3. Extending Classes by Means of Inheritance

Chapter Three explores one of the cornerstones of OOP, namely that of *inheritance*. Java classes can be easily modified and extended for the purpose of re-use by means of *inheritance* rather than by re-writing the source code of the class to be modified.

3.1 What Does Inheritance Mean?

In order to explain what inheritance means in the Java language, let us consider a very simple example. Suppose that we have a class called **MyClass**, with the following trivial class definition that declares a single member:

```
public class MyClass {

        private int someValue;

}
```

and suppose that we wish to *re-use* this class and, in doing so, we wish to give the replica a *different* name. To do this, we would write the following class definition:

```
public class MyOtherClass {

        private int someValue;

}
```

In other words, we have re-written the body of **MyClass** in the class definition of **MyOtherClass**. In our trivial example, the effort of re-writing the body of **MyClass** is minimal in that there in only one member to declare. In the general case, on the other hand, the effort of re-writing a large class definition ought to be avoidable.

The concept of *inheritance* is common to OOP languages and avoids re-writing code from one class to another. In the example above, the replica class definition is written as follows, to take advantage of the concept of inheritance as it applies in Java.

```
public class MyOtherClass extends MyClass {
        // the variable someValue is inherited and does not have to be declared again
}
```

The keyword **extends** indicates that the class **MyOtherClass** *inherits* the single member of the class **MyClass** so that it becomes a members of **MyOtherClass**.

The use of the keyword **extends** in the simple example above raises a question: *what is the purpose of replicating MyClass in the guise of MyOtherClass when we could simply use it as many times as we wish?* The answer to this question lies in the fact that inheritance means that inherited members can either be modified or left unchanged. This important concept means that we can modify an inherited method if we wish to and we are free to leave it (or other) inherited methods unchanged. We can also provide *additional* methods in an extended class.

The next example enhances the one above to illustrate the true value of extending a class.

```
public class MyClass {

        private int someValue;

        public void someMethod( String string ) {
                System.out.println( “This is the parent class.” );
        }

}
```

and

```
public class MyOtherClass extends MyClass{

        // MyOtherClass inherits both members of MyClass.

}
```

Given that **MyOtherClass** inherits both members of **MyClass**, consider the test class shown next.

```
public class TestClass {
         public static void main( String[ ] args ) {
                MyOtherClass moc = new MyOtherClass( );
                System.out.println( "The value of someValue is: " + moc.someValue );
        }
}
```

When an attempt is made to compile the test class, the compiler outputs the following message:

someValue has private access in MyClass

The purpose of this simple example is to show that although the private variable **someValue** is inherited by **MyOtherClass**, it is a private variable and, as such, is not directly accessible by selecting the member via a reference to an object of the class **MyOtherClass**. The compiler highlights the statement

```
System.out.println( "The value of someValue is: " + moc.someValue );
```

when it outputs its message because it is complaining that

moc . someValue

attempts to access a private variable, albeit an inherited one. Thus we can see that the compiler is consistent when we attempt to access a private variable directly whether or not the variable is inherited.

In order to access the variable **someValue**, we could write an accessor method in **MyClass** and invoke it via a reference to an object of **MyOtherClass** in a test class as shown next.

```
public class MyClass {
        private int someValue = 42;

        public void someMethod( String string ) {
                System.out.println( “This is the parent class.” );
        }

        public int getSomeValue( ) {
                return someValue;
        }
}
```

```
public class TestClass {
        public static void main( String[ ] args ) {
                MyOtherClass moc = new MyOtherClass( );
                System.out.println( "The value of someValue is: " +
                        moc.getSomeValue( ) );
        }
}
```

Executing **main** produces the following output.

The value of someValue is: 42

and shows that the public accessor method **getSomeValue** is inherited by **MyOtherClass**.

Alternatively, we could *modify* the inherited method **getSomeValue** in order to carry out some simple processing on **someValue** as shown next.

```
public class MyOtherClass extends MyClass {

        public int getSomeValue( ) {

                someValue ++ ;
                return someValue;

        }

}
```

The modified method does not compile because it attempts to access an inherited private variable. However, the following modification of the method *does* compile:

```
public class MyOtherClass extends MyClass {

        public int getSomeValue( ) {

                int value = 0;
                // invoke getSomeValue in MyClass
                value = super.getSomeValue( );
                value = value + 1;
                return value;

        }

}
```

The same test class produces the following output; note the value is 43, not 42:

The value of someValue is: 43

The statement

value = super . getSomeValue( );

includes a call to **getSomeValue** 'on' an object with the reference 'super'. The use of the keyword **super** in this context refers to an object of the class **MyClass** from which **MyOtherClass** inherits its members and shows that we can invoke methods of an extended class in the extending class by referring to the former via the object reference *super*.

