8. Inheritance, Polymorphism, and Interfaces

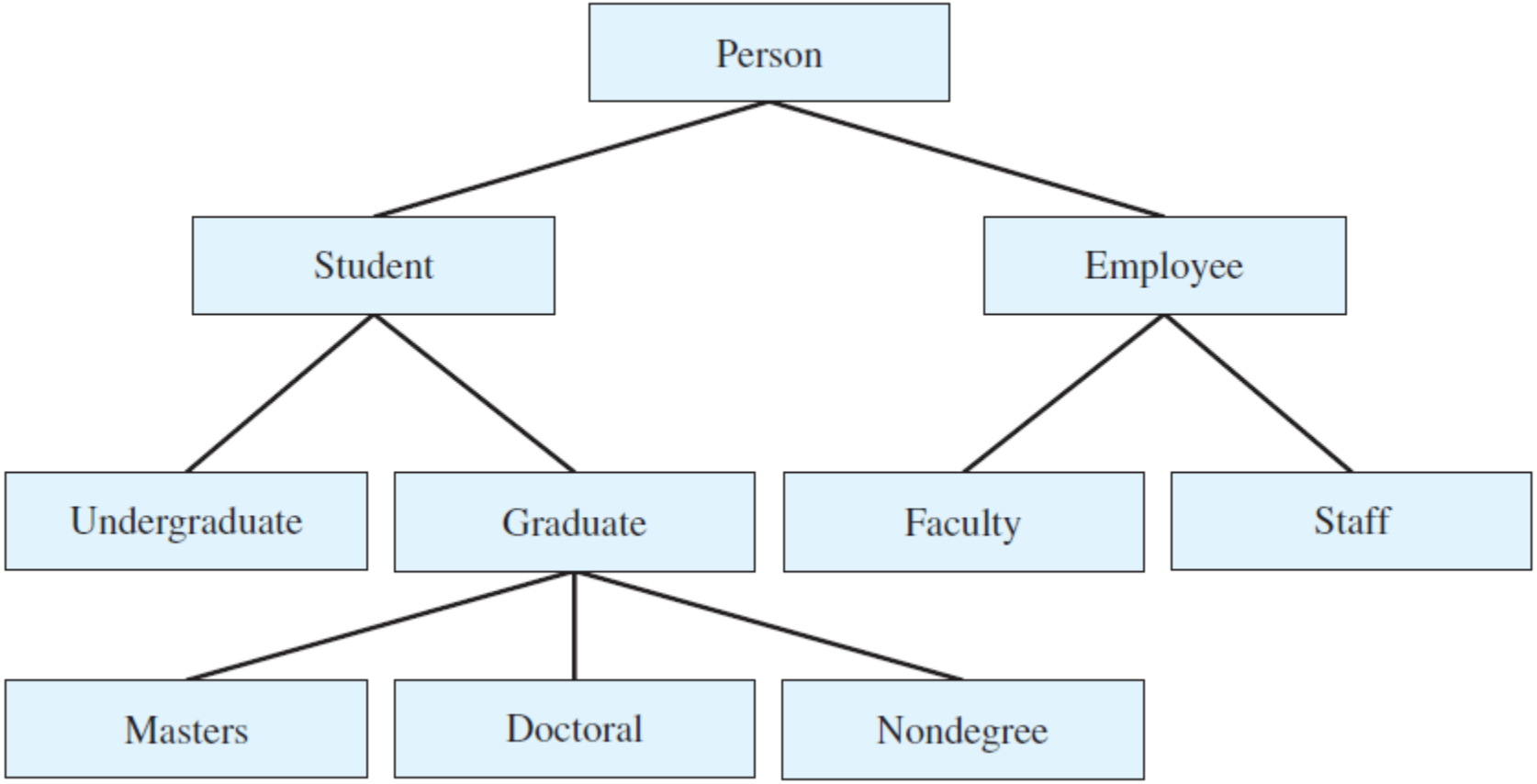
[ITP20003] Java Programming

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
  - Adds new details to general definition
  - New class inherits all properties of initial, general class

Base class and Derived Classes

For example,

- Class *Person* used as a base class
  - Also called superclass

- Now we declare derived class *Student*
  - Also called subclass
  - Inherits methods from the superclass

Derived Classes

- The class you start with is called the base class, or superclass. The derived class inherits all of the public methods and public instance variables from the base class and can add more instance variables and methods.

