15, 17, 18. Inheritance, Polymorphism, and Interfaces

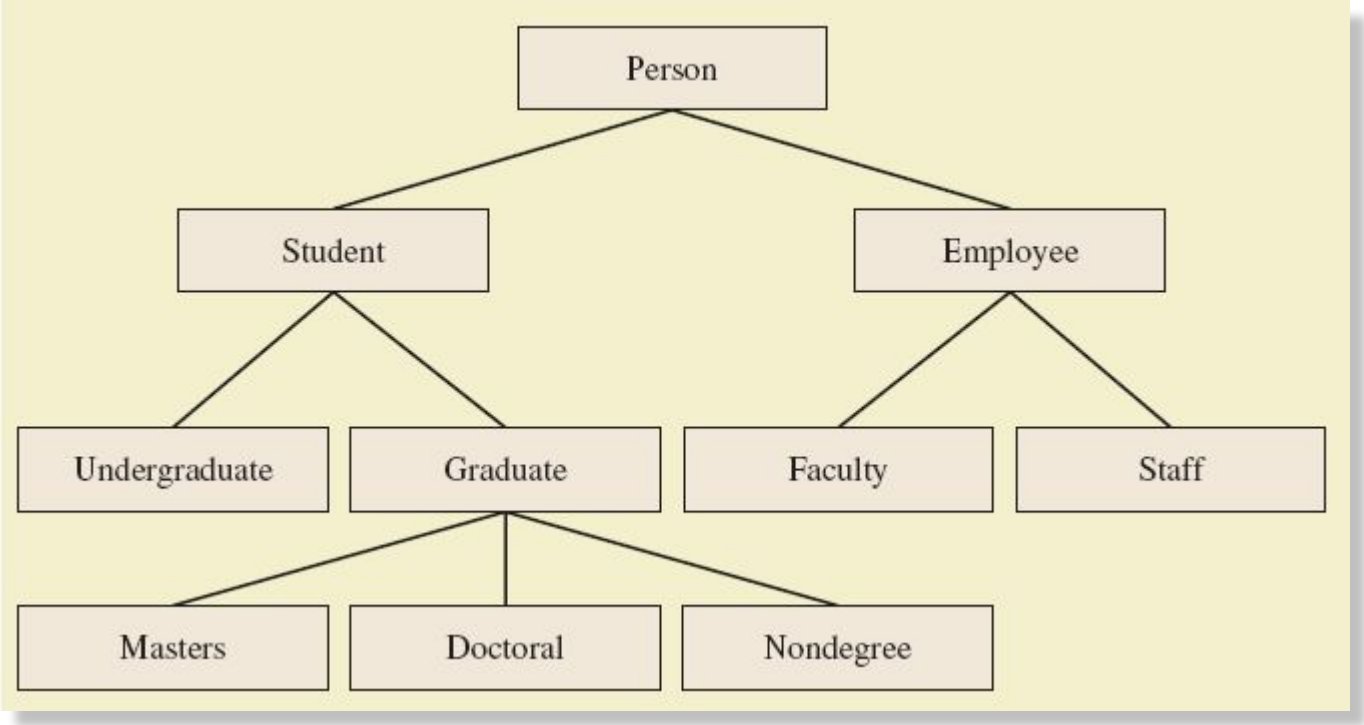
[ECE20016/ITP20003] Java Programming

Agenda

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

Class Hierarchy

- A class hierarchy



Inheritance Basics

- Inheritance allows programmer to define a general class.
- Later you define a more specific class
  - Add new details to general definition
  - New class inherits all properties of initial, general class
- class *Person*
  - https://github.com/lifove/InheritanceExample/blob/master/src/main/java/edu/handong/csee/java/example/inheritance/Person.java

Derived Classes

- Class *Person* used as a base class
  - Also called superclass
- Now we declare derived class *Student*
  - Also called subclass
  - Inherits methods from the superclass
- class *Student* **extends *Person***
  - https://github.com/lifove/InheritanceExample/blob/master/src/main/java/edu/handong/csee/java/example/inheritance/Student.java
- class InheritanceDemonstrator
  - https://github.com/lifove/InheritanceExample/blob/master/src/main/java/edu/handong/csee/java/example/inheritance/InheritanceDemonstrator.java

```
Name: Warren Peace
Student Number: 1234
```

Overriding Method Definitions

- Note method writeOutput in class Student
  - Class Person also has method with that name
- Method in subclass with same signature overrides method from base class
  - Overriding method is the one used for objects of the derived class
  - Overriding method must return same type of value

Overriding vs. Overloading

- Overriding takes place in subclass – new method with same signature

- Overloading
  - New method in same class with different signature

final Modifier for methods/classes

- Possible to specify that a method cannot be overridden in subclass
- Add modifier final to the heading

  Ex) public final void specialMethod()
- An entire class may be declared final
  - Thus cannot be used as a base class to derive any other class.

