4

Inheritance and Polymorphism

Say not you know another entirely,
till you have divided an inheritance with him.
—Johann Kaspar Lavater, Aphorisms on Man

Learning Objectives

In this chapter, you will learn:

- → Inheritance
- → Polymorphism
- → Type Compatibility

Inheritance

Superclass and Subclass

Inheritance defines a relationship between objects that share characteristics. Specifically it is the mechanism whereby a new class, called a *subclass*, is created from an existing class, called a *superclass*, by absorbing its state and behavior and augmenting these with features unique to the new class. We say that the subclass *inherits* characteristics of its superclass.

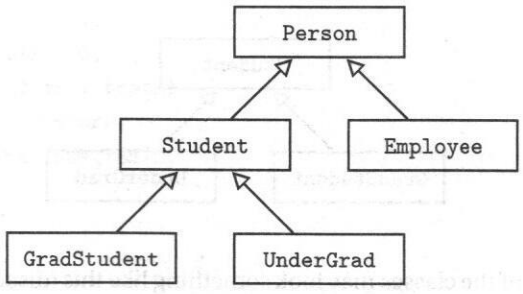
Don't get confused by the names: a subclass is bigger than a superclass—it contains more data and more methods!

Inheritance provides an effective mechanism for code reuse. Suppose the code for a superclass has been tested and debugged. Since a subclass object shares features of a superclass object, the only new code required is for the additional characteristics of the subclass.

Inheritance Hierarchy

A subclass can itself be a superclass for another subclass, leading to an *inheritance hierarchy* of classes.

For example, consider the relationship between these classes: `Person`, `Employee`, `Student`, `GradStudent`, and `UnderGrad`.



For any of these classes, an arrow points to its superclass. The arrow designates an inheritance relationship between classes, or, informally, an *is-a* relationship. Thus, an `Employee` *is-a* `Person`; a `Student` *is-a* `Person`; a `GradStudent` *is-a* `Student`; an `UnderGrad` *is-a* `Student`. Notice that the opposite is not necessarily true: A `Person` may not be a `Student`, nor is a `Student` necessarily an `UnderGrad`.

Note that the *is-a* relationship is transitive: If a `GradStudent` *is-a* `Student` and a `Student` *is-a* `Person`, then a `GradStudent` *is-a* `Person`.

Every subclass inherits the public or protected variables and methods of its superclass (see p. 142). Subclasses may have additional methods and instance variables that are not in the superclass. A subclass may redefine a method it inherits. For example, `GradStudent` and `UnderGrad` may use different algorithms for computing the course grade, and need to change a `computeGrade` method inherited from `Student`. This is called *method overriding*. If part of the original method implementation from the superclass is retained, we refer to the rewrite as *partial overriding* (see p. 143).

Implementing Subclasses

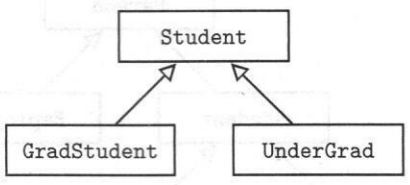
The `extends` Keyword

The inheritance relationship between a subclass and a superclass is specified in the declaration of the subclass, using the keyword `extends`. The general format looks like this:

```
public class Superclass
{
    //private instance variables
    //other data members
    //constructors
    //public methods
    //private methods
}

public class Subclass extends Superclass
{
    //additional private instance variables
    //additional data members
    //constructors  (Not inherited!)
    //additional public methods
    //inherited public methods whose implementation is overridden
    //additional private methods
}
```

For example, consider the following inheritance hierarchy:



The implementation of the classes may look something like this (discussion follows the code):

```
public class Student
{
    //data members
    public final static int NUM_TESTS = 3;
    private String name;
    private int[] tests;
    private String grade;

    //constructor
    public Student()
    {
        name = "";
        tests = new int[NUM_TESTS];
        grade = "";
    }

    //constructor
    public Student(String studName, int[] studTests, String studGrade)
    {
        name = studName;
        tests = studTests;
        grade = studGrade;
    }

    public String getName()
    { return name; }

    public String getGrade()
    { return grade; }

    public void setGrade(String newGrade)
    { grade = newGrade; }

    public void computeGrade()
    {
        if (name.equals(""))
            grade = "No grade";
        else if (getTestAverage() >= 65)
            grade = "Pass";
        else
            grade = "Fail";
    }

    public double getTestAverage()
    {
        double total = 0;
        for (int score : tests)
            total += score;
        return total/NUM_TESTS;
    }
}
```

```
public class UnderGrad extends Student
{
    public UnderGrad()   //constructor
    { super(); }

    //constructor
    public UnderGrad(String studName, int[] studTests, String studGrade)
    { super(studName, studTests, studGrade); }

    public void computeGrade()
    {
        if (getTestAverage() >= 70)
            setGrade("Pass");
        else
            setGrade("Fail");
    }
}

public class GradStudent extends Student
{
    private int gradID;

    public GradStudent()      //constructor
    {
        super();
        gradID = 0;
    }

    //constructor
    public GradStudent(String studName, int[] studTests,
            String studGrade, int gradStudID)
    {
        super(studName, studTests, studGrade);
        gradID = gradStudID;
    }

    public int getID()
    { return gradID; }

    public void computeGrade()
    {
        //invokes computeGrade in Student superclass
        super.computeGrade();
        if (getTestAverage() >= 90)
            setGrade("Pass with distinction");
    }
}
```

Inheriting Instance Methods and Variables

A subclass inherits all the public and protected methods of its superclass. It does not, however, inherit the private instance variables or private methods of its parent class, and therefore does

not have direct access to them. To access private instance variables, a subclass must use the accessor or mutator methods that it has inherited.

In the `Student` example, the `UnderGrad` and `GradStudent` subclasses inherit all of the methods of the `Student` superclass. Notice, however, that the `Student` instance variables `name`, `tests`, and `grade` are private and are therefore not inherited or directly accessible to the methods in the `UnderGrad` and `GradStudent` subclasses. A subclass can, however, directly invoke the public accessor and mutator methods of the superclass. Thus, both `UnderGrad` and `GradStudent` use `getTestAverage`. Additionally, both `UnderGrad` and `GradStudent` use `setGrade` to access indirectly—and modify—`grade`.

If, instead of `private`, the access specifier for the instance variables in `Student` were `public` or `protected`, then the subclasses could directly access these variables. The keyword `protected` is not part of the AP Java subset.

Classes on the same level in a hierarchy diagram do not inherit anything from each other (for example, `UnderGrad` and `GradStudent`). All they have in common is the identical code they inherit from their superclass.

Method Overriding and the `super` Keyword

Any public method in a superclass can be overridden in a subclass by defining a method with the same return type and signature (name and parameter types). For example, the `computeGrade` method in the `UnderGrad` subclass overrides the `computeGrade` method in the `Student` superclass.

Sometimes the code for overriding a method includes a call to the superclass method. This is called *partial overriding*. Typically this occurs when the subclass method wants to do what the superclass does, plus something extra. This is achieved by using the keyword `super` in the implementation. The `computeGrade` method in the `GradStudent` subclass partially overrides the matching method in the `Student` class. The statement

```
super.computeGrade();
```

signals that the `computeGrade` method in the superclass should be invoked here. The additional test

```
if (getTestAverage() >= 90)
    ...
```

allows a `GradStudent` to have a grade `Pass with distinction`. Note that this option is open to `GradStudents` only.

Note

Private methods cannot be overridden.

Constructors and `super`

Constructors are never inherited! If no constructor is written for a subclass, a *default constructor* with no parameters is generated, which calls the no-argument constructor from the superclass. If the superclass does not have a no-argument constructor, but only a constructor with parameters, a compiler error will occur. If there is a no-argument constructor in the superclass, inherited data members will be initialized as for the superclass. Additional instance

Be sure to provide at least one constructor for a subclass. Constructors are never inherited from the superclass.

variables in the subclass will get a default initialization—0 for primitive types and `null` for reference types.

> A default constructor is a no-argument constructor generated by the compiler when a subclass doesn't have an explicit constructor.

A subclass constructor can be implemented with a call to the `super` method, which invokes the superclass constructor. For example, the no-argument constructor in the `UnderGrad` class is identical to that of the `Student` class. This is implemented with the statement

```
super();
```

The second constructor in the `UnderGrad` class is called with parameters that match those in the constructor of the `Student` superclass.

```
public UnderGrad(String studName, int[] studTests, String studGrade)
{ super(studName, studTests, studGrade); }
```

For each constructor, the call to `super` has the effect of initializing the instance variables `name`, `tests`, and `grade` exactly as they are initialized in the `Student` class.

