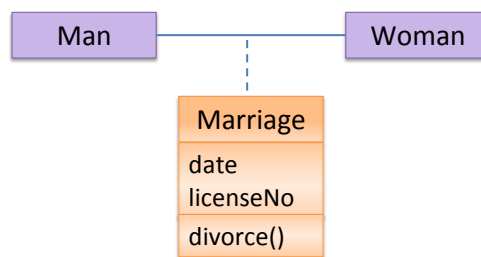


[Handout for L7]**Still doing detailed design, while getting more out of OO****Advanced OO concepts**

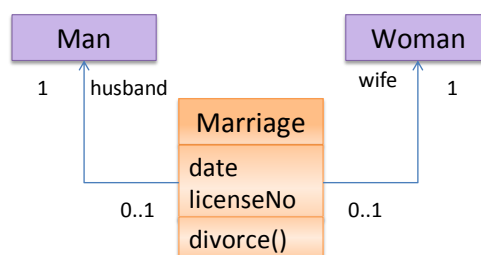
Please read pages 21-37 (compulsory reading of the document *Object-Oriented Programming with Objective-C* released by Apple Inc.

Association classes

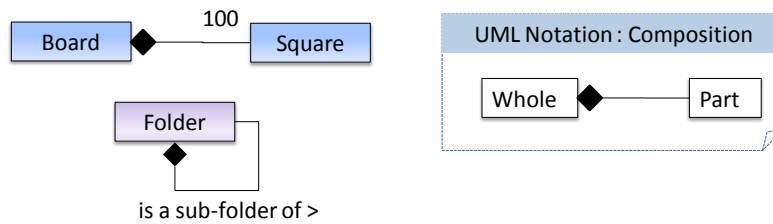
At times, we want to store additional information about an association. For example, if a Man class and a Woman class linked with a 'married to' association, we might also want to store the date of the marriage. However, that data is related to the association and not specifically owned by either the Man object or the Woman object. In such situations, we can introduce an additional class e.g. a class called Marriage, to store such information. We call those classes 'association classes' and denote them in a special way (i.e. connected to the association link with a dashed line), as shown below.



Note that an association class implies there can be only one 'association object' between any given two objects to which the association object is connected. i.e. there can be only one Marriage object between a given Man object and a Woman object. Note that while we use a special notation to indicate an association class, there is no special way to implement an association class. At implementation level, we are most likely to implement the above association class as follows.

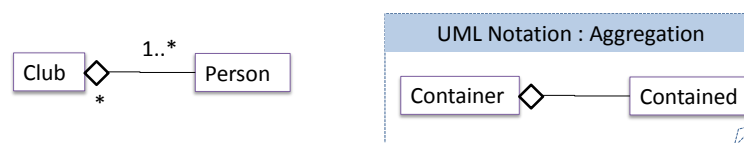
**Composition and aggregation**

A composition is an association that represents a strong "whole-part" relationship. When the "whole" is destroyed, "parts" are destroyed too. UML uses a solid diamond symbol to denote *composition*.



In addition, composition also implies that there cannot be cyclical links. In the example above, the notation represents a 'sub-folder' relationship between two folders while implying that a Folder cannot be a sub-folder of itself. If we remove the diamond, it is no longer a composition relationship and technically, allows a folder to be subfolder of itself.

Aggregation represents a 'part of' relationship. It is a weaker relationship than composition. UML provides a notation, a hollow diamond to indicate aggregation.

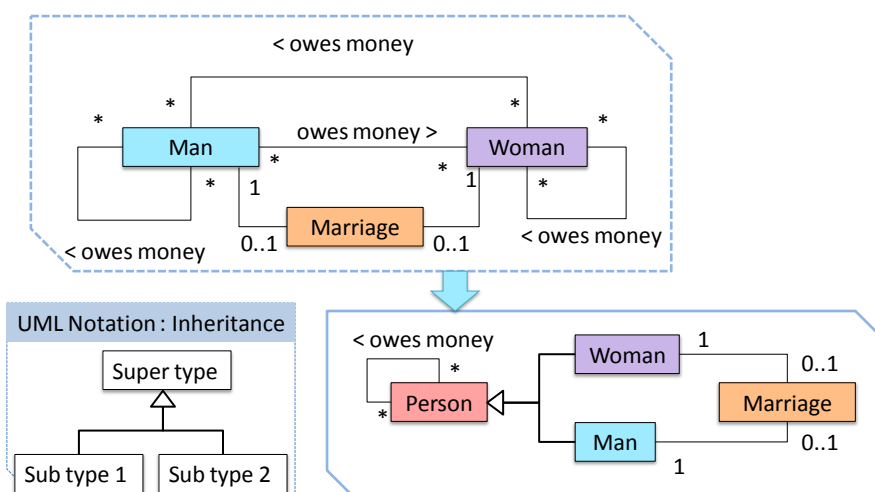


The line between composition and aggregation is rather blurred. Some practitioners (e.g., Martin Fowler, in his famous book *UML Distilled*) advocate not using the aggregation symbol altogether as using it adds more confusion than clarity.

Inheritance

Sometimes, it helps to visualize a more general object type that acts as a "super type". We can use the object-oriented concept of *inheritance* to achieve that.

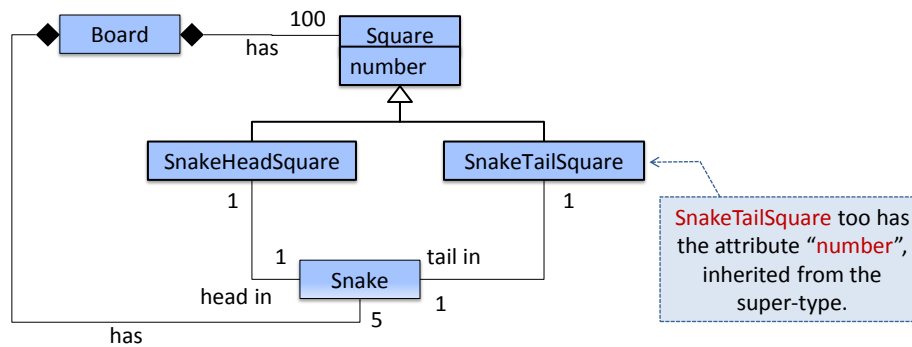
In the example below, Man and Woman behaves the same way for 'owes money' association. However, we cannot replace the two classes with a more general class Person because we still need to distinguish between Man and Woman for the 'marriage' association. Therefore, we add the Person class as a super class and let Man and Woman inherits from Person.



