Constructors in Derived Classes

A derived class, such as the class **Student** in **previous lesson,**, 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. 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.

Note: A derived class does not inherit constructors from its base class.

Note: Constructors in a derived class invoke constructors from the base class.

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. That is, you do *not* use

```
Person(initialName); //ILLEGAL
```

Note: Use super within a derived class as the name of a constructor in the base class (superclass) .

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. 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
}
```

Note: Any call to super must be first within a constructor.

Note: Without super, a constructor invokes the default constructor in the base class.

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;
}
```

When you omit a call to the base-class constructor in any derived-class constructor, the default constructor of the base class is called as the first action in the new constructor. This default constructor—the one without parameters—might not be the one that should be called. Thus, including your own call to the base-class constructor is often a good idea.

For example, omitting super(initialName) from the second constructor in the class Student would cause Person’s default constructor to be invoked. This action would set the student’s name to "No name yet" instead of to the string initialName.

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");
}
```

In this way, the default constructor calls the constructor

```
public Person(String initialName)
{
   name = initialName;
}
```

thereby setting the instance variable name to the string "No name yet".

As with super, any use of this must be the first action in a constructor definition. Thus, a constructor definition cannot contain both a call using super and a call using this.

Remember this and super Within a Constructor

When used in a constructor, **this** calls a constructor of the same class, but **super** invokes a constructor of the base class.

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, but in a slightly different way.

For example, consider the method writeOutput for the class Student. It contains the statement

```
System.out.println("Name: " + getName());
```

to display the name of the Student. Alternatively, you could display the name by calling the method **writeOutput** of the class **Person**, since the **writeOutput** method for the class **Person** will display the person's name. The only problem is that if you use the method name **writeOutput** within the class Student, it will invoke the method named **writeOutput** in the class **Student**. What you need is a way to say "**writeOutput()** as it is defined in the base class." The way you say that is **super.writeOutput()**. So an alternative definition of the **writeOutput** method for the class **Student** is the following:

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

Note: Using super as an object calls a base-class 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(); //Calls writeOutput in the base
                               //class
   System.out.println("Student Number: " + studentNumber);
}
```

A Derived Class of a Derived Class

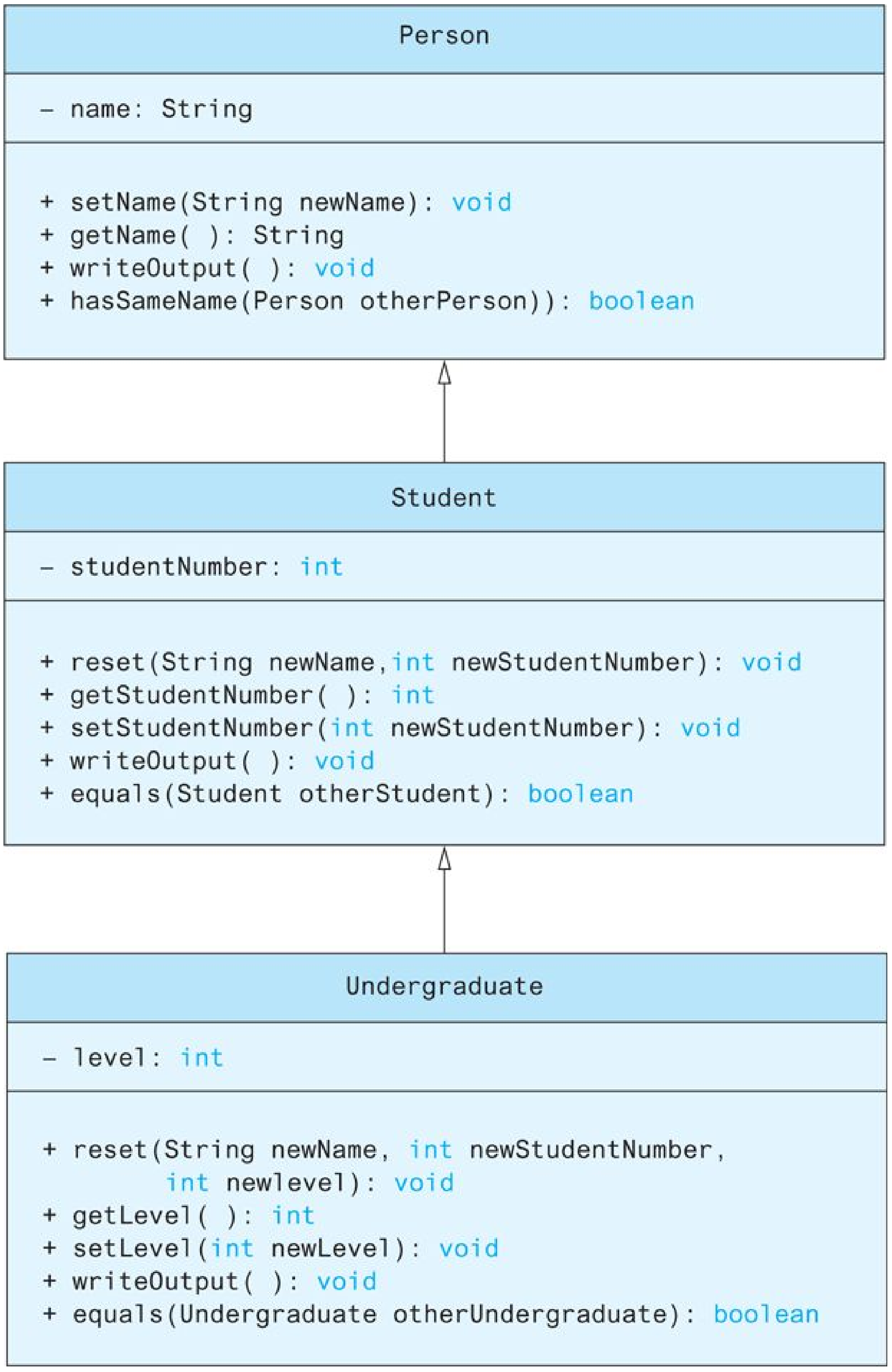
You can form a derived class from a derived class. For example, we previously derived the class **Student** from the class **Person** . We now derive a class **Undergraduate** from Student, as shown in **example below.**

```
public class Undergradute 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
   }
   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);
    }
}
```

