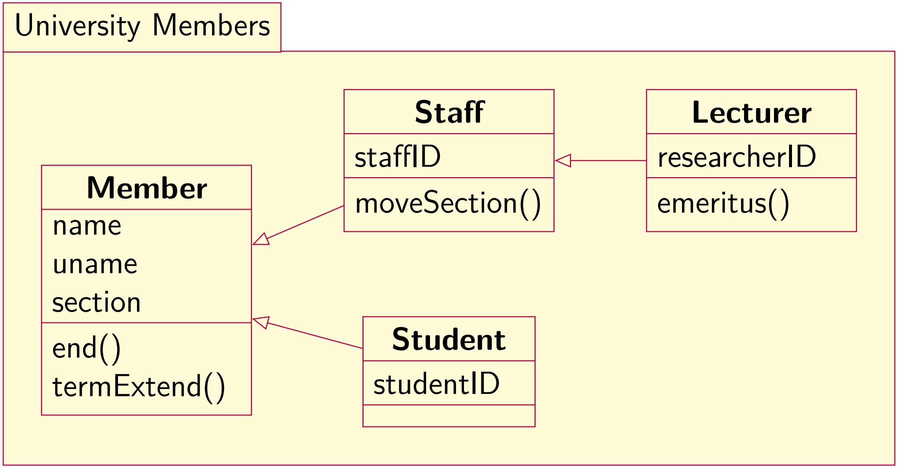
# Using Interfaces

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

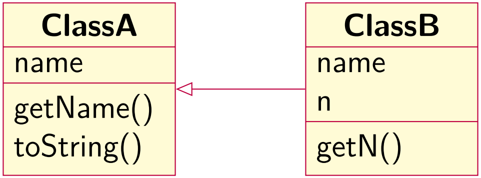
## Hierarchical structure of classes: クラス階層

- 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 1.1: University members



## Example 2.1: Class inheritance



sample code

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

## Inheritance: defining subclass: 継承

- A subclass inherits all fields and methods from 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 to its subclasses.
- The superclass can have abstract methods, which do not have implementations.
- Classes have abstract methods are called *abstract* class and have to be declared as `abstract`.

## 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 in superclass
- 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 2.2: inheritanceExample

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

## classA and classB

```java
public class ClassA {

    final private String label;

    public ClassA(String label) {this.label = label;}
    public String getLabel() {return label;}
    public String toString() {return label;}
}
```

```java
public class ClassB extends ClassA {

    final private int n;

    public ClassB(String label, int n) {
        super(label);
        this.n = n;
    }
    public int getN() {return n;}

    @Override
    public String toString() {
        return super.toString() + ":" + String.valueOf(n);
    }
}
```

## Using classA and classB

```java
public class NewMain {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        ClassA a = new ClassA("First");
        ClassA b = new ClassB("Second", 0);
        ClassB c = new ClassB("Third", 1);

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

- Observe the behavior of `toString()` method for these three instances

## Restrictions 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

- *Interfaces* define methods that classes must implement
  - `Comparable` defines `compareTo()` method
  - `Runnable` defines `run()` method
- 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 `compareTo()` 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 `<T>`
  - Replace all `Student` class specification by `T`

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

- Do not use any specific fields and methods of the `Student` class
  - use `compareTo()` method instead
- `<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.
- Class types can be omitted in the right-hand side of the assignment.
- Examples:

```
List<String> stringList = new ArrayList<>(); //stringList holds
↪  instances of String
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

## Unit tests

- 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 5.1: Testing `example0`

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

```java
public void testSort() {
    System.out.println("sort");
    MergeSort instance = new MergeSort(studentList);
    List<Student> expResult = sortedList;
    List<Student> result = instance.sort();
    Assert.assertEquals(expResult, result);
}
```

- Testing `isSorted()` method

## 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`

```java
public class MergeSort<T> {

    final private List<T> list;
    final private Comparator<T> comparator;

    public MergeSort(List<T> list, Comparator<T> comparator) {
        this.list = list;
        this.comparator = comparator;
    }

    private boolean less(int i, int j) {
        return (comparator.compare(list.get(i), list.get(j)) < 0);
    }

    ...
}
```

## Using Comparator

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