All attributes of the super-type are assumed to be 'inherited' by sub-types. Inheritance implies an 'is a' relationship; for example, in the class diagrams of a *Snakes and Ladders* board game given below:

- SnakeHeadSquare *is a* Square.

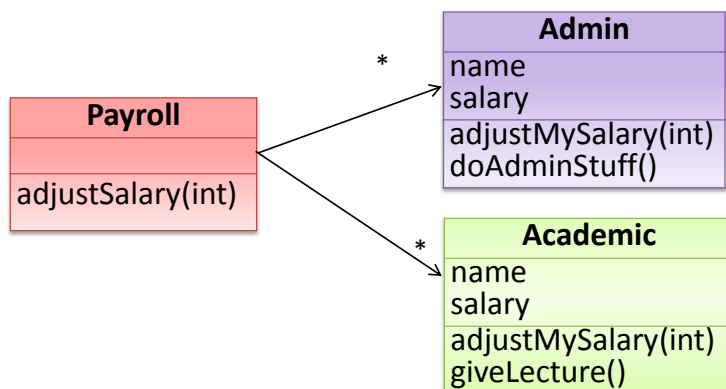
- SnakeTailSquare is a Square.



We assume you already know how to implement inheritance in your preferred programming language.

Polymorphism

Polymorphism is an important and useful concept in the object-oriented paradigm. Let us learn polymorphism using an example. Imagine you are writing a payroll application for a university. It helps in payroll processing of university staff. At this point you are trying to write the `adjustSalary(int)` operation that adjusts the salaries of all staff members. This operation will be executed whenever the university does a salary adjustment for its staff. Note that the adjustment formula is different for different staff categories. At the moment you have two categories of staff: admin and academic. Here is one possible way of designing the classes in the Payroll system.

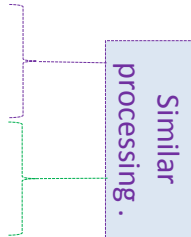


Here is the implementation of the `adjustSalary(int)` operation from the above design.

```

class Payroll1{
    ArrayList<Admin> admins;
    ArrayList<Academic> academics;
    //...
    void adjustSalary(int byPercent){
        for(Admin ad: admins){
            ad.adjustMySalary(byPercent);
        }
        for(Academic ac: academics){
            ac.adjustMySalary(byPercent);
        }
    }
}

```

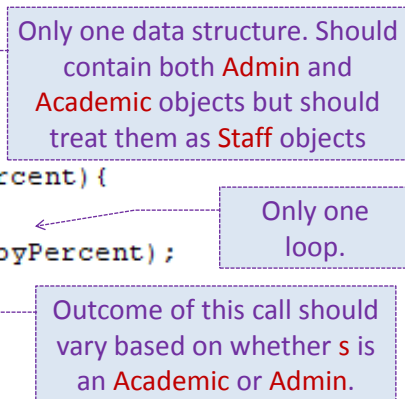


Note how processing is similar for the two staff types. It is as if the type of staff members is irrelevant to how we process them inside this operation! If that is the case, can we abstract away the staff type from this operation? After all, why keep irrelevant details? Here is such an implementation of adjustSalary(int):

```

class Payroll2{
    ArrayList<Staff> staff;
    //...
    void adustSalary(int byPercent){
        for(Staff s: staff){
            s.adjustMySalary(byPercent);
        }
    }
}

```

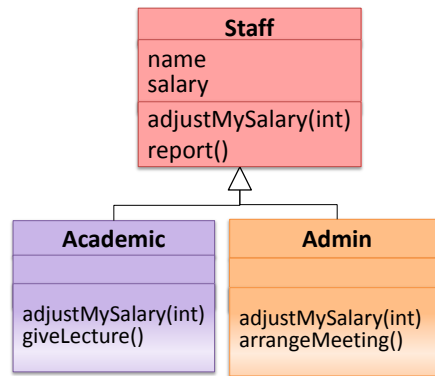


The above code is better in several ways:

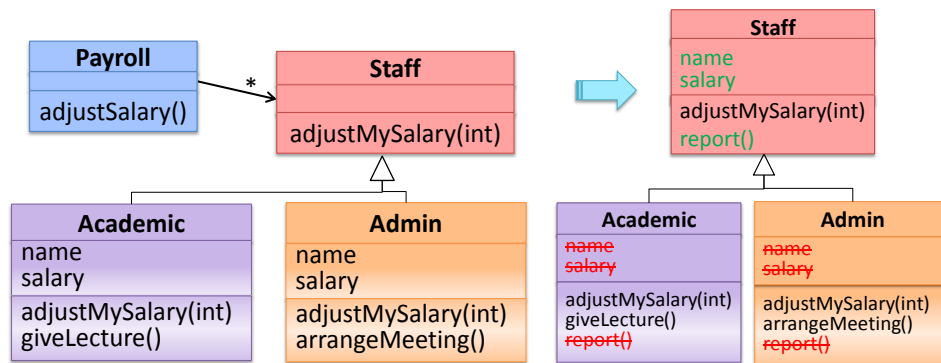
- It is shorter.
- It simpler.
- It is more flexible (this code will remain the same even if we add more staff types).

This does not mean that we have got rid of the Academic and Admin classes completely and replaced them with a more general class called Staff. Rather, this part of the code “treats” both Admin and Academic objects as one type called Staff. For example, the ArrayList staff contains both Admin and Academic objects although we treat it as an ArrayList of Staff objects. However, when we call the adjustMySalary(int) operation of these objects, the resulting salary adjustment will be different for Admin objects and Academic objects. Therefore, what we have here is an ability to treat different types of objects as a single general type and yet get different kinds of behavior from those objects. This is called *polymorphism* (literally, it means “ability to take many forms”). In this example, an object we see as of type Staff can be an Admin object or an Academic object.

