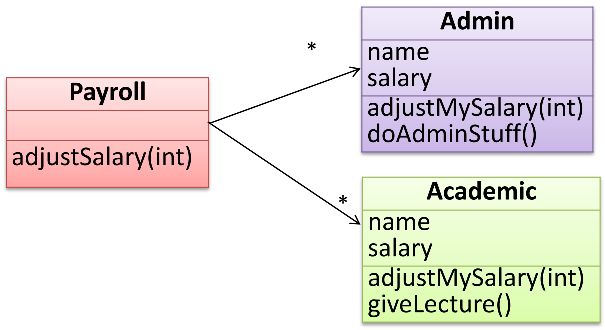
[L5P1]
Object-Oriented Programming: Advanced Concepts

Polymorphism

Polymorphism is an important and useful concept in the object-oriented paradigm. Take the example of writing a payroll application for a university to facilitate payroll processing of university staff. Suppose an adjustSalary(int) operation adjusts the salaries of all staff members. This operation will be executed whenever the university initiates a salary adjustment for its staff. However, the adjustment formula is different for different staff categories, say admin and academic. Here is one possible way of designing the classes in the Payroll system.

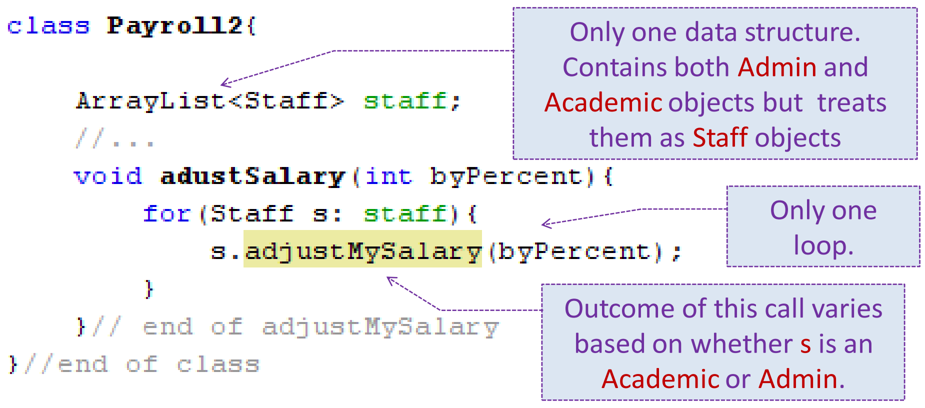


Here is the implementation of the adjustSalary(int) operation from the above design.

```
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);
        }
    }// end of adjustMySalary
}//end of class
```

Similar processing.

Note how processing is similar for the two staff types. It is <u>as if the type of staff members is irrelevant</u> to they are processed inside this operation! If that is the case, can the staff type be “abstracted away” 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);
        }
    }// end of adjustMySalary
}//end of class
```

Only one data structure. Contains both Admin and Academic objects but treats them as Staff objects

Only one loop.

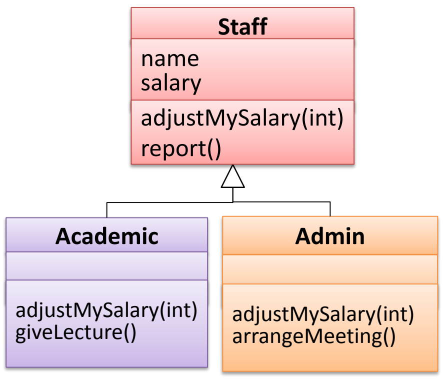
Outcome of this call varies based on whether s is an Academic or Admin.

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 more staff types are added).

This does not imply getting rid of the Academic and Admin classes completely and replacing 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, ArrayList staff contains both Admin and Academic objects although it is treated as an ArrayList of Staff objects. However, when the adjustMySalary(int)operation of these objects is called, the resulting salary adjustment will be different for Admin objects and Academic objects. Therefore, different types of objects are treated as a single general type, but yet each type of object exhibits a different kind of behavior. This is called *polymorphism* (literally, it means “ability to take many forms”). In this example, an object that is perceived as type Staff can be an Admin object or an Academic object.

To retain the multiple types while still allowing some parts of the system to treat them as one type, *inheritance* (*class inheritance* or *interfaces inheritance*) can be used. Here is a possible design that uses class inheritance.



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

