

# Lesson 7

# Inheritance

Absolute Java

Walter Savitch

# Inheritance

- Inheritance allows you to define a very general class and then later define more specialized classes that add some new details to the existing general class definition.
- This saves work, because the more specialized class inherits all the properties of the general class and you, the programmer, need only program the new features.

# Inheritance

- Suppose we define a class for vehicles that has instance variables to record the vehicle's number of wheels and maximum number of occupants. The class also has accessor and mutator methods.
- Imagine that we then define a class for automobiles that has instance variables and methods just like the ones in the class of vehicles.
- In addition, our automobile class would have added instance variables for such things as the amount of fuel in the fuel tank and the license plate number and would also have some added methods.
- Instead of repeating the definitions of the instance variables and methods of the class of vehicles within the class of automobiles, we could use Java's inheritance mechanism, and let the automobile class inherit all the instance variables and methods of the class for vehicles.

# Derived Classes

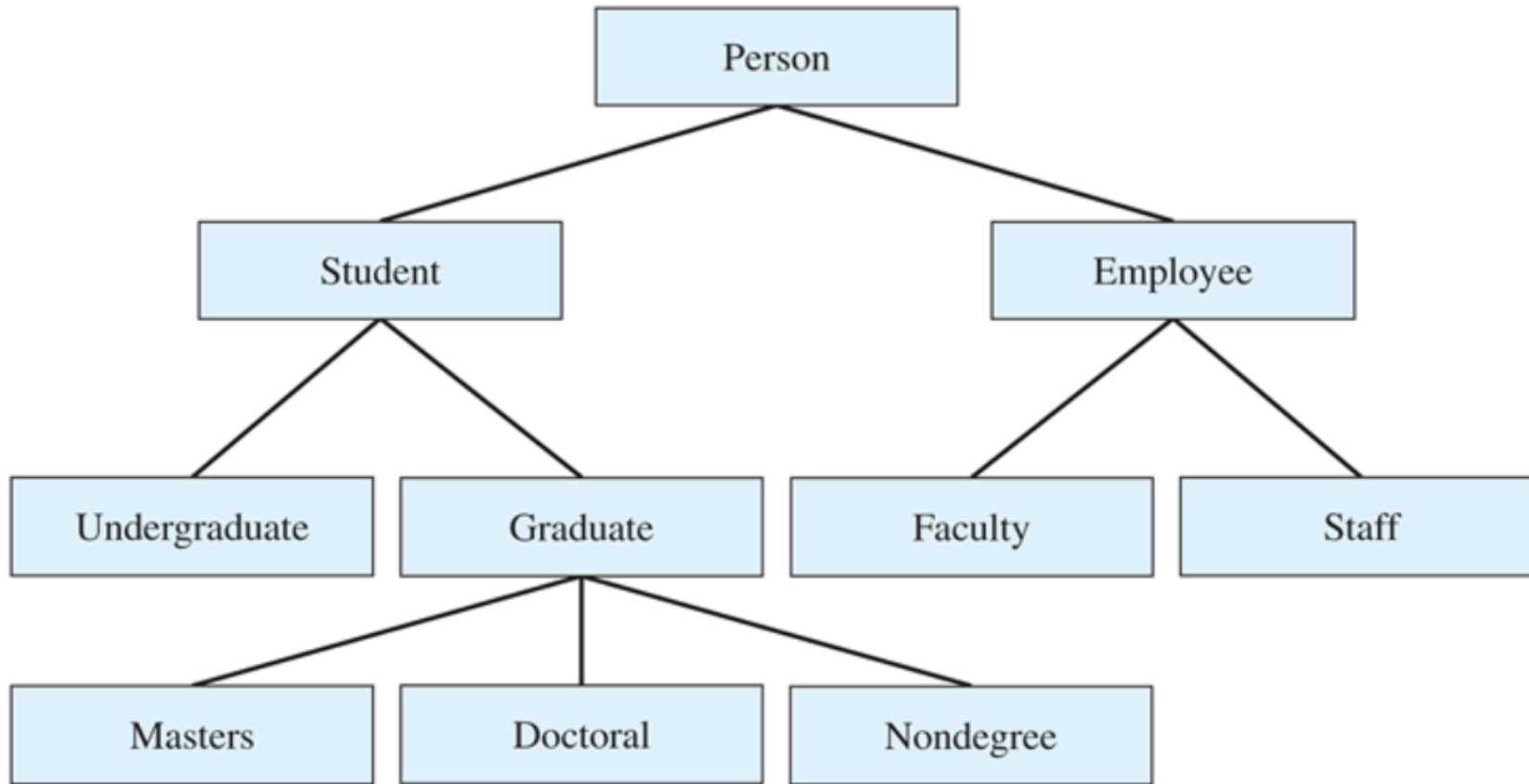
- A derived class is a class defined by adding instance variables and methods to an existing class.
- The derived class **extends** the existing class.
- The existing class that the derived class is built upon is called the **base class**, or **superclass**.