The relationship between **MyOtherClass** and **MyClass** is such that the latter is said to be the *superclass* of the former or, looking at the relationship the other way round, the former is a *subclass* of the latter. The IDE used to compile the three classes referred to in this section displays this relationship in the screen shot shown in the next figure.

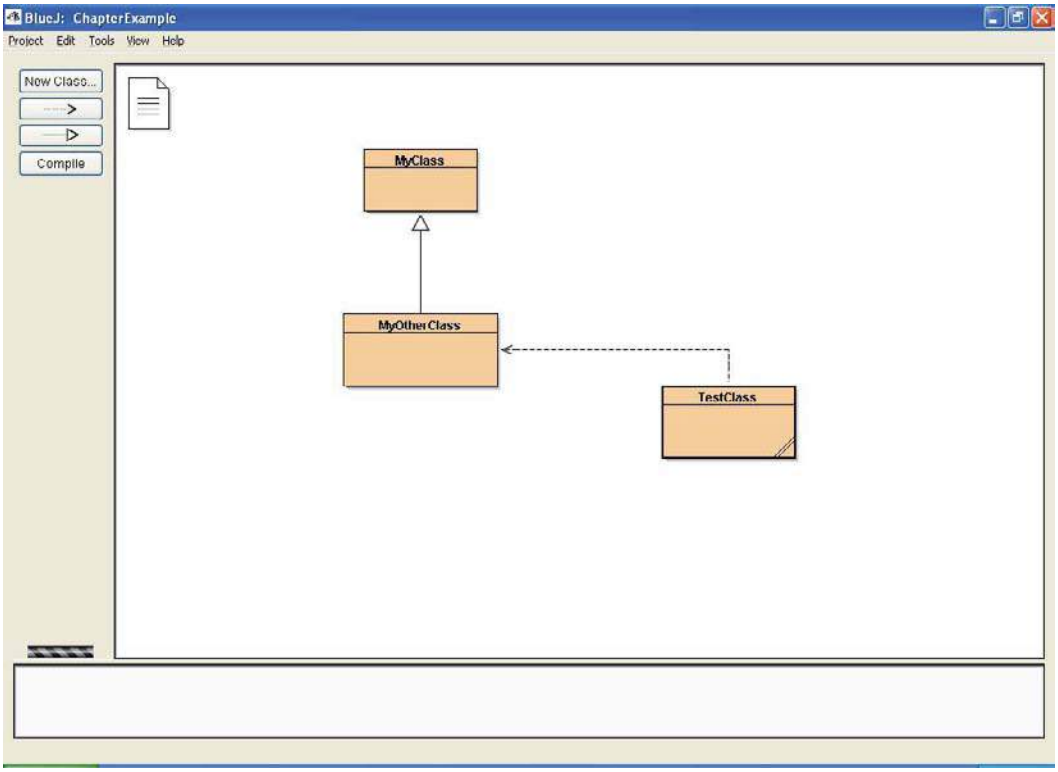

Figure 3.1 The relationship amongst classes used in Section 3.1

The solid line with the closed arrow indicates that **MyOtherClass** is a subclass of its superclass **MyClass**; in other words, **MyOtherClass** has an *'is a'* relationship with its superclass. The dotted line with the open arrow shows that **TestClass** has a *'has a'* relationship with **MyOtherClass** because **TestClass** declares a local variable of the **MyOtherClass** type in its **main** method. (The dotted line between **MyOtherClass** and **TestClass** should be a straight line according to the conventions of UML diagrams. It is a quirk of the IDE that it does not draw straight lines for 'has a' relationships.)

The simple example discussed in this section shows the superclass-to-subclass relationship between two classes. In fact, *all* classes written in Java inherit *implicitly* from a class whose type is **`Object`** and, as a consequence, inherit the members of the class **`Object`**. The following extract from the API shows some of the members of the **`Object`** class.

java.lang

Class Object

`java.lang.Object`

`public class` **`Object`**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Method Summary	
protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	**getClass**() Returns the runtime class of this Object.
int	**hashCode**() Returns a hash code value for the object.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
	wait is overloaded; not shown here.

