# CSE3040 Java Language

## Lecture 10: Object-Oriented Programming (4)

Dept. of Computer Engineering,
Sogang University

# Interface

- We can also define a variable inside an interface definition.
- These variables automatically become public static final variables.
  - No instance variable can be defined inside an interface definition.

```
interface Motion {
      int NORTH = 1;
      int EAST = 2;
      int SOUTH = 3;
      int WEST = 4;

      void move(int direction);
      int getX();
      int getY();
}

class TwoDMotion implements Motion {
      private int posX, posY;
      public TwoDMotion() { posX = 0; posY = 0; }
      public void move(int direction) {
            if(direction == NORTH) posY--;
            else if(direction == SOUTH) posY++;
            else if(direction == EAST) posX++;
            else if(direction == WEST) posX--;
      }
      public int getX() { return posX; }
      public int getY() { return posY; }
}
```

# Interface

- Before Java 8, all methods in an interface was abstract methods.
  - abstract method: a method without implementation
- From Java 8/9, three new types of methods could be included in an interface.
  - static method (from Java 8)
  - default method (from Java 8)
  - private method (from Java 9)
  - (Currently we are using Java 14.)

- Static method
  - The method digitsOf creates a DigitSequence object and returns it.
  - A method that creates an instance and returns it is called a factory method.
    - Factory methods are often defined as static methods in interface definitions.

```
interface IntSequence {
    static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }
    boolean hasNext();
    int next();
}
```

# Interface

- Using a factory method to create an object instance

```
IntSequence digits = IntSequence.digitsOf(1729);
```

# Interface

- Default method
  - Before default method was included, a class that implements an interface had to implement all methods specified in the interface.
  - For a default method, the class implementing the interface could
    - not define the method (using the default method)
    - define the method (overriding the default method)

```
interface IntSequence {
    default boolean hasNext() { return true; }
    int next();
}
```

```
class SquareSequence implements IntSequence {
    private int i;
    public int next() {
        i++;
        return i*i;
    }
}
```

# Interface

- Default method: Why are they necessary?
  - Suppose we have implemented an interface called IntSequence which has two methods: hasNext() and next().
  - Many classes implement the interface IntSequence.
  - Later on, someone adds an additional method f() in the interface IntSequence.
  - Now, all the old classes that implement the interface will no longer work because it does not implement f().
  - However, if f() is defined as a default method, old classes will still work using the default version of the new method.

# Interface

- Default methods and collisions
  - With default methods, a collision may occur when a class implement multiple interfaces.

```
interface Person {
    String getName();
    default int getId() { return 0; }  // Error: duplicate default methods with the same type of parameters
}

interface Identified {
    default int getId() { return 1; }  // Error: duplicate default methods with the same type of parameters
}

class Employee implements Person, Identified {
    private String name;
    public Employee(String name) { this.name = name; }
    public String getName() { return this.name; }
}

public class Lecture {
    public static void main(String[] args) {
        Employee m = new Employee("Peter");
        System.out.println(m.getId());
    }
}
```

# Interface

- Default methods and collisions
  - It is still an error if only one of the getId() method is defined as default.

```
interface Person {
    String getName();
    default int getId() { return 0; }  // Error: default method conflicts with another method
}

interface Identified {
    int getId();
}

class Employee implements Person, Identified {
    private String name;
    public Employee(String name) { this.name = name; }
    public String getName() { return this.name; }
}

public class Lecture {
    public static void main(String[] args) {
        Employee m = new Employee("Peter");
        System.out.println(m.getId());
    }
}
```

# Interface

- Default methods and collisions
  - If getId() is implemented in class Employee, then no problem.

```
interface Person {
    String getName();
    default int getId() { return 0; }
}

interface Identified {
    int getId();
}

class Employee implements Person, Identified {
    private String name;
    public Employee(String name) { this.name = name; }
    public String getName() { return this.name; }
    public int getId() { return 2; }
}

public class Lecture {
    public static void main(String[] args) {
        Employee m = new Employee("Peter");
        System.out.println(m.getId());
    }
}
```

# Interface

- Default methods and collisions
  - If the argument types are different, then it is ok because there is no ambiguity.

```
interface Person {
    String getName();
    default int getId() { return 0; }  // OK!
}

interface Identified {
    default int getId(int i) { return 1; }  // OK!
}

class Employee implements Person, Identified {
    private String name;
    public Employee(String name) { this.name = name; }
    public String getName() { return this.name; }
}

public class Lecture {
    public static void main(String[] args) {
        Employee m = new Employee("Peter");
        System.out.println(m.getId());
    }
}
```