Private Instance Variables, Methods

- Consider private instance variable in a base class
  - It is not inherited in subclass
  - It can be manipulated only by public accessor, modifier methods
- Similarly, private methods in a superclass not inherited by subclass

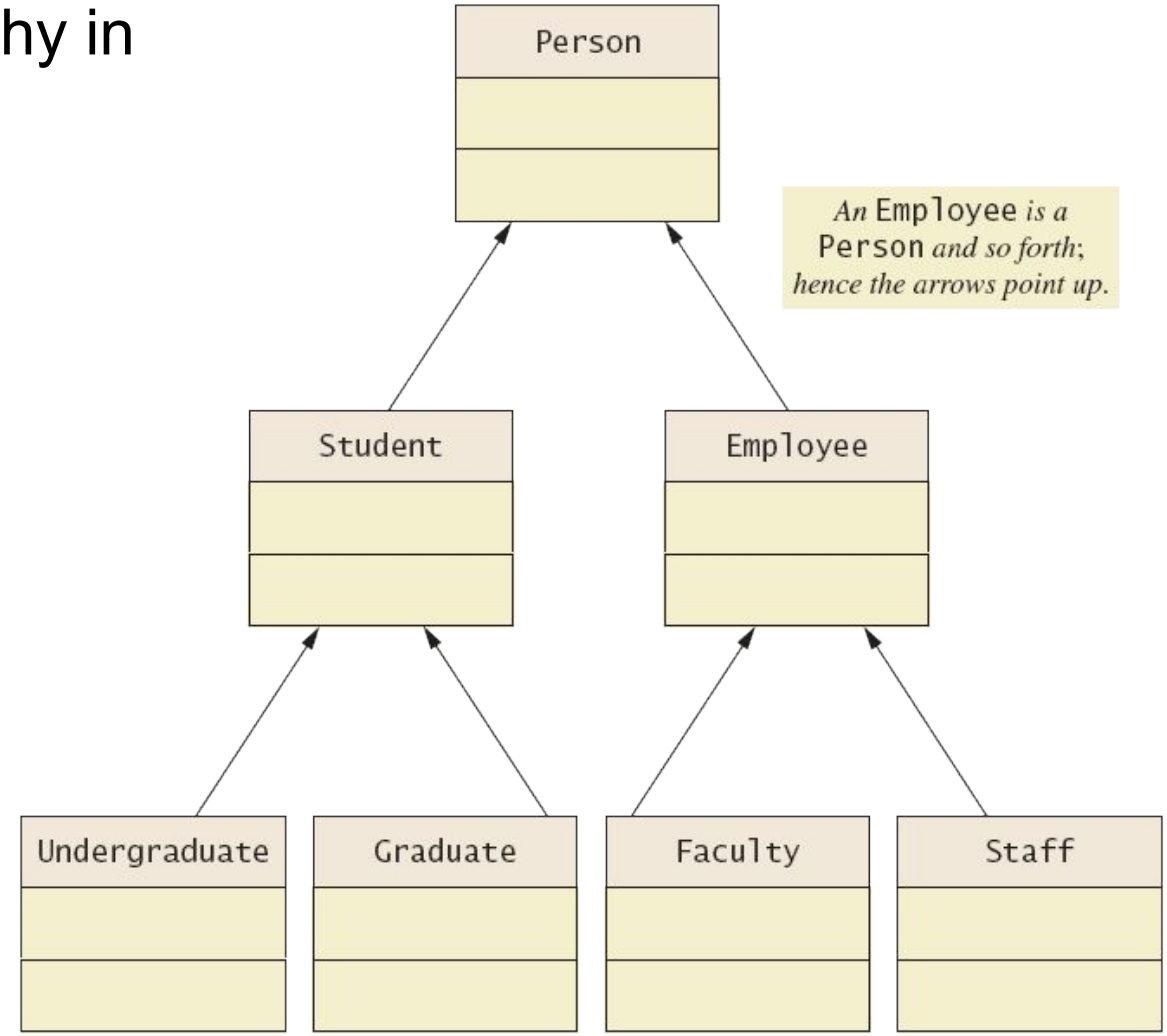
Visibility	Java Syntax	UML Syntax
public	`public`	+
protected	`protected`	#
package		~
private	`private`	-

Recall access modifiers for method

Scope \ Modifiers	public	protected	(default)	private
Same Class	O	O	O	O
Same Package	O	O	O	
Sud(derived) Class	O	O		
World	O			