```
class Staff {
    String name;
    double salary;

    void adjustMySalary(int percent) {
        // do nothing
    }
}

//-----------------------------------------
```

```
class Admin extends Staff {

    @Override
    void adjustMySalary(int percent) {
        salary = salary * percent;
    }
}

//------------------------------------------

class Academic extends Staff {

    @Override
    void adjustMySalary(int percent) {
        salary = salary * percent * 2;
    }
}
```

Using the above example, there are three issues that are at the center of how polymorphism is achieved: substitutability, operation overriding, and dynamic binding.

Operation overriding

To get polymorphic behavior from an operation, the operation in the superclass needs to be redefined in each of the subclasses. This is called *overriding*. i.e. Admin and Academic override the adjustMySalary(int) of Staff.

Note that Payroll cannot use the adjustMySalary(int) operation unless it is present in the Staff class. 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*? 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 arises when there are 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) ...
        }
        ...
}
```

The constructor has been overloaded

```
void calculateCAP (String matric) {  ...  }
void calculateCAP (int[] averages) {  ...  }
```

Overloading is used 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 arises 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* (sometimes called *late binding*[3] and *run-time binding*). Most OO languages support dynamic binding.

> Note: The opposite of dynamic binding is *static binding* (also called *early binding* and *compile-time binding*. As explained in the previous section, operation overloading is handled using static 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, only its operation header needs to be defined 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. no object instances can be created from it. This is logical because it does not make sense to

[3] There are subtle differences between late binding and dynamic binding, but they are beyond the scope of this handout.

create Staff objects which are neither Admin objects nor Academic objects. In any case, instances of an abstract Staff class cannot be created because the definition of the class is now incomplete. For example, line 1 below is not allowed but line 2 is allowed.

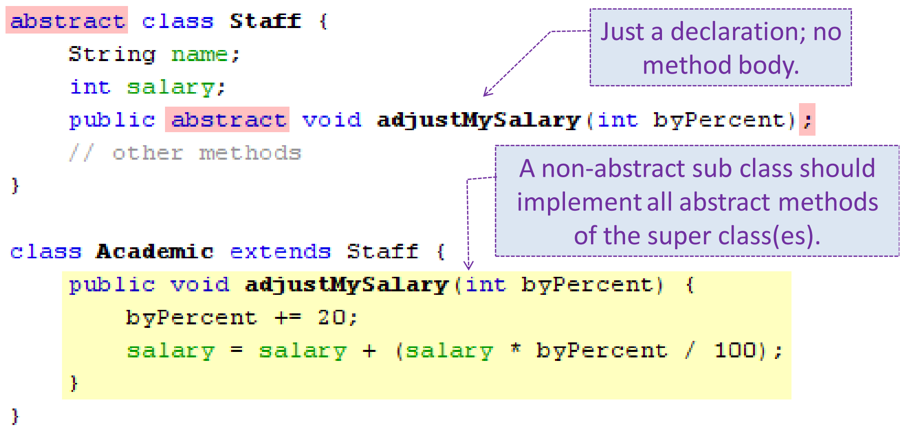
```
Staff s1 = new Staff(); // line 1 , not allowed
Staff s2 = new Admin(); //line 2, allowed
```

Furthermore, some OOP languages allow declaring a class as abstract even if it does not have any abstract operations so as to prevent it from being instantiated.

> Note: The term *concrete* class is used to distinguish a normal class from an abstract class. i.e. if a class is not an abstract class, then it is a concrete class.