Thus, the class declaration for MyClass

public class MyClass

actually implies

public class MyClass extends Object

The *extends* clause is omitted because *every* Java object inherits from the class **Object**.

The next extract from the API shows that all classes provided by the Java development environment inherit from the class **Object**.

java.lang
Class Short
java.lang.Object
└ java.lang.Number
└ **java.lang.Short**

The extract indicates that the class **Short** is a subclass of **Number** which is, in turn, a subclass of **Object**.

Before we move on to explore aspects of inheritance in Java more closely, let us summarise the principal concepts introduced or implied by the discussion in this section.

- Inheritance is the ability to create new classes from existing ones;
- Java exhibits *single* inheritance; i.e. a subclass can have only one superclass;
- fields and methods are inherited in a subclass; new ones can be introduced;
- constructors are not inherited;
- a class called Object is at the top of the inheritance tree, as shown by the Java API;
- all Java classes implicitly inherit from Object;
- a subclass can modify methods inherited from its parent class; this is known as *overriding*; thus, an instance method with the same signature and return type as a method in the superclass is said to 'override' it;
- the keyword *super* is used to refer to the members of a superclass, i.e. variables, methods and constructors; thus, an overriding method can invoke the overridden method using the keyword *super*;
- a method invocation does not have to be on a method in the superclass; it can be to a method further up the class hierarchy.

3.2 Overriding and Hiding Methods in a Subclass

Section 3.1 gives a simple example of overriding a method in a subclass. In essence, method overriding is a means by which inherited behaviour can be modified to suit the specific logic of a subclass, where this is derived from the general logic of its superclass. Thus, we can think of a superclass comprising members that are common to its subclasses. Common members are inherited and may or may not be modified in each subclass as required.

Consider, for example, the simple class hierarchy shown in the Figure 3.2 below.