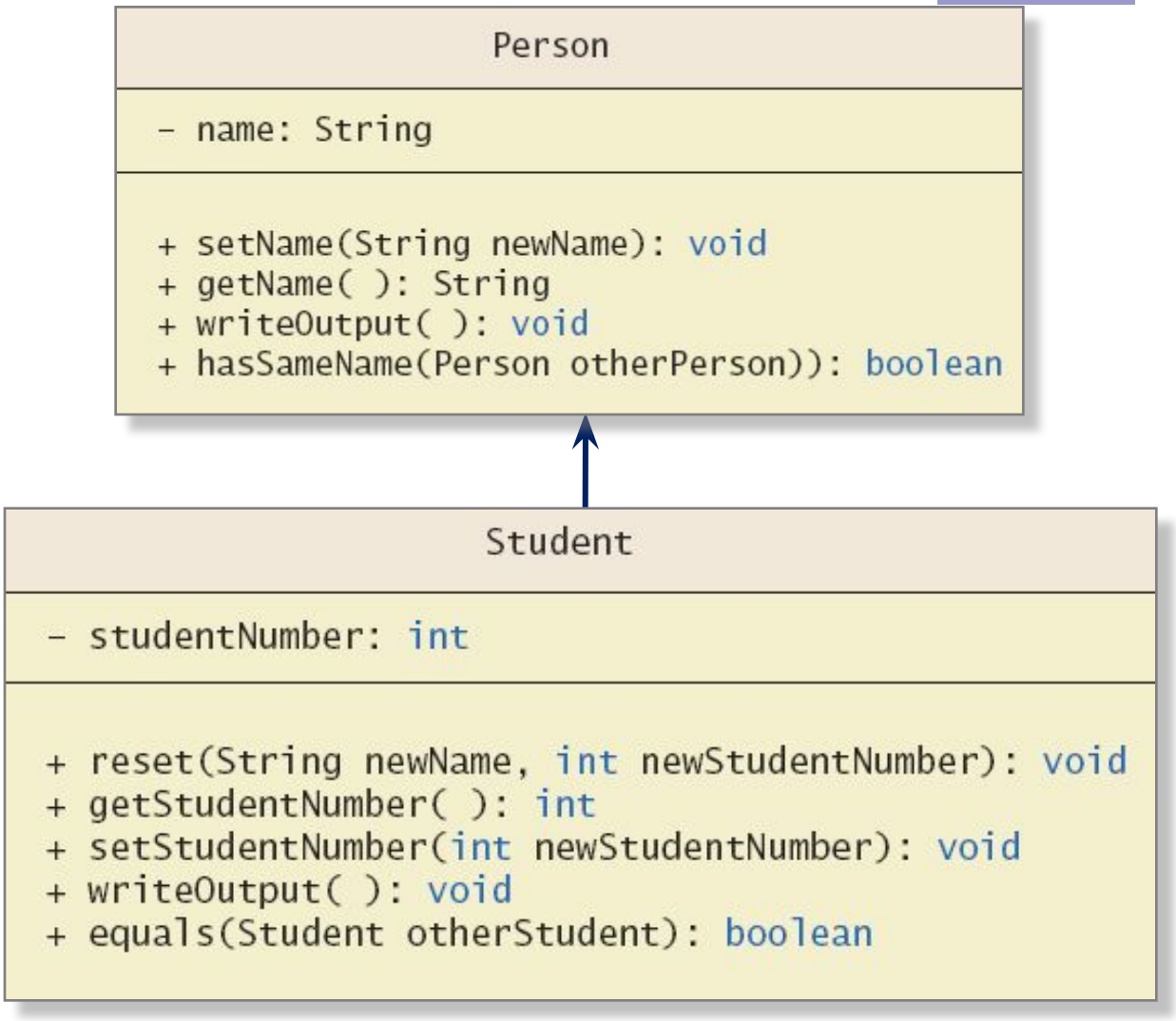
UML Inheritance Diagrams

- A class hierarchy in UML notation



UML Inheritance Diagrams

- Some details of UML class hierarchy



Agenda

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

Constructors in Derived Classes

- A derived class does not inherit constructors from base class
  - Constructor in a subclass must invoke constructor from base class

- Use the reserved word *super*

```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

  - Must be first action in the constructor

The *this* Method – Again

- Also possible to use the *this* keyword
  - Use to call any constructor in the class

```
public Person()
{
    this("No name yet");
}
```

  - Calling constructor using method name is not allowed in Java.
    Ex) Person(“No name yet”); // not valid.
  - Calling constructor from other methods is not allowed.
- When used in a constructor, this calls constructor in same class.
  - Contrast use of *super* which invokes constructor of base class

Calling an Overridden Method

- Reserved word *super* can also be used to call method in overridden method

