15 INHERITANCE

Terry Marris 14 May 2001

15.1 OBJECTIVES

By the end of this lesson the student should be able to

- explain the concept of inheritance
- implement specialisation/generalisation associations

15.2 PRE-REQUISITES

The students should be comfortable with implementing associations (Chapter 13) and with handling dates (Chapter 14).

15.3 PREVIEW

We have looked at associations (a student may register for up to two modules). We now look at sub-types (a student is a kind of person). We see how to create new classes from existing classes.

15.4 INTRODUCTION

People who pay for software development want

the software to be written in the shortest possible time (because time costs money and competitive edge)

without compromising

- accuracy (the program does what it is supposed to do)
- reliability (and keeps on doing it) and
- robustness (despite all manner of input).

This is achieved by re-using previously written, proven code.

We see how the inheritance mechanism creates new classes from old.

15.5 SPECIALISATION

We start with a familiar example. A person has a name and a date of birth. A student also has a name and a date of birth. But, in addition, a student has a reference number. We show the relation between *Student* and *Person* in a class diagram (Figure 15.1).

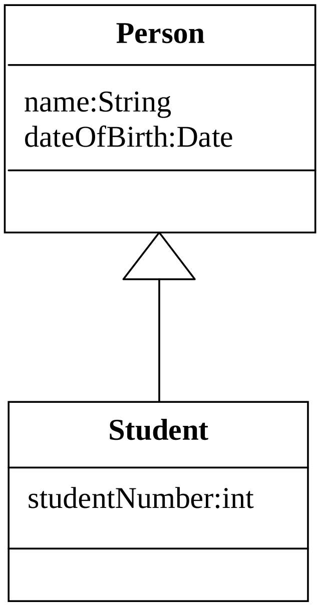


Figure 15.1 *Student is a kind of Person*

Even though a student has a name and date of birth, we do not repeat it in the *Student* class because a student is also a person. A student inherits a person's attributes.

But there are some things about *Student* that does not apply to *Person*. For example, a student has a student number, a person does not.

We say that *Student* extends, refines or specialises *Person*. We say that *Student* is a subclass of *Person*; and that *Person* is the superclass of *Student*. We say that *Person* is the parent of *Student*. We say that a student *is a kind of* person.

A subclass will have at least as many attributes as its superclass.

The △ symbol specifies specialisation/generalisation, or inheritance.

Everything we say about *Person* - attributes, operations and associations - also applies to *Student*.

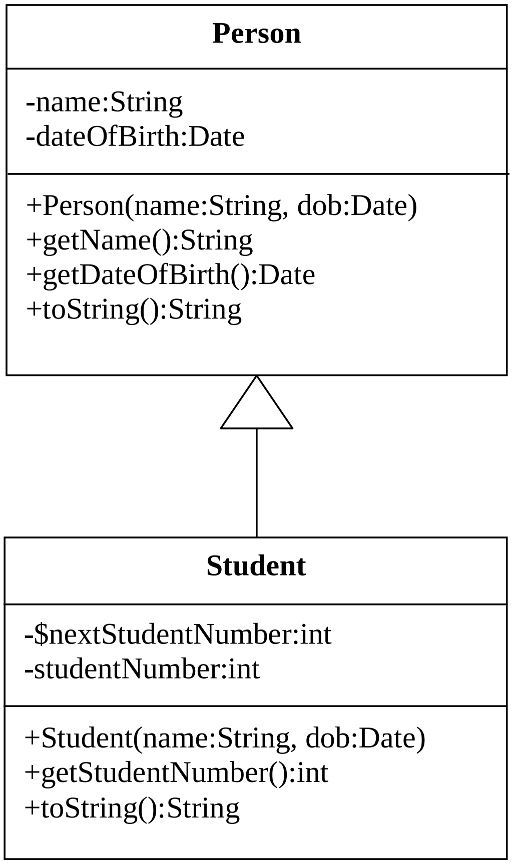


Figure 15.2 *Student inherits Person*

Look at Figure 15.2 above. It shows that

- the *Student* class *inherits* the operations *getName()* and *getDateOfBirth()* from *Person* unchanged. *Student* does not have its own *getName()* and *getDateOfBirth()* operations because they are already there in *Person*.