How do we keep the multiple types while allowing some parts of the system to treat them as one type? We can use *inheritance* or *interfaces* to achieve such an effect. First, let us learn how to achieve polymorphism using inheritance. Here is a possible design that we can use here.



[Side-Note] Reuse benefits of inheritance



As we mentioned previously, inheritance is a “is a” relationship between two classes. The subclass inherits all public and protected members (i.e. operations and variables) of the superclass. Often, but not always, inheritance reduces the need for code duplication. i.e. it gives us a way to reuse code. For example, we can put the attributes and operations common to Admin and Academic (i.e. name, salary, report()) in the Staff class without repeating them twice in Admin and Academic class.

Given below is the minimum code for Staff, Admin, and Academic classes.

```

class Staff {
    String name;
    int salary;

    public void adjustMySalary(int byPercent) {
        //do nothing
    }
} //end class

class Academic extends Staff {
    public void adjustMySalary(int byPercent) {
        byPercent += 20;
        salary = salary + (salary * byPercent / 100);
    }
} //end class

class Admin extends Staff {
    public void adjustMySalary(int byPercent) {
        if (byPercent < 20) byPercent += 20;
        salary = salary + (salary * byPercent / 100);
    }
} //end class

```

Java syntax for defining a sub-class

Next, let us take a look at three things that are at the center of how we achieved polymorphism in the above example: substitutability, operation overriding, and dynamic binding

Substitutability

We know that every instance of a subclass is an instance of the superclass, but not vice-versa. For example, an Academic is an instance of a Staff, but a Staff is not (necessarily) an instance of an Academic.

When implemented correctly, inheritance allows *substitutability*. i.e. wherever an object of the superclass is expected, we can substitute an object of any of its subclasses. For example, notice how the ArrayList staff (in Payroll2 class) is declared to contain Staff objects but actually contains Admin and Academic objects. That is because we can substitute objects of the superclass Staff (which the ArrayList expects) with objects of subclasses Admin and Academic.

Operation overriding

A subclass is a *specialization* of its superclass.

- A subclass may add members not present in the superclass.
- A subclass may redefine members of its superclass.

To get polymorphic behavior from an operation, we should have the operation in the superclass and redefine it in each of the subclasses. This is called *overriding*. i.e. Admin and Academic override the adjustMySalary(int) of Staff.

Note that unless we have the adjustMySalary(int) operation in the Staff class, Payroll cannot use that operation. That is because adjustSalary(int) of Payroll is treating all objects in the ArrayList as Staff objects. Therefore, it can use only those operations available in the Staff class.

How does overriding differ from *overloading* that you learned in other programming modules? The difference is related to the type signature of operations. The *type signature* of an operation is the type sequence of the parameters. The return type and parameter names are not part of the type signature. However, the parameter order is significant. Here are some examples:

Method	Type Signature
int Add(int X, int Y)	(int, int)
void Add(int A, int B)	(int, int)
void m(int X, double Y)	(int, double)
void m(double X, int X)	(double, int)

Operation overloading is when you have multiple operations with the same name but different type signatures. Given below are two examples.

```
class Account {
    Account () { // Signature: ( )
    ...
    }
    Account (String name, String number, double balance) {
        // Signature: (String,String,double) ...
    }
    ...
    void calculateCAP (String matrix) { ... }
    void calculateCAP (int[] averages) { ... }
}
```

The constructor has been overloaded

We use overloading to indicate that multiple operations do similar things but take different parameters. An operation can be overloaded inside the same class or in sub/super classes. Overloading is resolved at compile time: i.e. the compiler uses operation name and type signature to determine which operation to invoke.

a=new Account() //invokes the 1st constructor

b=new Account("a","b",2.0) //invokes the 2nd constructor

In contrast, overriding is when a sub-class redefines an operation using the same method name and the same type signature. E.g. adjustMySalary(int) of Admin is overriding the adjustMySalary(int) operation of Staff.

Dynamic binding

Overridden operations are resolved at runtime. That is, the runtime decides which version of the operation should be executed based on the actual type of the receiving object. This is also called *dynamic binding*. Most OO languages support dynamic binding.

Consider the example code given below. The declared type of s is Staff and it appears as if the adjustMySalary(int) operation of the Staff class is invoked. However, at runtime the adjustMySalary(int) operation of the actual object will be called (i.e. adjustMySalary(int) operation of Admin or Academic). If the actual object does not override that operation, the operation defined in the immediate superclass (in this case, Staff class) will be called.

```
void adjustSalary(int byPercent){
    for(Staff s: staff){
        s.adjustMySalary(byPercent);
    }
}
```

Abstract classes/operations

Since adjustMySalary(int) does not need a full implementation in the Staff class , we can put only its operation header in the Staff class without specifying its body. Such a 'declaration' without