# Derived Classes

- Suppose we are designing a college record-keeping program that has records for students, faculty, and other staff.
- There is a natural hierarchy for grouping these record types: They are all records of people.
- **Students are one subclass of people.**
- Another subclass is employees, which includes both faculty and staff.
- Students divide into two smaller subclasses: undergraduate students and graduate students.
- These subclasses may further subdivide into still smaller subclasses.

# Derived Classes

## A Class Hierarchy



# Derived Classes

- The derived class inherits all the public methods, all the public instance variables, and all the public and private static variables from the base class, and it can add more instance variables, more static variables, and more methods.
- The only members not inherited are private methods.
- A derived class is also called a **subclass**, in which case the base class is usually called a **superclass**.

# Derived Classes

- **Syntax**

```
public class Derived_Class_Name extends  
Base_Class_Name  
{  
    Declarations_of_Added_Static_Variables  
    Declarations_of_Added_Instance_Variables  
    Definitions_of_Added__And_Overridden_Methods  
}
```



# Derived Classes

- In our example, Person is the base class and Student is the derived class.
- It is indicated in the definition of Student by including the phrase extends Person on the first line of the class definition, so that the class definition of Student begins

# Derived Classes

```
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()
```

# Derived Classes

```
{  
    return name;  
}  
public void writeOutput()  
{  
    System.out.println("Name: " + name);  
}  
public boolean hasSameName(Person otherPerson)  
{  
    return this.name.equalsIgnoreCase(otherPerson.name);  
}  
}
```

# Derived Classes

```
public class Student extends Person
{
    private int studentNumber;
    public Student()
    {
        super();
        studentNumber = 0; //Indicating no number yet
    }
    public Student(String initialName, int initialStudentNumber)
    {
        super(initialName);
        studentNumber = initialNumber;|
    }
}
```

# Derived Classes

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

# Derived Classes

```
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);
}
}
```

# Derived Classes

- The class **Student**—like any other derived class—is said to **inherit** the private and public instance variables and public methods of the base class that it extends.
- When you define a derived class, you give only the added instance variables and the added methods.
- Even though the class **Student** has all the public instance variables and all the public methods of the class **Person**, we do not declare or define them in the definition of **Student**.
- For example, every object of the class **Student** has the method **getName**, but we do not define **getName** in the definition of the class **Student**.

# Derived Classes

- A derived class, such as **Student**, can also add some instance variables or methods to those it inherits from its base class.
- For example, **Student** defines the instance variable **studentNumber** and the methods **reset**, **getStudentNumber**, **setStudentNumber**, **writeOutput**, and **equals**, as well as some constructors.
- Notice that although **Student** does not inherit the private instance variable name from **Person**, it does inherit the method **setName** and all the other public methods of the base class.




# Derived Classes

- Thus, **Student** has indirect access to **name** and so has no need to define its own version.
- If **s** is a new object of the class **Student**, defined as  
`Student s = new Student();`
- we could write  
`s.setName("Warren Peace");`
- Because **name** is a private instance variable of **Person**, however, you cannot write **s.name** outside of the definition of the class **Person**, not even within the definition of **Student**.
- The instance variable exists, however, and it can be accessed and changed using methods defined within **Person**.

