clause is provided, then the interface being declared extends each of the other named interfaces and therefore inherits the methods and constants of each of the other named interfaces.

- These other named interfaces are the *direct superinterfaces* of the interface being declared.

- Any class that `implements` the declared interface is also considered to *implement all the interfaces that this interface extends and that are accessible to the class*.

  - We'll come back to this when discussing inheritance

Superinterfaces Example

```java
public interface Colorable {
    void setColor(int color);
    int getColor();
}
public interface Paintable extends
    Colorable {
    int MATTE = 0, GLOSSY = 1;
    void setFinish(int finish);
    int getFinish();
}
class Point {
    int x, y;
}
class ColoredPoint extends Point
    implements Colorable {
        int color;
        public void setColor(int
    color) {
        this.color = color;
     }
    public int getColor() {
       return color;
    }
}
```

```java
class PaintedPoint extends
    ColoredPoint implements Paintable
    {
     int finish;
     public void setFinish(int finish)
    {
            this.finish = finish;
     }
    public int getFinish() {
          return finish;
     }
}
```

- Relationships are as follows:
    - The interface **Paintable** is a *superinterface* of class **PaintedPoint**.
    - The interface **Colorable** is a *superinterface* of class **ColoredPoint** and of class **PaintedPoint**.
    - The interface **Paintable** is a *subinterface* of the interface **Colorable**, and **Colorable** is a *superinterface* of **Paintable**

Superinterfaces Example (cont'd)

- Class diagram for previous example
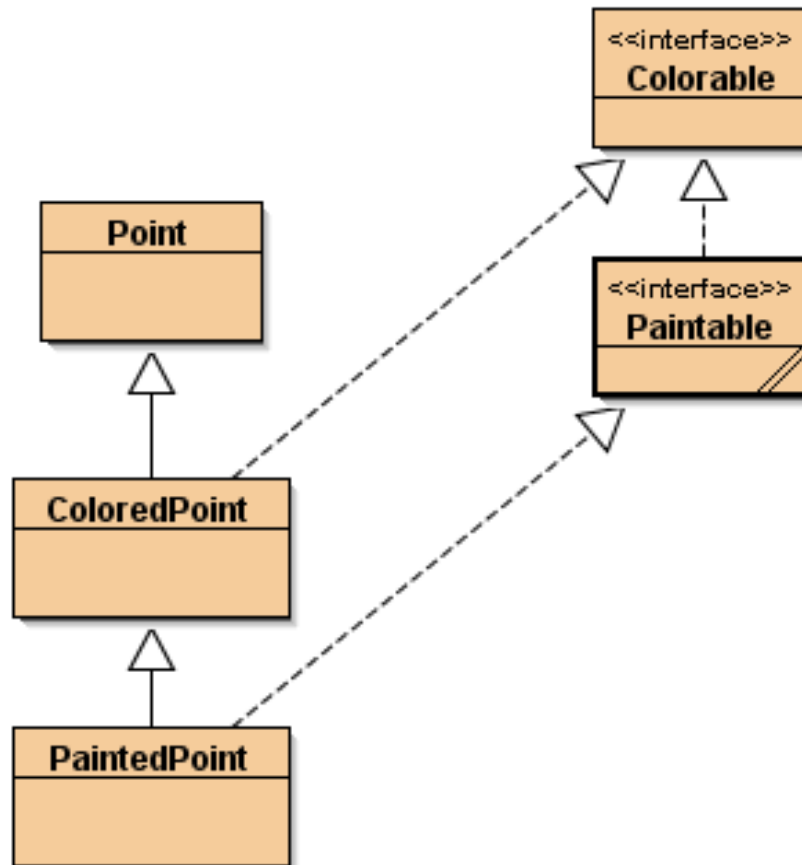
How interfaces are used

- This is difficult because there is no single, simple answer to the question.

- Like any semantic feature in a programming language, there are no hard and fast rules about in what situations it should best be exploited.

- However, there are many guidelines and general strategies (or else the feature would have never bee included).

- We'll cover a few ways in which interfaces are typically used.

Some interface uses

1. To simply clarify the the functionality associated with a particular abstraction.

2. To abstract functionality in a method to make it more general, such as pointers to functions are used in C. sort is a good example.

3. To implement callbacks, such as in GUI programming

4. To write more general, implementation depending code that is easy to extend and maintain.

5. To simulate global constants.

Clarifying functionality

- Often you just want to write a code to do something. It will never do anything else. You are not concerned about extensibility.

- Even in such a case, it can be nice to organize your program using interfaces. It makes the code easy to read and the intent of the author very clear.

Callbacks

- Callbacks are a general programming technique where a method call another method which then calls the calling method (typically to inform the caller when some action has taken place).

- `Timer` and Swing `ActionListeners` are good examples.

To write more general implementation

- A method that operates on a variable of type interface automatically works for any sub-type of that interface.

- This is much more general than writing your program to operate only on a particular subtype.

Abstracting functionality

- A method can often be made more general by customizing its work based on the implementation of some other function that it calls.

- sort(...) is a good example. A sort() function can sort any list of items as long as you tell it how to compare any two items.

- Numerical methods for solving differential equations often depend on taking discrete derivatives: you can make such a routine general by specifying the derivate technique independently.

The Software Life Cycle

- Encompasses all activities from initial analysis until obsolescence
- Formal process for software development
    - Describes phases of the development process
    - Gives guidelines for how to carry out the phases
- Development process
    - Analysis
    - Design
    - Implementation
    - Testing
    - Deployment

Analysis

- Decide what the project is suppose to do
- Do not think about how the program will accomplish tasks
- Output: requirements document
  - Describes what program will do once completed
  - User manual: tells how user will operate program
  - Performance criteria

Design

- Plan how to implement the system
- Discover structures that underlie problem to be solved
- Decide what classes and methods you need
- Output:
  - Description of classes and methods
  - Diagrams showing the relationships among the classes

Implementation. Testing. Deployment

- **Implementation**
  - Write and compile the code
  - Code implements classes and methods discovered in the design phase
  - Output: completed program
- **Testing**
  - Run tests to verify the program works correctly
  - Output: a report of the tests and their results
- **Deployment**
  - Users install program
  - Users use program for its intended purpose

Object-Oriented Design

1. Discover classes
2. Determine responsibilities of each class
3. Describe relationships between the classes

Unified Modeling Language

- **Unified Modeling Language**
  - UML is the international standard notation for object-oriented analysis and design.
  - It is defined by the Object Management Group
  - UML 2.0 defines thirteen types of diagrams, divided into three categories:
    - six diagram types represent static application structure;
    - three represent general types of behavior;
    - and four represent different aspects of interactions
  - **Structure Diagrams** include the *Class Diagram, Object Diagram*, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

Discovering Classes

- A class represents some useful concept
  - Concrete entities: bank accounts, ellipses, and products
  - Abstract concepts: streams and windows
- Find *classes* by looking for *nouns* in the task description
- Define the behavior for each class
- Find *methods* by looking for *verbs* in the task description