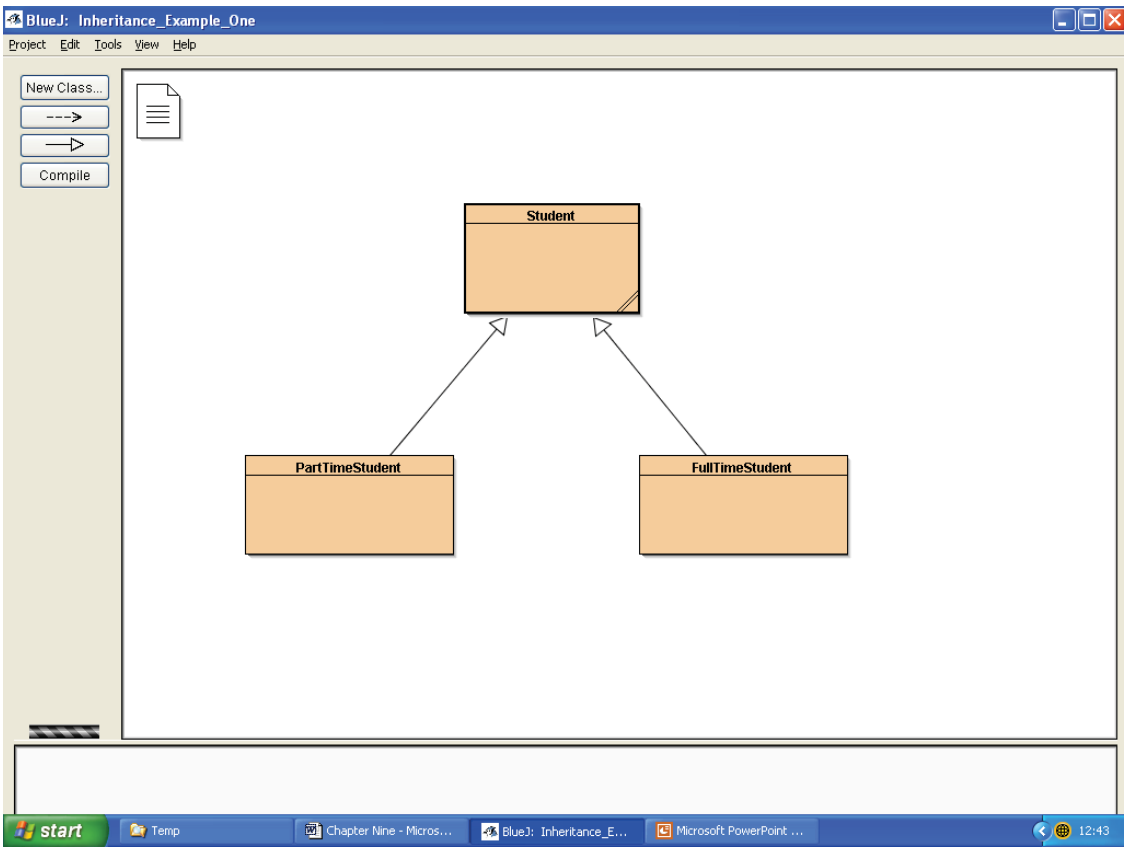

Figure 3.2 A superclass and two of its subclasses

The class definitions of the three classes follow on the next two pages: some of the documentation has been omitted for the sake of brevity.

```
public class Student {

    // Declare instance variables.
    private int idNumber;
    private String degreeCourseCode;
    private int yearOfEnrollment = 2008;

    public Student( int idNumber, String degreeCourseCode ) {
        // Intialise two of the instance variables.
```

```
            this.idNumber = idNumber;
            this.degreeCourseCode = degreeCourseCode;
        }

        public int getIdNumber( ) {
            return idNumber;
        }

        public String getDegreeCourseCode(  ) {
            return "course code " + degreeCourseCode;
        }

        public int getYearOfEnrollment( ) {
            return yearOfEnrollment;
        }

} // End of class definition.

public class FullTimeStudent extends Student {

        // Declare instance variables.
        private String fullName;

        public FullTimeStudent( int idNumber,
            String degreeCourseCode ) {

            // Call the constructor for the superclass.
            super( idNumber, degreeCourseCode );
        }


        // This method overrides the getDegreeCourseCode method in Student; it also calls
        // the overridden method.
        public String getDegreeCourseCode(  ) {
            return "This full-time student is enrolled on: "  +
                    super.getDegreeCourseCode( );
        }

} // End of class definition.

public class PartTimeStudent extends Student {

        // Declare instance variables.
        private String fullName;

        public PartTimeStudent(int idNumber,
```

```
            String degreeCourseCode) {
            // Call the constructor for the superclass.
            super( idNumber, degreeCourseCode );
    }

    // This method overrides the getDegreeCourseCode method in Student; it also calls
    // the overridden method.
    public String getDegreeCourseCode( ) {
            return "This part-time student is enrolled on: "
                    + super.getDegreeCourseCode( );
    }

} // End of class definition.
```

An examination of the source code illustrates how an instance method with the same signature and return type as a method in the superclass overrides it. Thus, the method **getDegreeCourseCode** in **PartTimeStudent** overrides **getDegreeCourseCode** in **Student**. The example also shows how the keyword *super* is used to invoke the overridden method in the body of the overriding method.

(One of the rules that govern method overriding states that an overriding method cannot throw different types of **Exception** objects than the overridden method. We will find out how exceptions are handled in the next chapter.)

A *class* method (i.e. a *static* method) with the same signature as a class method in the superclass is said to "hide" it. For class methods, the runtime system invokes the method defined in the *compile-time* type of the reference on which the method is called; for instance methods, the run-time system invokes the method defined in the *run-time* type of the reference on which the method is called. This is illustrated by providing the following method in the **Student** class definition.

```
public static String getDetails( ) {
        return "I am a student.";
}
```

When the **main** method shown next is run

```
public class TestStudents {

        public static void main( String[ ] args ) {

                // Create a Student object.  The variable s is of the Student type, but refers to
                // a FullTimeStudent object (at run-time).
                Student s = new FullTimeStudent( 1234, "Java" );
                // Call the static method of Student.
                System.out.println( s.getDetails( ) );
                // Call an instance method of FullTimeStudent.
                System.out.println( s.getDegreeCourseCode( ) );
        }
}
```

the output is:

```
I am a student.
This full-time student is enrolled on: course code Java
```

and illustrates the difference between a method invocation on a compile-time type and a run-time type. There is one further rule to state concerning class methods at this point: *an instance method cannot override a static method and a static method cannot hide an instance method.*