- *Student*'s *toString()* is different to *Person*'s *toString()*. A person's *toString()* will return a person's name and date of birth. A student's *toString()* will also return the name and date of birth; in addition it will return the student's number. A student's *toString() extends* a person's *toString()* by adding extra functionality.

- *Student* has its own constructor that will initialise *Person*'s name and date of birth, as well as initialising its own attributes.

We extend a class by

- adding extra attributes, or by
- adding extra methods, or by
- re-implementing existing methods.

15.6 INHERITANCE

Inheritance is the mechanism by which specialisation is implemented. We show the *Person* class first.

```
/* Person.java
   Terry Marris  14 May 2001
*/

import java.util.*;
import java.text.*;

public class Person {
  private String name;
  private Calendar dateOfBirth;

  public Person(String aName, int day, int month, int year)
  {
    name = aName;
    dateOfBirth = Calendar.getInstance();
    month--;  // Java counts January as month 0
    dateOfBirth.set(year, month, day);
  }


  public String getName()
  {
    return name;
  }


  public Calendar getDateOfBirth()
  {
    return dateOfBirth;
  }


  public String dateOfBirthString()
  {
    DateFormat df = DateFormat.getDateInstance();
    Date birthDate = dateOfBirth.getTime();
    String dateString = df.format(birthDate);
    return dateString;
  }


  public String toString()
  {
    return "Name: " + name + ", Date of birth: " +
            dateOfBirthString();
  }
  public static void main(String[] s)
  {
    Person p = new Person("james", 18, 12, 1959);
    System.out.println(p);
  }
}
```

Output from program run:

```
Name: james, Date of birth: 18-Dec-59
```

As you can see, there is nothing unusual in the *Person* class. Now we look at the *Student* class.

To show that *Student* inherits from *Person* we use the *extends* keyword in the class header.

```
     public class Student extends Person {
```

The *Student* constructor calls its superclass (i.e. *Person*) constructor with the keyword *super*. *super* always refers to the superclass.

```
     super(name, day, month, year);
```

as in

```
  public Student(String name, int day, int month, int year)
  {
    super(name, day, month, year);
    studentNumber = nextStudentNumber;
    nextStudentNumber++;
  }
```

A call to *super()* must be the first statement in a subclass constructor, so that the superclass instance variables are properly initialised.

The *Student toString()* method extends its superclass (*Person*) *toString()* method by calling *super.toString()* and adding onto to it the student number.

```
  public String toString()
  {
    return super.toString() + ", Number: " + studentNumber;
  }
```

Here is the *Student* class in its entirety.

```
/* Student.java
   Terry Marris  14 May 2001
*/

public class Student extends Person {
  private static int nextStudentNumber = 1001;
  private int studentNumber;

  public Student(String name, int day, int month, int year)
  {
    super(name, day, month, year);
    studentNumber = nextStudentNumber;
    nextStudentNumber++;
  }


  public int getStudentNumber()
  {
    return studentNumber;
  }


  public String toString()
  {
    return super.toString() + ", Number: " + studentNumber;
  }


  public static void main(String[] s)
  {
    Student pat = new Student("pat", 13, 5, 1982);
    System.out.println(pat);

    Student sam = new Student("sam", 31, 12, 1979);
    System.out.println("Name: " + sam.getName() +
                       ", Date of birth: " +
                        sam.dateOfBirthString() +
                       ", Number: " + sam.getStudentNumber());
    Student sue = new Student("sue", 1, 1, 2000);
    System.out.println(sue);
  }
}
```

Program run:

```
Name: pat, Date of birth: 13-May-82, Number: 1001
Name: sam, Date of birth: 31-Dec-79, Number: 1002
```

Notice that

- some methods in the *Person* class are not repeated in the *Student* class: *getName()* and *getDateOfBirth()* for example. These methods are inherited.

- some methods in the *Person* class are repeated in the *Student* class: *toString()* for example. These *Student* methods either re-implement or extend *Person* methods.