Relationships between entities
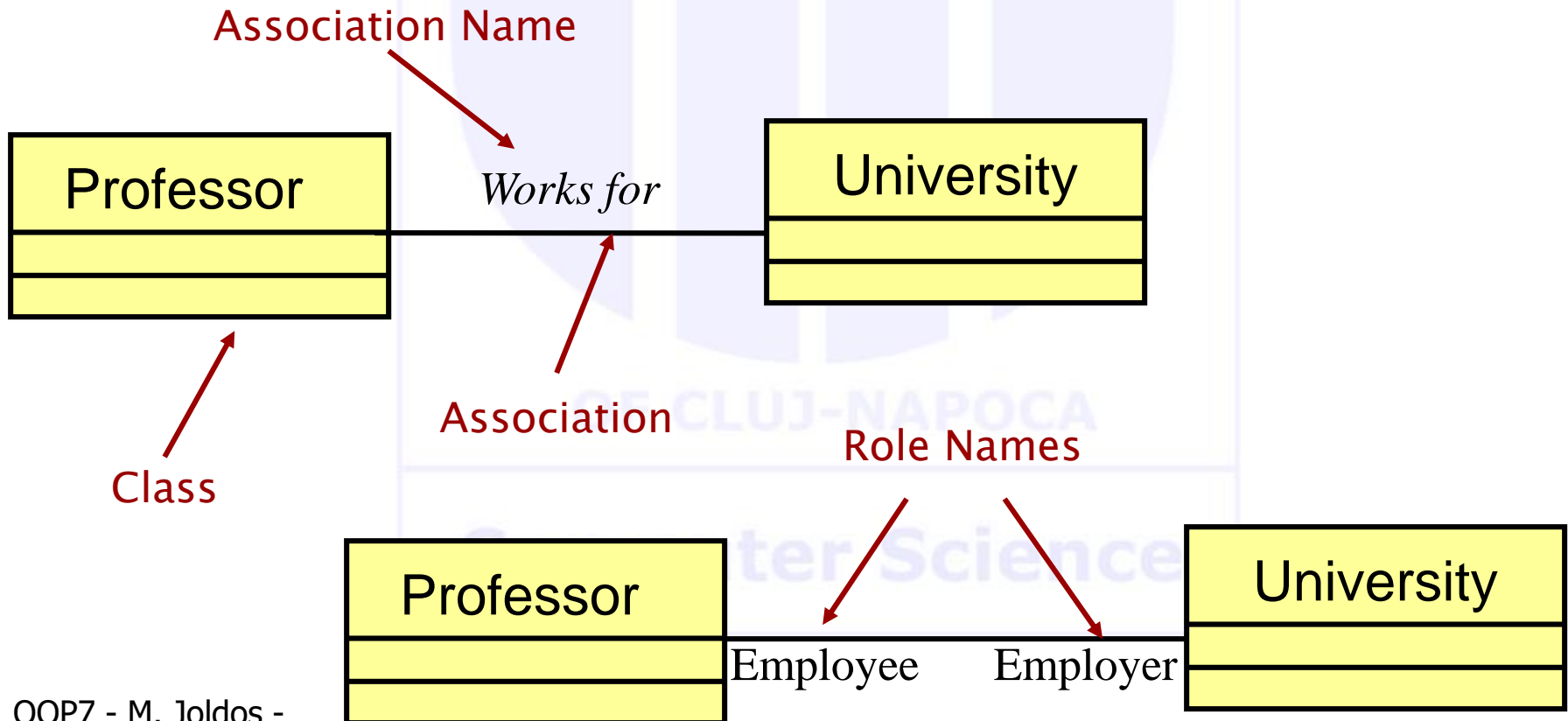
- ## Association
  - ### Aggregation
  - ### Composition
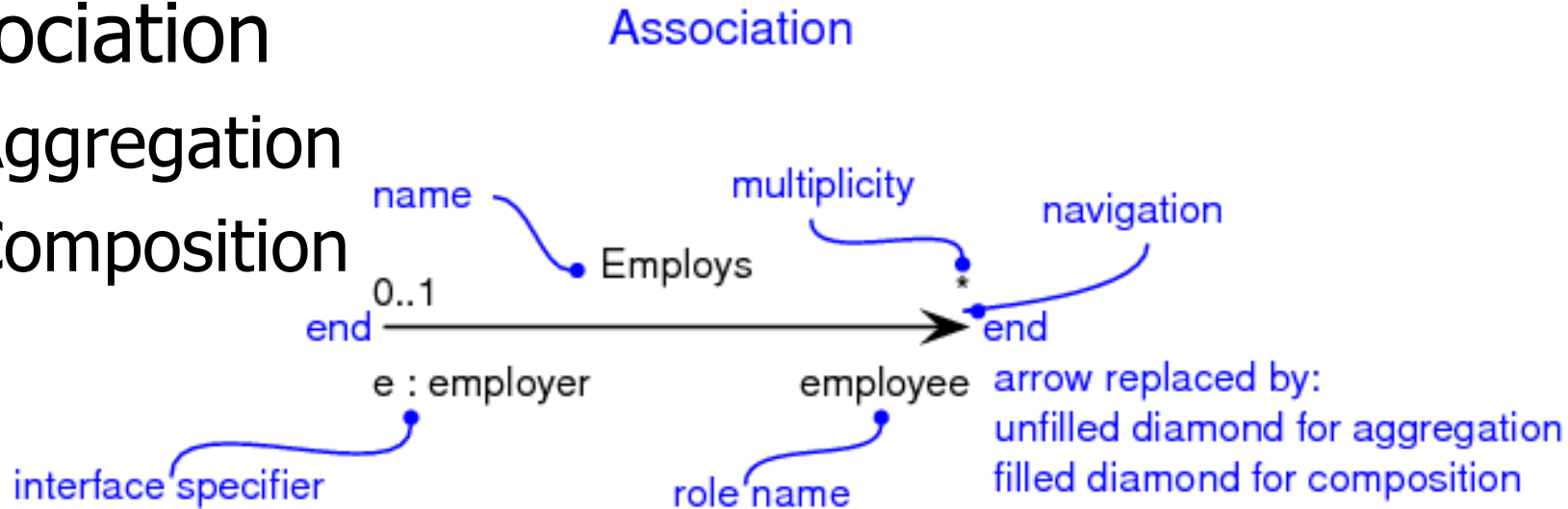
Association

- ## Dependency
- ## Generalization
- ## Realization

Relationships: Association

- Models a semantic connection among classes

Association Name

Professor —— *Works for* —— University

Class

Association

Role Names

Professor —— Employee —— Employer —— University

Use of Associations

- Three general purposes:
  - To represent a situation in which an object of one class *uses* the services of an object of another object, or they mutually use each others services - i.e. one object sends messages to the other, or they send messages back and forth. (In the former case, the navigability can be monodirectional; in the latter case it must be bidirectional.)
  - To represent aggregation or composition - where objects of one class are wholes that are composed of objects of the other class as parts. In this case, a uses relationship is implicitly present - the whole makes use of its parts to do its job, and the parts may also need to make use of the whole.
  - To represent a situation in which objects are related, even though they don't exchange messages. This typically happens when at least one of the objects is basically used to store information.

Relationships: Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts

Whole

Part

Student

Schedule
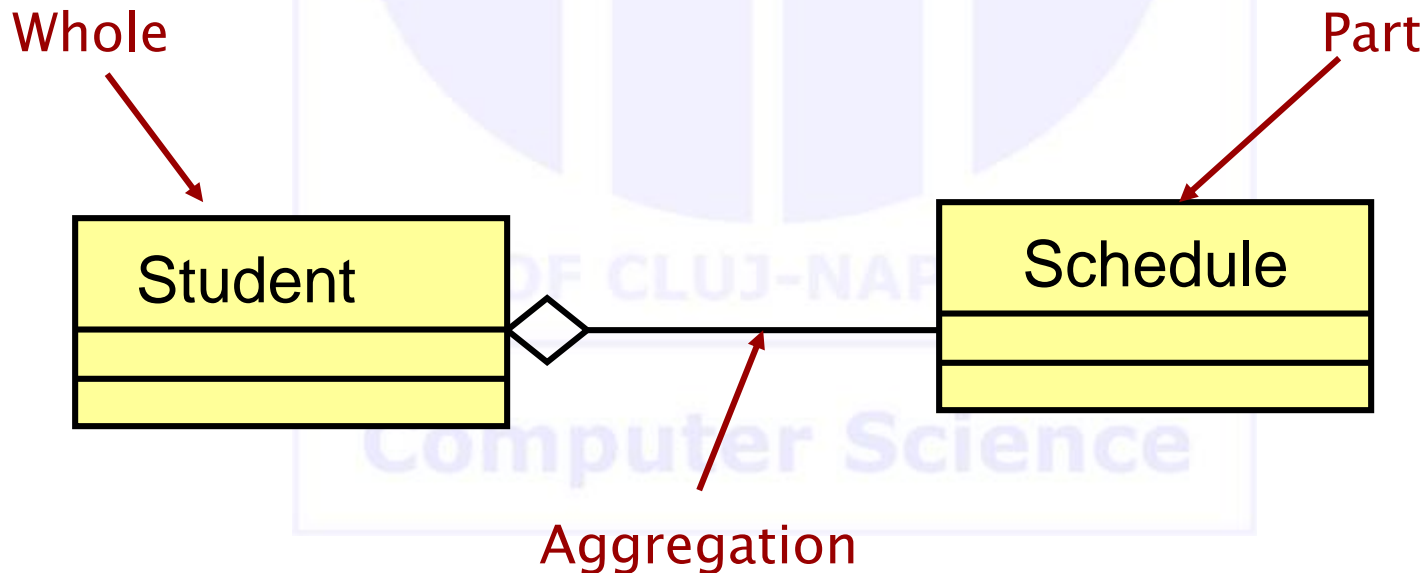
Aggregation

Relationships: Composition

- A form of aggregation with *strong* ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate

Whole

Part

Student

Grades

Aggregation
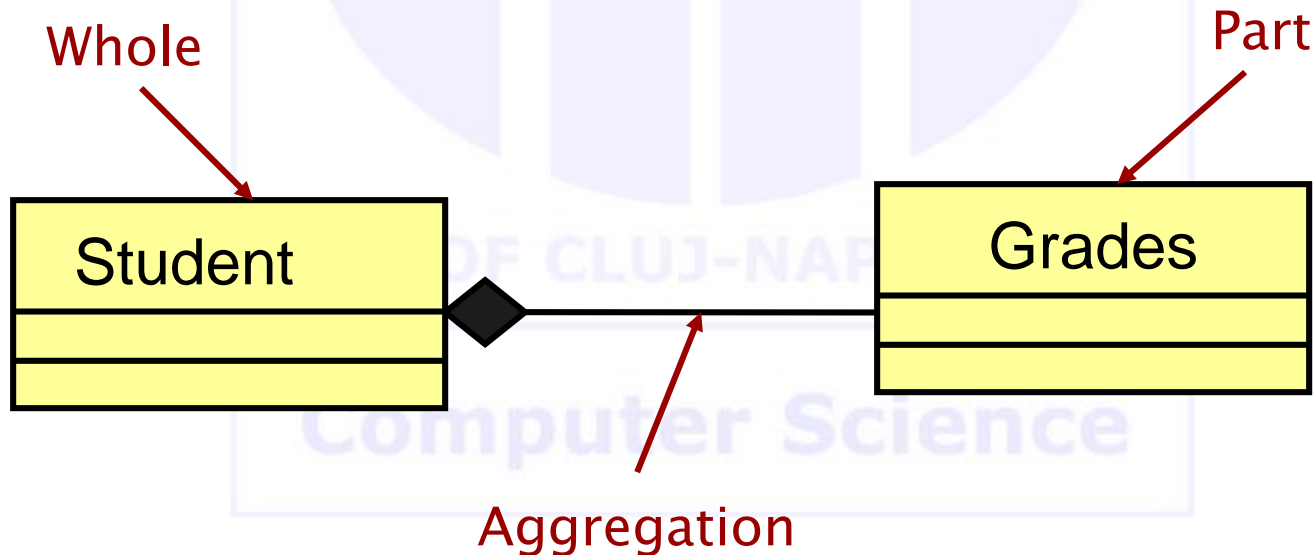
Association: Multiplicity and Navigation

- **Multiplicity defines how many objects participate in a relationship**
  - The number of instances of one class related to ONE instance of the other class
  - Specified for each end of the association
- **Associations and aggregations are bi-directional by default, but it is often desirable to restrict navigation to one direction**
  - If navigation is restricted, an arrowhead is added to indicate the direction of the navigation
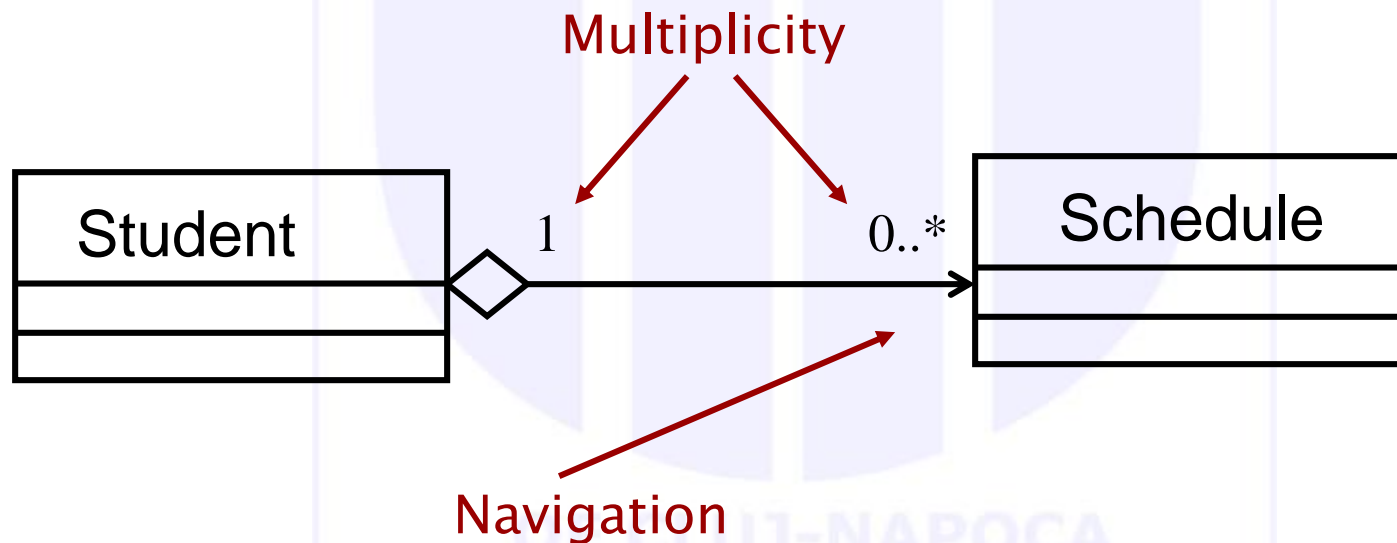
Association: Multiplicity