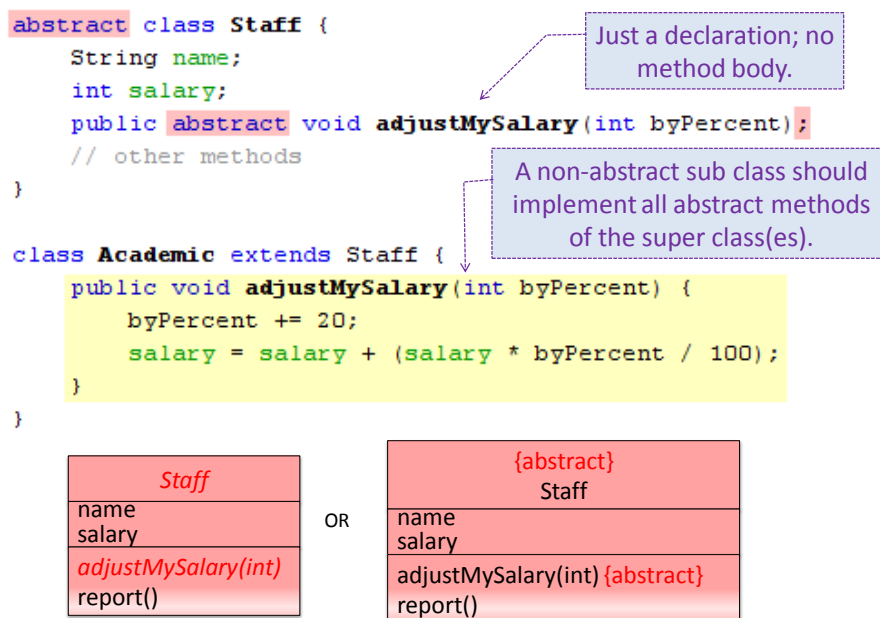
the operation body is called an *abstract* operation. In this context, abstract means ‘not fully specified’. A class that has at least one abstract operation becomes an abstract class itself. i.e. we cannot create instances of it. This is logical because it does not make sense to create Staff objects which are neither Admin objects nor Academic objects. In any case, we cannot create instances of the (now abstract) Staff class because the definition of the class is now incomplete. For example, line 1 below is not allowed but line 2 is allowed.

```
Staff s1 = new Staff(); // line 1
```

```
Staff s2 = new Admin(); //line 2
```

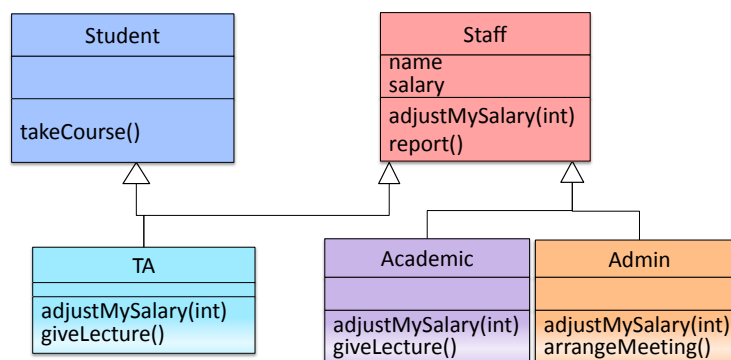
Furthermore, we can declare a class abstract (even if it does not have any abstract operations) to prevent it from being instantiated. A sub class should implement all abstract methods of all its super classes or should itself be declared abstract.

In UML, we use italics or the ‘{abstract}’ label to indicate abstract operations/classes.



Multiple inheritance

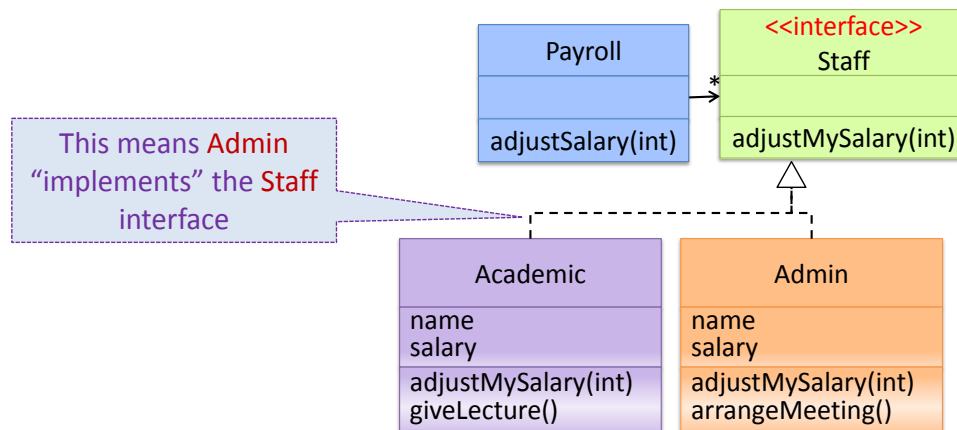
In the example below, the TA class inherits from Staff as well as Student. Such multiple inheritance is allowed among C++ classes but not among Java classes. However, Java interfaces can be used to achieve a similar effect.



Interfaces

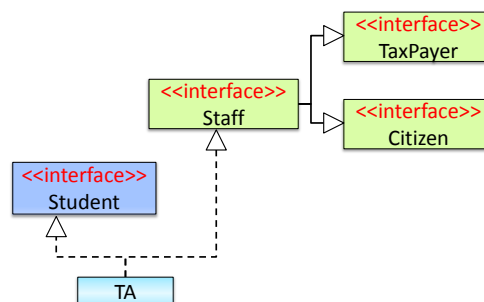
In an OO solution, an *interface* is a behavior specification. At code level, that roughly translates to a ‘class that has no implementation’. In Java, we can directly declare a class as an interface. In

C++, we can use a fully abstract class (i.e. all operations are *pure virtual*) to emulate an interface. We use an interface to specify the available operations of a class without implementing any. In UML, we show an interface with the keyword <<interface>>.



If a class implements all operations specified in an interface, it is said that the class “implements” the interface. In UML, the relationship is shown with a dashed line & an arrowhead similar to the one used for inheritance relationship.

If class C implements interface I, we treat C as a subtype of I. This means C is substitutable wherever I is expected. The diagram in the previous slide indicates that **Academic** and **Admin** implements the **Staff** interface and therefore, subtypes of **Staff**. It also illustrates how to achieve polymorphism using interfaces. Java allows multiple inheritance among interfaces. It also allows a class to implement multiple interfaces.



To summarize the relationship between interface, class, and abstract class:

- An interface is a behavior specification, with no implementation.
- A class is a behavior specification + implementation.
- An abstract class is a behavior specification + partial implementation.

At code level,

- In Java, an interface can be specified using an 'interface' construct.
- In C++, since there is no construct called 'interface', interfaces are specified as 'fully abstract' classes. That means as an abstract class that has no implementation at all.