Even though *Student* inherits all *Person* instance variables, *Student* methods have no direct access to these. *Student* (subclass) methods access its superclass instance variables only through the public *Person* (superclass) methods. For example, we could have implemented *Student*'s *toString()* like this:

```
public String toString()
{
  // access name field by superclass method
  return "Name: " + getName() +
         "Date of birth: " + dateOfBirthString() +
         "Student #: " + getStudentNumber();
}
```

Lets check out some of the statements in *main()*.

```
     Student pat = new Student("pat", 13, 5, 1982);
```

creates a new *Student* instance. The *Student()* constructor calls its parent's constructor, *Person(),* with the given arguments.

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

sends a *toString()* message to the *pat* object, which itself sends a *toString()* message to its superclass.

```
    System.out.println("... + sam.getName() + ...
```

sends the message *getName()* to the *sam* object. But a *Student* object has no *getName()* method. So the parent class, *Person*, becomes responsible for handling the message.

15.7 GENERALISATION

We approach the concept of generalisation with a simple example. We have two kinds of employee. A salaried employee has an annual salary and 20 days holiday entitlement. A waged employee has an hourly rate of pay and 10 days holiday entitlement. All employees have a number and a name. The class diagram (Figure 17.3) models the association between Salaried, Waged and Employee.

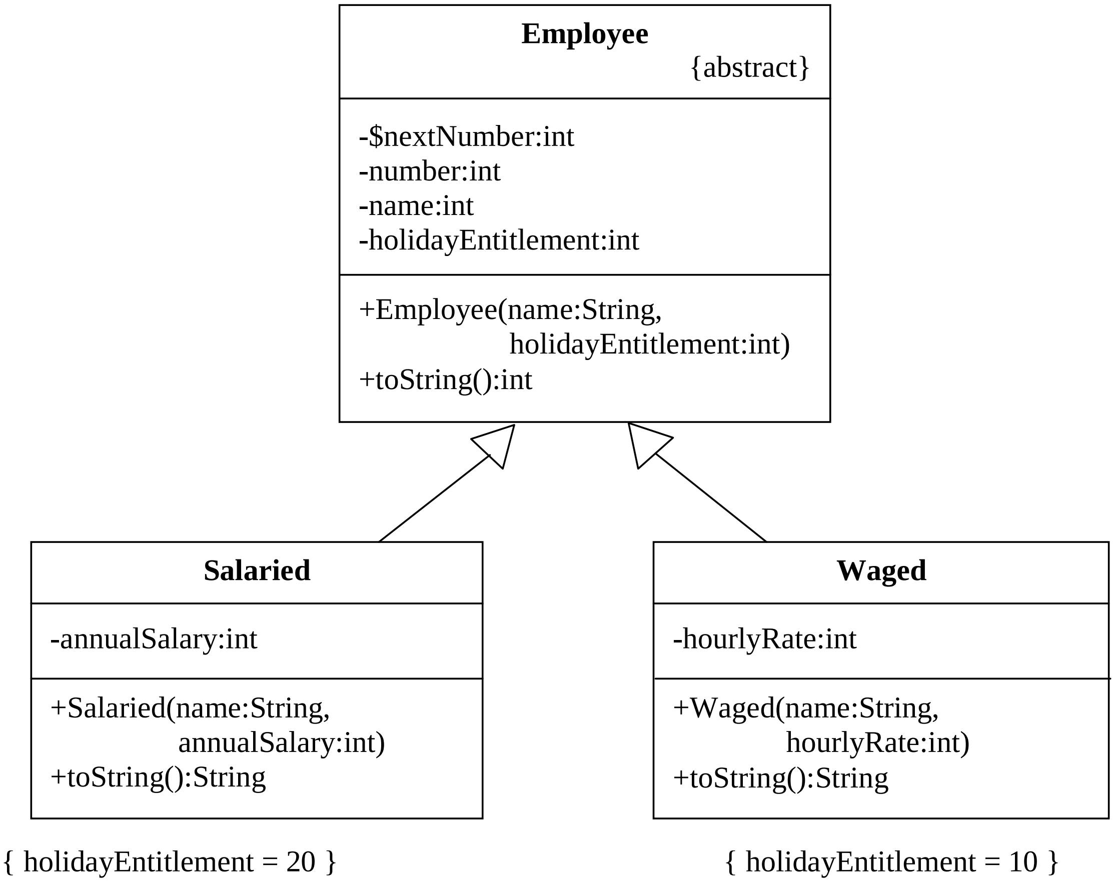


Figure 15.3 *Employee generalises Salaried and Waged*

Both *Salaried* and *Waged* are the same in that they both have *number*, *name* and *holidayEntitlement* attributes.

Salaried and *Waged* are different in that *Salaried* has an *annualSalary* attribute and *Waged* does not; *Waged* has *hourlyRate* attribute and *Salaried* does not.

Notice that we have specified the constraint *{abstract}* in the *Employee* class. This means that you cannot have *Employee* instances. But you can have *Salaried* and *Waged* instances. Which makes sense.

Notice also that for a *Salaried* instance we have constrained *holidayEntitlement* to be 20 and for a *Waged* instance we have constrained the *holidayEntitlement* to be 10.

Generalisation is just the inverse of specialisation - you cannot have one without the other.