- Unspecified ———————

- Exactly one ———————
  1

- Zero or more (many, unlimited) ———————
  0..*   *

- One or more ———————
  1..*

- Zero or one ———————
  0..1

- Specified range ———————
  2..4

- Multiple, disjoint ranges ———————
  2, 4..6

Example: Multiplicity and Navigation

Multiplicity

Student | 1 ——————————————> 0..* | Schedule

Navigation

Example of Associations
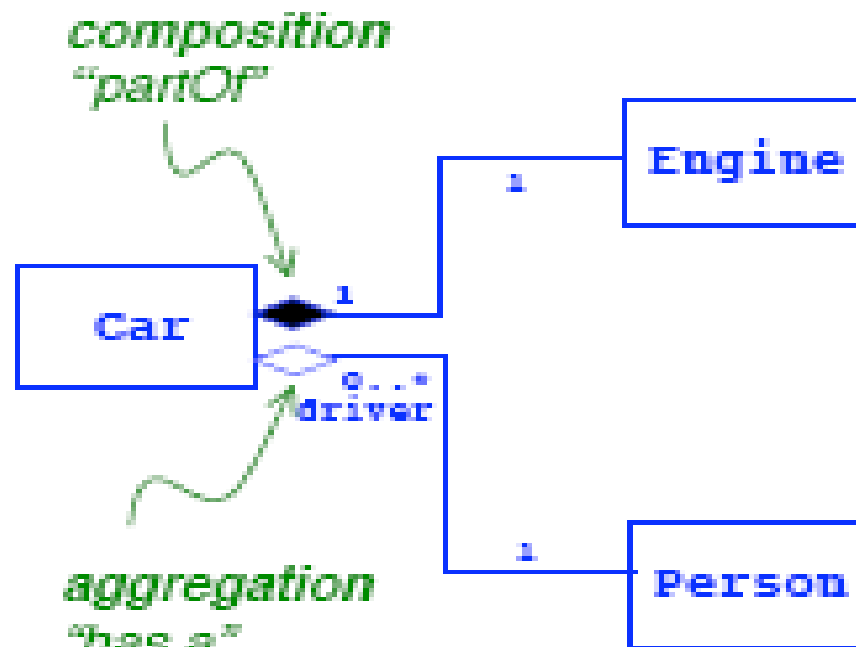
Remarks

- **Tests for "true" part-whole**
  - » **transitive: if "A partOf B" & "B partOf C" then "A partOf C"**
    - "The finger nail is part of the finger is part of the hand is part of the upper extremity is part of the body"
  - **A fault to the part is a fault in the whole**
    - Injury to the fingernail is injury to the body
  - "The tail-light is part of the electrical system is part of the car. A fault in the tail light is a fault in the car"
- **isPartOf *is different* from**
  - **isContainedIn** : shirt, pants,... --- suitcase [note that the fault test does not work here: faulty pants does not mean faulty suitcase]
  - **isConnectedTo** : suitcase --- person (pulling it)
  - **isBranchOf** : iliac artery,... --- aorta
  - **hasLocation** : house... --- street

When to use Aggregation

- **As a general rule, you can mark an association as an aggregation if the following are true:**
    - You can state that
        - the parts 'are part of' the aggregate
        - or the aggregate 'is composed of' the parts
    - When something owns or controls the aggregate, then they also own or control the parts

Aggregation and Composition

Association

Objects are aware about one another so they can work together

Aggregation

- **Protect the integrity of the configuration**

- **Function as a single unit**
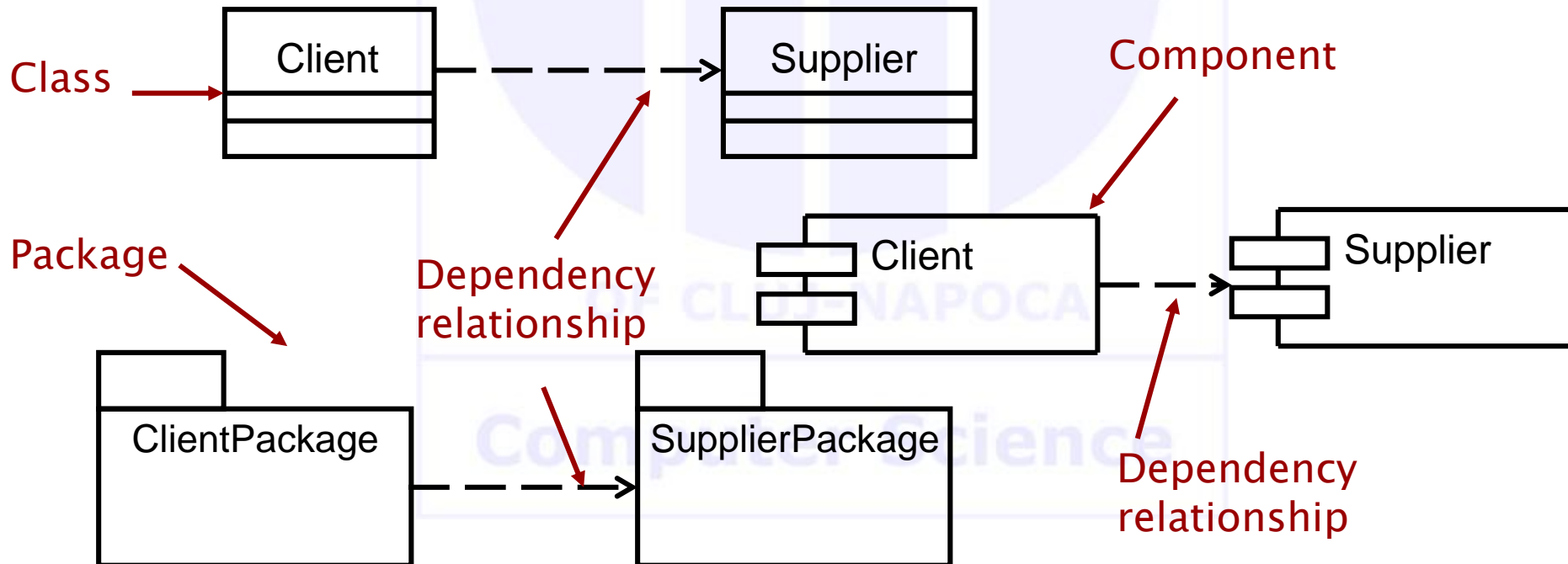
- **Control through one object – propagation downward**

Composition

Each part may only be a member of one aggregate object

Relationships: Dependency

- A relationship between two model elements where a change in one **may** cause a change in the other
- Non-structural, "using" relationship

Class → | Client | ⤍ → | Supplier |

Component

Dependency relationship

Package →

ClientPackage ⤍ → SupplierPackage

Client
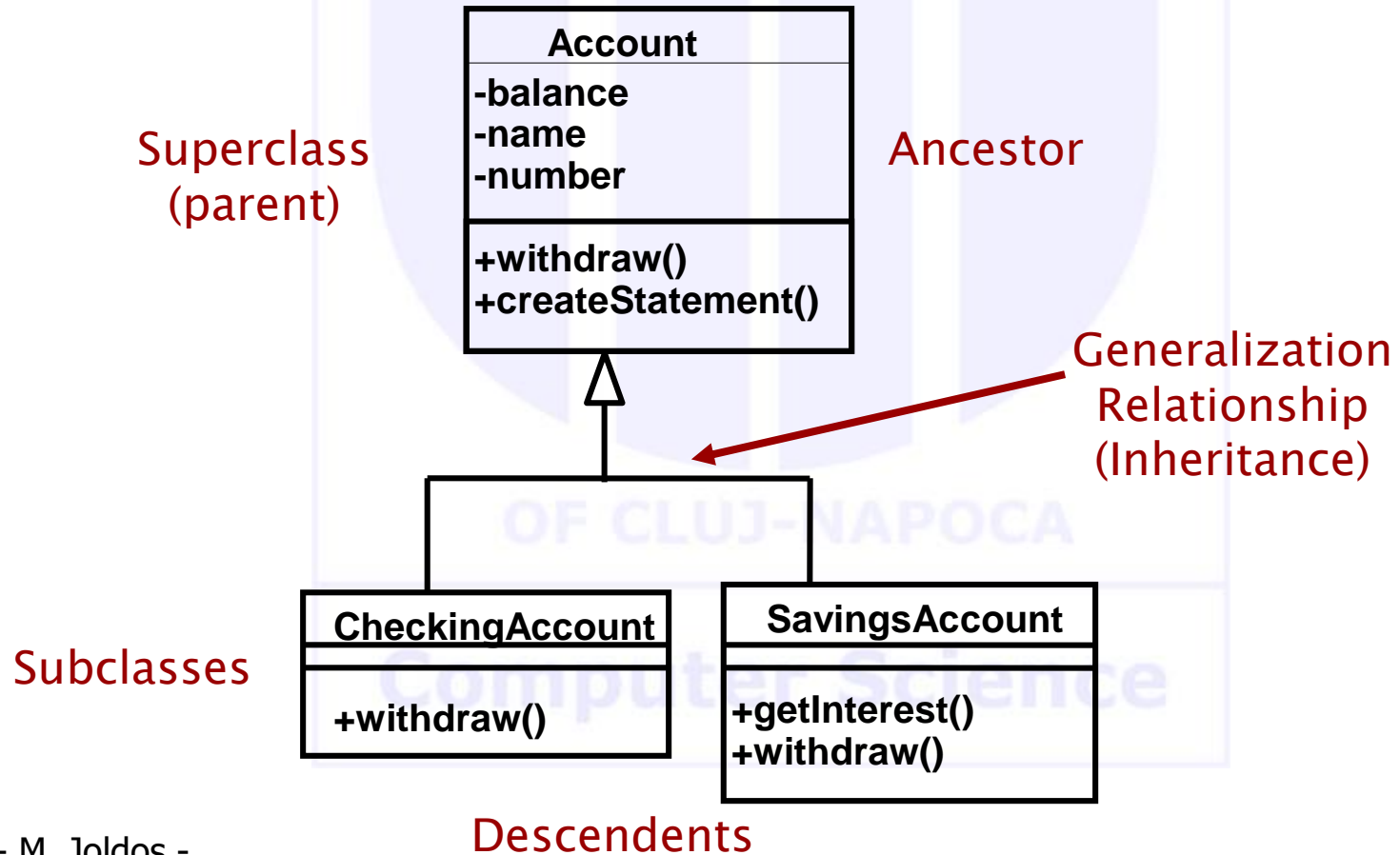
Supplier

Dependency relationship

Relationships: Generalization

- A relationship among classes where one class shares the structure and/or behavior of one or more classes