We need this

, and also this

SYNTAX

public class *Derived_Class_Name* extends *Base_Class_Name*
{
    Declarations_of_Added_Instance_Variables
    Definitions_of_Added__And_Changed_Methods
}

Write and run it!

The Class *Person*

```
public class Person
{
    private String name;
    public Person(){
        name = "No name yet";
    }
    public Person(String initialName){
        name = initialName;
    }
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public void writeOutput(){
        System.out.println("Name: " + name);
    }
    public boolean hasSameName(Person otherPerson){
        return this.name.equalsIgnoreCase(otherPerson.name);
    }
}
```

Write and run it!

A Derived Class Student

```
public class Student extends Person
{
    private int studentNumber;
    public Student(){     super();          studentNumber = 0;}
    public Student(String initialName, int initialStudentNumber){
        super(initialName);
        studentNumber = initialStudentNumber;
    }
    public void reset(String newName, int newStudentNumber){
        setName(newName);
        studentNumber = newStudentNumber;
    }
    public int getStudentNumber(){return studentNumber;}
    public void setStudentNumber(int newStudentNumber){
        studentNumber = newStudentNumber;
    }
    public void writeOutput(){
        System.out.println("Name: " + getName());
        System.out.println("Student Number: " + studentNumber);
    }
    public boolean equals(Student otherStudent){
        return this.hasSameName(otherStudent) &&
        (this.studentNumber == otherStudent.studentNumber);
    }
}
```

Write and run it!

InheritanceDemo

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        Student s = new Student();
        s.setName("Super Man");
        s.setStudentNumber(19380001);
        s.writeOutput();
    }
}
```

```
Name: Super Man
Student Number: 19380001
```

Inheritance

Example

List of methods

in Student class

```
public static void main(String[] args)
{
    Student s = new Student();
    s.setName("Super Man");
    s.setStudentNumber(19380001);
    s.writeOutput();
    s.
}
```

- equals(Object obj) : boolean - Object
- equals(Student otherStudent) : boolean - Student
- getClass() : Class<?> - Object
- getName() : String - Person
- getStudentNumber() : int - Student
- hashCode() : int - Object
- hasSameName(Person otherPerson) : boolean - Person
- notify() : void - Object
- notifyAll() : void - Object
- reset(String newName, int newStudentNumber) : void - Student
- setName(String newName) : void - Person
- setStudentNumber(int newStudentNumber) : void - Student
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object
- writeOutput() : void - Student

Press 'Ctrl+Space' to show Template Proposals

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 a method redefines it in a descendant class. *(actually replaces)*

Overriding Method Definitions

Person

- name: String

+ setName(String newName): void
+ getName(): String
+ writeOuput(): void
+ hasSameName(Person otherPerson): Boolean

Student

- name: String
- studentNumber: int

+ setName(String newName): void
+ getName(): String
+ writeOuput(): void
+ hasSameName(Person otherPerson): Boolean

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

New variable

Overriding

New methods

Overriding vs. Overloading

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

- Overloading
  - New method in same class with different signature

Overriding vs. Overloading

Person
- name: String
+ setName(String newName): void + getName(): String + writeOuput(): void + hasSameName(Person otherPerson): Boolean

- If you write the following method in the Student class.

```
public String getName(String title){
        return title + getName();
}
```

Let's try!

Student.java

```
public class Student extends Person{
    private int studentNumber;
    public Student(){…}
    public Student(String initialName, int initialStudentNumber) {…}
    public void reset(String newName, int newStudentNumber) {…}
    public int getStudentNumber() {…}
    public void setStudentNumber(int newStudentNumber) {…}
    public void writeOutput() {…}
    public boolean equals(Student otherStudent) {…}

    public String getName(String title){
        return title + getName();
    }
}
```

Add this part in Student.java

getNameTest_main.java

```
public class getNameTest_main {
    public static void main(String[] args) {
        Person p  = new Person(); p.setName("Super Man");
        Student s = new Student();s.setName("Super Man");

        System.out.println("p.getName(): " + p.getName());
        System.out.println("s.getName(): " + s.getName());
        System.out.println("s.getName(\" Hi \"): " + s.getName(" Hi "));
    }
}
```

```
p.getName(): Super Man
s.getName(): Super Man
s.getName(" Hi "):  Hi Super Man
```

Overriding vs. Overloading

- The Student class will have the two getName methods

```
public String getName()
```

From base class.