3.3 Invoking a Parent Class Constructor from a Subclass Constructor

A number of examples in previous sections show how the keyword *super* is used to invoke an overridden method. The keyword *super* is also used to invoke a parent class constructor from a constructor of a subclass. In fact, the call to super(< parameter list >) *must* be the first statement of a subclass constructor, as illustrated above in the constructor of **PartTimeStudent** and **FullTimeSudent.**

There are a number of rules that pertain to the call to super(< parameter list >):

- it is necessary to initialise all fields of a superclass; therefore its constructor must be called;
- a specific constructor is called as determined by the arguments that are passed to the call to *super*; this is illustrated by the first statement in the constructor for FullTimeStudent;
- if no call to *super* is used in a subclass constructor, the compiler adds an implicit call to super( ) that calls the parent, no argument constructor (which could be its default constructor);
- if the parent class define constructors but does not provide a no argument constructor, an error message is issued by the compiler if a call to super( ) is made from a subclass.

Given these rules, care should be taken when calling the correct superclass constructor from the first statement of a subclass constructor.

3.4 final and abstract Classes

Figure 3.3 shows an enhanced version of the simple class hierarchy shown in Figure 3.2.

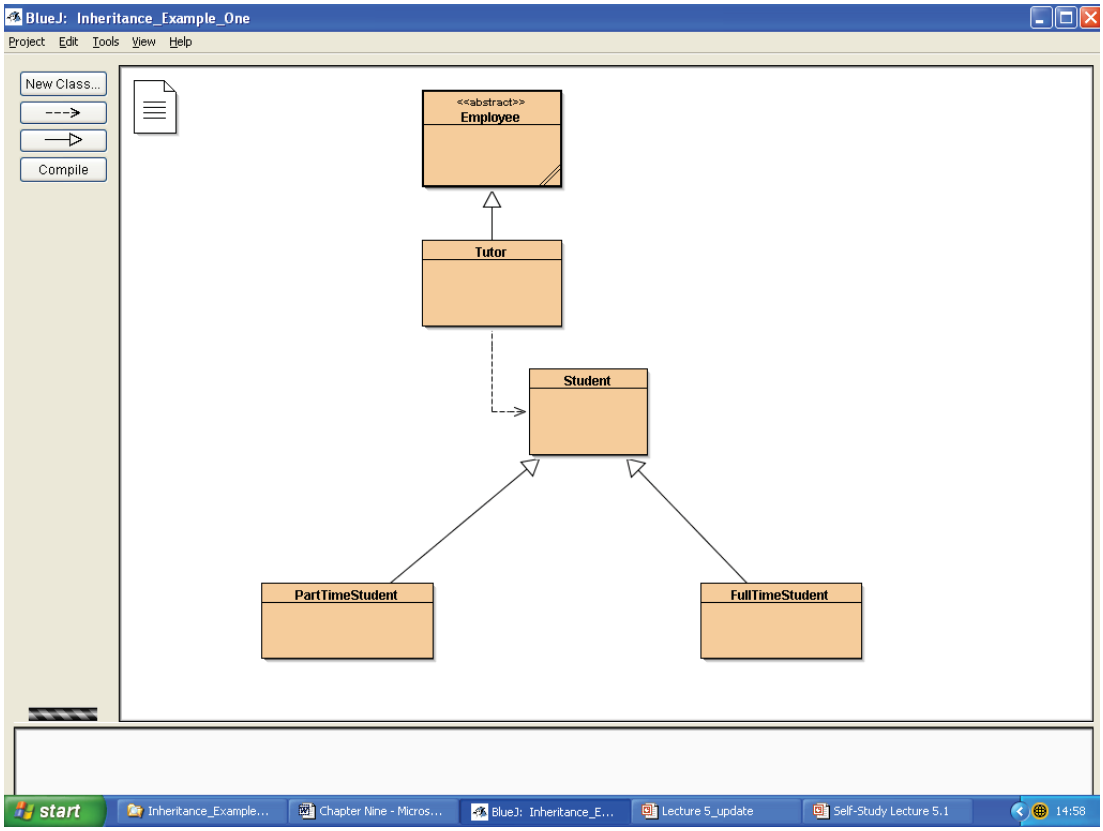

Figure 3.3 An enhanced version of Figure 3.2

The **Employee** class is labelled (by the IDE) as **abstract**. For the purposes of this simple example, the **Employee** class is defined to be **abstract** because we do not want to instantiate objects of the **Employee** class. Instead, we wish to instantiate objects of the **Tutor** class, in other words we wish to populate the application with *concrete* employees such as **Tutor**, rather than use the abstract and generalised notion of an employee. The class definitions for **Employee** is shown next.

```
public abstract class Employee {

    // Declare instance variables.
    String department;

    /**
    * This constructor initialises the variable with the identifier department.
     * @param department  The employee's department.
     */
    public Employee( String department ) {
            this.department = department;
    }

    // All subclasses of Employee must implement this abstract method.
    // Note how it is declared.
    public abstract String getDepartment( );

} // End of class definition.
```

