Jin L.C. Guo

M2 (c) - Types and Polymorphism

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

Recap -- Objective

- Programming mechanism:

Java Generics, Java Nested Classes

- Concepts and Principles:

Separation of concerns;

- Patterns and Antipatterns:

STRATEGY, SWITCH Statement 

- Design techniques:

Function objects

Objective of this class

- Concepts and Principles:

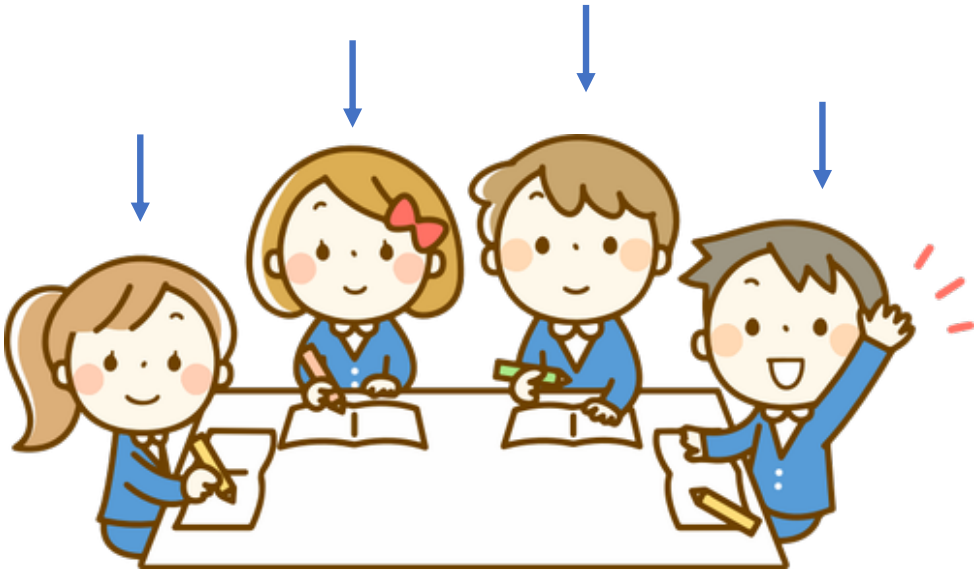
Interface Segregation Principle

- Patterns and Antipatterns:

ITERATOR

How to traverse students enrolled in the class?

- So that
  - I can add grade to each student
  - I can print each student's ID
  - I can …



Activity: How to allow the client code to traverse students enrolled in the class?

```
for(int i=0; i<course.getStudents().size; i++)
{
    Student s = course.getStudents().get(i);
    /* do something using Student instance*/
}
```

```
public class Course
{
        private List<Student> aEnrollment
                                = new ArrayList<>();

        … …

        public List<Student> getStudents()
        {
                return Collections.unmodifiableList(aEnrollment);
        }


}
```

Can we make the way of traversing the students irrelevant to how the students are stored internally?

think of what might change and what is being commited
eg what if students were stored in a different data structure

What is needed during traversing?

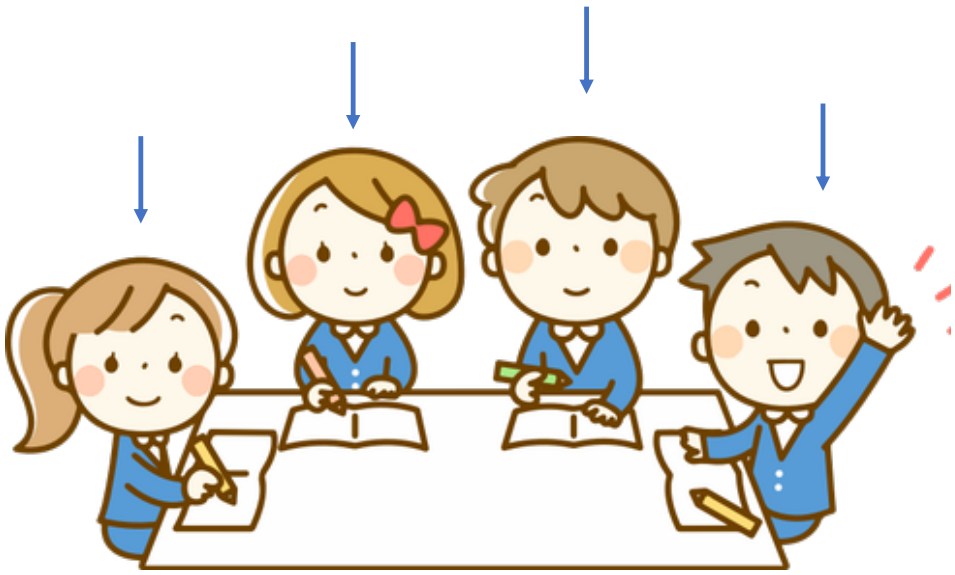
1. Keep track with the current element and know how to get to the next.

```
Student next()
```

