8. Inheritance, Polymorphism, and Interfaces

[ITP20003] Java Programming

Agenda

- Inheritance Basics
- Programming with Inheritance
- **Polymorphism**
- Interfaces and Abstract Classes

Polymorphism

- Inheritance allows you to define a base class and derive classes from the base class

- Polymorphism allows you to make changes in the method definition for the derived classes (by overriding) and have those changes apply to methods written in the base class.

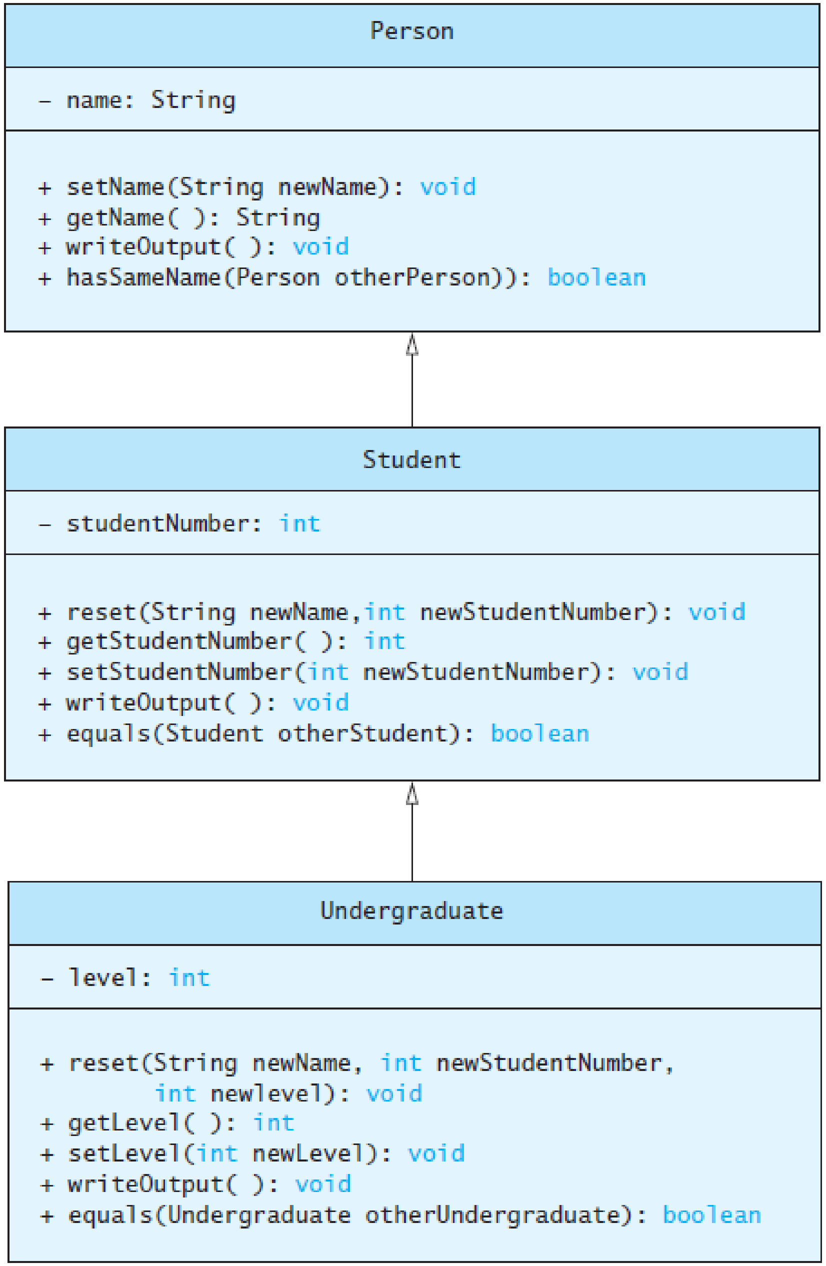
Polymorphism

- Consider an array of *Person*

  Person[] people = new Person[4];

- Since *Student* and *Undergraduate* are types of *Person*, we can assign them to *Person* variables

  people[0] = new Student(
                "DeBanque, Robin", 8812);
  people[1] = new Undergraduate(
                "Cotty, Manny", 8812, 1);