A subclass contains those attributes and methods that make it different to its superclass. A superclass contains those attributes and methods that are shared by all its subclasses.

Lets now look at some coding.

We specify that *Employee* is an abstract class.

```
     public abstract class Employee {
```

The rest of the *Employee* class is straightforward.

```
/* Employee.java
   Terry Marris  24 May 2001
*/


public abstract class Employee {
  private static int nextNumber = 1001;
  private String name;
  private int number;
  private int holidayEntitlement;


  public Employee(String aName, int aHolidayEntitlement)
  {
    number = nextNumber;
    nextNumber++;
    name = aName;
    holidayEntitlement = aHolidayEntitlement;
  }


  public String toString()
  {
    return "Number: " + number +
           ", Name: " + name +
           ", Holiday entitlement: " + holidayEntitlement;
  }
}
```

The *Salaried* class extends the *Employee* class in the usual way. Notice how the constraint that a salaried employee's holiday entitlement is 20 days is implemented in the constructor.

```
/* Salaried.java
   Terry Marris  24 May 2001
*/


public class Salaried extends Employee {
  private int annualSalary;


  public Salaried(String name, int salary)
  {
     super(name, 20);
     annualSalary = salary;
  }


  public String toString()
  {
    return super.toString() +
           ", Annual salary: " + annualSalary;
  }
}
```

Similarly, the *Waged* class is straightforward.

```
/* Waged.java
   Terry Marris  24 May 2001
*/


public class Waged extends Employee {
  int hourlyRate;


  public Waged(String name, int anHourlyRate)
  {
    super(name, 10);
    hourlyRate = anHourlyRate;
  }


  public String setHourlyRate(int aRate)
  {
    hourlyRate = aRate;
  }


  public String toString()
  {
    return super.toString() + ", Hourly rate: " + hourlyRate;
  }
}
```

Exercise: Explain each line of the *Waged* class.

A small program to test out the *Employee*, *Salaried* and *Waged* classes is shown below.

```
/* TestEmployee.java
   Terry Marris  24 May 2001
*/


public class TestEmployee {
  public static void main(String[] s)
  {
    Employee bond = new Salaried("james", 25000);
    Employee jones = new Waged("tom", 5);
    System.out.println(bond);
    System.out.println(jones);
  }
}
```

Output:

```
Number: 1001, Name: james, Holiday entitlement: 20,
Annual salary: 25000
Number: 1002, Name: tom, Holiday entitlement: 10,
Hourly rate: 5
```

15.8 CASTING

We introduce a new class, *WorkForce,* which manages a collection of *Employee* objects.