The class definition of **Tutor** is next.

```
public class Tutor extends Employee {

    // Declare instance variables.
    private String tutorsName;
    // An array to store references to students.
    private Student [ ] studentsInJavaGroup;

    /**
    * Constructor for objects of class Tutor.
    * @param department  The tutor's department.
    */
    public Tutor( String department ) {
        // Call the constructor of the superclass.
        super( department );
    }

    /**
    * Abstact method inherited from Employee; must be overridden.
    */
    public String getDepartment( ) {
        return department;
    }

} // End of class definition.
```

It should be noted that an **abstract** class may declare **abstract** methods. For example, the **abstract** class **Employee** declares an **abstract** method **getDepartment** that must be overridden in sublclasses of **Employee**.

The aim of the example shown in Figure 3.3 is to illustrate some of the following rules concerning **abstract** classes:

- the complier prevents an abstract class from being directly instantiated, though it usually has constructors that are called from the constructor of its subclasses;
- an abstract class may have abstract methods:
  - such methods contain no implementation;
  - non-abstract subclasses must override these methods and implement them;
  - if all methods are abstract, the class should be an *interface*; (we will find out what a Java interface is in Chapter Five);
- any class with one or more abstract methods is called an abstract class;
- abstract classes can have data attributes, concrete methods and constructors.

At this point, it is useful to note that there is another category of class known as a **final** class. A **final** class cannot be subclassed and a **final** method cannot be overridden.

3.5 What Does Type Compatibility Mean?

Now that we have explored some of the essential concepts associated with inheritance, we can address the deferred discussion about conversion of type variables from Chapter Three (An Introduction to Java Programming 3: The Fundamentals of Objects and Classes).

> Java is a *strongly-typed* language. This means that the compiler checks for *type compatibility* at compile time, preventing incompatible assignments.

Checking for type compatibility is carried out when an expression is assigned to a type variable as follows:

SomeClass sc = < expression that returns an object reference >;

The compiler will regard the types as compatible when one of three conditions applies to the expression:

1. the expression returns an object reference of the SomeClass type;
2. the expression returns an object reference to a subclass of SomeClass;
3. the expression returns an object reference to an object that implements the SomeClass interface. (Java interfaces are explored in Chapter Five.)

We can use the class hierarchy shown in Figure 3.3 to illustrate the first two conditions.

Consider the following statement:

Student s1 = new Student(111, "Java");

This statement compiles because it complies with the first condition.

Consider the next statement:

Student s2 = new PartTimeStudent(222, "Java");

This statement compiles because it complies with the second condition.

It is worthwhile dwelling on the general nature of the second condition:

SuperClass sc = new SubClass( );

Up to this point in the guide, we have created objects of a particular class by calling one of the constructors of that class. However as the statement shows, this condition of compatibility is permitted in Java. In other words, while the class of an existing object doesn't change during its lifetime, it can be referenced by a variable of either its own type or of its superclass type.

Therefore, we would expect the next statement to compile:

Student s3 = new FullTimeStudent(333, "Java");

In the statement above we are, in effect, converting between a subclass type and its superclass type when we make the association of a **FullTimeStudent** to a **Student**.

Types higher up a hierarchy are said to be *wider* than the *narrower* types lower down the hierarchy. In the statement

Student s3 = new FullTimeStudent(444, "Java");

the widening conversion, or *upcast*, is carried out automatically by the compiler. The implicit safe cast to a **Student** object is valid at run-time when it can be shown that **s3** refers to a **FullTimeStudent** object.

On the other hand, consider the next statement:

PartTimeStudent pts = new Student(555, "Java");

The compiler issues the following message:

incompatible types: found Student expected PartTimeStudent.