```
public void writeOutput()
{
    super.writeOutput(); //Display the name
    System.out.println("Student Number: " + studentNumber);
}
```

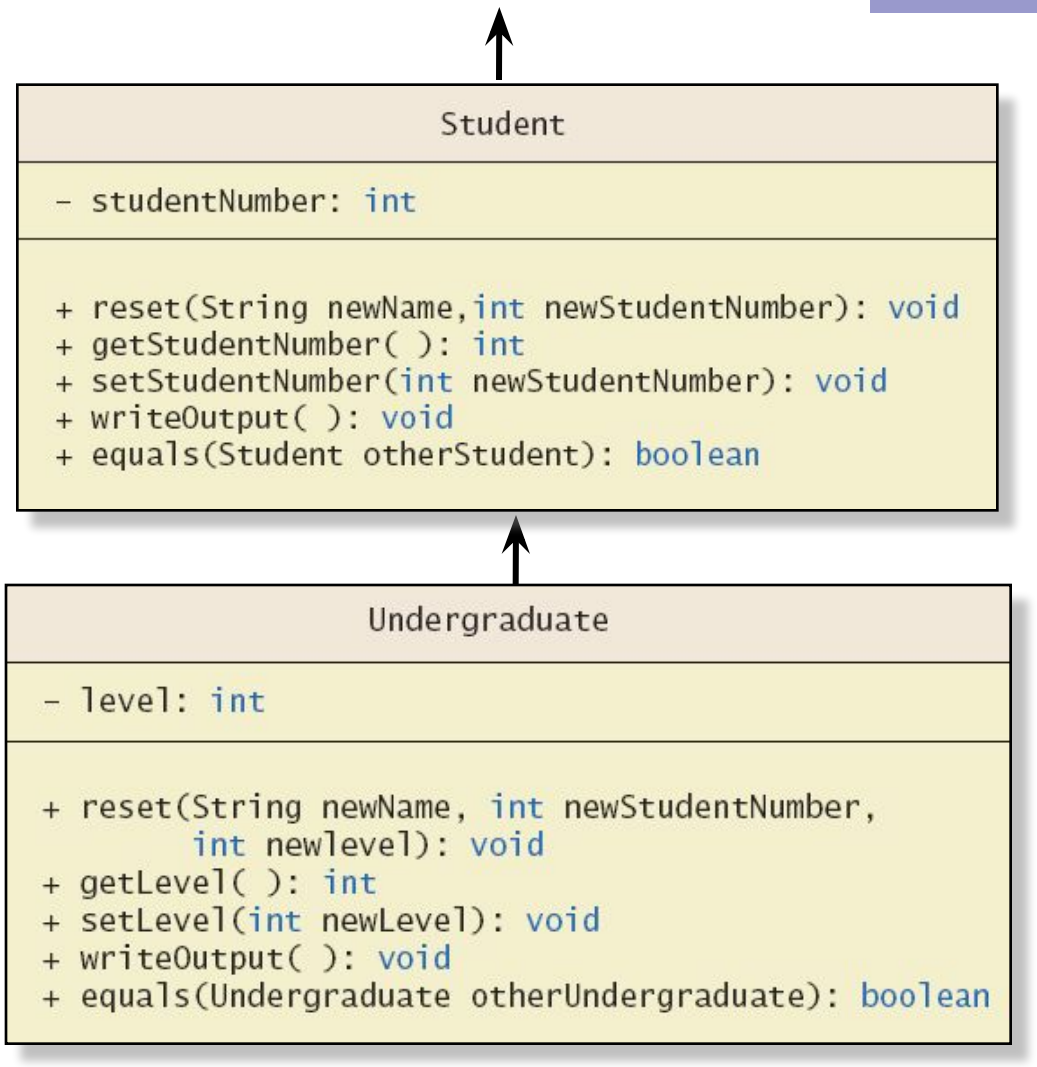
  - Calls method by same name in base class

Programming Example

- A derived class of a derived class
- class *Undergraduate*
  - https://github.com/lifove/InheritanceExample/blob/master/src/main/java/edu/handong/csee/java/example/inheritance/Undergraduate.java
- Same as having all public members of both *Person* and *Student* classes even though Undergraduate does not have such pubic methods in it.
  - getName() in *Person*
  - getStudentNumber() in *Student*
- This reuses the code in super classes

Programming Example

- More details of the UML class hierarchy



Type Compatibility

- In the class hierarchy
  - Each *Undergraduate* is also a *Student*
  - Each *Student* is also a *Person*

- An object of a derived class can serve as an object of the base class.
  - Note this is not typecasting
  - An object of a class can be referenced by a variable of an ancestor type.

Type Compatibility

- Be aware of the "is-a" relationship

  Ex) A *Student* is a *Person*

- Another relationship is the "has-a"
  - A class can contain (as an instance variable) an object of another type.
  - If we specify a date of birth variable for *Person* – it "has-a" *Date* object

The Class *Object*

- Java has a class that is the ultimate ancestor of every class
  - The class *Object*
- Thus possible to write a method with parameter of type *Object*
  - Actual parameter in the call can be an object of any type

Ex) println(Object theObject)

The Class *Object*

- Class *Object* has some methods that every Java class inherits

  Ex) equals, toString

- Method toString called when println(*theObject*) invoked
  - Best to define your own toString to handle this.

A Better *equals* Method

- The *Student* class has a equal method

```
public boolean equals (Student otherStudent) // overloaded equals method
{
    return this.hasSameName (otherStudent) &&
        (this.studentNumber == otherStudent.studentNumber);
}
```

- Programmer of a class should override method equals from Object