Contrast this with the constructors in `GradStudent`. In each case, the instance variables `name`, `tests`, and `grade` are initialized as for the `Student` class. Then the new instance variable, `gradID`, must be explicitly initialized.

```
public GradStudent()
{
    super();
    gradID = 0;
}

public GradStudent(String studName, int[] studTests,
            String studGrade, int gradStudID)
{
    super(studName, studTests, studGrade);
    gradID = gradStudID;
}
```

Note

1. If `super` is used in the implementation of a subclass constructor, it *must* be used in the first line of the constructor body.
2. If no constructor is provided in a subclass, the compiler provides the following default constructor:

```
public SubClass()
{
    super();      //calls no-argument constructor of superclass
}
```

3. If the superclass has at least one constructor with parameters, the code in Note 2 above will cause a compile-time error if the superclass does not also contain a no-argument constructor.

Rules for Subclasses

- A subclass can add new private instance variables.
- A subclass can add new public, private, or static methods.
- A subclass can override inherited methods.
- A subclass may not redefine a public method as private.
- A subclass may not override static methods of the superclass.
- A subclass should define its own constructors.
- A subclass cannot directly access the private members of its superclass. It must use accessor or mutator methods.

Declaring Subclass Objects

When a superclass object is declared in a client program, that reference can refer not only to an object of the superclass, but also to objects of any of its subclasses. Thus, each of the following is legal:

```
Student s = new Student();
Student g = new GradStudent();
Student u = new UnderGrad();
```

This works because a `GradStudent` *is-a* `Student`, and an `UnderGrad` *is-a* `Student`.

Note that since a `Student` is not necessarily a `GradStudent` nor an `UnderGrad`, the following declarations are *not* valid:

```
GradStudent g = new Student();
UnderGrad u = new Student();
```

Consider these valid declarations:

```
Student s = new Student("Brian Lorenzen", new int[] {90,94,99},
        "none");
Student u = new UnderGrad("Tim Broder", new int[] {90,90,100},
        "none");
Student g = new GradStudent("Kevin Cristella",
        new int[] {85,70,90}, "none", 1234);
```

Suppose you make the method call

```
s.setGrade("Pass");
```

The appropriate method in `Student` is found and the new grade assigned. The method calls

```
g.setGrade("Pass");
```

and

```
u.setGrade("Pass");
```

achieve the same effect on g and u since GradStudent and UnderGrad both inherit the setGrade method from Student. The following method calls, however, won't work:

```
int studentNum = s.getID();
int underGradNum = u.getID();
```

These don't compile because Student does not contain getID.

Now consider the following valid method calls:

```
s.computeGrade();
g.computeGrade();
u.computeGrade();
```

Since s, g, and u have all been declared to be of type Student, will the appropriate method be executed in each case? That is the topic of the next section, *polymorphism.*

Note

The initializer list syntax used in constructing the array parameters—for example, `new int[] {90,90,100}`— will not be tested on the AP exam.

Polymorphism

A method that has been overridden in at least one subclass is said to be *polymorphic.* An example is computeGrade, which is redefined for both GradStudent and UnderGrad.

Polymorphism is the mechanism of selecting the appropriate method for a particular object in a class hierarchy. The correct method is chosen because, in Java, method calls are always determined by the type of the *actual object,* not the type of the object reference. For example, even though s, g, and u are all declared as type Student, s.computeGrade(), g.computeGrade(), and u.computeGrade() will all perform the correct operations for their particular instances. In Java, the selection of the correct method occurs *during the run of the program.*

Here is a simple example.