Person

- name: String

+ setName(String newName): void
+ getName( ): String
+ writeOutput( ): void
+ hasSameName(Person otherPerson)): boolean

Student

- studentNumber: int

+ reset(String newName,int newStudentNumber): void
+ getStudentNumber( ): int
+ setStudentNumber(int newStudentNumber): void
+ writeOutput( ): void
+ equals(Student otherStudent): boolean

Undergraduate

- level: int

+ reset(String newName, int newStudentNumber,
        int newlevel): void
+ getLevel( ): int
+ setLevel(int newLevel): void
+ writeOutput( ): void
+ equals(Undergraduate otherUndergraduate): boolean

Polymorphism

- Given:

```
Person[] people = new Person[4];
people[0] = new Student("DeBanque, Robin", 8812);
```

- When invoking:

```
people[0].writeOutput();
```

- Which writeOutput() is invoked, the one defined for *Student* or the one defined for *Person*?

An Inheritance as a Type

- The method can substitute one object for another.
  - Called polymorphism

- This is made possible by mechanism,
  - Dynamic binding
  - Also known as late binding

Dynamic Binding and Inheritance

- Static binding (or early binding)
  The method invocation is determined at compile time.

- Dynamic binding
  The method invocation is not bound to the method definition until the program executes
  (but at run time).

Dynamic Binding and Inheritance

- When an overridden method invoked
  - Action matches method defined in class used to create object using *new*
  - Not determined by type of variable naming the object

- Variable of any ancestor class can reference object of descendant class
  - Object always remembers which method actions to use for each method name.

Ex) // *Person* is an ancestor of *Undergraduate*.

```
Person a = new Undergraduate();
a.writeOutput();     // Undergraduate.writeOutput();
a.setLevel(3);       // error (Person does not have setLevel())
```

Write the following PolymorphismDemo.java, run it.

Polymorphism Example

```
public class PolymorphismDemo {
    public static void main(String[] args){
        Person[] people = new Person[4];
        people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
        people[1] = new Undergraduate("Kick, Anita", 9931, 2);
        people[2] = new Student("DeBanque, Robin", 8812);
        people[3] = new Undergraduate("Bugg, June", 9901, 4);

        for (int i=0; i < people.length; i++){
            Person p = people[i];
            p.writeOutput();
            System.out.println();
        }
    }
}
```

Polymorphism Example

- Output

```
Name: Cotty, Manny
Student Number: 4910
StudentLevel: 1

Name: Kick, Anita
Student Number: 9931
StudentLevel: 2

Name: DeBanque, Robin
Student Number: 8812

Name: Bugg, June
Student Number: 9901
StudentLevel: 4
```

Dynamic Binding and Inheritance

- When an overridden method is invoked, its action is the one defined in the class used to create the object using the new operator.

- It is not determined by the type of the variable naming the object. A variable of any ancestor class can reference an object of a descendant class, but the object always remembers which method actions to use for every method name.

- The type of the variable does not matter. What matters is the class name when the object was created.

- This is because Java uses dynamic binding.

Agenda

- Inheritance Basics
- Programming with Inheritance
- Polymorphism
- **Interfaces and Abstract Classes**

Class Interfaces

- Let’s imagine a person calling her pets to dinner by whistling. Each animal responds in its own way.

- For the pets we can specify their common behaviors.
  - Be named
  - Eat
  - Respond to a command

- Maybe…

```
/** Sets a pets name to petName. */
public void setName(String petName)

/** Returns true if a pet eats the given food.*/
public boolean eat(String food)

/** Returns a description of a pet’s response to
the given command. */
public String respond(String command)
```