A *narrowing conversion* or *downcast* in this statement will satisfy the compiler, as follows:

PartTimeStudent pts = (PartTimeStudent) new Student(555, “Java”);

If this kind of cast survives a compile-time check, a second check occurs at run-time to determine whether the class of the object being cast is compatible with the object reference. In other words, a downcast may not be a safe cast in the sense that it may not be valid at run-time.

When the statement

PartTimeStudent pts = (PartTimeStudent) new Student(555, “Java”);

is included in a **main** method and **main** is run, a run-time **Exception** occurs:

java.lang.ClassCastException: Student cannot be cast to PartTimeStudent

and shows that this kind of cast *is* invalid at run-time.

In the context of the class hierarchy shown in Figure 3.3, the run-time rules imply that **Student** objects cannot be cast to **PartTimeStudent** or **FullTimeStudent** objects, but that **PartTimeStudent** and **FullTimeStudent** objects can be cast to **Student** objects. In other words, all **PartTimeStudent** and **FulTimeStudent** objects are **Student** objects in that the former inherit from the latter, but all **Student** objects are not necessarily FullTimeStudent or **PartTimeStudent objects**: a **Student** object may be a **Student** object, as in the next statement:

Student s = new Student(999, “Java”);

While the type of an object reference may be obvious at compile-time, the actual class of the object referenced in memory may be less obvious or may not be known until run-time. For example, consider the following statement taken from the **MediaStore** class of the themed application.

Member[ ] members = someStream.readObject( );

The purpose of the input stream **someStream** is to read the array of existing members of the Media Store from a file into the application by calling the stream’s **readObject** method. (We will examine some of the stream classes in Chapter One in *An Introduction to Java Programming 3: Graphical User Interfaces*). The statement does not compile because the API states that the **readObject** method returns an object of the **Object** type. Therefore, a cast is required, as follows:

Member[ ] members = (Member[ ])someStream.readObject( );

On the face of it, (down)casting an **Object** object to an object of the Member[ ] type may not be valid at run-time. In this case, however, the actual object stored in memory *is* of the **Member[ ]** type; therefore, run-time compatibility is preserved.

Finally in this section, it is worth reminding the learner that the **instanceof** operator can be used to find out the type of an object held in memory, so that a cast can be used to restore full functionality to the object. For example, one of the test classes of the object hierarchy shown in Figure 3.3 includes the method shown on the next page.

```
public static void getDetails( Student student ) {
        // find out the type of the object passed when the method is called
        if ( student instanceof PartTimeStudent )
        {
                // note the cast
                PartTimeStudent pt = ( PartTimeStudent )student;
                // call an inherited method
                System.out.println( "This part-time student enrolled in "
                        + pt.getYearOfEnrollment( ) );
        }
        else if ( student instanceof FullTimeStudent )
        {
                FullTimeStudent pt = ( FullTimeStudent )student;
                System.out.println( "This full-time student enrolled in "
                        + pt.getYearOfEnrollment( ) );
        }
        else if ( student instanceof Student )
        {
                System.out.print( "This student enrolled in "
                        + student.getYearOfEnrollment( ) );
        }
}
```

The reader should note how the **instanceof** operator and casting is used, in the method shown above, to find out the type of **Student** object passed to it as an argument so that it can be processed accordingly.

3.6 Virtual Method Invocation

The rules of compatibility discussed in Section 3.5, raise a question when invoking methods: *how do we know which object is being used when invoking a method?*

To illustrate the answer to this question, consider the following code snippet from a test class of the simple application shown in Figure 3.3.

```
Student student = new PartTimeStudent( 1234, "Java" );
System.out.println( student . getDegreeCourseCode( ) );
```

The output is:

```
This part-time student is enrolled on: course code Java
```

The output shows that when you invoke a method via an object reference, it is the *run-time type* of the object referred to which governs which implementation is used. Thus, in the statement above, the object reference **student** refers to a **PartTimeStudent** object and the call to

```
student . getDegreeCourseCode( );
```

invokes the **getDegreeCourseCode** method implemented in the **PartTimeStudent** class definition and not that in the **Student** class definition.

For class methods, on the other hand, the run-time system invokes the method defined in the *compile-time type* of the reference on which the method is called. Thus, a call to the static method **getDetails** as follows

```
student . getDetails( );
```

invokes the **getDetails** method of the **Student** class and not that of **PartTimeStudent**, as shown in Section 3.2 above.

