Type Compatibility

Consider the class **Undergraduate**. It is a derived class of the class **Student**. In the real world, every undergraduate is also a student. This relationship holds true in our Java example as well. Every object of the class **Undergraduate** is also an object of the class **Student**. Thus, if you have a method that has a formal parameter of type **Student**, the argument in an invocation of this method can be an object of type **Undergraduate**. In this case, the method could use only methods defined in the class **Student**, but every object of the class **Undergraduate** has all these methods.

For example, suppose that the classes **Student** and **Undergraduate** are defined , and consider the following method definition that might occur in some class:

```
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 than " + s2.getName());
  }
  . . .
}
```

A program that uses **SomeClass** might contain the following code:

```
Student studentObject = new Student("Jane Doe", 1234);
Undergraduate undergradObject =new Undergraduate("Jack Buck", 1234, 1);
SomeClass.compareNumbers(studentObject, undergradObject);
```

If you look at the heading for the method compareNumbers, you will see that both parameters are of type **Student**. However, the invocation

```
SomeClass.compareNumbers(studentObject, undergradObject);
```

uses one argument of type **Student** and one object of type **Undergraduate**. How can we use an object of type Undergraduate where an argument of type Student is required? The answer is that every object of type **Undergraduate** is also of type **Student**. To make the point a little more dramatically, notice that you can reverse the two arguments and the method invocation will still be valid, as shown below:

SomeClass.compareNumbers(undergradObject, studentObject);

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**.

An object can actually behave as if it has more than two types as a result of inheritance. Recall that the class **Undergraduate** is a derived class of the class **Student** and that **Student** is a derived class of the class **Person** . This means that every object of the class **Undergraduate** is also an object of type **Student** as well as an object of type **Person**. Thus, everything that works for objects of the class **Person** also works for objects of the class **Undergraduate**.

Note: An object of a derived class can serve as an object of the base class

Note: An object can have several types because of inheritance

For example, suppose that the classes **Person** and **Undergraduate** are defined,and consider the following code, which might occur in a program:

```
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");
```

If you look at the heading for the method **hasSameName** , you will see that it has one parameter, and that parameter is of type **Person**. However, the call in the preceding if-else statement,

joePerson.hasSameName(someUndergrad)

is perfectly valid, even though the argument **someUndergrad** is an object of the class **Undergraduate**—that is, its type is **Undergraduate**—but the corresponding parameter in **hasSameName** is of type **Person**. Every object of the class Undergraduate is also an object of the class Person.

Even the following invocation is valid:

someUndergrad.hasSameName(joePerson)

The method **hasSameName** belongs to **Person**, but it is inherited by the class **Undergraduate**. So the **Undergraduate** object **someUndergrad** has this method. An object of type **Undergraduate** is also of type **Person**. **Everything that works for objects of an ancestor class also works for objects of any descendant class.** Or, to say this another way, an object of a descendant class can do the same things as an object of an ancestor class. As we have already seen, if class **A** is derived from class **B**, and class **B** is derived from class **C**, then an object of class **A** is of type **A**. It is also of type **B**, and it is also of type **C**. This works for any chain of derived classes, no matter how long the chain is.

Because an object of a derived class has the types of all of its ancestor classes in addition to its "own" type, you can assign an object of a class to a variable of any ancestor type, but not the other way around. For example, because **Student** is a derived class of **Person**, and **Undergraduate** is a derived class of **Student**, the following code is valid:

```
Student s = new Student();
Undergraduate ug = new Undergraduate();
Person p1 = s;
Person p2 = ug;
```

Note: An object of a class can be referenced by a variable of any ancestor type

You can even bypass the variables **s** and **ug** and place the new objects directly into the variables **p1**and **p2**, as follows:

```
Person p1 = new Student();
Person p2 = new Undergraduate();
```

However, the following statements are all illegal:

Student s = new Person(); *//ILLEGAL!*
Undergraduate ug = new Person(); *//ILLEGAL!*
Undergraduate ug2 = new Student(); *//ILLEGAL!*

And if we define **p** and s as follows:

Person p = new Person(); *//valid*
Student s = new Student(); *//valid*

even the following statements, which may look more innocent, are similarly illegal:

Undergraduate ug = p; *//ILLEGAL!*
Undergraduate ug2 = s; *//ILLEGAL!*

This all makes perfectly good sense. For example, a **Student** *is a* **Person**, but a **Person** is not necessarily a **Student**. Some programmers find the phrase "is a" to be useful in deciding what types an object can have and what assignments to variables are valid. As another example, if **Employee** is a derived class of **Person**, **an Employee *is a* Person,** so you can assign an Employee object to a variable of type Person. However, a **Person** *is not necessarily* an **Employee**, so you cannot assign an object created as just a plain **Person** to a variable of type **Employee**.

Note: 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.

The Class Object

In Java, every class is derived from the class **Object**. If a class **C** has a different base class **B**, this base class is derived from **Object**, and so **C** is a derived class of **Object**. Thus, every object of every class is of type **Object**, as well as being of the type of its class and all its ancestor classes. Even classes that you define yourself without using inheritance are descendant classes of the class **Object**. If you do not make your class a derived class of some class, Java will automatically make it a derived class of **Object**.

Note: The class Object is the ultimate ancestor of all classes

The class **Object** does have some methods that every Java class inherits. For example, every class inherits the methods **equals** and **toString** from some ancestor class, either directly from the class **Object** or from a class that ultimately inherited the methods from the class **Object**. 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.

Note: Every class inherits the methods toString and equals from Object

The inherited method **toString** takes no arguments. The method **toString** is supposed to return all the data in an object, packaged into a string. However, the inherited version of **toString** is almost always useless, because it will not produce a nice string representation of the data. You need to override the definition of **toString** so it produces an appropriate string for the data in objects of the class being defined.

For example, the following definition of **toString** could be added to the class **Student**

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

After adding this **toString** method to the class Student, we can use it to display output in the following way:

```
Student joe = new Student("Joe Student", 2001);
System.out.println(joe.toString());
```

The output produced would be

```
Name: Joe Student
Student number: 2001
```

The object System.out has several definitions of println—in other words, **println** is overloaded. One of these println methods has a parameter of type **Object**.

Note: println is an example of a real-life overloaded method