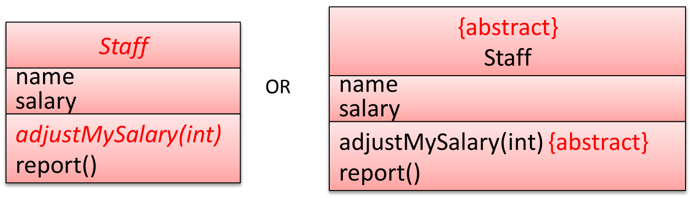
A sub class should implement all abstract methods of all its super classes, or otherwise declare itself as abstract.

In UML, italics or the '{abstract}' label are used to indicate the abstract operations/classes.



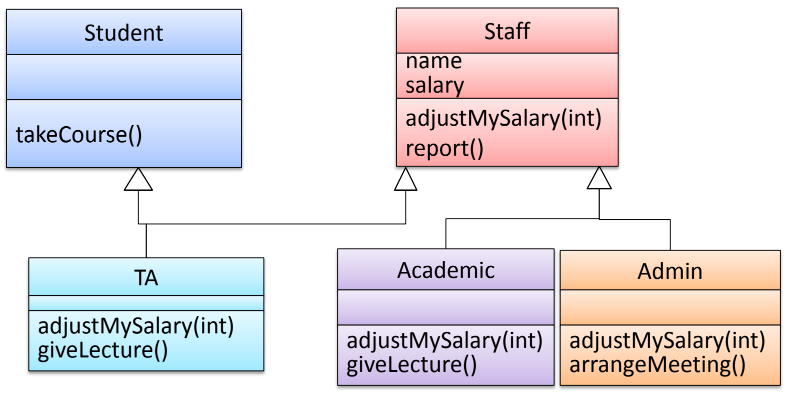
```
abstract class Staff {
    String name;
    int salary;
    public abstract void adjustMySalary(int byPercent);
    // other methods
}

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

Just a declaration; no method body.
A non-abstract sub class should implement all abstract methods of the super class(es).

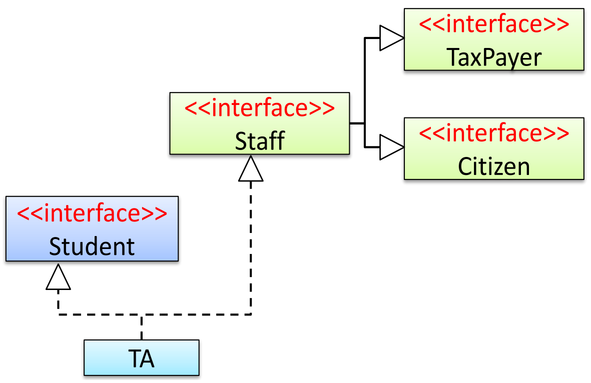


Multiple inheritance

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.



Java allows multiple inheritance among interfaces. It also allows a class to implement multiple interfaces. The design given below is allowed in Java.



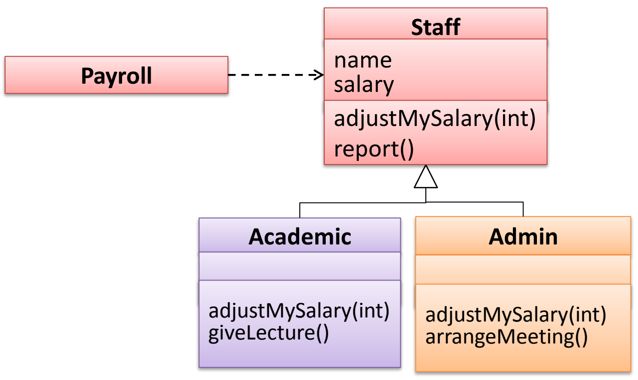
Class vs Abstract Class vs Interface

- 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.

Liskov Substitution Principle (LSP)

> Liskov Substitution Principle states that if a program module is using a super class, then the reference to the super class can be replaced with a sub class without affecting the functionality of the program module.

As we know, Java already allows substituting objects of sub classes where an object of the super class is expected. However, that does not mean that doing so will not break the functionality of the code. To follow LSP, we should be careful not to contradict the behavior specified by the super class.



For example, let us assume the Payroll class depends on the adjustMySalary(int percent) method of the Staff class. Furthermore, the Staff class states that the adjustMySalary method will work for all positive percent values. Both Admin and Academic classes override the adjustMySalary method. Now consider the following:

- Admin::adjustMySalary method works for both negative and positive percent values.
- Academic::adjustMySalary method works for percent values 1..100 only.

In the above scenario, Admin class follows LSP because it fulfills Payroll's expectation of Staff objects (i.e. it works for all positive values) but the Academic class violates LSP because it will not work for percent values over 100 as expected by the Payroll class. That is, substituting Admin objects for Staff objects will not break the Payroll class, but substituting Academic objects for Staff objects will break the Payroll functionality.

Worked examples

[Q1]

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

a. Inheritance
b. Dynamic binding
c. Overriding
d. Substitutability

However, Jim has no idea what polymorphism is. You 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.

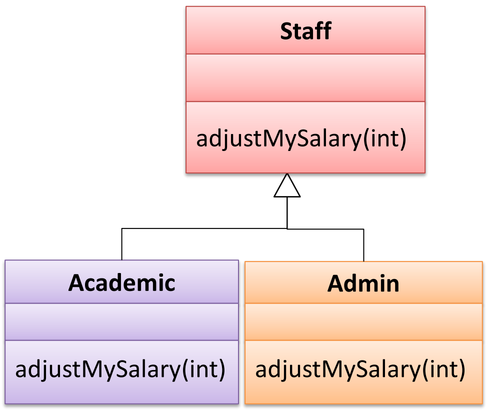
Note that Jim already knows what each of *a-d* means individually. We do not have to explain each in-detail again. What he does not know is what polymorphism is and how *a-d* combines to achieve polymorphism.

[A2]

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, the loop at line 5 in the 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) { //line 5
        s.adjustMySalary();
}
```