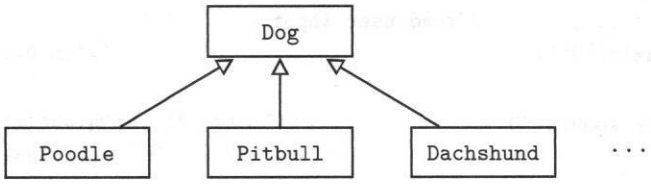
```
public class Dog
{
    private String name;
    private String breed;

    public Dog (String aName, String aBreed)
    {
        name = aName;
        breed = aBreed;
    }
        ...
}
```

```
public class Poodle extends Dog
{
    private boolean needsGrooming;

    public Poodle (String aName, String aBreed, boolean grooming)
    {
        super(aName, aBreed);
        needsGrooming = grooming;
    }
        ...
}
```

Now consider this hierarchy of classes, and the declarations that follow it:



Suppose the `Dog` class has this method:

```
public void eat()
{ /* implementation not shown */ }
```

And each of the subclasses, `Poodle`, `PitBull`, `Dachshund`, etc., has a different, overridden `eat` method. Now suppose that `allDogs` is an `ArrayList<Dog>` where each `Dog` declared above has been added to the list. Each `Dog` in the list will be processed to eat by the following lines of code:

```
for (Dog d: allDogs)
    d.eat();
```

Polymorphism is the process of selecting the correct `eat` method, during run time, for each of the different dogs.

Recall that constructors are not inherited, and if you use the keyword `super` in writing a constructor for your subclass, the line containing it should precede any other code in the constructor.

Recall also that if the superclass does not have a no-argument constructor, then you must write an explicit constructor for the subclass. If you don't do this, later code that tries to create a subclass object will get a compile-time error. This is because the compiler will generate a default constructor for the subclass that tries to invoke the no-argument constructor of the superclass.

Dynamic Binding (Late Binding)

Making a run-time decision about which instance method to call is known as *dynamic binding* or *late binding*. Contrast this with selecting the correct method when methods are *overloaded* (see p. 106) rather than overridden. The compiler selects the correct overloaded method at compile time by comparing the methods' signatures. This is known as *static binding*, or *early*

binding. In polymorphism, the actual method that will be called is not determined by the compiler. Think of it this way: The compiler determines *if* a method can be called (i.e., does the method exist in the class of the reference?), while the run-time environment determines *how* it will be called (i.e., which overridden form should be used?).

Example 1

```
Student s = null;
Student u = new UnderGrad("Tim Broder", new int[] {90,90,100},
        "none");
Student g = new GradStudent("Kevin Cristella",
        new int[] {85,70,90}, "none", 1234);
System.out.print("Enter student status: ");
System.out.println("Grad (G), Undergrad (U), Neither (N)");
String str = ...;       //read user input
if (str.equals("G"))
    s = g;
else if (str.equals("U"))
    s = u;
else
    s = new Student();
s.computeGrade();
```

When this code fragment is run, the `computeGrade` method used will depend on the type of the actual object `s` refers to, which in turn depends on the user input.

Example 2

```
public class StudentTest
{
    public static void computeAllGrades(Student[] studentList)
    {
        for (Student s : studentList)
            if (s != null)
                s.computeGrade();
    }

    public static void main(String[] args)
    {
        Student[] stu = new Student[5];
        stu[0] = new Student("Brian Lorenzen",
                             new int[] {90,94,99}, "none");
        stu[1] = new UnderGrad("Tim Broder",
                             new int[] {90,90,100}, "none");
        stu[2] = new GradStudent("Kevin Cristella",
                             new int[] {85,70,90}, "none", 1234);
        computeAllGrades(stu);
    }
}
```

Polymorphism applies only to overridden methods in subclasses.

Here an array of five `Student` references is created, all initially null. Three of these references, `stu[0]`, `stu[1]`, and `stu[2]`, are then assigned to actual objects. The `computeAllGrades`

method steps through the array invoking for each of the objects the appropriate `computeGrade` method, using dynamic binding in each case. The null test in `computeAllGrades` is necessary because some of the array references could be null.

Using super in a Subclass

A subclass can call a method in its superclass by using `super`. Suppose that the superclass method then calls another method that has been overridden in the subclass. By polymorphism, the method that is executed is the one in the subclass. The computer keeps track and executes any pending statements in either method.

Example

```
public class Dancer
{
    public void act()
    {
        System.out.print (" spin ");
        doTrick();
    }

    public void doTrick()
    {
        System.out.print (" float ");
    }
}

public class Acrobat extends Dancer
{
    public void act()
    {
        super.act();
        System.out.print (" flip ");
    }

    public void doTrick()
    {
        System.out.print (" somersault ");
    }
}
```

Suppose the following declaration appears in a class other than `Dancer` or `Acrobat`:

```
Dancer a = new Acrobat();
```

What is printed as a result of the call `a.act()`?