# Derived Classes

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        Student s = new Student();
        s.setName("Warren Peace");
        s.setStudentNumber(1234);
        s.writeOutput();
    }
}
```

*setName is inherited  
from the class Person.*



# Derived Classes

- **Output**

**Name: Warren Peace**

**Student Number: 1234**

- An object of **Student** has all of the methods of **Person** in addition to all of the methods of **Student**.
- **Note: Inheritance should define a natural is-a relationship between two classes.**
- If an is-a relationship does not exist between two proposed classes, do not use inheritance to derive one class from the other.
- Instead, consider defining an object of one class as an instance variable within the other class.
- That relationship is called has-a.

# Derived Classes

- A base class is often called a parent class.
- A derived class is then called a child class.
- This analogy is often carried one step further.
- A class that is a parent of a parent of a parent of another class (or some other number of “parent of” iterations) is often called an ancestor class.
- If class A is an ancestor of class B, then class B is often called a descendent of class A.
- You define a derived class, or subclass, by starting with another already defined class and adding (or changing) methods and instance variables.
- 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.

# Derived Classes

- **Syntax:**

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

- A derived class is also called a subclass, a child class, and a descendant class.

# Derived Classes

- **Syntax:**

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

- A derived class is also called a subclass, a child class, and a descendant class.

# Overriding Method Definitions

- If a derived class defines a method with the same name, the same number and types of parameters, and the same return type as a method in the base class, the definition in the derived class is said to override the definition in the base class.
- In other words, the definition in the derived class is the one that is used for objects of the derived class.

# Overriding Method Definitions

- The class **Student** defines a method named **writeOutput** that has no parameters.
- But the class **Person** also has a method by the same name that has no parameters.
- If the class **Student** were to inherit the method **writeOutput** from the base class **Person**, **Student** would contain two methods with the name **writeOutput**, both of which have no parameters.
- Java has a rule to avoid this problem.



# Overriding Method Definitions

- For example, the invocation

**s.writeOutput();**

- will use the definition of **writeOutput** in the class **Student**, not the definition in the class **Person**, since **s** is an object of the class **Student**.
- When overriding a method, you can change the body of the method definition to anything you wish, but you cannot make any changes in the method's heading, including its return type.

# Overriding Versus Overloading

- If the method in the derived class were to have the same name and the same return type but a different number of parameters or a parameter of a different type from the method in the base class, the method names would be overloaded.
- In such cases, the derived class would have both methods.
- **A method overloads another if both have the same name and return type but different parameter lists.**

# Overriding Versus Overloading

- **For Example** if we added the following method to the definition of the class **Student**,

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

- The class **Student** would have two methods named **getName**: It would inherit the method **getName**, with no parameters, from the base class **Person** , and it would also have the method named **getName**, with one parameter, that we just defined.
- This is because the two **getName** methods have different numbers of parameters, and thus the methods use overloading.

# Changing the Return Type of an Overridden Method

- In a derived class, you can override (change) the definition of a method from the base class.
- As a general rule, when overriding a method definition, you may not change the type returned by the method, and you may not change a void method to a method that returns a value, nor a method that returns a value to a void method.

# Changing the Return Type of an Overridden Method

- The one exception to this rule is if the returned type is a class type, then you may change the returned type to that of any descendent class of the returned type.
- For example, if a function returns the type Employee , when you override the function definition in a derived class, you may change the returned type to HourlyEmployee, SalariedEmployee , or any other descendent class of the class Employee.
- This sort of changed return type is known as a **covariant return type** and is new in Java version 5.0; it was not allowed in earlier versions of Java.

# Changing the Return Type of an Overridden Method

- For example, suppose one class definition includes the following details:

```
public class BaseClass  
{
```

```
    ...
```

```
    public Employee getSomeone(int someKey)
```

```
    ...  
}
```

- In this case, the following details would be allowed in a derived class:

```
public class DerivedClass extends BaseClass  
{
```

```
    ...
```

```
    public HourlyEmployee getSomeone(int someKey)
```

# Changing the Return Type of an Overridden Method

- When the method definition for **getSomeone** is overridden in **DerivedClass**, the returned type is changed from **Employee** to **HourlyEmployee**.
- It is worth noting that when you change the returned type of an overridden method in this way, such as from **Employee** to **HourlyEmployee**, you are not really changing the returned type so much as placing additional restrictions on it.

# Changing the Return Type of an Overridden Method

- Every HourlyEmployee is an Employee with some additional properties that, while they are properties of every HourlyEmployee, are not properties of every Employee.
- Any code that was written for a method of the base class and that assumed the value returned by the method is Employee will be legal for an overridden version of the method that returns an HourlyEmployee.
- This is true because every HourlyEmployee is an Employee.



# Changing the Return Type of an Overridden Method

- **Changing the Access Permission of an Overridden Method**
- You can change the access permission of an overridden method from private in the base class to public in the derived class (or in any other way that makes access permissions more permissive).
- For example, if the following is a method heading in a base case:

**private void doSomething()**

- then you can use the following heading when overriding the method definition in a derived class:

**public void doSomething()**

# Changing the Return Type of an Overridden Method

- **Changing the Access Permission of an Overridden Method**
- You can change the access permission of an overridden method from private in the base class to public in the derived class (or in any other way that makes access permissions more permissive).
- For example, if the following is a method heading in a base case:

**private void doSomething()**

- then you can use the following heading when overriding the method definition in a derived class:

**public void doSomething()**

# Changing the Return Type of an Overridden Method

- You cannot change permissions to make them more restricted in the derived class.
- You can change private to public, but you cannot change public to private.
- This makes sense, because you want code written for the base class method to work for the derived class method.
- You can use a public method anywhere that you can use a private method, but it is not true that you can use a private method anywhere that you can use a public method.

# Check your understanding

1. Suppose the class named DiscountSale is a derived class of a class called Sale. Suppose the class Sale has instance variables named price and numberOfItems. Will an object of the class DiscountSale also have instance variables named price and numberOfItems?
2. Suppose the class named DiscountSale is a derived class of a class called Sale, and suppose the class Sale has public methods named getTotal and getTax. Will an object of the class DiscountSale have methods named getTotal and getTax? If so, do these methods have to perform the exact same actions in the class DiscountSale as in the class Sale?

# Check your understanding

1. Suppose the class named DiscountSale is a derived class of a class called Sale, and suppose the class Sale has a method with the following heading and no other methods named getTax, as follows:

*public double* getTax()

And suppose the definition of the class DiscountSale has a method definition with the following heading and no other method definitions for methods named getTax, as follows:

*public double* getTax(double rate)

How many methods named getTax will the class DiscountSale have and what are their headings?

# The final Modifier

- If you want to specify that a method definition cannot be overridden by a new definition within a derived class, you can add the final modifier to the method heading .

- **Example:**

**public final void specialMethod()**

- A final method cannot be overridden.
- **Final Class:** An entire class can be declared final, in which case you cannot use it as a base class to derive any other class.
- So a final class cannot be a base class.

# Private Instance Variables and Private Methods of a Base Class

- An object of the derived class **Student** does not inherit the instance variable **name** from the base class **Person** , but it can access or change name's value via the public methods of Person.
- For example, the following statements create a Student object and set the values of the instance variables name and studentNumber:

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

# Private Instance Variables and Private Methods of a Base Class

- Private instance variables in a base class are not inherited by a derived class; they cannot be referenced directly by name within a derived class.
- So the following code is INVALID:

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



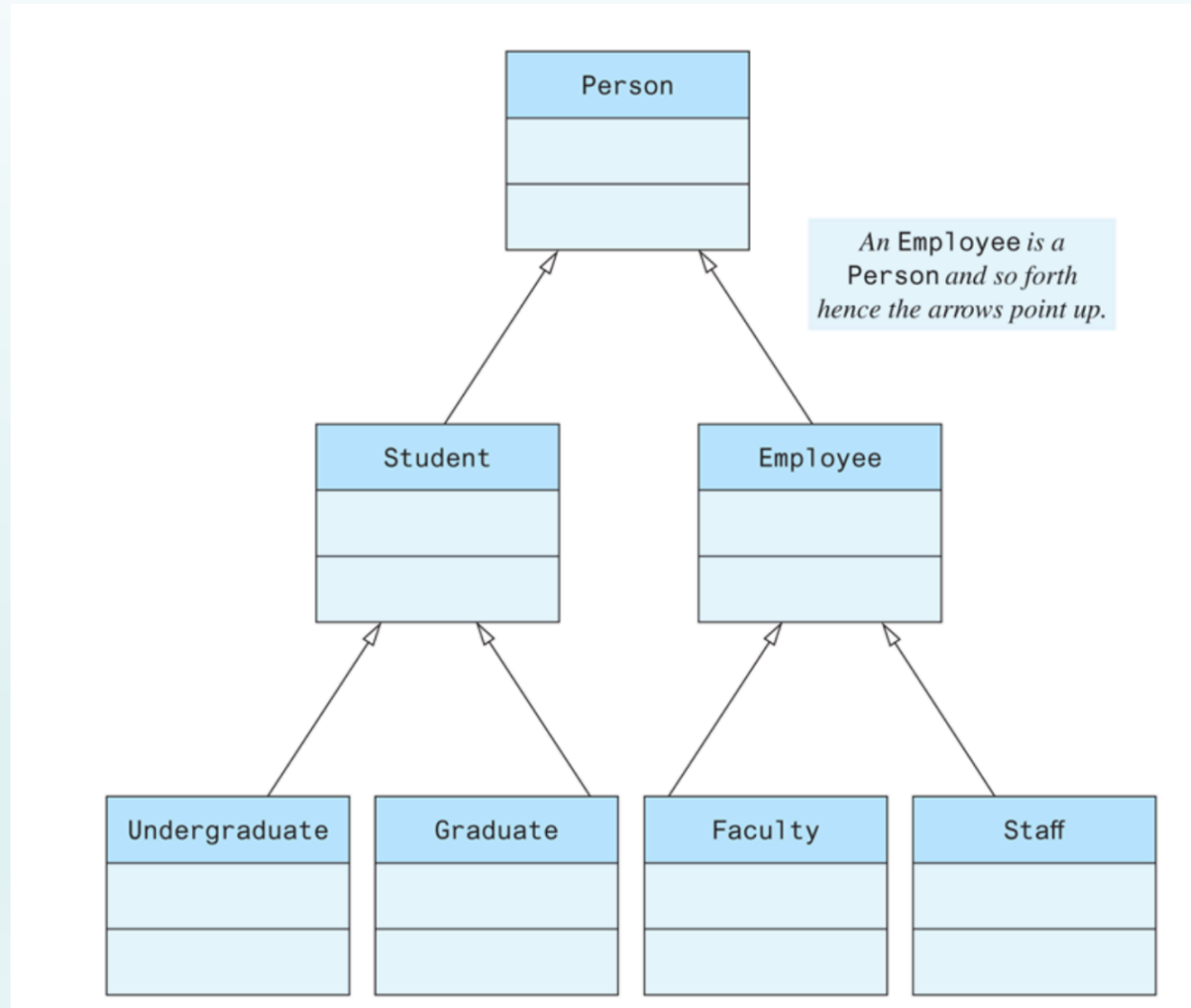
# Private Instance Variables and Private Methods of a Base Class

- A derived class cannot access the private instance variables of its base class directly by name.
- It knows only about the public behavior of the base class.
- The derived class is not supposed to know—or care—how its base class stores data.
- However, an inherited public method may contain a reference to a private instance variable.

# UML Inheritance Diagrams

- Below is an example of class hierarchy using UML.
- Note that the arrowheads point up from the derived class to the base class. These arrows show the is-a relationship.
- For example, a **Student** is a **Person**. In Java terms, an object of type **Student** is also of type **Person**.

# UML Inheritance Diagrams



# UML Inheritance Diagrams

- The arrows also help in locating method definitions.
- If you are looking for a method definition for some class, the arrows show the path you (or the computer) should follow.
- If you are looking for the definition of a method used by an object of the class **Undergraduate**, you first look in the definition of the class **Undergraduate**; if it is not there, you look in the definition of **Student**; if it is not there, you look in the definition of the class **Person**.

# UML Inheritance Diagrams

- Look at the example below.
- It shows more details of the inheritance hierarchy for two classes: **Person** and one of its derived classes, **Student**.
- Suppose **s** references an object of the class **Student**. The diagram tells you that definition of the method **getStudentNumber** in the call

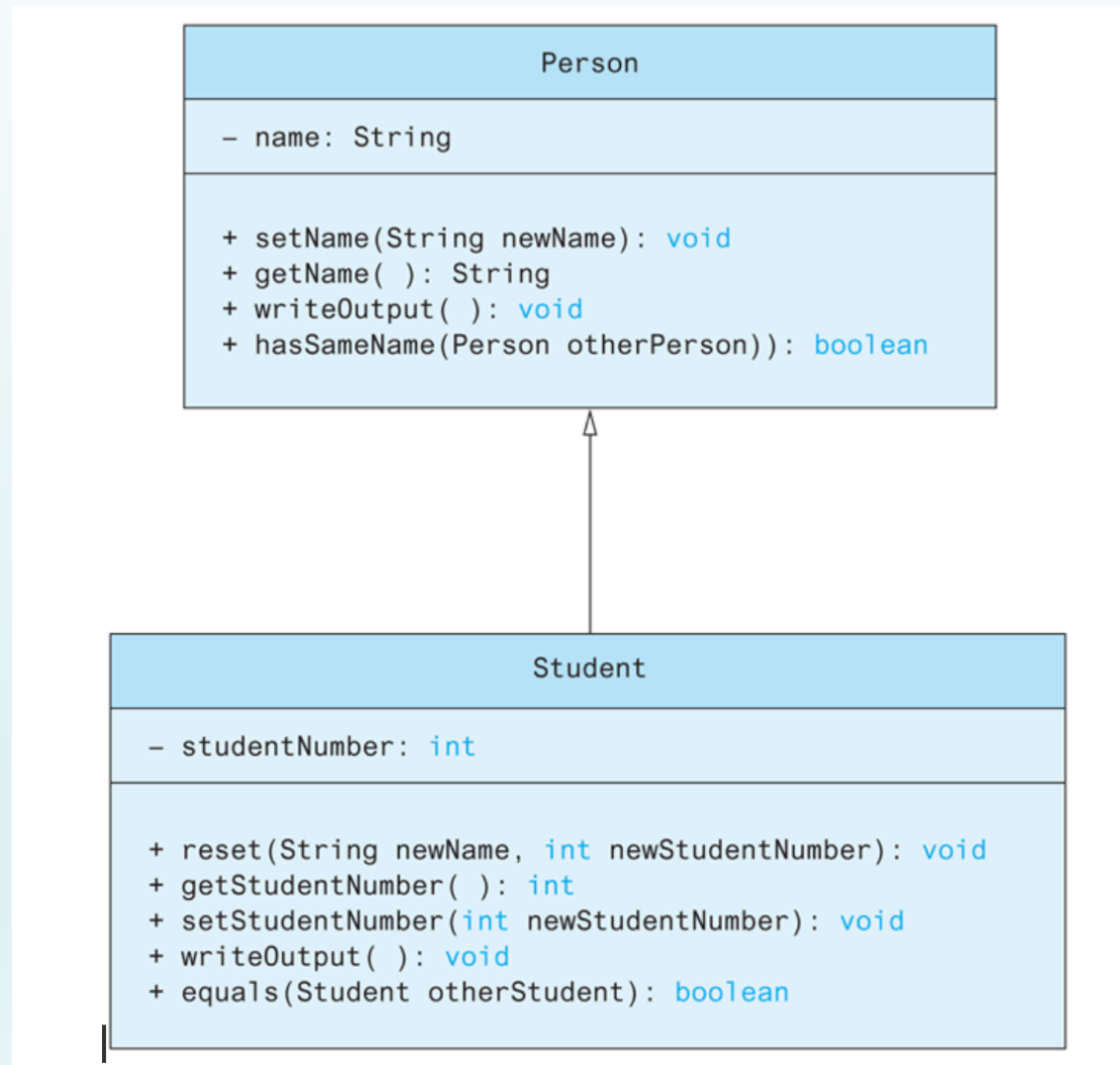
```
int num = s.getStudentNumber();
```

- is found within the class **Student**, but that the definition of **setName** in