```
public boolean equals (Object otherObject)
{
    boolean isEqual = false;
    if ((otherObject != null) && (otherObject instanceof Student)){
        Student otherStudent = (Student) otherObject;
        isEqual = this.sameName (otherStudent)
                && (this.studentNumber == otherStudent.studentNumber);
    }

    return isEqual;
}
```

Agenda

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

Polymorphism

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

- Polymorphism can let an object have many different forms. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

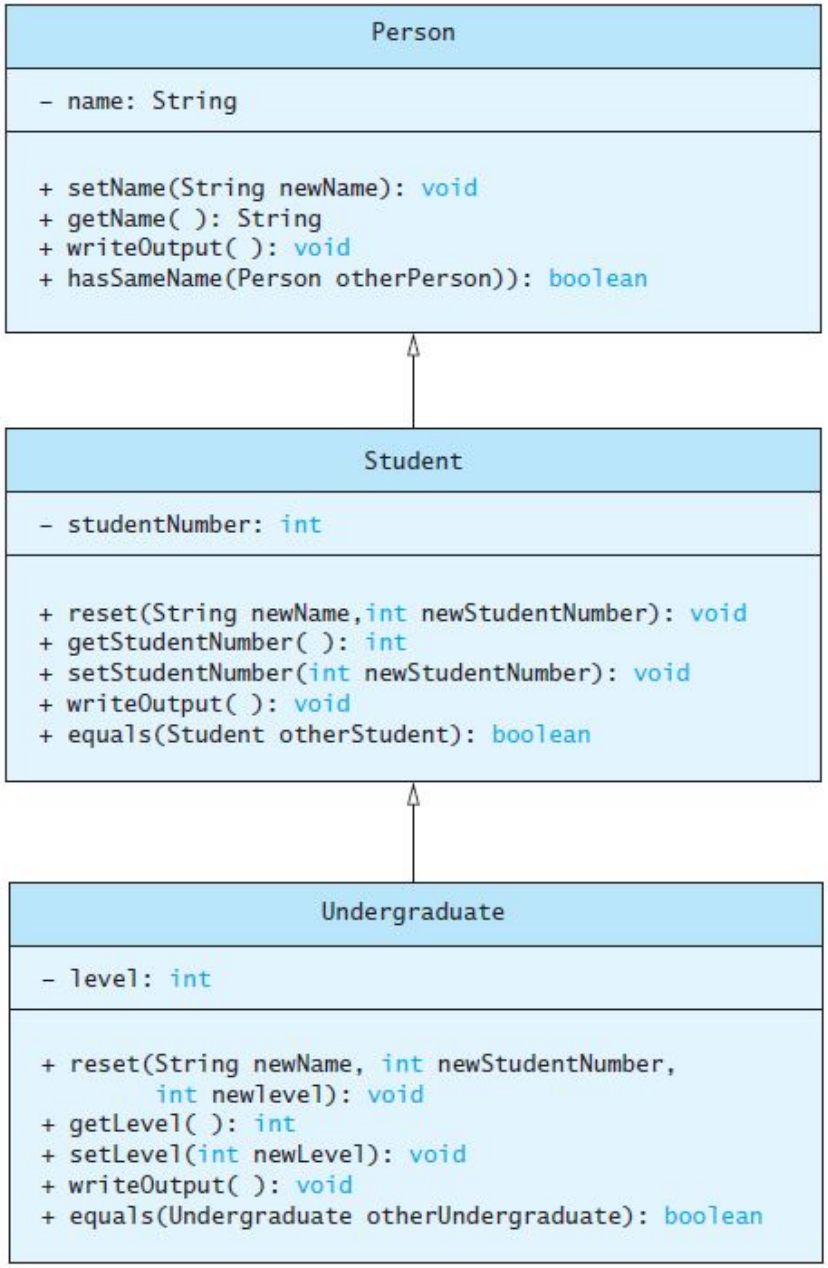
Polymorphism

- 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);



Polymorphism Examples

```
Undergraduate ug = new Undergraduate("Nam, JC", 1111, 4);
Student st = ug;
Person ps = ug;
Object obj = ug;
```

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*?
  - Answer: The one defined for *Student*

An Inheritance as a Type

- The method can substitute one object for another.
  - Called polymorphism
    - e.g. call writeOutput() of a Person object (another) but writeOutput() of a sub class (one object) is actually called.

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

Dynamic Binding and Inheritance

- **When an overridden method invoked**
  - Action matches method defined in class used to create object using *new* *(i.e., actual instance!)*
  - Not determined by type of variable naming the object

- Variable of any ancestor class can have reference the 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())
```

Polymorphism Example

```
public class PolymorphismDemonstrator
{
	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 (Person p : people) {
			p.writeOutput();
			System.out.println();
		}
	}
}
```