```
public String getName(String title)
```

New one.

=> This is overloading!!

Overloading places an additional “load” on a method name by using it for another method, whereas overriding replaces a method’s definition.

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.

Try, the final keyword for class Person.

```
public final class Person{
                    …
}
```

Person.java

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`	–

Private Instance Variables and Methods

Person
- name: String
+ setName(String newName): void + getName(): String + writeOuput(): void + hasSameName(Person otherPerson): Boolean

Exist, but not
A direct member.

Only accessible by
public access method

Student
- name: String - studentNumber: int
+ setName(String newName): void + getName(): String + writeOuput(): void + hasSameName(Person otherPerson): Boolean + reset(String newName, int newStudentNumber): void + getStudentNumber( ): int + setStudentNumber(int newStudentNumber): void + equals(Student otherStudent): Boolean

Private Instance Variables and Methods

```
Student joe = new Student();
joe.reset("Joesy", 9892);
```

Wrong!

```
public void reset(String newName, int newStudentNumber)
{
        name = newName;//ILLEGAL!
        studentNumber = newStudentNumber;
}
```

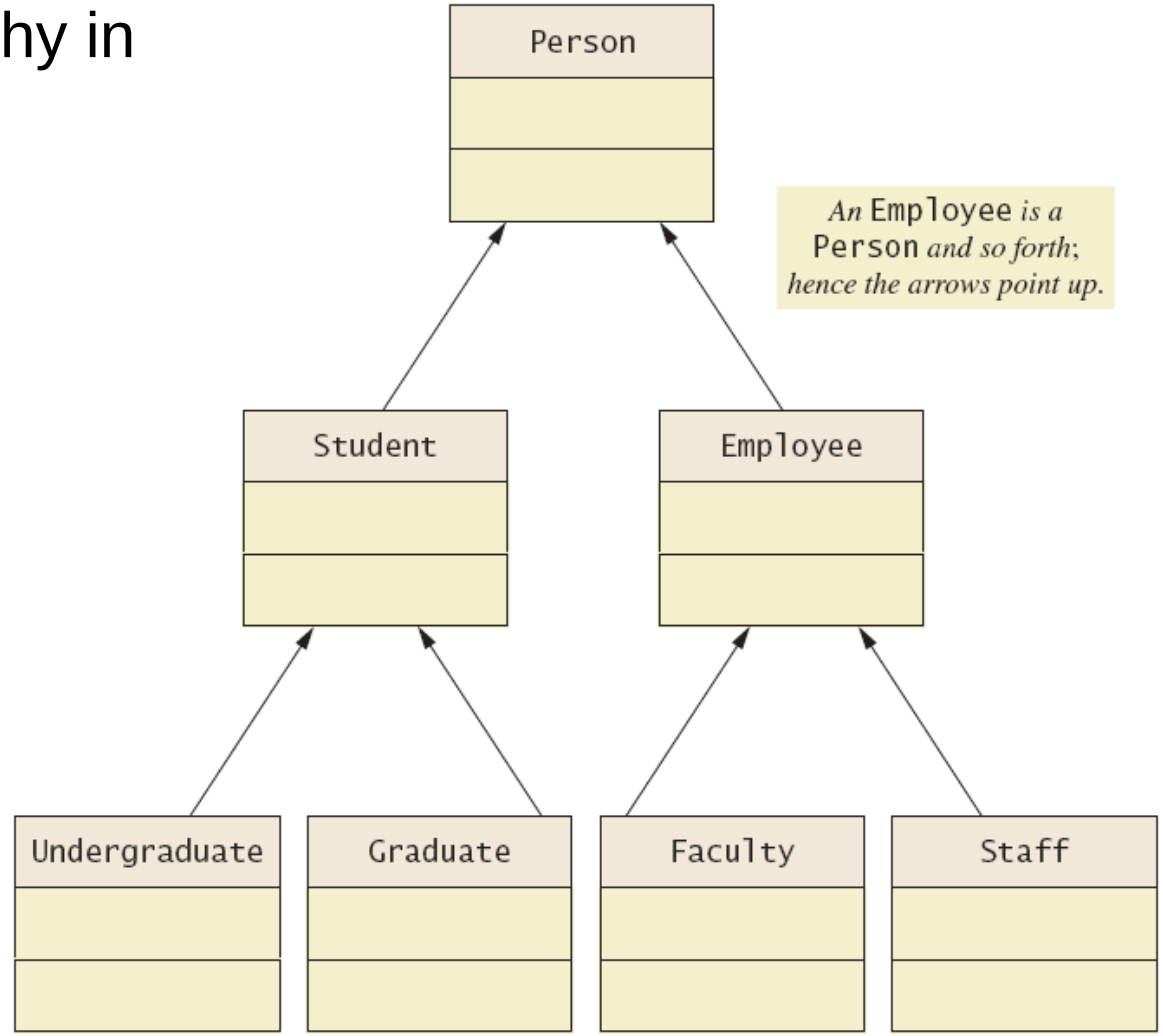
Better way.