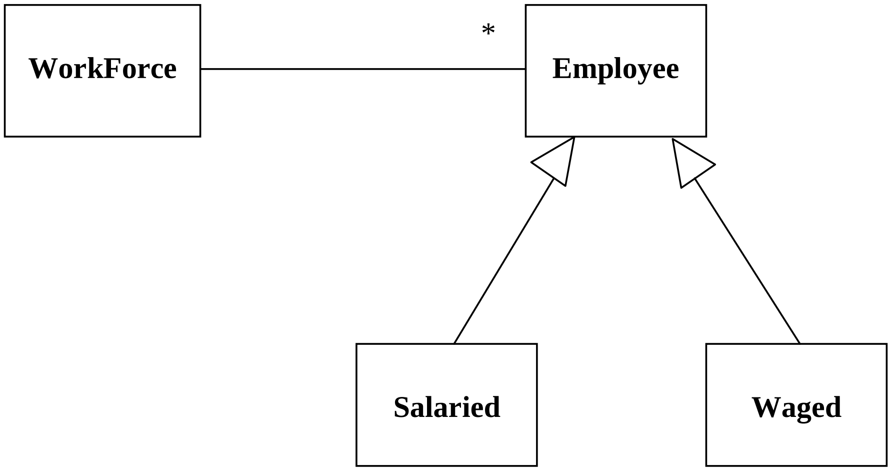


Figure 15.4 *WorkForce has zero, one or many Employees, who may be either Salaried or Waged*

WorkForce contains its *Employee* objects in a *HashSet* and has methods *add(Employee)* and *toString().*

We want to change the hourly rate for each waged employee from five to six. So, we look through the set containing *Employee* objects and, for each object retrieved determine whether it is a *Waged* instance; if so we set its new hourly rate.

```
public void changeHourlyRate(int newRate)
{
  Iterator it = workForce.iterator();
  while (it.hasNext()) {
    Object obj = it.next();
    Employee employee = (Employee)obj;
    if (employee instanceof Waged) {
      Waged wagedEmployee = (Waged)employee;
      wagedEmployee.setHourlyRate(newRate);
    }
  }
}
```

As far as the *HashSet, workForce,* is concerned, it contains objects; it has no idea what kind of objects it holds. So, having retrieved an object from the set we use the *(Employee)* cast operator to convert *Object*s to *Employee*s because, we, as programmers, know that the set cannot contain anything else.

```
     Object obj = it.next();
     Employee employee = (Employee)obj;
```

But only *Waged* employees have an hourly rate. So we first check that an employee is a *Waged* employee with the *instanceof* operator, and, if, so, we use the cast operator *(Waged)* to convert the employee into a *Waged* object.

```
     if (employee instanceof Waged) {
         Waged wagedEmployee = (Waged)employee;
```

The complete class, and a small test program, are shown below.

```
/* WorkForce.java
   Terry Marris  24 May 2001
*/

import java.util.*;


public class WorkForce {
  private HashSet workForce;


  public WorkForce()
  {
    workForce = new HashSet();
  }


  public void add(Employee anEmployee)
  {
    workForce.add(anEmployee);
  }
  public void changeHourlyRate(int newRate)
  {
    Iterator it = workForce.iterator();
    while (it.hasNext()) {
      Object obj = it.next();
      Employee employee = (Employee)obj;
      if (employee instanceof Waged) {
        Waged wagedEmployee = (Waged)employee;
        wagedEmployee.setHourlyRate(newRate);
      }
    }
  }


  public String toString()
  {
    String s = "";
    Iterator it = workForce.iterator();
    while (it.hasNext()) {
      s += it.next() + "\n";
    }
    return s;
  }


  public static void main(String[] s)
  {
    WorkForce workForce = new WorkForce();
    Employee bond = new Salaried("james", 25000);
    Employee jones = new Waged("tom", 5);
    workForce.add(bond);
    workForce.add(jones);
    System.out.println(workForce);
    workForce.changeHourlyRate(6);
    System.out.println(workForce);
  }
}
```

Output:

```
Number: 1002, Name: tom, Holiday entitlement: 10,
Hourly rate: 5
Number: 1001, Name: james, Holiday entitlement: 20,
Annual salary: 25000

Number: 1002, Name: tom, Holiday entitlement: 10,
Hourly rate: 6
Number: 1001, Name: james, Holiday entitlement: 20,
Annual salary: 25000
```