UML Class Hierarchy : A UML diagram showing the relationship among the classes Person, Student, and Undergraduate.


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

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**. 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**.

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 **Perso**n and **Student** are invoked, and then all the code following super in the constructor for **Undergraduate** is executed.

The classes **Student** and **Undergraduate** both define a method named **reset**. In **Student**, **reset** has two parameters, while in **Undergraduate**, **reset** has three parameters, so **reset** is an overloaded name. The **reset** method in the class **Undergraduate**, begins by invoking **reset**, passing it only two arguments:

```
public void reset(String newName, int newStudentNumber, int newLevel)
{
   reset(newName, newStudentNumber); //Student's reset
   setLevel(newLevel); //Checks 1 <= newLevel <= 4
}
```

Thus, this invocation is to the method named **reset** defined in the base class **Student**, which changes the values of the instance variables **name** and **studentNumber**. The **Undergraduate** reset method then changes the value of the new instance variable level by calling **setLevel**.

Remember that within the definition of the class **Undergraduate**, the private instance variables **name** and **studentNumber**—of the base classes **Person** and **Student,** respectively—cannot be referenced by name, and so a mutator method is needed to change them.

Because the version of **reset** defined in the class **Undergraduate** has a different number of parameters than the version of **reset** defined in the class **Student**, there is no conflict in having both versions of reset in the derived class **Undergraduate**. In other words, we can overload the reset method. In contrast, the definition of the method **writeOutput** in **Undergraduate,** has exactly the same parameter list as the version of **writeOutput** in the base class Student:

```
public void writeOutput()
{
   super.writeOutput();
   System.out.println("StudentLevel: " + level);
}
```

Thus, when **writeOutput** is invoked, Java must decide which definition of **writeOutput** to use. For an object of the derived class **Undergraduate**, it uses the version of **writeOutput** given in the definition of the class **Undergraduate**. The version in Undergraduate overrides the definition given in the base class Student. To invoke the version of **writeOutput** defined in **Student** within the definition of the derived class **Undergraduate**, you must **place super and a dot** in front of the method name **writeOutput**, as shown previously.

Note: writeOutput overrides the base-class method having the same signature

Now consider the methods named **equals** in the classes **Student** and **Undergraduate.** They have different parameter lists. The one in the class **Student** has a parameter of type **Student**, while the one in the class **Undergraduate** has a parameter of type **Undergraduate**. They have the same number of parameters—namely, one—but that one parameter is of a different type in each of the two definitions. Recall that a difference in type is enough to qualify for overloading. To help us analyze the situation, we reproduce the definition of equals within the derived class Undergraduate here:

```
public boolean equals(Undergraduate otherUndergraduate)
{
return equals((Student)otherUndergraduate) &&

                (this.level == otherUndergraduate.level);
        }
```

Why did we cast **otherUndergraduate** to **Student** in the invocation of **equals**? Because otherwise Java would invoke the definition of equals in the class **Undergraduate.** That is, this **equals** method would invoke itself. By casting the argument from **Undergraduate** to **Student,** we cause Java to call **Student's** equals method.

Another Way to Define the equals Method in Undergraduate

The **equals** method in the class **Undergraduate** casts its parameter **otherUndergraduate** from **Undergraduate** to its base class **Studen**t when it passes it to **Student's** equals method. Another way to force Java to call the definition of **equals** in **Student** is to use **super and a dot**, as follows:

```
public boolean equals(Undergraduate otherUndergraduate)
{
   return super.equals(otherUndergraduate) &&
        (this.level == otherUndergraduate.level);
}
```

Note: 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.