Dependency injection: Polymorphism in action

Now let us look at another situation where polymorphism comes in handy. During developer testing, we often have to ‘inject’ stubs into the code to isolate the SUT from other collaborating objects so that it can be tested independently. This is called *dependency injection* i.e. replacing the collaborating objects with test-friendly objects such as stubs or mock objects.

Given next is a sample testing scenario that tests the totalSalary of the Payroll class. The production version of the totalSalary method collaborates with the SalaryManager object to calculate the return value. During testing, we substitute the SalaryManager object with a SalaryManagerStub object which responds with hard-coded return values.

```
public class PayrollTestDriver {
    public static void main(String[] args) {
        //test setup
        Payroll p = new Payroll();
        p.setSalaryManager(new SalaryManagerStub()); //dependency injection
        //test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        print("Test 1 output " + p.totalSalary());
        //test case 2
        p.setEmployees(new String[]{"E001"});
        print("Test 2 output " + p.totalSalary());
        //more tests
        System.out.println("Testing completed");
    }
}
//-----
class Payroll{
    private SalaryManager s = new SalaryManager();
    private String[] employees;

    void setEmployees(String[] employees) {
        this.employees = employees;
    }
    /*the operation below is used to substitute the actual SalaryManager
    with a stub used for testing */
    void setSalaryManager(SalaryManager s) {
        this.s = s;
    }
    double totalSalary(){
        double total = 0;
        for(int i=0;i<employees.length; i++){
            total += s.getSalaryForEmployee(employees[i]);
        }
        return total;
    }
}
//-----
class SalaryManager{
    double getSalaryForEmployee(String empID){
        //code to access employee's salary history
        //code to calculate total salary paid and return it
    }
}
//-----
class SalaryManagerStub extends SalaryManager{
    double getSalaryForEmployee(String empID){
        if(empID.equals("E001"))return 2300.0;
        else if(empID.equals("E002"))return 4100.0;
```

```

    else throw new Error("unknown id");
  }
}

```

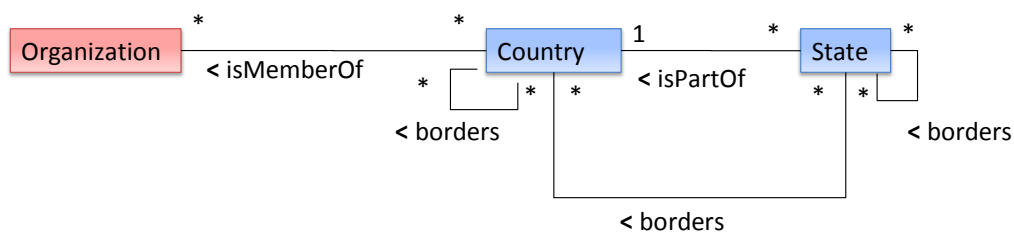
References

- [1] Object-Oriented Programming with Objective-C , A document by Apple inc., retrieved from http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/OOP_ObjC/OOP_ObjC.pdf

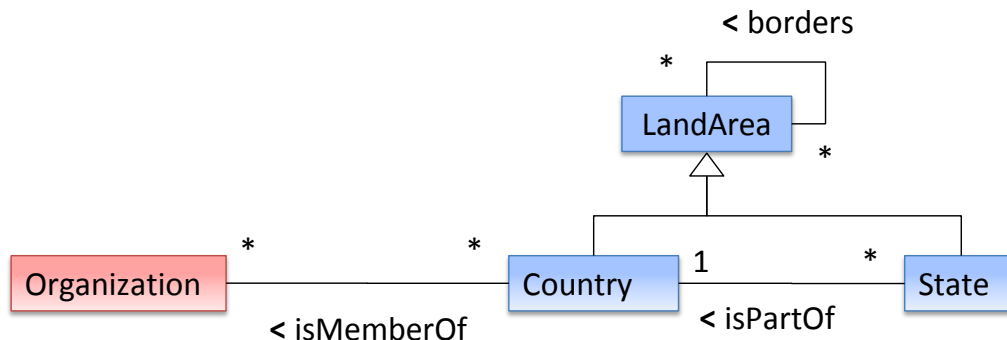
Worked examples

[Q1]

Refine the class diagram model (given in L6 handout) so that it uses inheritance.

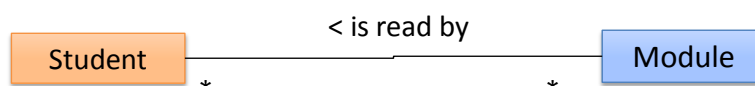


[A1]



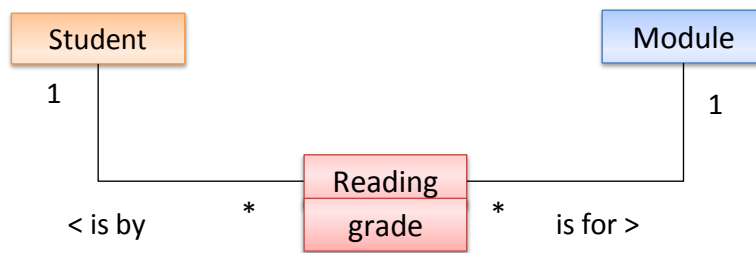
[Q2]

Modify the class diagram below so that we can keep track of the grade a Student earned every time he/she reads a module. Here, 'reads' means taking the module.



[A2]

It appears that now we are required to store some data (i.e. grade) about an association (i.e. 'is read by'). Note that storing the grade inside the Student or the Module is not appropriate as the data is about a particular association between the two objects. Besides, the Student can read a module multiple times. To tackle this situation, we can introduce a class called 'Reading' to represent the association.



Based on the new class diagram, a Reading object represents exactly one student reading exactly one Module and the grade he/she obtained for the Module in that reading. A module can have any number of Reading objects associated with it and a Student can have any number of Reading objects. Furthermore, a Student can have multiple Reading objects for the same Module.

Note: Reading class can be shown as an association class too.