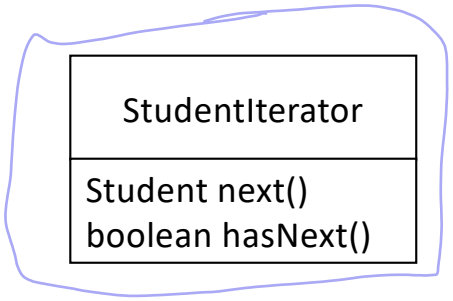
2. Know if the end has been reached

```
boolean hasNext()
```

```
for(int i=0; i<course.getStudents().size; i++)
{
    Student s = course.getStudents().get(i);
    /* do something using Student instance*/
}
```



How to traverse students enrolled in the class?



create a class/interface with desired functionalities

```
for(int i=0; i<course.getStudents().size; i++)
{
    Student s = course.getStudents().get(i);
    /* do something using Student instance*/
}
```

```
public class Course
{
        private List<Student> aEnrollment
                                = new ArrayList<>();

        … …


        public StudentIterator getIterator()
        {
                /* create student iterator*/
                return sIterator;
        }

        public List<Student> getStudents()
        {
                return Collections.unmodifiableList(aEnrollment);
        }

}
```

Client code won't have to change

Java Iterator Interface

- Interface Iterator<E>

  E - the type of elements returned by this iterator

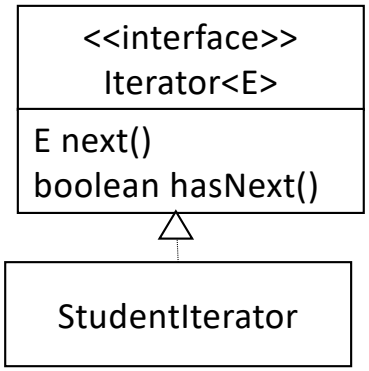
  ```
  boolean hasNext();
  ```

  Returns true if the iteration has more elements.

  ```
  E next();
  ```

  Returns the next element in the iteration.

How to traverse students enrolled in the class?




```
public class Course
{
        private List<Student> aEnrollment
                                = new ArrayList<>();

        … …

        public StudentIterator getIterator()
        {
                 /* create student iterator*/
                return sIterator;
        }


}
```

Iterator<Student>

```
StudentIterator sIterator = course.getIterator();
while(sIterator.hasNext())
{
    Student s = sIterator.next();
    /* do something using Student instance*/
}
```

Adding even more flexibility: how to traverse students in data type such as Club, Committee, …?

```
public class Course
{
        private List<Student> aEnrollment
                                = new ArrayList<>();

        … …


        public Iterator<Student> getIterator()
        {
                 /* create student iterator*/
                return sIterator;
        }


}
```

```
Iterator<Student> sIterator = course.getIterator();
while(sIterator.hasNext())
{
    Student s = sIterator.next();
    /* do something using Student instance*/
}
```

