5 Programming with Classes

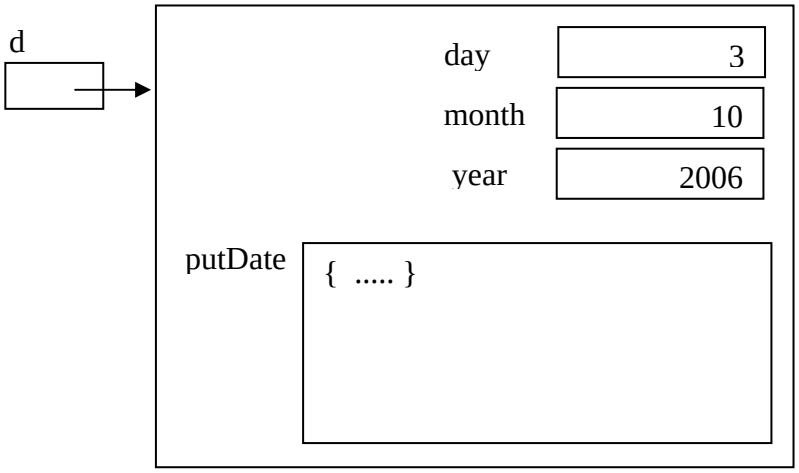
1 Static and instance elements together

It is common for classes to include both instance and static methods. A method should be static if it has no good reason to access instance variables or instance methods in the class. This is illustrated in the following class for dates. The class includes an instance `putDate` method which prints the date in a form typified by `03-10-06` (day followed by month followed by year, each occupying two digits).

```
class Date {

  private int day; private int month; private int year;

  Date(int d, int m, int y) {
       day = d; month = m; year = y;
 }

  private static String dig2(int n) { // rightmost two digits of n
       return("" +  n/10 + n%10);
 }

  void putDate() {
       System.out.println(dig2(day)+"-"+dig2(month)+"-"+dig2(year%100));
  }

}
```

`putDate` uses *static* method `dig2` to compute a 2-character string representation of the rightmost digits of its argument. `dig2` makes no reference to any instance variables in the class and so it would be both misleading and computationally expensive to write it as an instance method, although it would be legal Java.

Remember that static methods exist as soon as the program is loaded – they are not included in objects. For example, an execution of `Date d = new Date(3,10,2006)` creates:



A static method may be invoked by an instance method. When both are defined in the same class, it suffices for to invoke using just the simple name of the static method (as in the example above). If the static method is defined in a different class, then the name of the static method must be prefixed with the name of the class in which it is defined, as in Character.isDigit(c).

Static methods may call instance methods. They must *always* identify the object in which the instance method resides using the usual prefix notation.

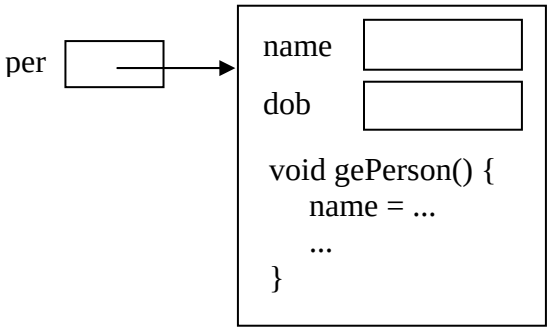
2 Nested objects

Classes may contain instance variables of a class type, and so an object of such a class will contain (references to) other objects within it. The inner objects are said to be "nested" in the outer objects. Consider the following example:

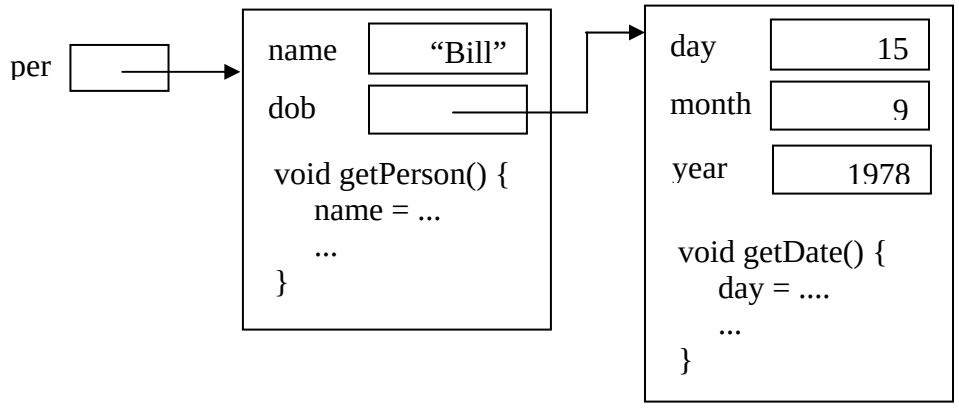
```
class Date {
        private int day, month, year;

        void getDate() {
```

```
            day = Console.readInt(); month = Console.readInt();
            year = Console.readInt();
        }
}

class Person {
        private String name;
        private Date dob; // Nested object!

        void getPerson() {
            name = Console.readToken();
            dob = new Date();  dob.getDate();
        }

}

class MyProg {

        public static void main(String[] args) {
            Person per = new Person();
            per.getPerson();
            ....
        }
}
```