```
public void reset(String newName, int newStudentNumber)
{
        setName(newName);
        studentNumber = newStudentNumber;
}
```

UML Inheritance Diagrams

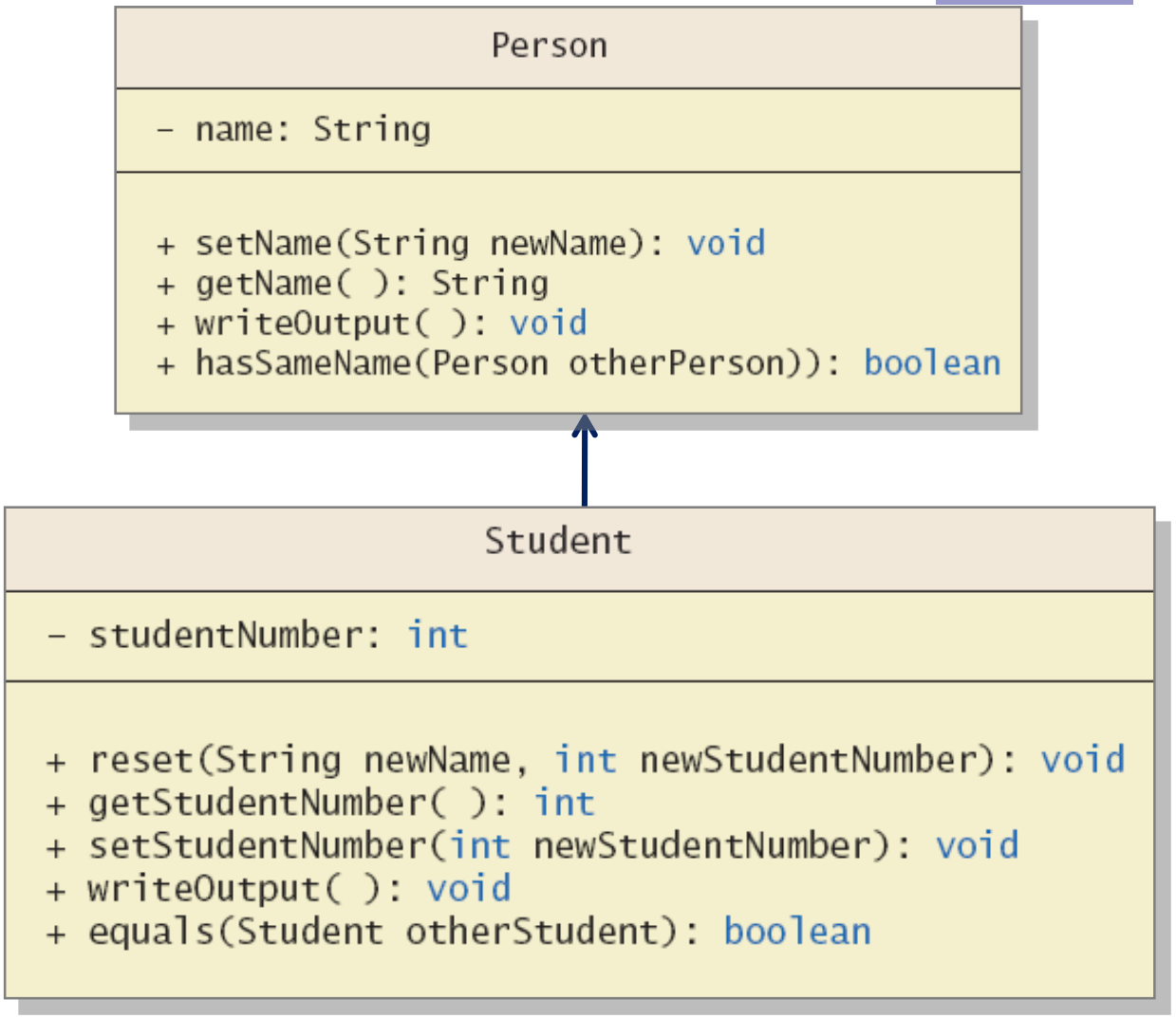
- A class hierarchy in UML notation

  ‘arrow’



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

'name' is an private instance variable of the base class, and it should be set in the constructor of the base class.

Constructors in Derived Classes

- If you do not include an explicit call to the base-class constructor in any constructor for a derived class, Java will automatically include a call to the base class's default constructor.