Encapsulate Iterable Behavior

- **Java Iterable<T> Interface**

  T - the type of elements returned by the iterator

```
public Iterator<T> iterator()
```

can be iterated through an iterator

Adding even more flexibility

Same client code to traverse students in data type such as Club, Committee, …

```
public class Course implements Iterable<Student>
{
        private List<Student> aEnrollment
                                = new ArrayList<>();

        … …

        @Override
        public Iterator<Student> iterator()
        {       /* create student iterator*/
                return aEnrollment.iterator();
        }

}
```

```
Iterator<Student> sIterator = course.getIterator();
while(sIterator.hasNext())
{
    Student s = sIterator.next();
    /* do something using Student instance*/
}
```

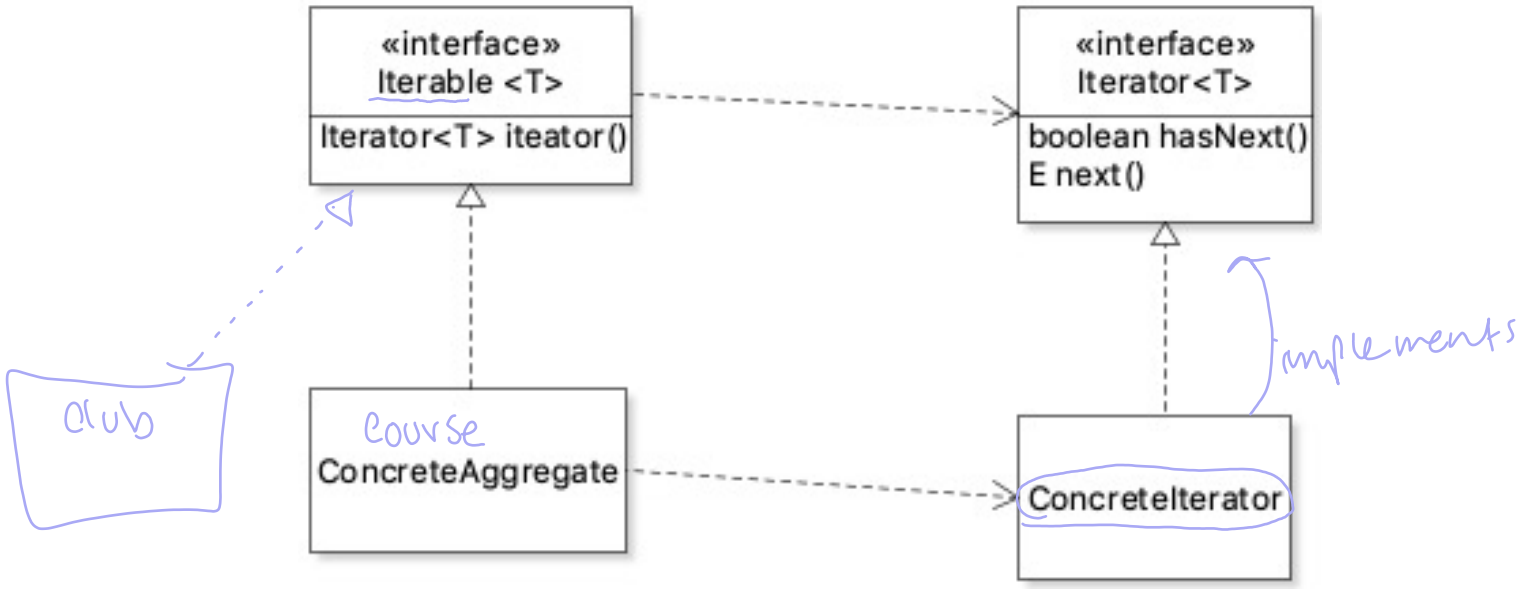
any object that implements Iterable<Student>

want this to be applicable to clubs etc

make the student iterable

Iterator Design Pattern

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation



Adding even more flexibility

Same client code to traverse students in data type such as Club, Committee, …

```
Iterator<Student> sIterator = course.getIterator();
while(sIterator.hasNext())
{
    Student s = sIterator.next();
    /* do something using Student instance*/
}
```

Club, Committee, …

```
public class Course implements Iterable<Student>
{
        private List<Student> aEnrollment
                                = new ArrayList<>();

        … …

        @Override
        public Iterator<Student> iterator()
        {       /* create student iterator*/
                return aEnrollment.iterator();
        }


}
```

using the iterator provided by List<>

Objective of this class

- Concepts and Principles:

Interface Segregation Principle

- Patterns and Antipatterns:

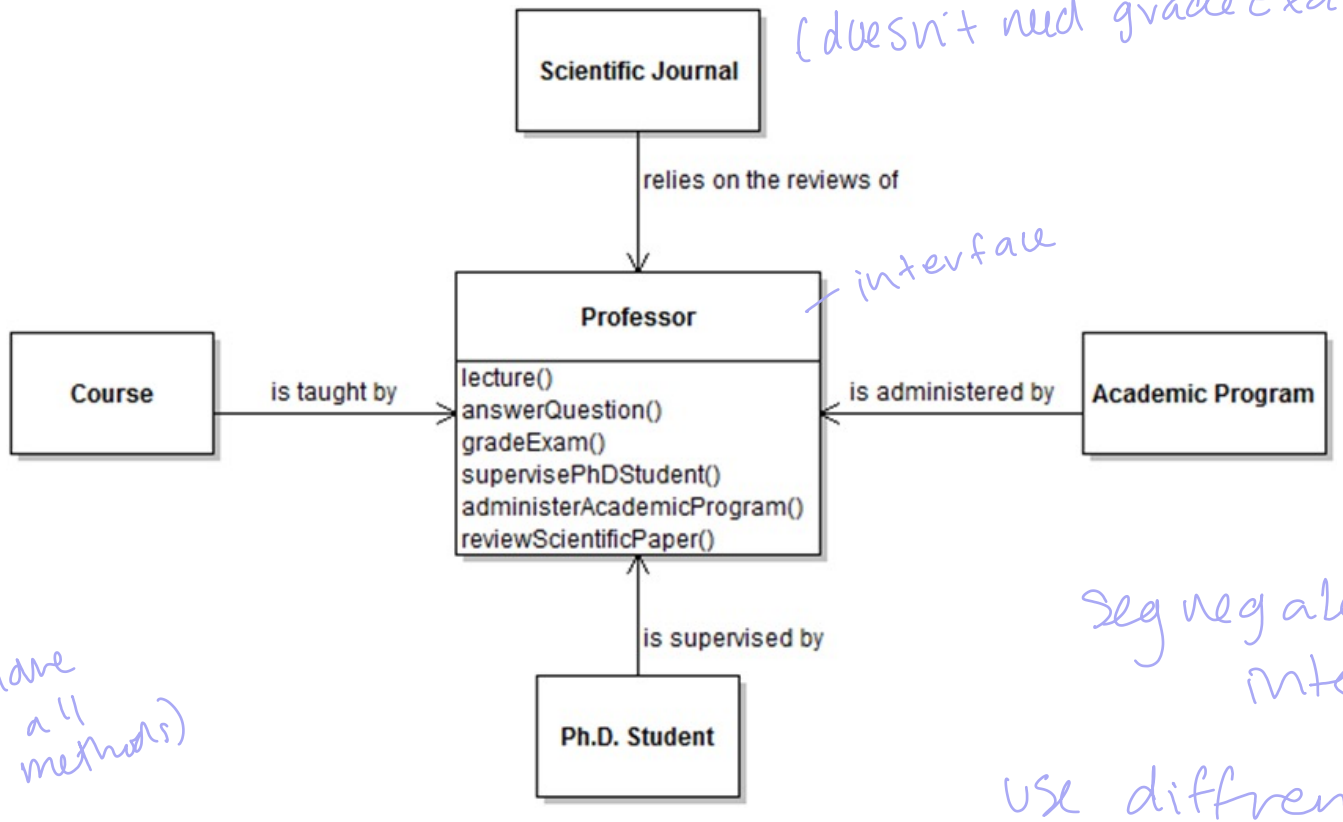
ITERATOR

Interface Segregation Principle

Clients should not be forced to depend on interfaces they do not need.

makes your code safe
Client should not access things not relevent to current scenario