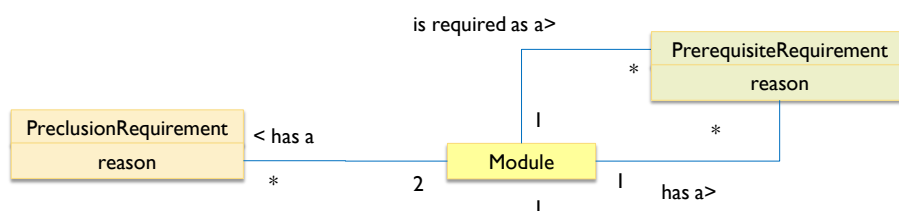
[Q3]

Create an OO class diagram to represent the following description:

Some modules require other prerequisite modules to be taken first. If two modules cover nearly same material, taking one of them would preclude a student from taking the other. Such modules are said to be mutually exclusive. The reasons or descriptions for preclusion as well as for the prerequisite also need to be captured.

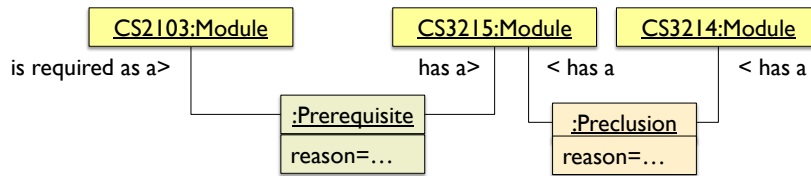
[A3]

We cannot represent a prerequisite as a mere association between two modules because there is some data (i.e. the reason for the prerequisite) we have to store about the association. Therefore, we introduce a class to represent a prerequisite. Note that the Prerequisite class does not represent a module that is a prerequisite for another module. It simply represents the fact/requirement/constraint that a certain Module is a prerequisite for a certain other module. A similar reasoning is behind the Preclusion class. Please refer practice question PQ2.4 for another example of a similar situation.



A Prerequisite has two associations with the Module class because the roles played by each of the two Module objects connected to a Prerequisite object are different (one is the module that requires the other module as the prerequisite). In the case of the Preclusion, both Module objects play the same role (either one is a preclusion with respect to the other), letting us show it as one association with multiplicity of 2.

In the object diagram below (for your info only, not required by the question), CS2103 is a prerequisite for CS3215 while CS3215 is a preclusion for CS3214 (and vice versa).

**[Q4]**

- Draw the corresponding sequence diagram for the interactions resulting from the highlighted statement below.
- Draw a class diagram for the code shown in the previous question. Show associations, dependencies, navigabilities, and multiplicities.

```

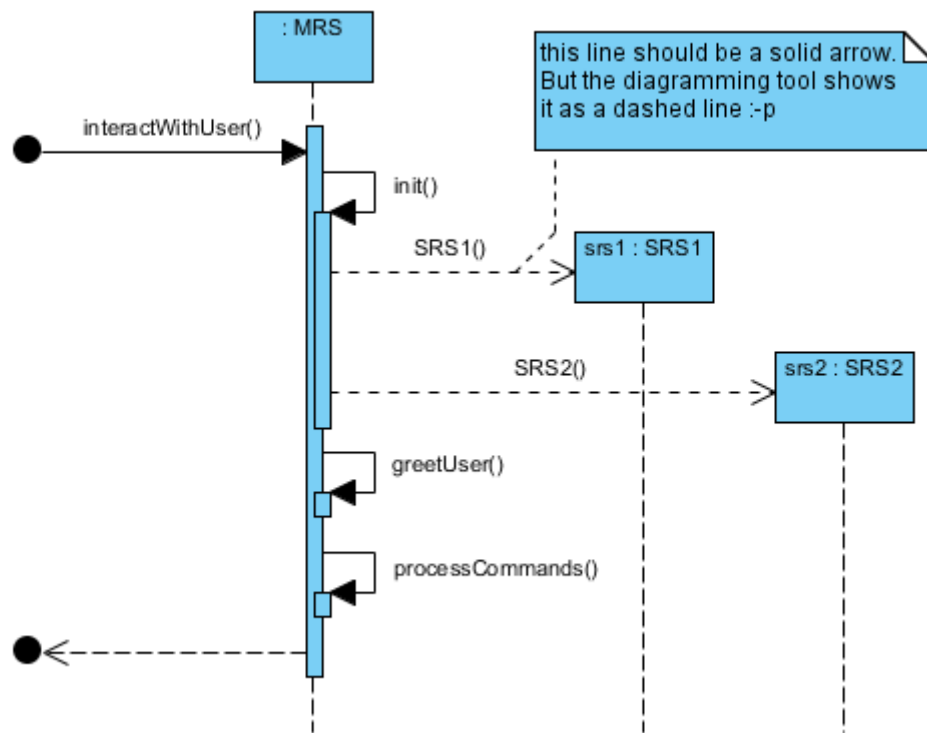
class MRS {
    private int BUFFER_SIZE = Config.BS;
    //BS is a class-level constant of the Config class
    private SRS1 srs1 = null;
    private SRS2 srs2 = null;

    public static void main(String[] args){
        MRS mrs = new MRS();
        mrs.interactWithUser();
    }
    private void interactWithUser(){
        init();
        greetUser();
        processCommands();
    }
    private void init(){
        srs1 = new SRS1();
        srs2 = new SRS2();
    }

    // TextDiff and Result are part of the solution (they are not library classes).
    private boolean compareResults(Result[] results){
        TextDiff diff = new TextDiff();
        return diff.compare(results);
    }
}
  
```

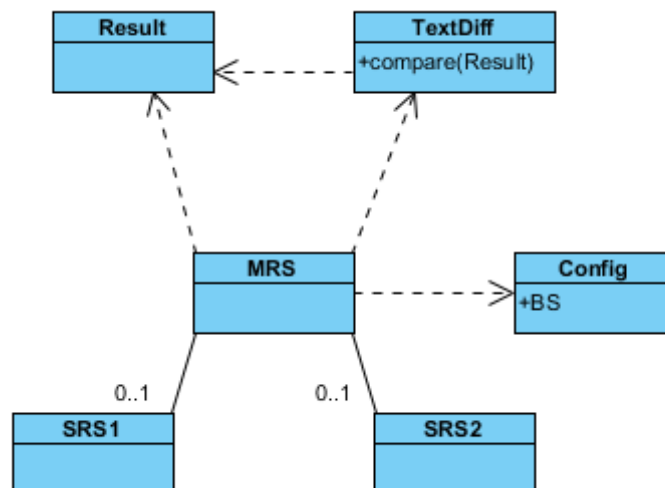
[A4]

(a)



Note slight variations in the notation. Unfortunately, no two UML tools draw diagrams exactly alike and apparently, none follow the UML standard precisely. It is better for you to get used to such slight variations.