Class Interfaces

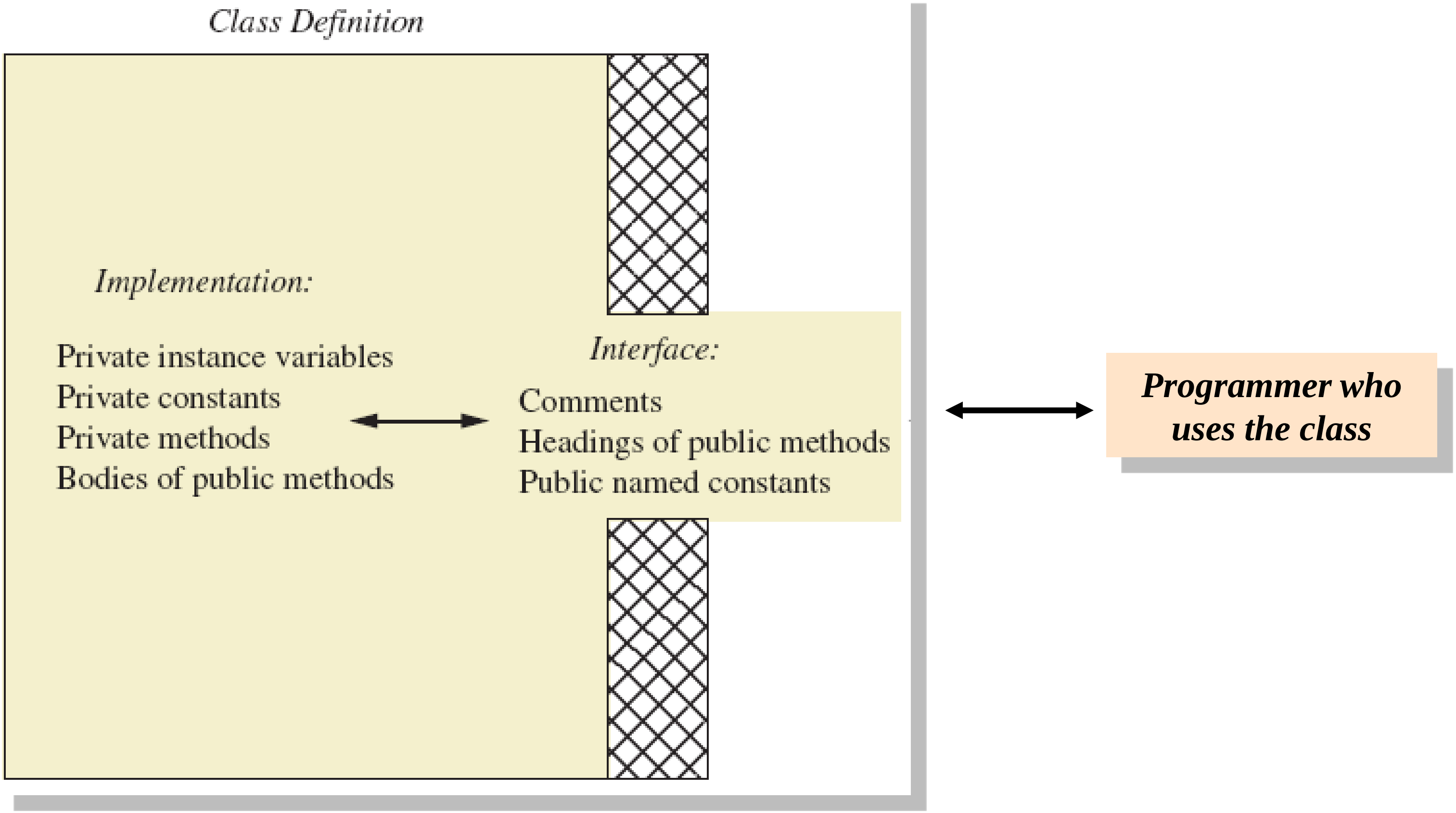
- We can create the objects for the pets.
- Each object can be named, can eat, and can respond.
- For example, although dogs, birds, and fish respond to a command, the way they respond differs.

- We can use the following statement for the pets,

  ```
  String response = myPet.respond("Come!");
  ```

- The value of the string response, however, differs according to the type of object that myPet names.

Class Definition vs. Interfaces



Class Definition

Implementation:

Private instance variables
Private constants
Private methods
Bodies of public methods

Interface:

Comments
Headings of public methods
Public named constants

Programmer who uses the class

Class Interfaces

- Now consider different classes that implement this interface.
  - They will have the same behaviors.
  - Nature of the behaviors will be different.

- Each of the classes implements the behaviors/methods differently.

