Using Interfaces

Object Oriented Programming
2024 First Semester
Shin-chi Tadaki (Saga University)

Unit test

- Testing in software developments
  - unit tests: testing small size codes, usually functions and methods
  - integration tests: testing systems by combining components
- `JUnit` is a commonly used tool for unit tests in Java.
  - `JUnit` is available in various IDEs such as `netbeans`.

Using `JUnit4` in `netbeans`

- Preparation
  - Install `junit4` library
  - Install `hamcrest` library
- Select a class for testing
  - `Tools` → `Create/Update Tests`
  - Test templates are generated

Utility methods initially created by `JUnit4`

- `setUpClass()`: sets up various common features and runs once before all testing methods
- `tearDownClass()`: clearing various results of tests and runs once after all testing methods
- `setUp()`: set up some features for one test and runs before each testing method
- `tearDown()`: clearing various results of one test and runs once after each testing method

Example 1.1: Testing example0

- Creating sort target in the constructor
- Testing `sort()` method

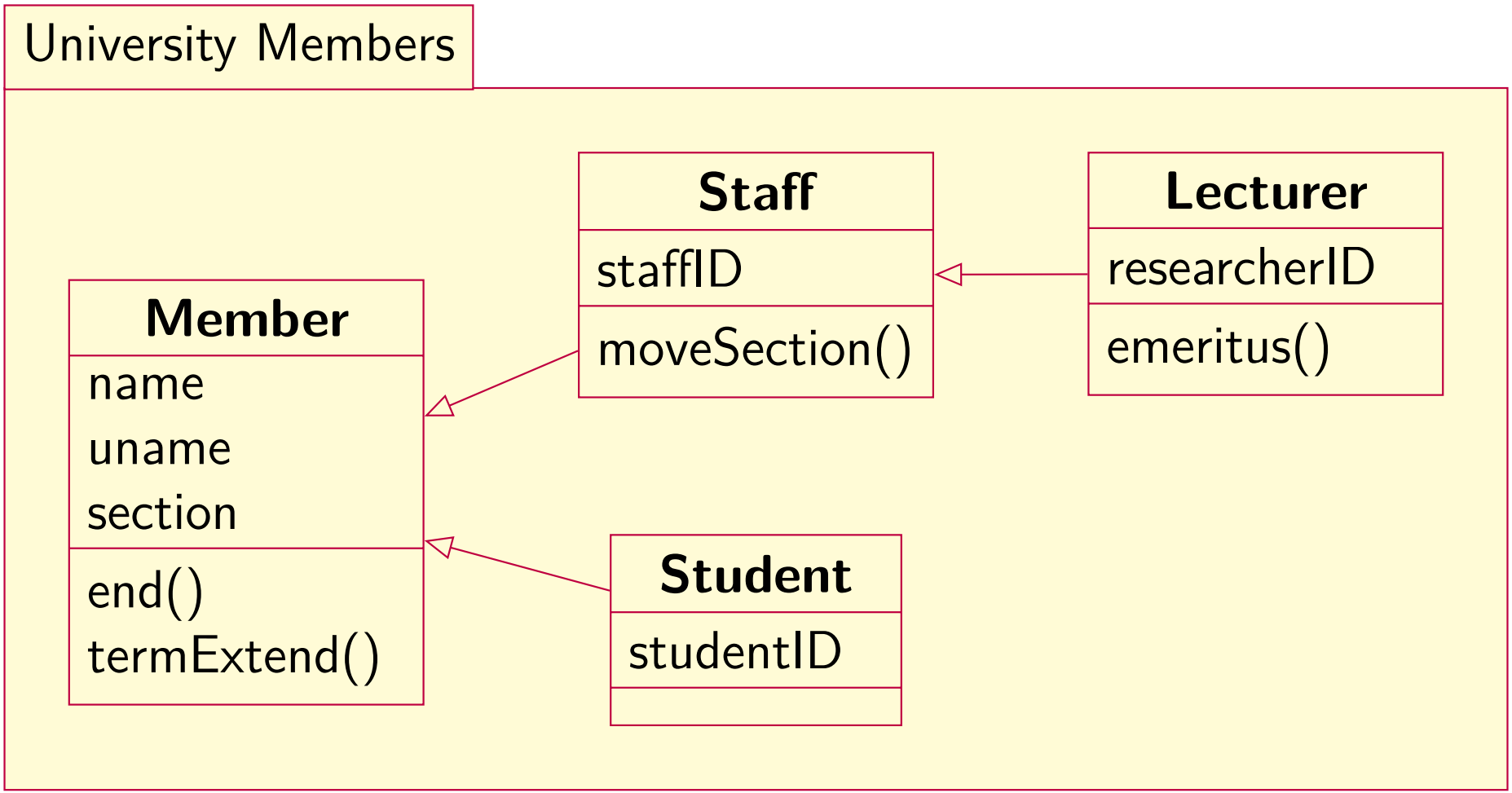
```
1            public void testSort() {
2                System.out.println("sort");
3                MergeSort instance = new MergeSort(studentList);
4                List<Student> expResult = sortedList;
5                List<Student> result = instance.sort();
6                Assert.assertEquals(expResult, result);
7            }
```