Interface Segregation Principle



(doesn't need gradeExam())

interface

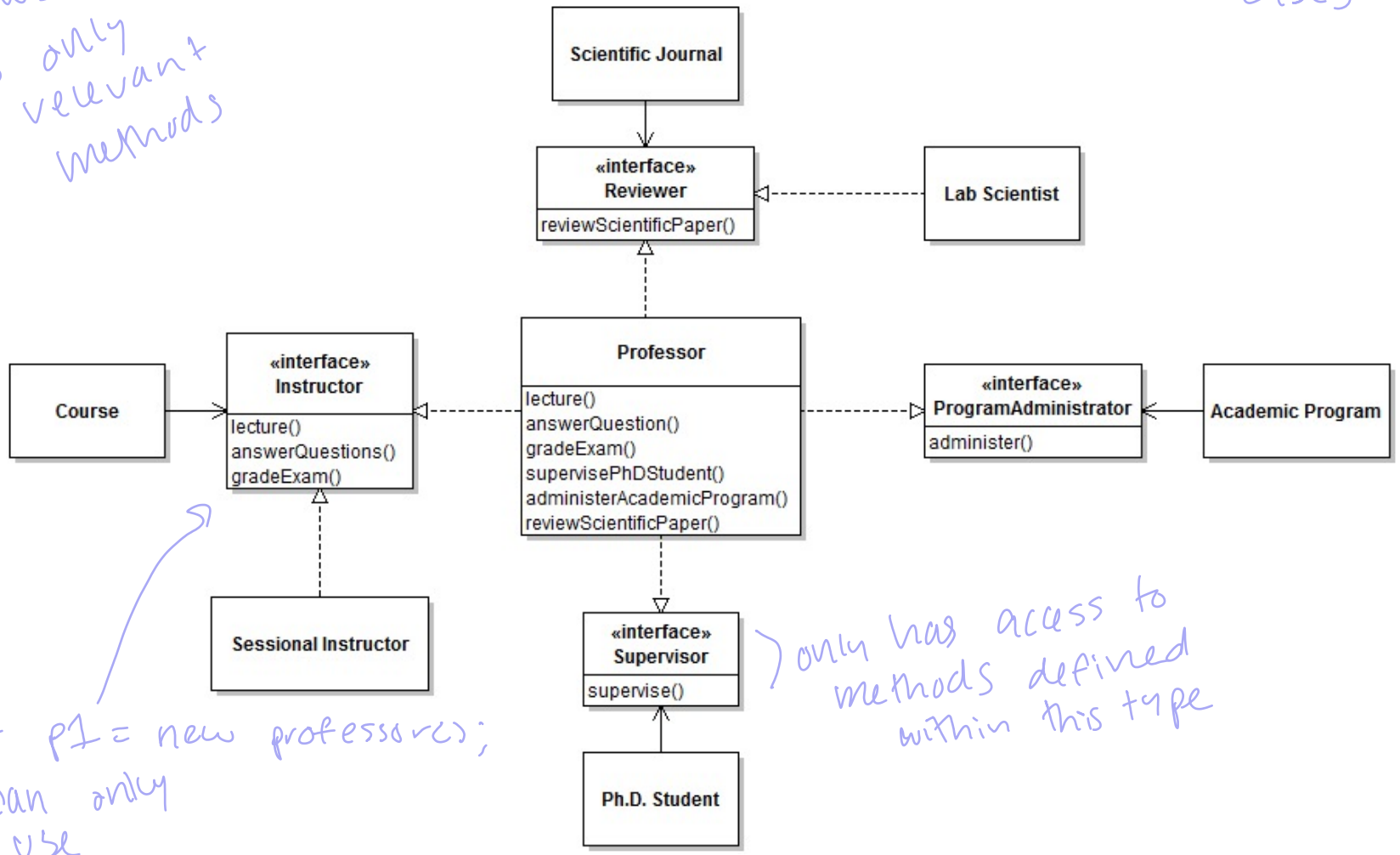
student (shouldn't have access to all professor methods)

segregate the interface!

use different types that correspond to scenarios

different types of profs for different cases

allows access to only relevant methods



only has access to methods defined within this type

instructor p1 = new professor();

can only use
lecture()
answerQuestions()
gradeExam()