- Defines a hierarchy of abstractions in which a subclass inherits from one or more superclasses
    - Single inheritance
    - Multiple inheritance

- Generalization is an "is-a-kind of" relationship

Example: Single Inheritance

- **One class inherits from another**

Superclass (parent)

Ancestor

```
         Account
  -balance
  -name
  -number

  +withdraw()
  +createStatement()
```

Generalization Relationship (Inheritance)

Subclasses

```
  CheckingAccount

  +withdraw()
```

```
  SavingsAccount

  +getInterest()
  +withdraw()
```

Descendents

Relationships: Realization

- One classifier serves as the contract that the other classifier agrees to carry out
- Found between:
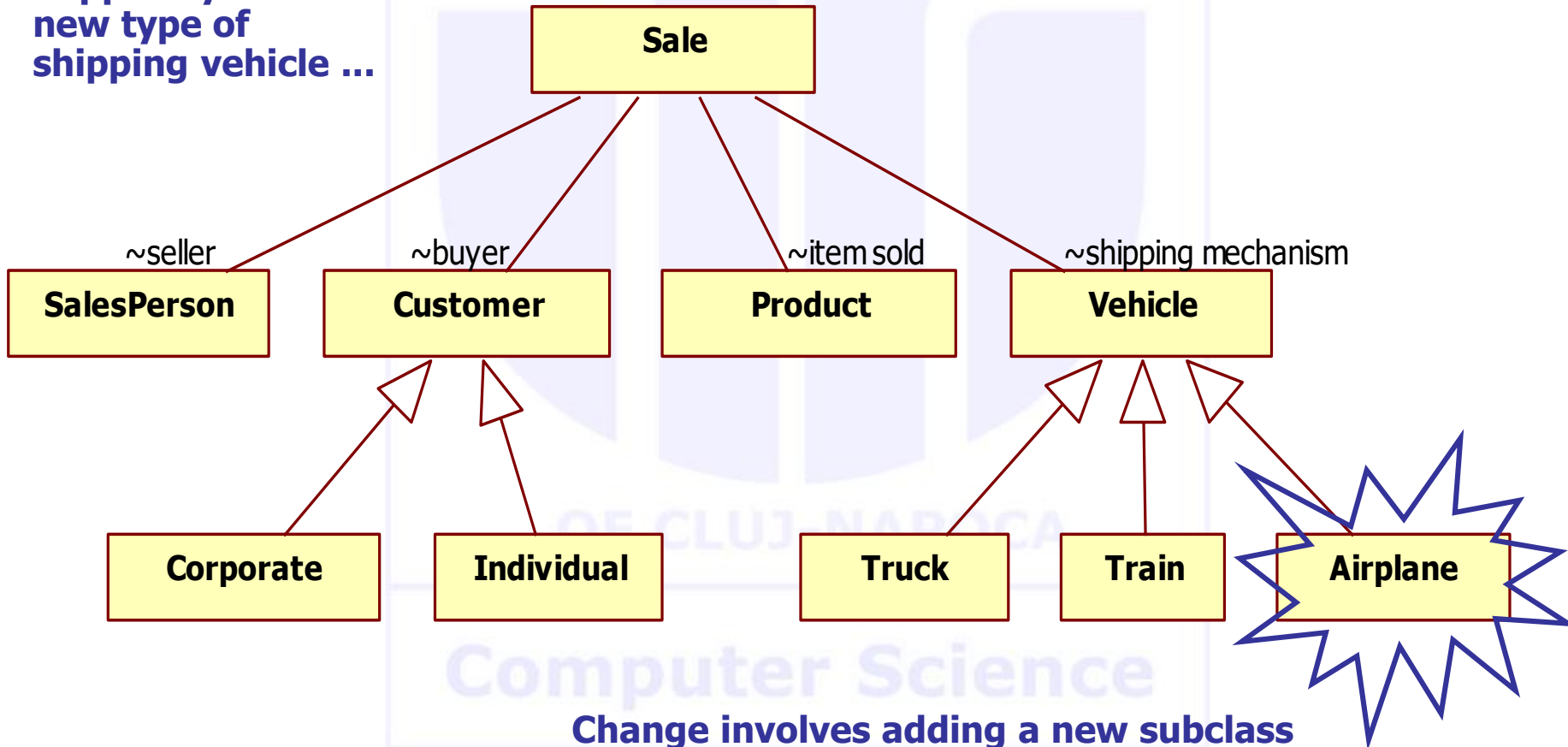 - Interfaces and the classifiers that realize them

Interface Class

Interface Subsystem

Interface Component

Elided form
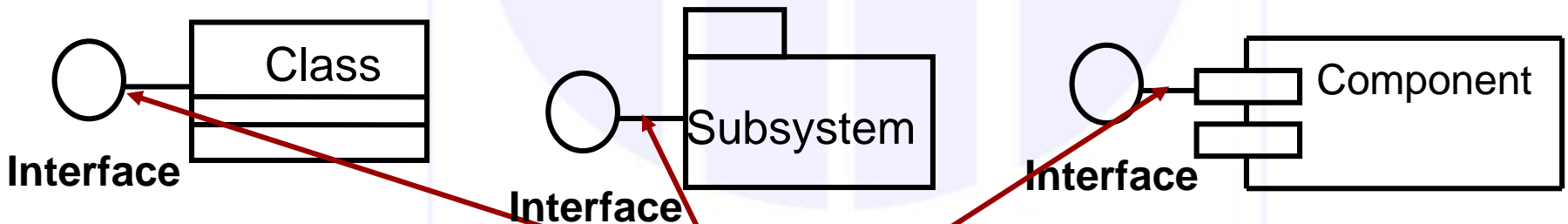
  - Use cases and the collaborations that realize them

Canonical form

Use-Case

Use-Case Realization

Effect of Requirements Change

Suppose you need a new type of shipping vehicle ...

```
                          ┌──────────────┐
                          │     Sale     │
                          └──────────────┘

   ~seller          ~buyer          ~item sold      ~shipping mechanism
┌──────────────┐ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
│ SalesPerson  │ │  Customer    │ │   Product    │ │   Vehicle    │
└──────────────┘ └──────────────┘ └──────────────┘ └──────────────┘

┌──────────────┐ ┌──────────────┐   ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
│  Corporate   │ │  Individual  │   │    Truck     │ │    Train     │ │   Airplane   │
└──────────────┘ └──────────────┘   └──────────────┘ └──────────────┘ └──────────────┘
```

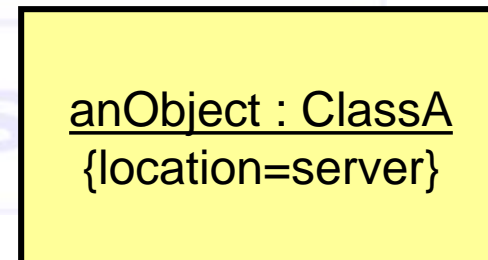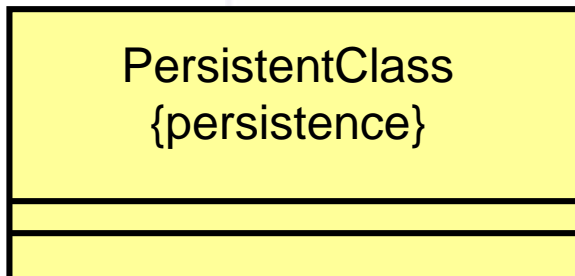**Change involves adding a new subclass**