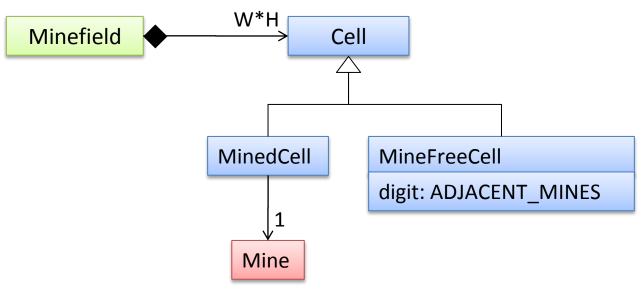
According to line 1, staff ArrayList expects Staff objects. Yet, we can add Admin and Academic objects to it (lines 2-4). That is an example of "the ability to treat multiple types as a single general type". This ability is a result of substitutability, which states that wherever an object of super type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



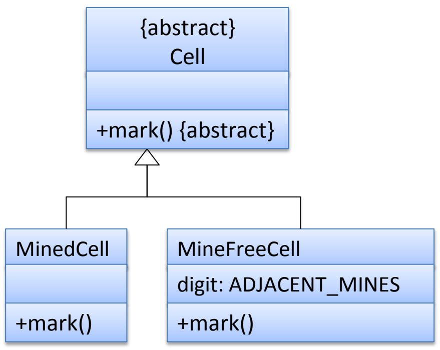
The ability to "still get different behavior from each of those types" is given by dynamic binding. That is, the runtime dynamically checks the actual type of the object (irrespective of its declared type) and binds to the "most concrete" definition of an operation (i.e. the one that is defined in the 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.]

[Q2]
Given below is a partial class diagram from the Minesweeper game. 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.



[A2]
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 handout --