3.7 Controlling Access to the Members of a Class

Access modifiers are used to determine whether other classes have access to a member of a class. Up to this point in the guide, we have met the access modifiers *public* and *private* as they are applied to modify declarations of fields and methods of a class. The access modifier *public* means that all other classes have access to such members of a class and the access modifier *private* means that other members of a class have access to private members of that class.

Extended classes give us an opportunity to explain a further access modifier: that of *protected*. The access levels for the access modifiers *public*, *private* and *protected* are summarised in Table 3.1 shown on the next page.

Modifier	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default: no specifier	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Table 3.1 Access levels

The first row confirms what we know about the access modifier *private*: i.e. the class itself has access to other private members, as we would expect.

The second row indicates that if no modifier is specified, classes in the same *package* have access to such members. We will encounter packages in Chapter Six. Until then, suffice it to say for the present purposes that a package is a convenient way to group together a number of related classes to provide namespace management.

The fourth row shows that *all* classes have access to public members, regardless of their package and parentage.

The third row indicates the level of access provided when a class member is declared to be *protected*. The first column indicates that other members of the class itself have access to the protected member of that class; the second column indicates that classes in the same package, regardless of their parentage, have access to the protected member of the class; the third column indicates that subclasses of the class have access to the protected member, regardless of what package they are in. However, the subclass-protected table entry has an interesting twist that we will defer until Chapter Six.

The example code that follows on the next page illustrates the ‘rules’ encapsulated in Table 3.1, but ignores the second column for the time being.

Example One

```
public class MySuperClass {

        private int privateInt;
        protected int protectedInt;

        public void aMethod( ) {

                System.out.println( privateInt );
                System.out.println( protectedInt );
        }

}
```

The class compiles and merely shows that one of the members of the class – **aMethod** – has access to the private and protected variables of the class.

Example Two

```
public class MySubClass extends MySuperClass {

        public void aMethod( ) {

                // System.out.println( privateInt );
                // the statement above is illegal: privateInt has private access
                // in MySuperClass
                System.out.println( protectedInt );
        }

}
```

The class compiles and shows that the subclass does not have access to the private variable of the superclass but that it *does* have access to the protected variable of the superclass.

Whilst on the face of it, the access level known as *protected* might be regarded as implying a high degree of 'protection' from other classes, the table and example code above shows that this is not the case. The class **MySubClass** shows that all subclasses of **MySuperClass** have access to protected members of **MySuperClass**. Thus we can see that the *protected* access level is not as protected as *private*. Nevertheless, it may be the case that the developer wishes to make a number of members of a superclass protected in order to provide easy access to them from subclasses.

3.8 Summary of Inheritance

Although it is not made explicit at the beginning of this chapter, the examples used aim to illustrate *two* forms of inheritance. Rather than merely mention them both at the outset, it is to be hoped that both forms emerge from the explanations and code examples used in the chapter.

The previous sections aim to show how a class can be extended or subclassed and that a subclass can be used in code designed to work with the superclass. For example, **FullTimeStudent** objects can be used by code designed to work with **Student** objects. If a method expects a parameter of the type **Student**, you can pass it a **FullTimeStudent** object and it will work, although you are likely to have to find out the type of the argument – by using the **instanceof** operator – and restore the full functionality of the object by a suitable cast, as shown in Section 3.5. This feature of object references is known as *polymorphism*. An object that is declared to be of the **Student** type can have many forms in that it can be used as a **Student** object, a **FullTimeStudent** object, or a **PartTimeStudent** object.

The behaviour of a **Student** object is inherited by a **FullTimeStudent** object: the latter is said to *extend* the former. Extended behaviour can be entirely new – by means of adding new methods to the subclass – or it can modify inherited behaviour by overriding a method with the same signature and return type as the overridden method. Thus an extended class can override the behaviour of its superclass by providing new implementations of one or more of the inherited methods.

Overall, extending classes gives rise to two forms of inheritance:

1. *inheritance of type*, where the subclass acquires the type of the superclass so that it can be used polymorphically in code designed to work with the superclass type;
2. *inheritance of implementation*, where the subclass acquires the implementation of the superclass in terms of its members.

Inheritance is a fundamental concept in OOP languages; as a consequence, further examples could have been provided in this chapter to illustrate further the essential features of extending classes. At this stage, however, it is probably wise to conclude this chapter and leave it to the leaner to work with extended classes in practice and encounter concepts in practical situations.

The next chapter explains how errors are handled in Java programmes.