(b)



Notes: `Result` is simply a parameter to a method of `MRS`. It does not indicate a structural relationship between `MRS` and `Result`. As far as we can see from the given code, it is merely a dependency. Similarly, `TextDiff` is just a local variable inside a method, not a structural relationship.

[Q5]

Consider the code given below. Draw a class diagram to match the code. Include attributes, operations, visibilities, navigabilities, multiplicities, and dependencies.

```

class ReportGenerator{

    //getInstance always returns a valid Storage object.

    private Storage storage = Storage.getInstance();

    public static void main(){
        ReportGenerator rg = new ReportGenerator();
        rg.printReport(342);
    }

    public void printReport(int reportID){
        //getReport(int) returns a Report object.
        displayReport ( storage.getReport(reportID).getText() );
    }

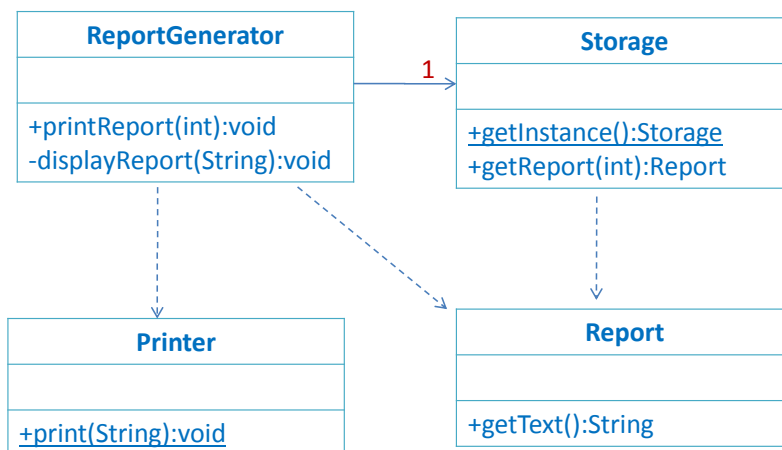
    private void displayReport(String s){
        //call the static method print() of Printer class
        Printer.print(s);
    }

}

```

[A5]

Note how only the link between ReportGenerator and Storage is an association. The other links are merely dependencies because they do not represent structural links between classes. The multiplicity 1 represents the fact that a ReportGenerator will always be linked to exactly one Storage object.

**[Q6]**

Jim recently learned the following four things in a programming class.

- Inheritance
- Dynamic binding
- Overriding
- Substitutability

However, Jim has no idea what polymorphism is. Your job is to prepare a short write-up (about 1 page) to explain what polymorphism is and how *a – d* above work together to achieve polymorphic behavior. You may use diagrams of any sort, code snippets, and pseudo-code in your answer. You may also use the Payroll example from the handout.

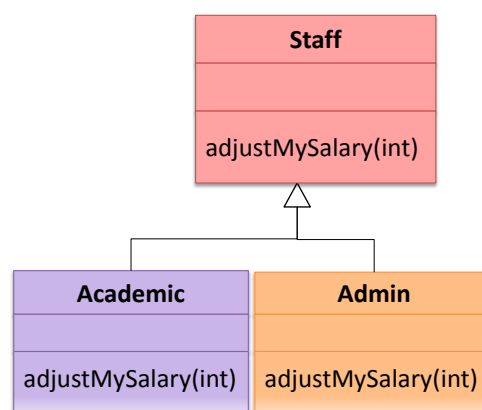
Note that Jim already knows what each of *a-d* means individually. We do not have to explain each in-detail again. What he does not know is what polymorphism is and how *a-d* combines to achieve polymorphism.

[A6]

Polymorphism is the ability to treat multiple types as a single general type and still get different behavior from each of those types. For example, line 5 in below code treats Admin and Academic objects as a single type called Staff and still the result of the `adjustMySalary` differs based on whether *s* is an Academic object or an Admin object.

```
ArrayList<Staff> staff = new ArrayList<Staff>(); //line 1
staff.add(new Admin("Sam")); //line 2
staff.add(new Academic("Dilbert")); //line 3
staff.add(new Admin("Mui Kiat")); //line 4
for (Staff s: staff) { s.adjustMySalary();} //line 5
```

According to line 1, `staff` `ArrayList` expects `Staff` objects. Yet, we can add `Admin` and `Academic` objects to it (lines 2-4). That is an example of “the ability to treat multiple types as a single general type”. This ability is a result of substitutability, which states that wherever an object of super type is expected, we should be able to substitute a subtype. This implies that `Admin` and `Academic` are subtypes of `Staff`. One way we can achieve this is using inheritance. That is, we can make `Admin` and `Academic` subclasses of `Staff`.