- Testing `isSorted()` method

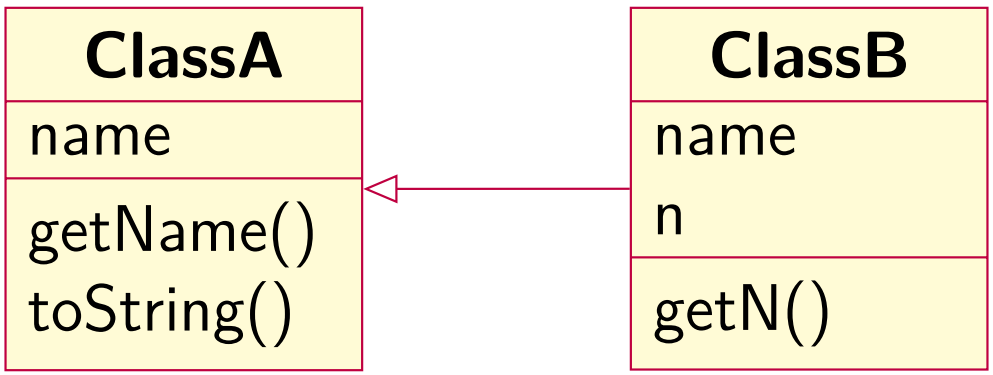
Hierarchical structure of classes: クラス階層

- Superclasses
  - Generalization / Abstraction : 一般化 / 抽象化
  - Having common methods and fields
  - Defining abstract methods: *without implementations*
- Subclasses
  - embodiment / Specialization : 実装 / 具体化
  - Implementing (Overriding) methods
  - Adding new methods and fields

Example 2.1: University members



Example 3.1: Class inheritance



sample code
`https://github.com/oop-mc-saga/JavaIntroduction`

Inheritance: defining subclass: 継承

- A subclass inherits all fields and methods of its superclass.
- The subclass can have additional fields and methods
- The subclass can change implementations of some methods of its superclass.

Generalization: defining superclass: 一般化

- The superclass provides common fields and methods of its subclasses.
- The superclass can have abstract methods, which do not have implementations.
- Classes have abstract methods are called *abstract* class

Method Override: メソッドの再定義

- Note identifiers of methods in Java: contact /signature
  - method name
  - argument list
- Define implementations of abstract methods
- Allow different implementations for subclasses

Polymorphism: 多形

- A method in subclasses is allowed to behave differently from the corresponding method of their superclass
- An instance of subclasses also can be treated as an instance of its superclass
  - The method in the superclass is invoked, if the new implementation is not given in subclasses.

Example 3.2: inheritanceExample

- Superclass: `SuperClass`
- Subclasses: `SubClassA` and `SubClassB`
- Observe the behavior of `getResult()` and `getValue()` methods

Limitation of inheritance in Java

- General difficulties in multiple inheritance
  - Superclasses may have fields or methods with the same name
- Java allows
  - a class inherits only one superclass
- Interfaces as special Superclasses
  - Java classes can inherit multiple interfaces

Interfaces

- Restriction on fields
  - Interfaces have only static final fields
  - Allowing to define constants
- Restriction on methods
  - Interfaces can only have abstract methods: without implementation

Using interfaces

- Declaring to use interface at class definition
- All abstract methods must be implemented
- Users of classes with interfaces need to know only the methods of the interfaces

Comparable: example of interfaces

- Read API document
  `https://www.oracle.com/jp/java/technologies/documentation.html`
- Understand
  - purposes of methods
  - their return values

Today's tasks

- Working with `example1` package
- Add `Comparable` interface to the `Student` class
- Implement the `compareTo()` method
- Change `MergeSort` to be compatible with `Comparable` instances

Implement Comparable interface to Student class

- Copy example0/Student.java into example1 package
- Confirm package name in example1/Student.java
- Modify class definition

```
public class Student implements Comparable<Student>
```

- Understand the meaning of Comparable<Student> phrase
- Implement the compare\To() method

Modify MergeSort

- Make MergeSort to be compatible with Comparable
- Copy example0/MergeSort.java into example1
- Generalize the target class of sorting
  - Specify the target using parametrized types
  - Delete all Student class specification

```
public class MergeSort<T extends Comparable<T>>
```

- Do not use any specific fields and methods of the Student class
  - use compareTo() method
- <T extends Comparable<T>> means the type T implementing Comparable

Parametrized Types

- In the form `C<T1,...,Tn>`, where `C` is the name of a class or interface.
- `Ti` are parametrized types, used for specifying types of parameters of the class or interface.
- Examples:

```
1 List<String> stringList = new ArrayList<>(); //stringList holds
  ↪  instances of String
2 Map<String,Integer> map = new HashMap<>();//The key is String and
  ↪  the value is Integer
```

- You can define classes not by specifying class details by using parametrized types.

Homework

- Write a class for bubble sort being compatible `Comparable`
- Use parametrized types

Appendix: Comparator

- `Comparator` interface provides `compare` method
- `compare(o1,o2)` returns negative, zero, and positive integers depending on `o1` is less than, equal, and greater than `o2`

```
1  public class MergeSort<T> {
2
3      final private List<T> list;
4      final private Comparator<T> comparator;
5
6      public MergeSort(List<T> list, Comparator<T> comparator) {
7          this.list = list;
8          this.comparator = comparator;
9      }
10
11     private boolean less(int i, int j) {
12         return (comparator.compare(list.get(i), list.get(j)) < 0);
13     }
14
15     ...
16 }
```

Using Comparator

```
1  Comparator<Student> comparator
2          = (s1, s2) -> s1.getRecord() - s2.getRecord();
3  MergeSort<Student> sort
4          = new MergeSort<>(Arrays.asList(list), comparator);
```