15.9 FURTHER READING

FOWLER & SCOTT *UML Distilled* pp 53, 68, 91
HORSTMANN & CORNELL *Core Java 2 Volume 1* pp 163..224
PARSONS *Object Oriented Programming with C++* pp 123
HOPKINS & HORAN *Smalltalk An Introduction* pp 4,5

15.10 REVIEW

Inheritance

models the is-a-kind-of relationship

common operations and fields are placed in a superclass

a subclass inherits all the fields of its superclass

a subclass accesses its superclass fields via public superclass methods

a class may be extended by providing extra fields or methods or by re-implementing existing methods

a subclass constructor must first call its superclass constructor

the super keyword refers to a superclass

use instanceof before casting from a parent to child class

abstract classes have no instances (objects)

15.11 EXERCISES

1 Explain the meaning of each of the terms

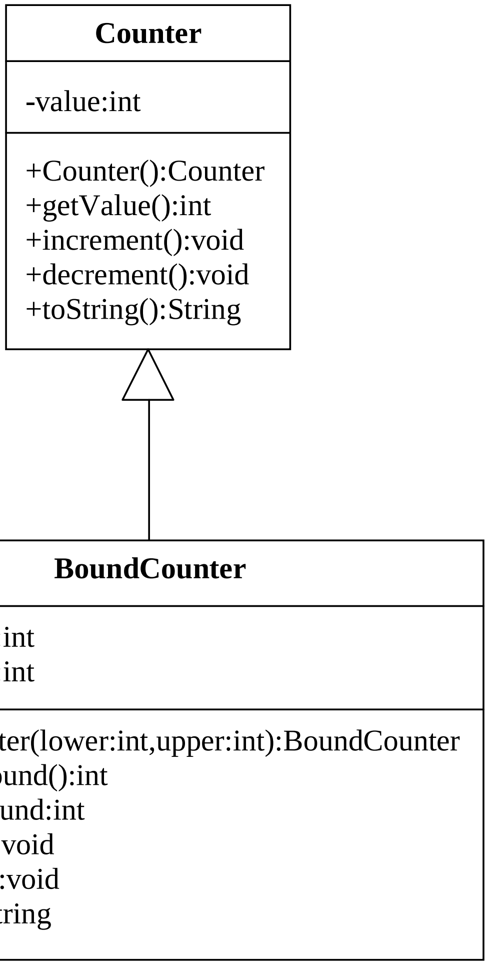
(a) subclass **(b)** inheritance **(c)** extends **(d)** super

2 In what circumstances should inheritance be considered?

3 *Counter* is a class with just one private instance variable named *value*. It has a constructor, which initialises *value* to zero, and three other public methods: *increment()* adds one to *value*, *decrement()* subtracts one from *value* and *toString()*, which returns a string representation of *value*. The code for class *Counter* is given below.

```
public class Counter {
  private int value;

  public Counter()
  {
    value = 0;
  }



  public int getValue()
  {
    return value;
  }


  public void increment()
  {
    value++;
  }


  public void decrement()
  {
    value--;
  }


  public String toString()
  {
    return "" + value;
  }
}
```

BoundCounter is-a-kind-of *Counter*. It differs from *Counter* in that it has two additional instance variables, *upperBound* and *lowerBound* (both set by *BoundCounter*'s constructor) and its *increment()* method adds one to *value* but only if *value* has not reached *upperBound* and its *decrement()* method subtracts one from *value* only if *value* has not reached its *lowerBound.*



(a) Implement and test *BoundCounter*. Suggest a situation where a *BoundCounter* object might be used.

(b) *RollCounter* is a kind of *BoundCounter* but different in its *increment()* and *decrement()* methods. *increment()* adds one to value. If this operation makes *value* exceed the *upperBound,* then *value* is re-set to zero. *decrement()* subtracts one from *value*. If this operation makes *value* less than the *lowerBound,* then it is reset to zero. Implement and test *RollCounter.*