```
public Student()
{
      super();
      studentNumber = 0;
}
```

```
public Student()
{

      studentNumber = 0;
}
```

Completely equivalent

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

Constructors in Derived Classes

Calling the constructor in the base class using 'super'

```
public Student()
{
    super();
    studentNumber = 0;
}
```

Calling the constructor in the same class using 'this'

```
public Student()
{
    this("sman",19380001)
}
```

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);
}
```

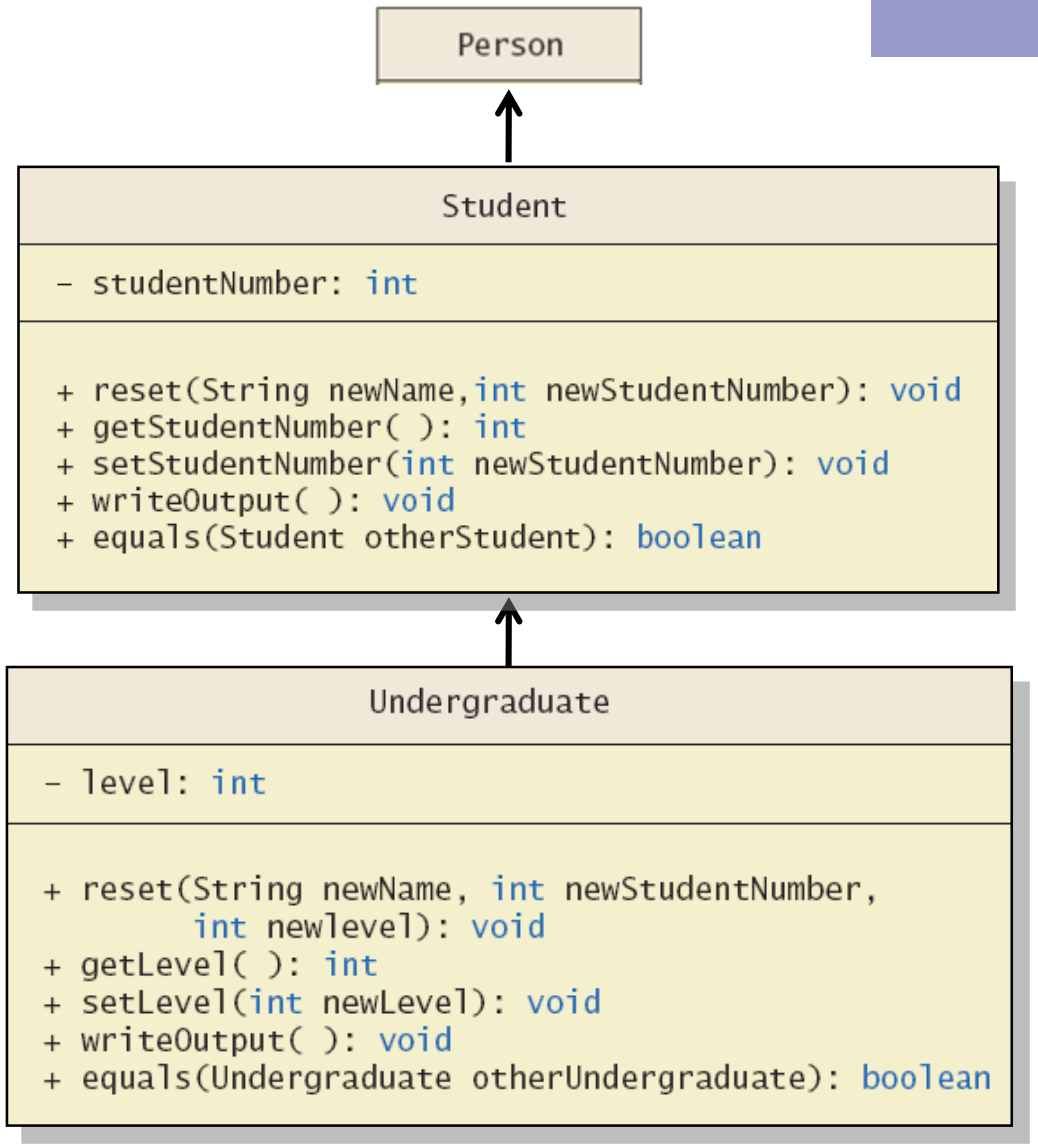
  - Calls method by same name in base class