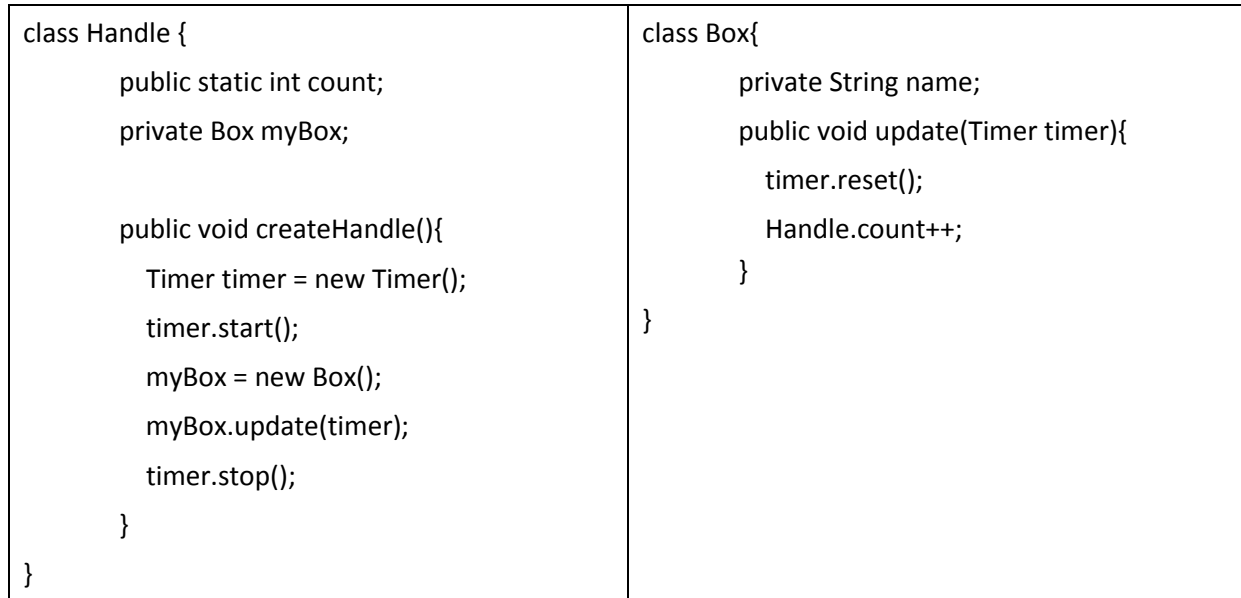


The ability to “still get different behavior from each of those types” is given by dynamic binding. That is, the runtime dynamically checks the actual type of the object (irrespective of its declared type) and binds to the “most concrete” definition of an operation (i.e. the one that is defined in the lowest level of the inheritance hierarchy). That means, if we override the `adjustMySalary` operation in `Academic` and `Admin` objects, those operations will be the ones invoked during runtime instead of the one defined in the `Staff` class. This way, we get the behavior of `Admin` and `Academic` objects although the declared type of *s* is `Staff`.

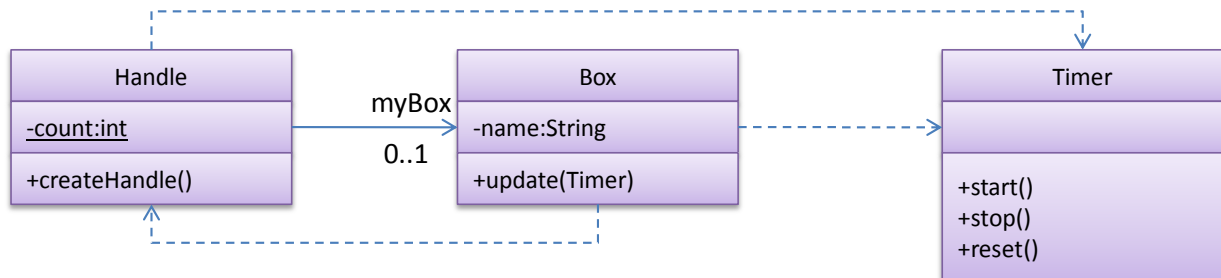
[There you have it Jim; that is how Inheritance, Dynamic binding, Overriding, and Substitutability combine to achieve polymorphism.]

[Q7]

Draw the corresponding class diagram for the following code. Include navigabilities, visibilities, dependencies, multiplicities, attributes, association role names, and operations.



[A7]



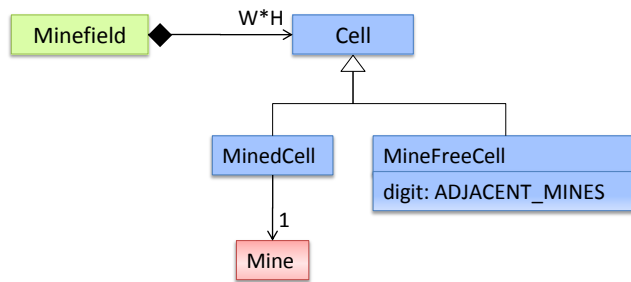
Points to note:

- Associations are structural links between objects. An association link should be supported by an instance-level variable that holds the link. Box accessing a static variable in Handle class simply means a dependency, not an association link. Local variables (e.g. timer variable inside createHandle method) and parameters (e.g. Timer parameter in the update method) too indicate dependencies.
- An association link already implies a dependency; there is no need to add a dependency link from Handle to Box.

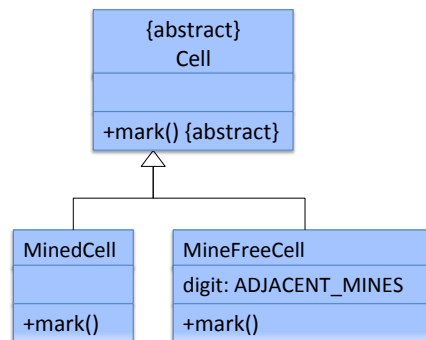
[Q8]

Given below is a partial class diagram from the Minesweeper game discussed in lectures.

Explain at least one instance where you can take advantage of polymorphism when implementing this class diagram. State which classes and which operations will be involved and what is the polymorphic behavior you expect.

**[A8]**

The mark operation in Cell class can be abstract and can be overridden in MinedCell and MineFreeCell classes. The Minefield object can treat all cells as of type Cell. When a cell is marked by the player, Minefield object can simply call the abstract mark() operation of the Cell class and still get a different behavior depending on whether the actual cell object is a MinedCell or a MineFreeCell. The same can be applied to clear() operation.



---End of document---