Notes

- A note can be added to any UML element
- Notes may be added to add more information to the diagram
- It is a 'dog eared' rectangle
- The note may be anchored to an element with a dashed line

MaintainScheduleForm

There can be up to one MaintainScheduleForm per user session.
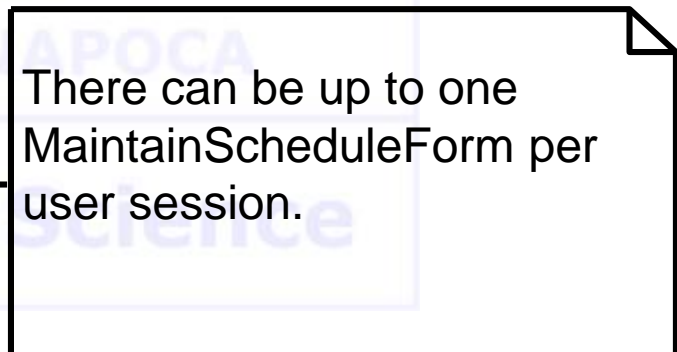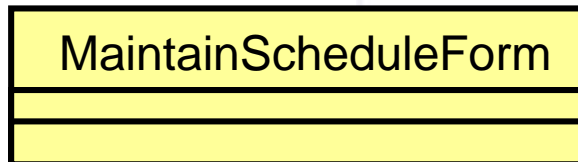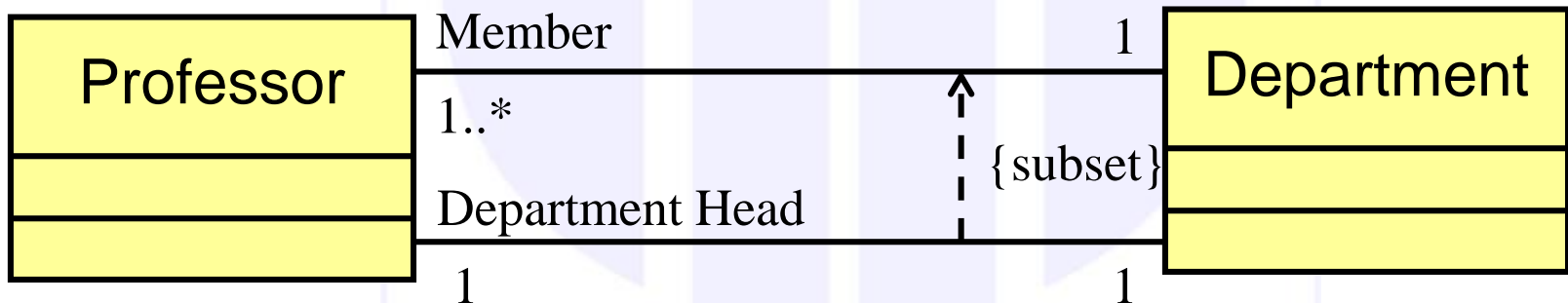
Tagged Values

- **Extensions of the properties, or specific attributes, of an UML element**

- Some properties are defined by UML
    - Persistence
    - Location (e.g., client, server)

- Properties can be <u>created</u> by UML modelers for any purpose

```
┌───────────────────────────┐
│   PersistentClass          │
│   {persistence}            │
├───────────────────────────┤
│                            │
├───────────────────────────┤
│                            │
└───────────────────────────┘
```

```
┌───────────────────────────┐
│                            │
│   anObject : ClassA        │
│   {location=server}        │
│                            │
└───────────────────────────┘
```

Constraints

- ## Supports the addition of new rules or modification of existing rules

Professor	Member 1	Department

```
                    Member                        1
 ┌──────────────┐                                    ┌──────────────┐
 │              ├─ 1..*                           ────┤              │
 │  Professor   │                        ↑            │  Department  │
 │              │   Department Head       ┆ {subset}  │              │
 └──────────────┘─ 1                     ┆ 1          └──────────────┘
```

- This notation is used to capture two relationships between Professor-type objects and Department-type objects; where one relationship is a subset of another….
- Shows how UML can be tailored to correctly modeling exact relationships….

CRC Card

- CRC Card
- Describes a class, its responsibilities, and its collaborators
- Use an index card for each class
- Pick the class that should be responsible for each method (verb)
- Write the responsibility onto the class card
- Indicate what other classes are needed to fulfill responsibility (collaborators)

CRC Card. Class Relationships

- ## CRC card

Class Name	
Responsibilities	Collaborators

- ## Relationships Between Classes
  - Inheritance
  - Aggregation
  - Dependency

Inheritance

- *Is-a* relationship

- Relationship between a more general class (superclass) and
  a more specialized class (subclass)

- Examples:
  - Every savings account is a bank account
  - Every circle is an ellipse (with equal width and height)

- It is sometimes abused
  - Should the class `Tire` be a subclass of a class `Circle`?
  - The *has-a* relationship would be more appropriate

Aggregation

- *Has-a* relationship
- Objects of one class contain references to objects of another class
- Use an instance variable
  - A tire has a circle as its boundary:

    ```
    class Tire {

        . . .
        private String rating;
        private Circle boundary;
    }
    ```

- Every car has a tire (in fact, it has four)

    ```
    class Car extends Vehicle{

        . . .
        private Tire[] tires;
    }
    ```

Dependency

- *Uses* relationship
- Example: many of our applications depend on the Scanner class to read input
- Aggregation is a stronger form of dependency
- Use aggregation to remember another object between method calls

UML Relationship Symbols

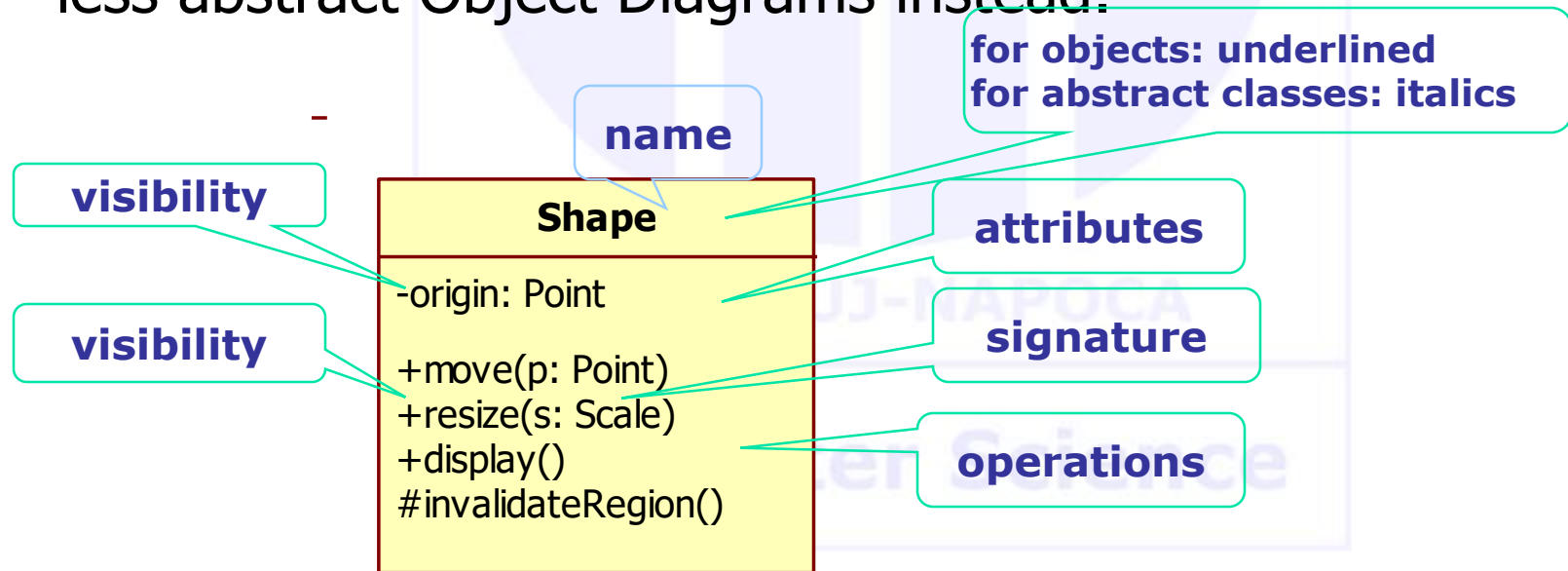| Relationship	Symbol	Line Style	Arrow Tip
Inheritance	———————▷	Solid	Triangle
Interface Implementation	- - - - - -▷	Dashed	Triangle
Aggregation	◇———————	Solid	Diamond
Dependency	- - - - - -→	Dashed	Open