Programming Example

- A derived class of a derived class
- Has all public members of both *Person* and *Student* classes
- This reuses the code in super classes

Programming Example

- More details of the UML class hierarchy



Write and run it!

```java
public class Undergraduate extends Student {
    private int level; //1 for freshman, 2 for sophomore . . .
    public Undergraduate(){super();level = 1;}
    public Undergraduate(String initialName, int initialStudentNumber,int initialLevel){
        super(initialName, initialStudentNumber);
        setLevel(initialLevel); //checks 1 <= initialLevel <= 4
    }
    public void reset(String newName, int newStudentNumber,int newLevel){
        reset(newName, newStudentNumber); //Student's reset
        setLevel(newLevel); //Checks 1 <= newLevel <= 4
    }
    public int getLevel(){return level;}
    public void setLevel(int newLevel){
        if ((1 <= newLevel) && (newLevel <= 4))
                level = newLevel;
        else{
            System.out.println("Illegal level!");
            System.exit(0);
        }
    }
    public void writeOutput(){
        super.writeOutput();
        System.out.println("StudentLevel: " + level);
    }
    public boolean equals(Undergraduate otherUndergraduate){
        return equals((Student)otherUndergraduate) &&
                (this.level == otherUndergraduate.level);
    }
}
```

Write and run it!

InheritanceDemo2

```
public class InheritanceDemo2 {
    public static void main(String[] args) {
        Person p = new Person("Super Man");
        Student s = new Student("Super Man", 20180101);
        Undergraduate u = new Undergraduate("Super Man", 20180101, 1);

        p.writeOutput(); System.out.println(" ");
        s.writeOutput(); System.out.println(" ");
        u.writeOutput(); System.out.println(" ");
    }
}
```

InheritanceDemo2 (result)

```
Name: Super Man

Name: Super Man
Student Number: 20180101

Name: Super Man
Student Number: 20180101
StudentLevel: 1
```

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.

* Will be explained again later.

Type Compatibility

```
public class SomeClass{
    public static void compareNumbers(Student s1, Student s2){
        if (s1.getStudentNumber() == s2.getStudentNumber())
                System.out.println(s1.getName() + " has the same " +
                                                "number as " + s2.getName());
        else
                System.out.println(s1.getName() + " has a different " +
                                                "number from " + s2.getName());
    }
}
```

```
public class SomeClassDemo{
public static void main(String[] args){
    Student sobj                = new Student("SMan", 1938);
    Student sobj2               = new Student("BMan", 1939);
    Undergraduate uobj          = new Undergraduate("IronMan", 1963, 1);
    Undergraduate uobj2         = new Undergraduate("Hulk", 1963, 1);
    SomeClass.compareNumbers(sobj, uobj);
}}
```