Polymorphism Example

- Output

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

Name: Kick, Anita
Student Number: 9931
Student Level: 2
```

```
Name: DeBanque, Robin
Student Number: 8812

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

instanceof: Check a type of objects

```
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 (Person p : people) {
                System.out.println("Student Name: " + p.getName());

                // we can call getLevel only in Undergraduate by casting p (Person).
                // However, before cating we need to check if p is actually Undergraduate type by using
'instanceof'
                if(p instanceof Undergraduate){
                        Undergraduate studentObj = (Undergraduate) p;
                        System.out.println("Student Level: " + studentObj.getLevel());
                }
                System.out.println();
        }
}
```

Agenda

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

Class Interfaces

- Consider a set of behaviors for pets
  - Be named
  - Eat
  - Respond to a command
- We could specify method headings for these behaviors
- These method headings can form a class interface

Class Interfaces

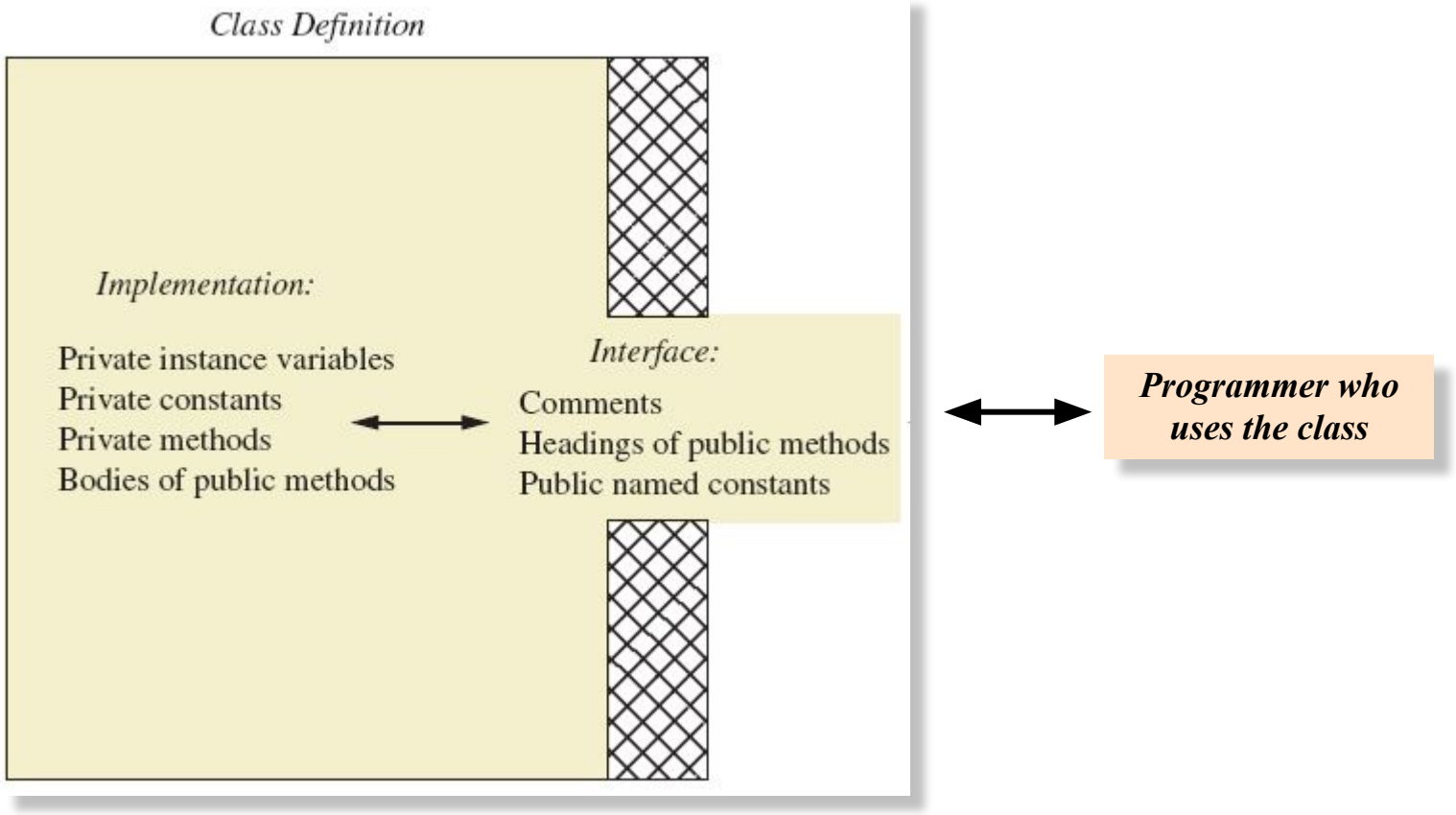
Without interfaces, we can still implement our own programs!

Then why do we need such method headings?????

Interfaces are contracts that spell out how their software interacts

https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html

Class Definition vs. Interfaces



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.

