Polymorphism allows you to make changes in the method definition for the derived classes and have those changes apply to the methods written in the base class.

Dynamic Binding and Inheritance

Consider a program that uses the **Person, Student,** and **Undergraduate** classes. Let's say that we would like to set up a committee that consists of four people who are either students or employees. If we use an array to store the list of committee members, then it makes sense to make the array of type **Person** so it can accommodate any class derived from it. Here is a possible array declaration:
Person[ ] people = new Person[4];

Next we might add objects to the array that represent members of the committee. In the example below we have added three objects of type **Undergraduate** and one object of type **Student** (perhaps we don't know if this person is an undergraduate or graduate):

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);

In this case we are assigning an object of a derived class (either **Student** or **Undergraduate**) to a variable defined as an ancestor of the derived class (**Person**). This is valid because **Person** encompasses the derived classes. In other words, **Student "is-a" Person and Undergraduate "is-a" Person**, so we can assign either one to a variable of type **Person**.

Next, let's output a report containing information about all of the committee members. The report should be as detailed as possible. For example, if a **student** is an **undergraduate**, then the report should contain the **student's name, student number, and student level**. If the student is of type **Student**, then the report should contain the **name and student number**. The **writeOutput** method contains this detail, but which one is invoked? There are three of them, one defined for **Undergraduate, Student, and Person.**

If we focus on just **people[0]**, then we can see that it is declared to be an object of type **Person**. If we invoke:

people[0].writeOutput();

then it is logical to assume that the **writeOutput** method defined in the **Person** object will be invoked. But that is not what happens! Instead, Java recognizes that an object of type **Undergraduate** is stored in **people[0]**. As a result, even though **people[0]** is declared to be of type **Person**, the method associated with the class used to create the object is invoked. This is called **dynamic binding** or **late binding.**

More precisely, **when an overridden method is invoked, its action is the one defined in the class used to create the object using the new operator.** It is not determined by the type of the variable naming the object. **A variable of any ancestor class can reference an object of a descendant class, but the object always remembers which method actions to use for every method name.** The type of the variable does not matter. **What matters is the class name when the object was created.**

Returning to our report, the code below could be used to generate it:

```
for (Person p : people)
{
   p.writeOutput();
   System.out.println();
}
```

This code would output:

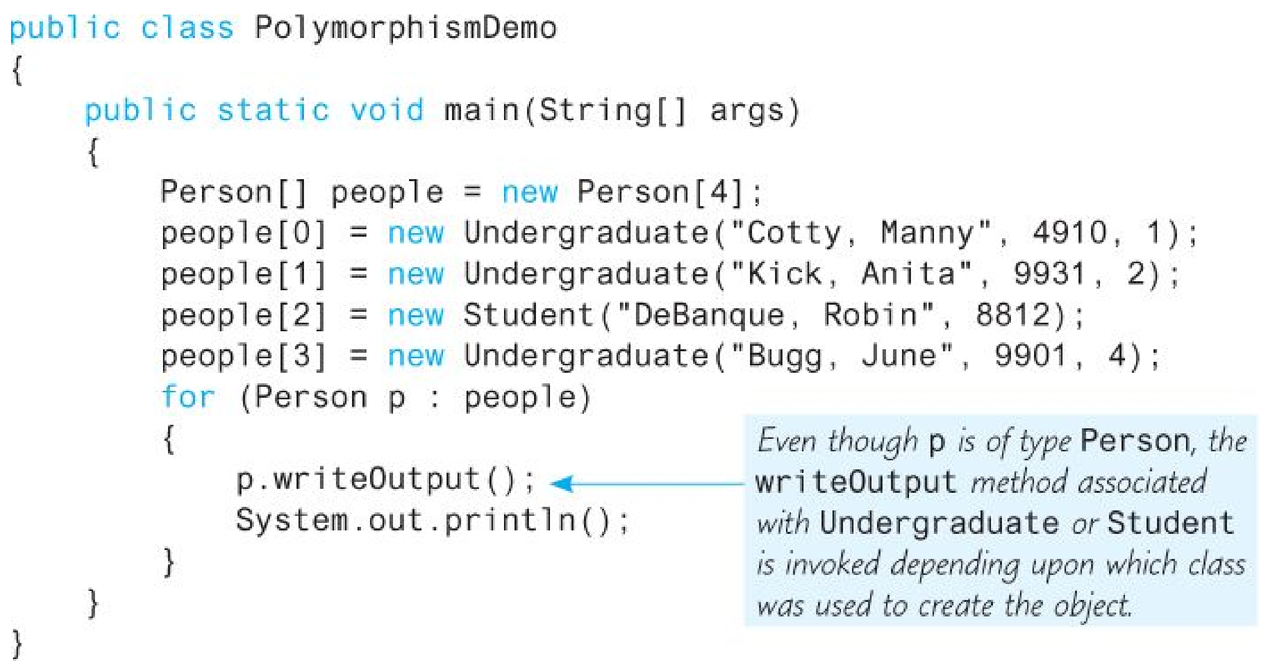
```
Name: Cotty, Manny
Student Number: 4910
Student Level: 1

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

Name: DeBanque, Robin
Student Number: 8812

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

A Demo of Polymorphism



```java
public class PolymorphismDemo
{
    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();
        }
    }
}
```

Even though `p` *is of type* `Person`, *the* `writeOutput` *method associated with* `Undergraduate` *or* `Student` *is invoked depending upon which class was used to create the object.*

Screen 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
```

One of the amazing things about polymorphism is it lets us invoke methods that might not even exist yet! For example, assume the program in above exists and runs with only the **Person, Student,and Undergraduate** classes defined. At some later date we could write the **Employee, Faculty, and Staff classes**. Since all of these classes would be derived from the **Person** class, as long as each implements a **writeOutput** method then we could add one of these objects to the array and its **writeOutput** method would be invoked in the for loop. We wouldn't even need to recompile the **PolymorphismDemo** class to invoke the new methods via dynamic binding.

Note: When an overridden method is invoked, its action is the one defined in the class used to create the object using the new operator. It is not determined by the type of the variable naming the object. A variable of any ancestor class can reference an object of a descendant class, but the object always remembers which method actions to use for every method name. The type of the variable does not matter. What matters is the class name when the object was created. This is because Java uses dynamic binding.

Dynamic Binding with toString

If you include an appropriate **toString** method in the definition of a class, then you can output an object of the class using **System.out.println**. For example, in the previous lesson we described adding a **toString** method to the **Student** class:

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

For the **Student object joe**, we can invoke the method with the statement

```
System.out.println(joe.toString());
```

However, we can get the exact same result without the **toString**:

```
System.out.println(joe);
```

This happens because Java uses dynamic binding with the **toString** method. The various **println** methods that belong to the object **System.out** were written long before we defined the class **Student**. Yet the invocation calls the definition of **toString** in the class **Student**, not the definition of **toString** in the class Object, because **joe** references an object of type **Student**. Dynamic binding is what makes this work. Because **System.out.println** invokes **toString** in this manner, always defining a suitable **toString** method for your classes is a good idea.

Note: With dynamic, or late, binding the definition of a method is not bound to an invocation of the method until run time when the method is called. Polymorphism refers to the ability to associate many meanings to one method name through the dynamic binding mechanism. Thus, polymorphism and dynamic binding are really the same topic.