# Interface

- Private methods
  - private methods can only be called from other methods in the class.

```
interface CustomInterface {

    public abstract void method1();

    public default void method2() {
        method4();  // private method inside default method
        method5();  // static method inside other non-static method
        System.out.println("default method");
    }

    public static void method3() {
        method5(); //static method inside other static method
        System.out.println("static method");
    }

    private void method4() {
        System.out.println("private method");
    }

    private static void method5() {
        System.out.println("private static method");
    }
}
```

# Interface

- Private methods
  - private methods can only be called from other methods in the class.

```
public class Lecture implements CustomInterface {

    public void method1() {
        System.out.println("abstract method");
    }

    public static void main(String[] args){
        CustomInterface instance = new Lecture();
        instance.method1();
        instance.method2();
        CustomInterface.method3();
    }
}
```

- Rules for private interface methods
  - Private interface methods cannot be abstract.
  - Private method can only be used inside interface.
    - They are not used in sub-interfaces or classes implementing the interface.
  - Private static method can be used inside other static and non-static interface methods.
  - Private non-static methods cannot be used inside private static methods.

# Four Principles of Objected Oriented Programming

- Encapsulation
  - hide internal implementation by restricting access to public methods
  - instance variables and some methods are kept private

- Abstraction
  - use of “Interface”, a specification without implementation
  - abstract classes

- Inheritance
  - “is-a” and/or “has-a” relationship between two objects
  - super class (parent class) vs. sub class (child class)
  - reuse the code of existing super classes

- Polymorphism
  - one name can have many different forms
  - static polymorphism: method overloading
  - dynamic polymorphism: method overriding

# Extending a class

- Suppose we have a class called Employee.

```
class Employee {
    private String name;
    private int salary;
    public Employee() {
        this.name = "NoName";
        this.salary = 0;
    }
    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }
    public int getSalary() { return this.salary; }
    public void setSalary(int salary) { this.salary = salary; }
}
```

- Now we can implement a class that extends class Employee.
  - class Manager inherits all members (variables and methods) of class Employee
  - class Manager also has new members defined in its own class definition.
    - bonus, setBonus()

```
class Manager extends Employee {
    private int bonus;
    public void setBonus(int bonus) { this.bonus = bonus; }
}
```

# Super class vs. Sub class

- We can use class Manager as follows.
  - Notice how the methods defined in class Employee (getName and getSalary) are called from an instance of class Manager.

```
public class Lecture {
    public static void main(String[] args) {
        Manager m = new Manager();
        System.out.println(m.getName() + " " + m.getSalary());
    }
}
```

- class Employee is a superclass of class Manager.
- class Manager is a subclass of class Employee.

# Method overriding

- If class Manager does not define methods like getName and getSalary, the methods from its superclass (Employee) are used.
- It is also possible to define getSalary in the definition of class Manager.
- In this case, the getSalary method of class Manager overrides the getSalary method of class Employee and is used.
- This is called Method Overriding.
  - The keyword super is a directive that indicates superclass.
  - The overriding method should have the same parameter type as the superclass method.

```
class Manager extends Employee {
    private int bonus;
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() {
        return super.getSalary() + bonus;
    }
}

public class Lecture {
    public static void main(String[] args) {
        Manager m = new Manager();
        System.out.println(m.getName() + " " + m.getSalary());
    }
}
```

# Method overriding

- Can't we directly access the instance variable salary, instead of calling super.getSalary()?
  - No, because the variable salary is defined as private.
  - If it is a public or a protected variable, it is accessible from the subclass.
  - protected: visible to classes of the same package and subclasses.

```
class Employee {
    private String name;
    private int salary;
    public Employee() {
        this.name = "NoName";
        this.salary = 0;
    }
    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }
    public int getSalary() { return this.salary; }
    public void setSalary(int salary) { this.salary = salary; }
}
class Manager extends Employee {
    private int bonus;
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() {
        return this.salary + bonus;    // Error: salary is private in class Employee.
    }
}
```

# Method overriding

- If a method in the subclass has the same name as a method in its superclass but different parameter type, then the method is not an overriding method.

```
class Employee {
    private String name;
    protected Employee supervisor;
    public Employee() {
        this.name = "NoName";
    }
    public boolean worksFor(Employee supervisor) {
        System.out.println("Employee.worksFor");
        return (this.supervisor == supervisor);
    }
}

class Manager extends Employee {
    public boolean worksFor(Manager supervisor) {
        System.out.println("Manager.worksFor");
        return (this.supervisor == supervisor);
    }
}
```

```
public class Lecture {
    public static void main(String[] args) {
        boolean rv;
        Manager m = new Manager();
        Manager n = new Manager();
        Employee e = new Employee();
        rv = m.worksFor(n);
        rv = m.worksFor(e);
    }
}
```