```
s.setName("Joe Student");
```

- is in the class **Person**.

# UML Inheritance Diagrams



# Constructors in Derived Classes

- A derived class, such as the class **Student**, has its own constructors.
- It does not inherit any constructors from the base class.
- A base class, such as **Person**, also has its own constructors.
- **You can invoke a constructor of the base class within the definition of a derived class constructor.**
- **In the definition of a constructor for the derived class, the typical first action is to call a constructor of the base class.**
- For example, consider defining a constructor for the class Student.
- One thing that needs to be initialized is the student's name.
- Since the instance variable name is defined in the definition of **Person**, it is normally initialized by the constructors for the base class Person.

# Constructors in Derived Classes

- **Example:**

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

- This constructor uses the reserved word `super` as a method name to call a constructor of the base class.
- Although the base class `Person` defines two constructors, the invocation `super(initialName);`
- is a call to the constructor in that class that has one parameter, a string.
- Notice that you use the keyword `super`, not the name of the constructor.



# Constructors in Derived Classes

- That is, you do *not* use

`Person(initialName); //ILLEGAL`

- **The use of super involves some details:** It must always be the first action taken in a constructor definition.
- You cannot use super later in the definition.
- 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.

# Constructors in Derived Classes

- For example, the definition of the default constructor for the class Student,