Class Interfaces

SYNTAX

```
public interface Interface_Name
{
    Public_Named_Constant_Definitions
    . . .
    Public_Method_Heading_1;
    . . .
    Public_Method_Heading_n;
}
```

EXAMPLE

```
/**
An interface of static methods to convert measurements
between feet and inches.
*/
public interface Convertible
{
    public static final int INCHES_PER_FOOT = 12;
    public static double convertToInches(double feet);
    public static double convertToFeet(double inches);
}
```

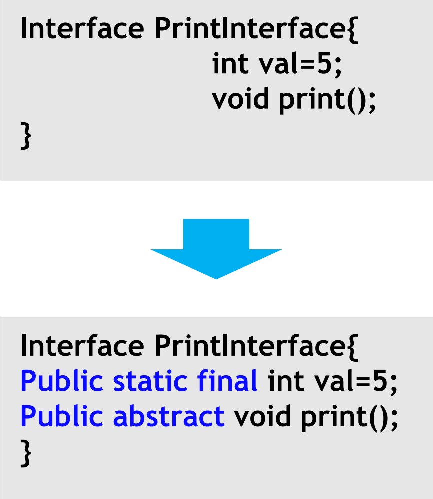
Java Interfaces

- Interface name begins with uppercase letter
- Stored in a file with suffix .java
- Interface does not include
  - Declarations of constructors
  - Instance variables
  - Method bodies

Class Interfaces

- A **Java interface** is a program component that contains the headings for a number of public methods.

- You can define your interface that can contain Methods, Named constant, comments etc.
  - Methods should be public, abstract.
  - Variables should be public, static and final.



```
Interface PrintInterface{
            int val=5;
            void print();
}
```

```
Interface PrintInterface{
Public static final int val=5;
Public abstract void print();
}
```

- Of course, Java Class Library contains interfaces that are already written.

Implementing an Interface

- When you write a class that defines the methods declared in an interface, we say that the class **implements the interface.**

- A class that implements an interface must define a body for every method that the interface specifies.

Declare in an interface

Define in a class

Implementing an Interface

- Two steps to implement an interface.
- In your class,
  step1: Include the phrase

```
        implements Interface_Name
```

  step2: Define each method declared in the interface(s)

Interfaces Help Designers and Programmers

- Writing an interface is a way for a class designer to specify methods for another programmer.
- Implementing an interface is a way for a programmer to guarantee that a class defines certain methods.

Interface as a Type

- You can declare a variable of an Interface type.

  Ex)
  ```
  // Rectangle implements Measurable
     Measurable a = new Rectangle(100, 200);
     System.out.println("a.getArea() = " + a.getArea());
  ```

- You cannot create an object of an Interface type using the new operator.
  - An Interface cannot have a Constructor.

  Ex)
  ```
  Measurable a = new Measurable ();        // error
  ```

Implementing an Interface

- Different classes can implement the same interface, perhaps in different ways.

- For example, many classes can implement the interface `Measurable` and provide their own version of the methods `getPerimeter` and `getArea`.

```
/**
An interface for methods that return
the perimeter and area of an object.
*/
public interface Measurable {
    /** Returns the perimeter. */
    public double getPerimeter();
    /** Returns the area. */
    public double getArea();
}
```

Write the following classes, run it.

(1)

```
public interface Measurable {
   public double getPerimeter();
   public double getArea();
}
```

(2)

```
public class Circle implements Measurable{
   private double myRadius;
   public Circle(double radius){
     myRadius = radius;
   }
   public double getPerimeter(){
     return 2 * Math.PI * myRadius;
   }
   public double getArea(){
     return Math.PI * myRadius * myRadius;
   }
   public double getCircumference(){
     return getPerimeter();
   }
}
```

(3)

```
public class Rectangle implements Measurable{
  private double myWidth;
  private double myHeight;
  public Rectangle(double width, double height){
    myWidth = width;  myHeight = height;
  }
  public double getPerimeter(){
    return 2 * (myWidth + myHeight);
  }
  public double getArea(){
    return myWidth * myHeight;
  }
}
```

Write the following classes, run it.

(4)

```
public class DisplayDemo {
    public static void main(String[] args) {
        Measurable box     = new Rectangle(5.0, 5.0);
        Measurable disc    = new Circle(5.0);
        display(box);      display(disc);
    }

    public static void display(Measurable figure){
        double perimeter   = figure.getPerimeter();
        double area        = figure.getArea();
        System.out.println("Perimeter = " + perimeter
                           +"; area = " + area);
    }
}
```

Implementing an Interface

```java
public static void main(String[] args) {
    Measurable m;
    Rectangle box = new Rectangle(5.0, 5.0);
    m = box;
    display(m);
    Circle disc = new Circle(5.0);
    m = disc;
    display(m);
}
```

Result

```
Perimeter = 20.0; area = 25.0
Perimeter = 31.41592653589793;
area = 78.53981633974483
```

- The invocations of `getPerimeter` and `getArea` within display are identical. Yet these invocations use different definitions for `getPerimeter` and `getArea`, and so the two invocations of display produce different output, just as they did in our earlier example.
- **Dynamic binding**
  A variable of an interface type can reference an object of a class that implements the interface, but the object itself always determines which method actions to use for every method name.