** An object of a derived class can serve as an object of the base class.*

Type Compatibility

- Note that there is no automatic type casting here.
- An object of the class Undergraduate *is* an object of the class Student, and so it *is* of type Student.
- It need not be, and is not, type cast to an object of the class Student

```
Student sobj              = new Student("SMan", 1938);
Undergraduate uobj        = new Undergraduate("IronMan", 1963, 1);

SomeClass.compareNumbers(sobj, uobj);
```

Result SMan has a different number from IronMan

* *An object can have several types because of inheritance.*

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

Type Compatibility

```
import java.util.Scanner;
public class TypeCompatibilityDemo {
   public static void main(String[] args) {
   Person joePerson   = new Person("Josephine Student");
   System.out.println("Enter name:");

   Scanner keyboard   = new Scanner(System.in);
   String newName     = keyboard.nextLine();
   Undergraduate someUndergrad = new Undergraduate(newName, 222, 3);

   if (joePerson.hasSameName(someUndergrad))
       System.out.println("Wow, same names!");
   else
       System.out.println("Different names");
}
```

- Every object of the class *Undergraduate* is also an object of the class *Person*. Even the following invocation is valid:

  `someUndergrad.hasSameName(joePerson)`

Assignment Compatibility

- An object of a derived class has the type of the derived class, but it can be referenced by a variable whose type is any one of its ancestor classes.
- Thus, you can assign an object of a derived class to a variable of any ancestor type, but not the other way around.

```
public class AssignmentTest {
    public static void main(String[] args) {
        Person p1          = new Student();
        Person p2          = new Undergraduate();
        Student s1         = new Student();
        Undergraduate ug1  = new Undergraduate();
        Person p3 = s1;
        Person p4 = ug1;

        Student s2         = new Person();
        Undergraduate ug2  = new Person();
        Undergraduate ug3  = new Student();

        Person p5          = new Person();
        Student s3         = new Student();
        Undergraduate ug4  = p5;
        Undergraduate ug5  = s3;
    }
}
```

?

check which statements are correct or incorrect?

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 object of any type

Ex) println(Object theObject)

The Class *Object*

- Class *Object* has some methods that every Java class inherits, for example
  - `toString` : Returns a string representation of the object.
  - `Equals` : Checks if two objects are same.

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

- However, the methods `equals` and `toString` inherited from Object will not work correctly for almost any class you define.

- Thus, you need to override the inherited method definitions with new, more appropriate definitions.

Define Your Own toString Method

- Object's `toString` method will not display any data related to your class. You usually should override `toString` in the classes that you write.

```
public String toString()
{
      return "Name: " + getName() +
            "\nStudent number: " + studentNumber;
}
```

A Better equals Method

- Remember that class *Student* has two equal methods

```
public boolean equals(Student otherStudent)
public boolean equals(Object otherObject)
```

} Overriding or Overloading?

```
public static void main(String[] args)
{
    Student s = new Student();
    s.setName("Super Man");
    s.setStudentNumber(19380001);
    s.writeOutput();
    s.
}
```

- equals(Object obj) : boolean - Object
- equals(Student otherStudent) : boolean - Student
- getClass() : Class<?> - Object
- getName() : String - Person
- getStudentNumber() : int - Student

In some cases, Java may be confused about what it should calls.

We'd better specify what we want to call.

A Better equals Method

- To specify that we want to use equal method for Student object in any cases, we can correct the equals() as follows,

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

```
public boolean equals(Object otherObject){
        Student otherStudent = (Student)otherObject;
        return this.hasSameName(otherStudent) &&
                (this.studentNumber == otherStudent.studentNumber);
}
```

- However, there is another problem.
- If a user use Student object as an parameter, it is OK.
- But users can input any object, this case may make an error.

A Better equals Method

```
public boolean equals(Object otherObject){
        Student otherStudent = (Student)otherObject;
        return this.hasSameName(otherStudent) &&
                (this.studentNumber == otherStudent.studentNumber);
}
```

```
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;
}
```

* The **java** `instanceof` **operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).