Java Interfaces

- A program component that contains headings for a number of public methods
  - Will include comments that describe the methods
- Interface can also define public named constants

Ex)

```
public interface Measurable
{
    /** Returns the perimeter. */
    public double getPerimeter ();
    /** Returns the area. */
    public double getArea ();
}
```

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

Implementing an Interface

- To implement a method, a class must
  - Include the phrase
    implements *Interface_name*
  - Define each specified method

- class Rectangle implements Measurable

- class Circle implements Measurable

https://github.com/lifove/InterfaceExample/tree/master/src/main/java/edu/handong/csee/java/example

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

Interface as a Type

- Possible to write a method that has a parameter as an interface type
  - An interface is a reference type
    - public void myMethod(Measurable measure) { … }
- An object of any class which implements that interface can be an argument for the method.
  - Measurable myMeasurable = new Rectangle(100,200);
    myMethod(myMeasurable);

Extending an Interface

- Possible to define a new interface which builds on an existing interface
  - It is said to extend the existing interface
  - public interface MeasurableForVolume extends Measurable {
    public double getVolume();
    }
- A class that implements the new interface must implement all the methods of both interfaces
  - e.g. ,getPerimeter(), getArea(), getVolume()

Implement a built-in interface

- Java has many **predefined interfaces**
- One of them, the Comparable interface, is used to impose an ordering upon the objects that implement it
  - A class implementing Comparable can be sorted by Arrays.sort() method.
- Requires that the method compareTo be written

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

Comparable example

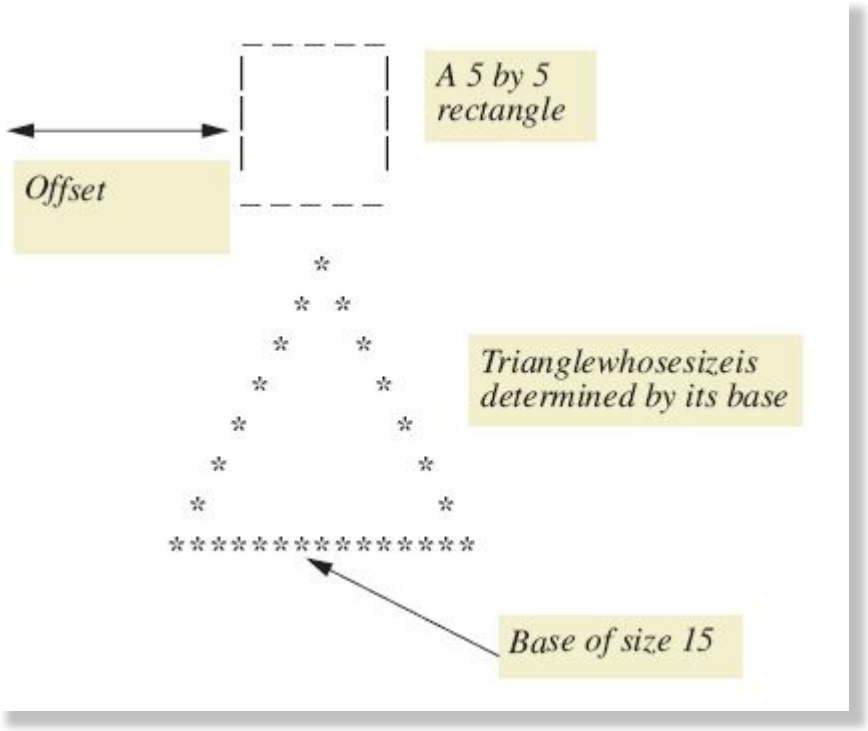
- Apply Arrays.sort(...) for Salesman
  - https://github.com/lifove/SalesReporter/tree/master/src/main/java/edu/handong/csee/java/example

Case Study

- Character Graphics
- interface ShapeInterface
- If we wish to create classes that draw rectangles and triangles
  - We could create interfaces that extend ShapeInterface

Case Study

- A sample rectangle and triangle



Base Interface *ShapeInterface*

```
public interface ShapeInterface
{
    /**
    Sets the offset for the shape.
    */
    public void setOffset (int newOffset);
    /**
    Returns the offset for the shape.
    */
    public int getOffset ();
    /**
    Draws the shape at lineNumber lines down
    from the current line.
    */
    public void drawAt (int lineNumber);
    /**
    Draws the shape at the current line.
    */
    public void drawHere ();
}
```

Derived Interfaces

```
/**
Interface for a rectangle to be drawn on the screen.
*/
public interface RectangleInterface extends ShapeInterface
{
    /**
    Sets the rectangle's dimensions.
    */
    public void set (int newHeight, int newWidth);
}
/**
Interface for a triangle to be drawn on the screen.
*/
public interface TriangleInterface extends ShapeInterface
{
    /**
    Sets the triangle's base.
    */
    public void set (int newBase);
}
```

Case Study

- A base class which uses (implements) previous interfaces
  class ShapeBasics

- Note
  - Method drawAt calls drawHere
  - Derived classes must override drawHere
  - Modifier extends comes before implements