Implementing an Interface

- You can assign an object of type Circle to a variable of type Measureable.

- But you cannot call `getcircumference`(). Why?

```
Measurable m = new Circle(5.0);
System.out.println(m.getCircumference()); //ILLEGAL!
```

- Because `getCircumference()` is not the name of a method in the Measurable interface. (not exist)

- The following statement is correct.

```
Circle c = (Circle)m;
System.out.println(c.getCircumference());//Legal
```

What is legal and what happens

- Please remember that

  A variable's type (ex: `Measureable`) determines what method names can be used, but the object (ex: `Circle`) the variable references determines which definition of the method will be used.

Interface and Polymorphism

- Dynamic binding applies to interfaces just as it does with classes.

- The process enables objects of different classes to substitute for one another, if they have the same interface.

- This ability—called polymorphism—allows different objects to use different method actions for the same method name.

Comparable interface

- Java has many predefined interfaces that are used by many classes. One of them is the Comparable interface, and it is used to impose an ordering upon the objects that implement it.

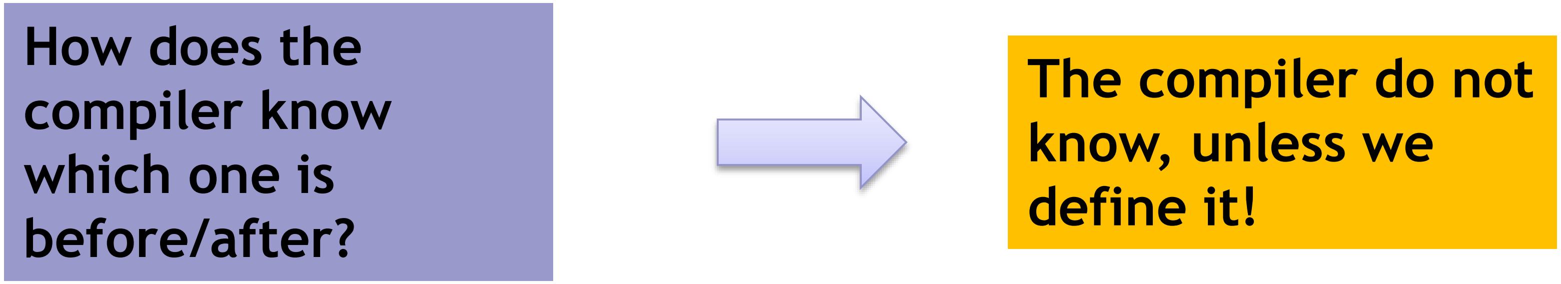
- The Comparable interface has only one method heading. The method compareTo() must be written for a class to implement the Comparable interface.

```
public int compareTo(Object other);
```

Comparable interface

The compareTo method should return

- A negative number (< 0) if the calling object “comes before” the parameter other.
- A zero (= 0) if the calling object “equals” the parameter other.
- A positive number (0 <) if the calling object “comes after” the parameter other.



How does the compiler know which one is before/after?

The compiler do not know, unless we define it!

Write the following classes, run it.

```
public class Fruit{
    private String fruitName;

    public Fruit()                     {fruitName = "";}
    public Fruit(String name)          {fruitName = name;}
    public void setName(String name) {fruitName = name;}
    public String getName()            {return fruitName;}
}
```

```
import java.util.Arrays;
public class FruitDemo{
    public static void main(String[] args){
        Fruit[] fruits = new Fruit[4];
        fruits[0]     = new Fruit("Orange");
        fruits[1]     = new Fruit("Apple");
        fruits[2]     = new Fruit("Kiwi");
        fruits[3]     = new Fruit("Durian");
        Arrays.sort(fruits);
        // Output the sorted array of fruits
        for (Fruit f : fruits)
                { System.out.println(f.getName());}
    }
}
```