Solution

When `a.act()` is called, the `act` method of `Acrobat` is executed. This is an example of polymorphism. The first line, `super.act()`, goes to the `act` method of `Dancer`, the superclass. This prints `spin`, then calls `doTrick()`. Again, using polymorphism, the `doTrick` method in `Acrobat` is called, printing `somersault`. Now, completing the `act` method of `Acrobat`, `flip` is printed. So the complete output is the following:

```
spin somersault flip
```

Note

Even though there are no constructors in either the `Dancer` or `Acrobat` classes, the declaration

```
Dancer a = new Acrobat();
```

compiles without error. This is because `Dancer`, while not having an explicit superclass, has an implicit superclass, `Object`, and a default constructor is generated that calls `super()`, `Object`'s no-argument constructor. Similarly the `Acrobat` class gets this constructor slotted into its code.

The statement `Dancer a = new Acrobat();` will not compile, however, if the `Dancer` class has at least one constructor with parameters but does not have a no-argument constructor.

Type Compatibility

Downcasting

Consider the statements

```
Student s = new GradStudent();
GradStudent g = new GradStudent();
int x = s.getID();              //compile-time error
int y = g.getID();              //legal
```

Both `s` and `g` represent `GradStudent` objects, so why does `s.getID()` cause an error? The reason is that `s` is of type `Student`, and the `Student` class doesn't have a `getID` method. At compile time, only nonprivate methods of the `Student` class can appear to the right of the dot operator when applied to `s`. Don't confuse this with polymorphism: `getID` is not a polymorphic method. It occurs in just the `GradStudent` class and can therefore be called only by a `GradStudent` object.

The error shown above can be fixed by casting `s` to the correct type:

```
int x = ((GradStudent) s).getID();
```

Since `s` (of type `Student`) is actually representing a `GradStudent` object, such a cast can be carried out. Casting a superclass to a subclass type is called a *downcast.*

Note

1. The outer parentheses are necessary, so

   ```
   int x = (GradStudent) s.getID();
   ```

   will still cause an error, despite the cast. This is because the dot operator has higher precedence than casting, so `s.getID()` is invoked before `s` is cast to `GradStudent`.
2. The statement

   ```
   int y = g.getID();
   ```

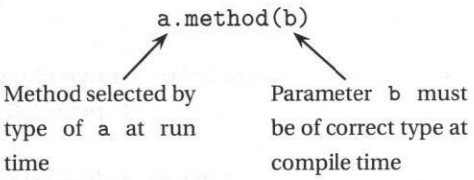
   compiles without problem because `g` is declared to be of type `GradStudent`, and this is the class that contains `getID`. No cast is required.
3. Class casts will not explicitly be tested on the AP exam. You should, however, understand why the following statement will cause a compile-time error:

```
int x = s.getID(); //No getID method in Student class
```

And the following statement will compile without error:

```
int y = g.getID(); //getID method is in GradStudent class
```

Type Rules for Polymorphic Method Calls



- For a declaration like

  ```
  Superclass a = new Subclass();
  ```

  the type of `a` at compile time is `Superclass`; at run time it is `Subclass`.
- At compile time, `method` must be found in the class of `a`, that is, in `Superclass`. (This is true whether the method is polymorphic or not.) If `method` cannot be found in the class of `a`, you need to do an explicit cast on `a` to its actual type.
- For a polymorphic method, at run time the actual type of `a` is determined—`Subclass` in this example—and `method` is selected from `Subclass`. This could be an inherited method if there is no overriding method.
- The type of parameter `b` is checked at compile time. It must pass the *is-a* test for the type in the method declaration. You may need to do an explicit cast to a subclass type to make this correct.

Abstract Classes

Abstract classes are no longer part of the AP Java subset.

Interfaces

Interfaces are no longer part of the AP Java subset.

Chapter Summary

You should be able to write your own subclasses, given any superclass.

Be sure you understand the use of the keyword `super`, both in writing constructors and calling methods of the superclass.

You should understand what polymorphism is: Recall that it only operates when methods have been overridden in at least one subclass. You should also be able to explain the difference between the following concepts:

- An overloaded method and an overridden method
- Dynamic binding (late binding) and static binding (early binding)
- A default constructor and a no-argument constructor