Case Study

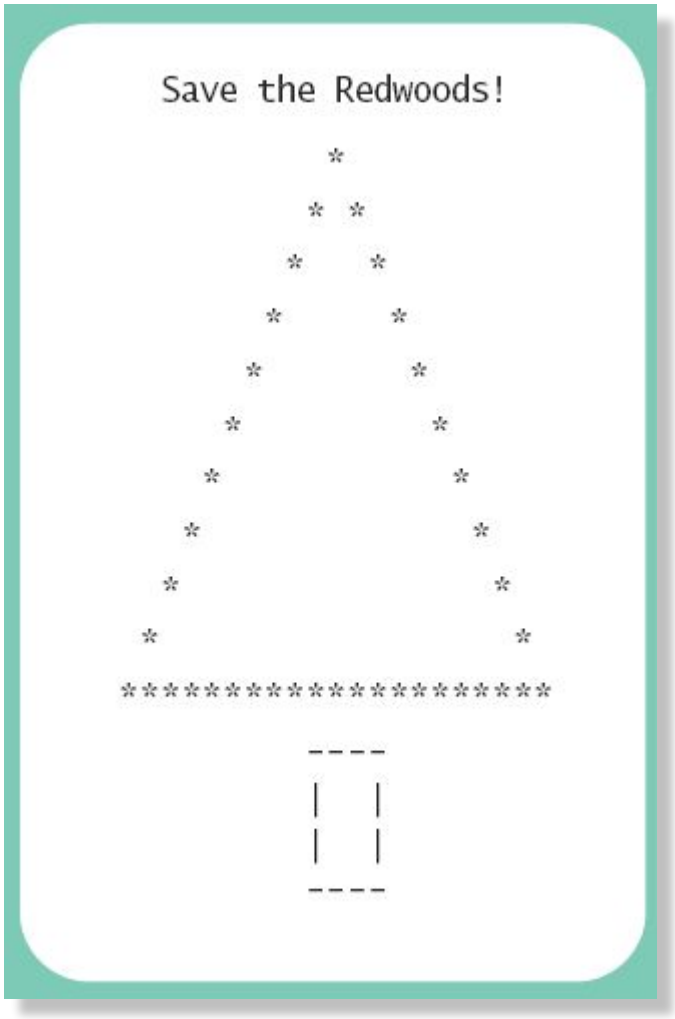
- Note algorithm used by method drawHere to draw a rectangle
  - Draw the top line
  - Draw the side lines
  - Draw the bottom lines
- Subtasks of drawHere are realized as private methods
- class Rectangle

Case Study

- class Triangle
- It is a good practice to test the classes as we go
- class TreeDemonstrator

https://github.com/lifove/InterfaceExample/tree/master/src/main/java/edu/handong/csee/java/example/keyboard/characters

Case Study



Sorting an Array of Fruit Objects

- Initial (non-working) attempt to sort an array of *Fruit* objects
- class *Fruit* and class *FruitDemontrator*
  - https://github.com/lifove/InterfaceExample/tree/master/src/main/java/edu/handong/csee/java/example/comparable/without
- Result: Exception in thread “main”
  - Sort tries to invoke *compareTo* method but it doesn’t exist

Sorting an Array of Fruit Objects

- Working attempt to sort an array of Fruit objects – implement *Comparable*, write *compareTo* method

- class *Fruit*
  - https://github.com/lifove/InterfaceExample/blob/master/src/main/java/edu/handong/csee/java/example/comparable/Fruit.java

compareTo Method

- An alternate definition that will sort by length of the fruit name

```
public int compareTo(Fruit otherFruit) {
        if(fruitName.length() > otherFruit.fruitName.length())
                return 1;
        else if (fruitName.length() < otherFruit.fruitName.length())
                return -1;
        else
                return 0;
}
```

Agenda

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

Abstract Classes

- We can *add method headings without implementation of its body* in a super class by defining it as an ***abstract class***.

- Class *ShapeBasics* is designed to be a base class for other classes
  - Method drawHere will be redefined for each subclass
  - It should be declared abstract – a method that has no body

- This makes the class abstract

- You cannot create an object of an abstract class – thus its role as base class.

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

Abstract Classes

https://github.com/lifove/InterfaceExample/blob/master/src/main/java/edu/handong/csee/java/example/keyboard/characters/ShapeBase.java

```
public abstract class ShapeBase implements ShapeInterface
{
    private int offset;
    public abstract void drawHere ();
    /*
    The rest of the class is identical to ShapeBasics, except for the
    names of the constructors. Only the method drawHere is abstract.
    Methods other than drawHere have bodies and do not have the keyword
    abstract in their headings.
    We repeat one such method here:
    */
    public void drawAt (int lineNumber)
    {
        for (int count = 0 ; count < lineNumber ; count++)
            System.out.println ();
        drawHere ();
    }
}
```

Dynamic Binding and Inheritance

- Note how drawAt(in ShapeBasics) makes a call to drawHere
- Class Rectangle overrides method drawHere
  - How does drawAt know where to find the correct drawHere?
- Happens with dynamic or late binding
  - Address of correct code to be executed determined at run time