Revised Fruit Class (using Comparable interface)

```java
public class Fruit implements Comparable
{
    private String fruitName;
    public Fruit(){fruitName = "";}
    public Fruit(String name){fruitName = name;}
    public void setName(String name){fruitName = name;}
    public String getName(){return fruitName;}

    public int compareTo(Object o){
        if ((o != null) && (o instanceof Fruit)){
            Fruit otherFruit = (Fruit) o;
            return (fruitName.compareTo(otherFruit.fruitName));
        }
        return -1; // Default if other object is not a Fruit
    }
}
```

Results

```
Apple
Durian
Kiwi
Orange
```

Abstract Classes

- Remember the *final* keyword?
  If you want to specify that a method definition cannot be overridden by a new definition within a derived class, you can use the final modifier.
  `public final void specialMethod()`

- If you plan to override a method later, use ‘abstract’.
  With the abstract keyword, just write only the method heading without definition.
  `public abstract void reservedMethod();`

Abstract Classes

```java
public abstract class PersonAbstract
{
    private String name;
    public PersonAbstract(){
        name = "No name yet";
    }
    public PersonAbstract(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public abstract void writeOutput();
}
```

Abstract Classes

- You cannot create an object of an abstract class.
- For example, given the abstract class PersonAbstract, the following statement is illegal:

```
PersonAbstract p = new PersonAbstract(); // NO
```

Abstract Classes

- Not all methods of an abstract class are abstract methods
- Abstract class makes it easier to define a base class
  - Specifies the obligation of designer to override the abstract methods for each subclass
- Cannot have an instance of an abstract class
  - But OK to have a parameter of that type

Write the following classes, run it.

Inherit abstract class and implement interface

P11_PersonAbstract.java

```java
public abstract class PersonAbstract
{
    private String name;
    public P11_PersonAbstract(){
        name = "No name yet";
    }
    public P11_PersonAbstract(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public abstract void writeOutput();
}
```

P11_StudentInterface.java

```java
public interface P11_StudentInterface {
        void learn();
        void study();
}
```

Write the following classes, run it.

Inherit abstract class and implement interface

P11_Student.java

```
public class P11_Student extends P11_PersonAbstract implements P11_StudentInterface {
     private int studentNumber;
     private String courseName;

     public void learn() {System.out.println("I am taking " + courseName + " course.");}
     public void study() {System.out.println("I am studying " + courseName);}

     public P11_Student(){    super();
                              System.out.println("Student()");
                              studentNumber = 0;}
     public P11_Student(String inName, int inSNum, String inCourse){
                              super(inName);
                              studentNumber = inSNum;
                              courseName = inCourse;               }

     public void writeOutput(){
                              System.out.println("Name:  " + getName());
                              System.out.println("Student Number:  " + studentNumber);

                              learn();  study();}
}
```

Write the following classes, run it.

Inherit abstract class and implement interface

P11_Student_main.java

```

public class P11_Student_main {

public static void main(String[] args) {
    P11_Student sman = new P11_Student("Super Man", 19380001,
                                  "Java Programming");
    sman.writeOutput();
}
}
```

Inherit abstract class and implement interface

```
Name: Super Man
Student Number: 19380001
I am taking Java Programming course.
I am studying Java Programming course.
```

Add another interface, and run the "P11_Student_main" again.

Implement multiple interfaces

P11_KoreanInterface.java

```
public interface P11_KoreanInterface {
        void saykorean();
}
```

P11_Student.java   (revised)

```
public class P11_Student extends P11_PersonAbstract
                            implements P11_StudentInterface, P11_KoreanInterface {
    private int studentNumber;
    private String courseName;

    public void learn() {System.out.println("I am taking " + courseName + " course.");}
    public void study() {System.out.println("I am studying " + courseName);}

    public P11_Student(){…}
    public P11_Student(String inName, int inSNum, String inCourse){…}

    public void writeOutput(){
                            System.out.println("Name: " + getName());
                            System.out.println("Student Number: " + studentNumber);

                            learn(); study(); saykorean();}
    public void saykorean() {System.out.println("I am a Korean.");}
}
```

Implement multiple interfaces

```
Name: Super Man
Student Number: 19380001
I am taking Java Programming course.
I am studying Java Programming
I am a Korean
```