Aggregation and Association

- Association: more general relationship between classes
- Use *early* in the design phase
- A class is associated with another if you can navigate from objects of one class to objects of the other
- Given a `Bank` object, you can navigate to `Customer` objects

Bank	serves →	Customer

Class Diagram

- A set of classes, interfaces, collaborations, and relationships

- Reflects the *static design* of a system.

- Can be confusing if used to explain system dynamics; use less abstract Object Diagrams instead.

for objects: underlined
for abstract classes: italics

name

visibility

attributes

Shape

-origin: Point

visibility

signature

+move(p: Point)
+resize(s: Scale)
+display()
#invalidateRegion()

operations

Object Diagrams

- A set of objects (instances of classes) and their relationships.

- A static snapshot of a dynamic view of the system.

- Represents real or prototypical cases.

- Very useful before developing class diagrams.

- Worth saving as elaborations of class diagrams.

Object Diagrams

- Useful for explaining small pieces with complicated relationships, especially *recursive relationships*.

- Each rectangle in the object diagram corresponds to a single instance.

- Instance names (object names) are <u>underlined</u> in UML diagrams.

Department
-degree:String[]={"graduate","undergraduate","both"}

0..*

subdepartment

1
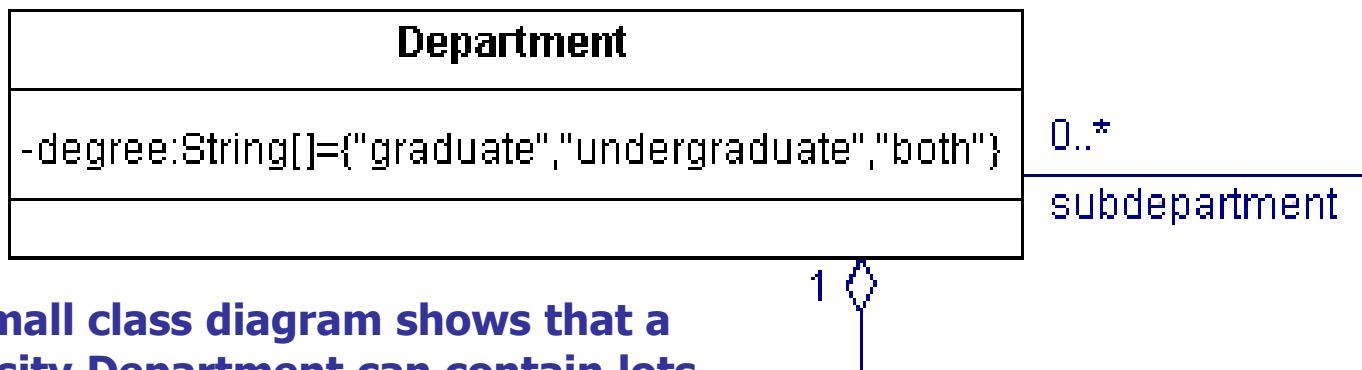
This small class diagram shows that a university Department can contain lots of other Departments.
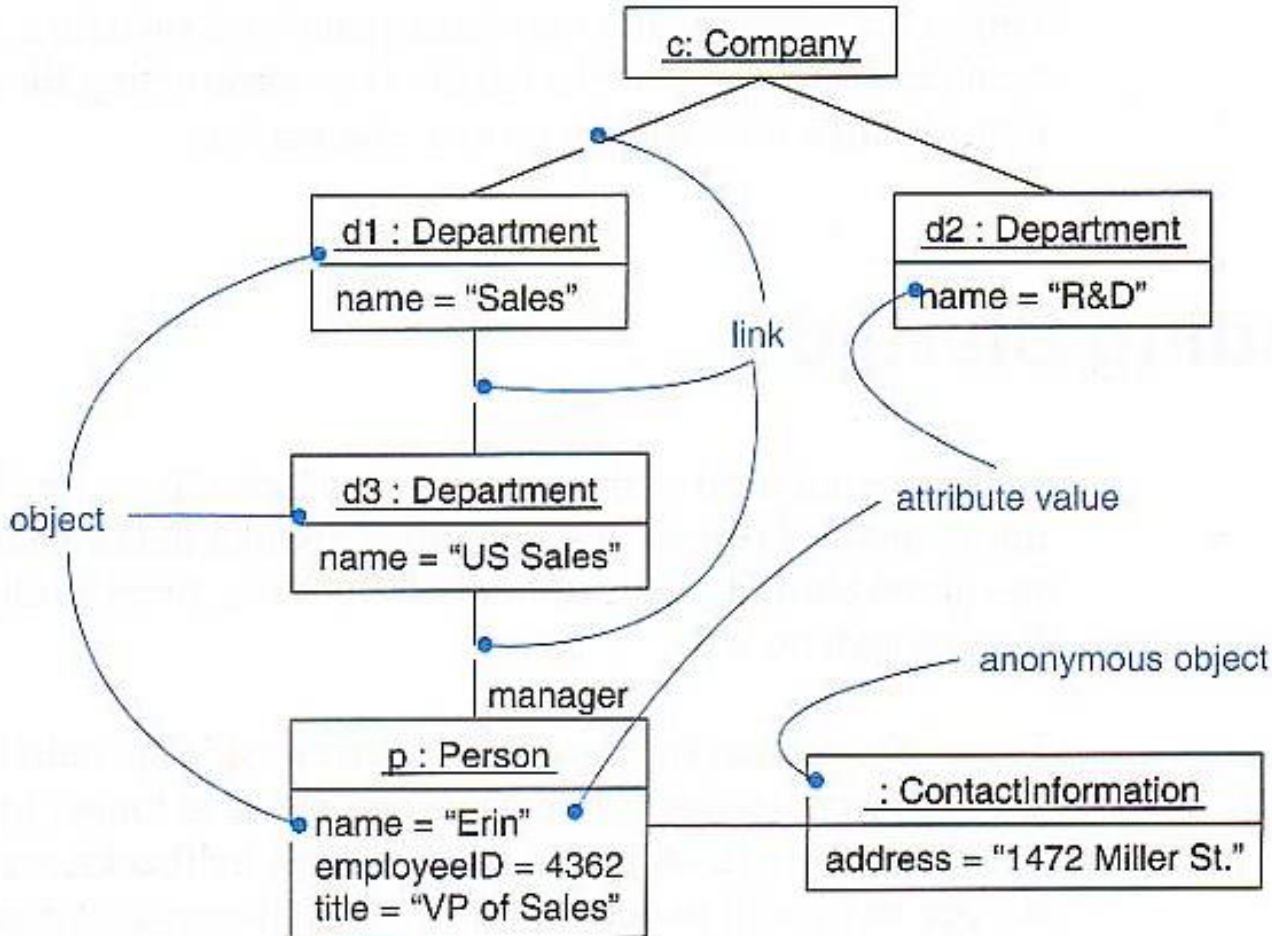
Object Diagrams