# Method overriding

- If the programmer intents to implement an overriding method, we can use annotation @Override to prevent mistakes.
  - If there is @Override annotation, a compile error will occur if the method is not an overriding method.

```
class Employee {
    private String name;
    protected Employee supervisor;
    public Employee() {
        this.name = "NoName";
    }
    public boolean worksFor(Employee supervisor) {
        System.out.println("Employee.worksFor");
        return (this.supervisor == supervisor);
    }
}

class Manager extends Employee {
    @Override   // this will cause compile error!
    public boolean worksFor(Manager supervisor) {
        System.out.println("Manager.worksFor");
        return (this.supervisor == supervisor);
    }
}
```

```
public class Lecture {
    public static void main(String[] args) {
        boolean rv;
        Manager m = new Manager();
        Manager n = new Manager();
        Employee e = new Employee();
        rv = m.worksFor(n);
        rv = m.worksFor(e);
    }
}
```

# Method overriding

- When overriding a method, it is possible to change the return type to the subclass type.
- The getSupervisor method is an overriding method, although it returns Manager instead of Employee.

```
class Employee {
    private String name;
    protected Employee supervisor;
    public Employee() {
        this.name = "NoName";
    }
    public Employee getSupervisor() {
        System.out.println("Employee");
        return supervisor;
    }
}

class Manager extends Employee {
    @Override   // this is ok!
    public Manager getSupervisor() {
        System.out.println("Manager");
        return (Manager)supervisor;
    }
}
```

```
public class Lecture {
    public static void main(String[] args) {
        Manager m = new Manager();
        System.out.println(m.getSupervisor());
        Employee e = new Employee();
        System.out.println(e.getSupervisor());
    }
}
```

# Method overriding

- When overriding a method, the overriding method must have at least the equal accessibility with the superclass method.
- The overriding method cannot reduce the visibility of the inherited method.

```
class Employee {
    private String name;
    protected Employee supervisor;
    public Employee() {
        this.name = "NoName";
    }
    public Employee getSupervisor() {
        System.out.println("Employee");
        return supervisor;
    }
}

class Manager extends Employee {
    @Override
    protected Manager getSupervisor() {    // Error: the overriding method has less visibility
        System.out.println("Manager");
        return (Manager)supervisor;
    }
}
```

# Programming Lab #10

# 10-01. Default Interface Methods

- The following code contains error. Fix the code so that it runs without error.

```
interface Person {
    String getName();
    default int getId() { return 0; }
}

interface Identified {
    default int getId() { return 1; }
}

class Employee implements Person, Identified {
    private String name;
    public Employee(String name) { this.name = name; }
    public String getName() { return this.name; }
}

public class Ex10_01 {
    public static void main(String[] args) {
        Employee m = new Employee("Peter");
        System.out.println(m.getId());
    }
}
```

# 10-02. Different Types of Methods in Interface

- Try running the code and understand the result.

```
interface CustomInterface {

    public abstract void method1();

    public default void method2() {
        method4();  // private method inside default method
        method5();  // static method inside other non-static method
        System.out.println("default method");
    }

    public static void method3() {
        method5();  // static method inside other static method
        System.out.println("static method");
    }

    private void method4() {
        System.out.println("private method");
    }

    private static void method5() {
        System.out.println("private static method");
    }
}
```

# 10-02. Different Types of Methods in Interface

- (continued)

```
public class Ex10_02 implements CustomInterface {

    public void method1() {
        System.out.println("abstract method");
    }

    public static void main(String[] args){
        CustomInterface instance = new Ex10_02();
        instance.method1();
        instance.method2();
        CustomInterface.method3();
    }
}
```

# 10-03. Method Overriding 1

- Try running the code and understand the result.

```
class Employee {
    private String name;
    private int salary;
    public Employee() {
        this.name = "NoName";
        this.salary = 0;
    }
    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }
    public int getSalary() { return this.salary; }
    public void setSalary(int salary) { this.salary = salary; }
}
class Manager extends Employee {
    private int bonus;
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() {
        return super.getSalary() + bonus;
    }
}
public class Ex10_03 {
    public static void main(String[] args) {
        Manager m = new Manager();
        System.out.println(m.getName() + " " + m.getSalary());
    }
}
```

# 10-04. Method Overriding 2

- Try running the code and understand the result.

```
class Employee {
    private String name;
    protected Employee supervisor;
    public Employee() {
        this.name = "NoName";
    }
    public boolean worksFor(Employee supervisor) {
        System.out.println("Employee.worksFor");
        return (this.supervisor == supervisor);
    }
}

class Manager extends Employee {
    public boolean worksFor(Manager supervisor) {
        System.out.println("Manager.worksFor");
        return (this.supervisor == supervisor);
    }
}
```

```
public class Ex10_04 {
    public static void main(String[] args) {
        boolean rv;
        Manager m = new Manager();
        Manager n = new Manager();
        Employee e = new Employee();
        rv = m.worksFor(n);
        rv = m.worksFor(e);
    }
}
```

# End of Class


Instructor office: AS818A

Email: jso1@sogang.ac.kr