```
public Student()  
{  
    super();  
    studentNumber = 0; //Indicating no number yet  
}
```

- is completely equivalent to the following definition:

```
public Student()  
{  
    studentNumber = 0; //Indicating no number yet  
}
```

# Constructors in Derived Classes

**Restrictions on how you can use the base class constructor call super:**

- You cannot use an instance variable as an argument to super.
- When defining a constructor for a derived class, you can use super as a name for the constructor of the base class.
- Any call to super must be the first action taken by the constructor.
- **Example**

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

# The this Method

- Another common action when defining a constructor is to call another constructor in the same class.
- We can revise the default constructor in the class **Person** to call another constructor in that class by using this, as follows:

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

# The this Method

**The restrictions on how you can use the base class constructor call super also apply to the this constructor.**

- You cannot use an instance variable as an argument to this.
- Also, any call to the constructor this must always be the first action taken in a constructor definition.
- Thus, a constructor definition cannot contain both an invocation of super and an invocation of this.
- If you want to include both a call to super and a call to this, use a call with this, and have the constructor that is called with this have super as its first action.

# Calling an Overridden Method

- A method of a derived class that overrides (redefines) a method in the base class can use `super` to call the overridden method.
- Within the definition of a method of a derived class, you can call an overridden method of the base class by prefacing the method name with **super and a dot**.

- **Syntax:**

*`super.Overridden_Method_Name(Argument_List)`*

- **Example:**

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

- If you just use the method name **writeOutput** within the class **Student**, it will invoke the method named **writeOutput** in the class **Student**.

# An Object of a Derived Class Has More than One Type

- An object of a derived class has the type of the derived class.
- It also has the type of the base class, and more generally, it has the type of every one of its ancestor classes.
- **Example:**