- The object diagram below instantiates the class diagram, replacing it by a concrete example
- Class or instance names may be omitted from object diagrams as long as the diagram meaning is still clear.
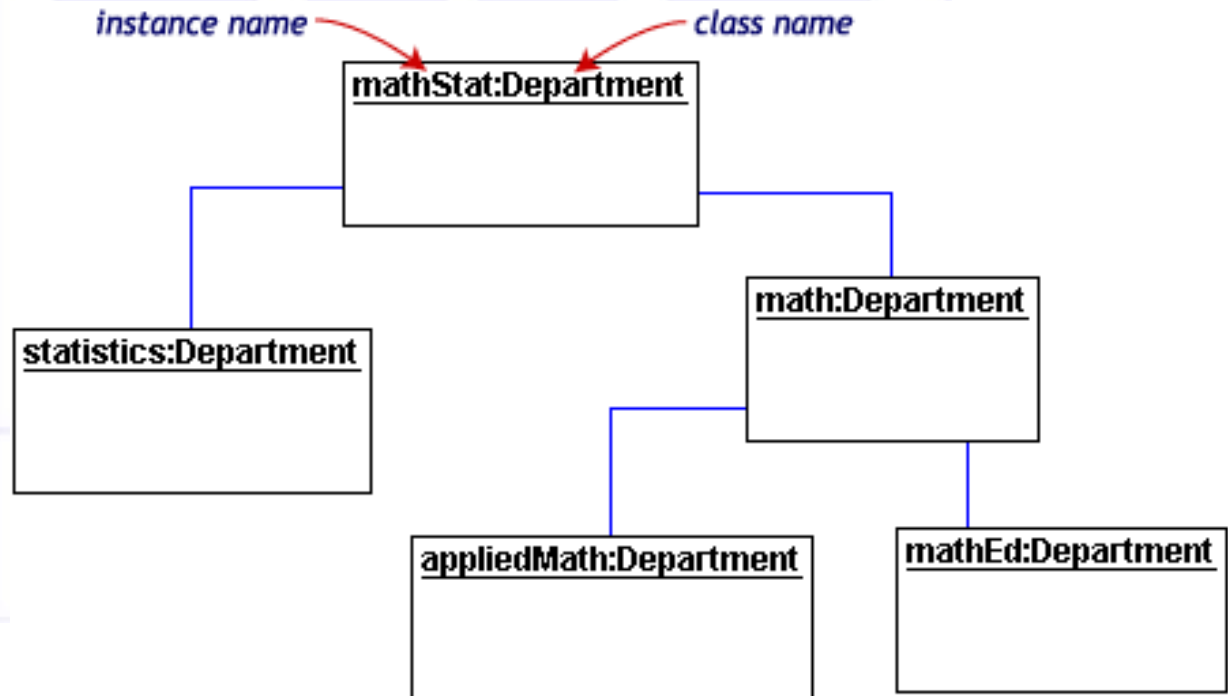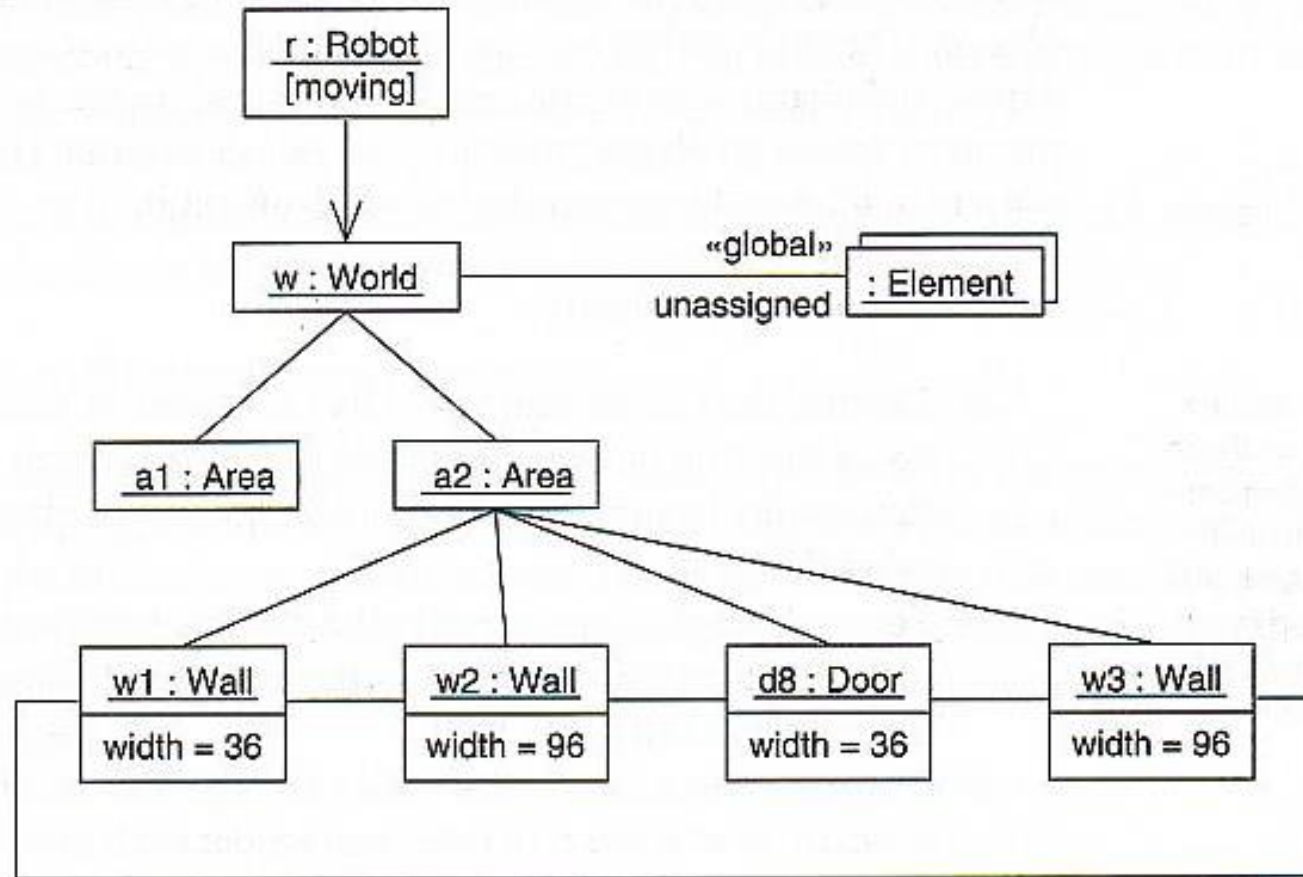
Object Diagram Example

Some Useful UML Links

- OMG UML Resource Pages
  - http://www.uml.org
- Dzone Java Zone UML2 diagrams
  - https://dzone.com/articles/uml2-class-diagram-java
- Practical UML: A Hands on introduction for developers
  - http://edn.embarcadero.com/article/31863
- Robert Martin: UML Class Diagrams for Java Programmers
  - http://www.informit.com/articles/article.aspx?p=336264&seqNum=3
- Creately: The Complete Guide to UML Diagram Types with Examples
  - http://creately.com/blog/diagrams/uml-diagram-types-examples
- Holub Associates: UML Reference Card
  - http://www.holub.com/goodies/uml
- TutorialsPoint: UML
  - https://www.tutorialspoint.com/uml/index.htm

Reading

- Eckel: chapter 10
- Barnes: chapter 5, 10.6, 10.7
- Deitel: 10.7, chapters 12, 13

Summary

- Java interface
  - definition
  - declaration
  - usefulness
  - restrictions
  - examples
- The class `Object`
  - everything inherits from it
  - methods to override
- The class `Class`
  - useful methods
- The `instanceof` operator vs `getClass()` method

- Software life cycle
- UML diagrams
  - Relationships between entities
  - Class diagrams
  - Object Diagrams