```
public HourlyEmployee(HourlyEmployee originalObject)
{
    super(originalObject);
    wageRate = originalObject.wageRate;
    hours = originalObject.hours;
}
```

# An Object of a Derived Class Has More than One Type

- The line

`super(originalObject);`

- is an invocation of a constructor for the base class Employee.
- The class Employee has no constructor with a parameter of type HourlyEmployee, but originalObject is of type HourlyEmployee.
- Fortunately, every object of type HourlyEmployee is also of type Employee.
- So, this invocation of super is an invocation of the copy constructor for the class Employee.



# A Derived Class of a Derived Class

- The line

`super(originalObject);`

- is an invocation of a constructor for the base class Employee.
- The class Employee has no constructor with a parameter of type HourlyEmployee, but originalObject is of type HourlyEmployee.
- Fortunately, every object of type HourlyEmployee is also of type Employee.
- So, this invocation of super is an invocation of the copy constructor for the class Employee.

# A Derived Class of a Derived Class

- You can form a derived class from a derived class.
- For example, we previously derived the class **Student** from the class **Person** .
- We now derive a class **Undergraduate** from **Student**.

# A Derived Class of a Derived Class

```
public class Undergraduate extends Student
{
    private int level; //1 for freshman, 2 for sophomore
                        //3 for junior, or 4 for senior.

    public Undergraduate()
    {
        super();
        level = 1
    }

    public Undergraduate(String initialName, int initialStudentNumber, int initialLevel)
    {
        super(initialName, initialStudentNumber);
        setLevel(initialLevel); //checks 1 <= initialLevel <= 4
    }
}
```

# A Derived Class of a Derived Class

```
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;
}
```

# A Derived Class of a Derived Class

```
}

public void setLevel(int newLevel)
{
    if ((1 <= newLevel) && (newLevel <= 4))
        level = newLevel;
    else
    {
        System.out.println("Illegal level!");
        System.exit(0);
    }
}
```

# A Derived Class of a Derived Class

```
public void writeOutput()
{
    super.writeOutput();
    System.out.println("StudentLevel: " + level);
}
public boolean equals(Undergraduate otherUndergraduate)
{
    return equals(Student)otherUndergraduate) &&
        (this.level == otherUndergraduate.level);
}
}
```

# A Derived Class of a Derived Class

- An object of the class **Undergraduate** has all the public members of the class **Student**.
- But **Student** is already a derived class of **Person**.
- This means that an object of the class **Undergraduate** also has all the public members of the class **Person**.
- An object of the class **Person** has the instance variable **name**.
- An object of the class **Student** has the instance variable **studentNumber**, and an object of the class **Undergraduate** has the instance variable **level**.

# A Derived Class of a Derived Class

- Although an object of the class **Undergraduate** does not inherit the instance variables **name** and **studentNumber**, since they are private, it can use its inherited accessor and mutator methods to access and change them.
- In effect, the classes **Student** and **Undergraduate**—as well as any other classes derived from either of them—reuse the code given in the definition of the class **Person**, because they inherit all the public methods of the class **Person**.



# A Derived Class of a Derived Class

- Each of the constructors in the class **Undergraduate** begins with an invocation of super, which in this context stands for a constructor of the base class **Student**.
- But the constructors for the class **Student** also begin with an invocation of super, which in this case stands for a constructor of the base class **Person**.
- Thus, when we use new to invoke a constructor in **Undergraduate**, constructors for **Person** and **Student** are invoked, and then all the code following super in the constructor for **Undergraduate** is executed.

# A Derived Class of a Derived Class

- As we already noted, within the definition of a method of a derived class, you can call an overridden method of the base class by prefacing the method name with `super` and a dot.
- However, **you cannot repeat the use of `super` to invoke an overridden method from some ancestor class other than a direct parent.**
- Suppose that the class **Student** is derived from the class **Person**, and the class **Undergraduate** is derived from the class **Student**.
- You might think that you can invoke a method of the class **Person** within the definition of the class **Undergraduate** by using `super.super`, as in  
`super.super.writeOutput(); //ILLEGAL!`
- However, as the comment indicates, it is illegal to have such a train of supers in Java.