Observe in class Person that `dob` is an instance variable of a class type – type `Date`. The effect of executing `Person per = new Person()` can be visualised as:



Note that as yet `dob` does not reference a `Date` object. The `Date` object is created by the invocation `per.get()` (take a look at the code and notice it is created just before invoking `dob.get()`), after which the situation is (assuming the input consists of `Bill 15 9 1978`):



(Aside: Variables of `String` type contain references to strings rather than a string itself. However, it is a consequence of the fact that strings are immutable (i.e. they cannot be changed once created) that we may envisage string variables containing an actual string. Hence, the representation of "Bill" above.)

The nested object trap

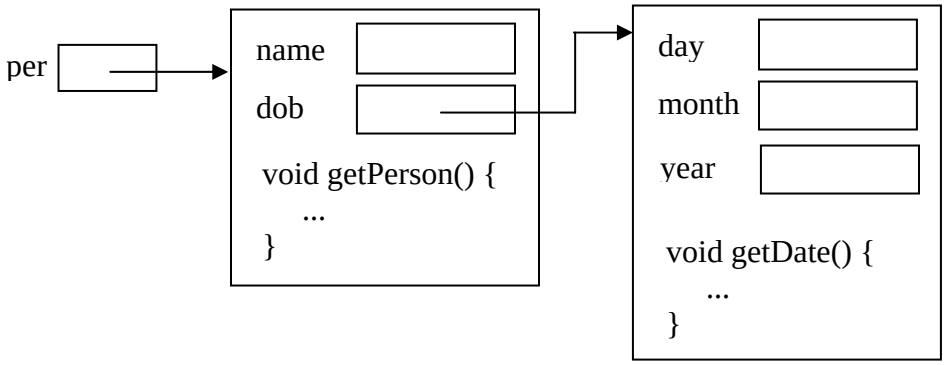
It is a common error to fail to assign an object reference to an instance variable of a class type. For example, many beginners forget to include the statement `dob = new Date()` in method `getPerson()` above. A good practice to avoid this pitfall is to code the creation of an object into the variable's declaration:

```
class Person {
        private String name;
        private Date dob = new Date(); // Note default creation!

        void getPerson() {
                name = Console.readToken();
                dob.getDate();              // No Date creation now
        }

}
```

Now, each time an instance of Person is created, a new instance of `Date` is also created, just as we want. For example, the declaration

```
Person per = new Person();
```

gives rise to:



The failure-to-delegate trap

This potential error is a stylistic one which is extremely important to understand and avoid!

A class should encapsulate *everything* about whatever concept or notion it is intended to represent. Whenever we want to carry out some action pertaining to the concept, we should do so by invoking a method in the class. It takes beginners some time to grasp this. Take a look at method `getPerson` above, which reads a name and a date. Observe in particular that it does not read the date directly, but delegates it to the method `getDate` in class `Date`, as it should. There is no other acceptable way to code `getPerson`.

Some beginners who fail to delegate when appropriate might wrongly code `getPerson` as follows:

```
void getPerson() {  // This is bad!
        name = Console.readToken();
        day = Console.readInt();
        month = Console.readInt();
        year = Console.readInt();
}
```

This is seriously wrong. To begin with, the compiler will legitimately complain that you do not have access to variables `day`, `month`, and `year` because they are instance variables of another class (class `Date`, whereas the code here is in class `Person`). So the programmer may make a

second attempt:

```
void getPerson() {  // This is bad!
        name = Console.readToken();
        dob = new Date();
        dob.day = Console.readInt();
        dob.month = Console.readInt();
        dob.year = Console.readInt();
}
```

This is still seriously wrong! The compiler will complain that you are accessing *private* variables in another class (class `Date`, whereas the code here is in class `Person`), which you may not do.

Heaping error upon error, the programmer makes a third version, this time by deleting the attribute `private` from the `day`, `month`, and `year` variables in `Date`. This time the compiler no longer complains, and the program works in so far as it delivers the correct output. But the program remains seriously wrong stylistically!

The style is bad because it builds details of dates into class `Person`. If subsequently it was decided to handle dates differently, not only would class `Date` have to be amended but also class `Person`. For example, if it was decided to implement the program in a country where dates are specified in the order month-day-year, the original design only requires that class `Date` be amended No change in class `Person` is needed, and that is how it should be.

The correct way to write `getDate` is as we wrote it originally – any other way of writing is either technically or stylistically wrong.

Copying nested objects

When making a copy of an object that contains a nested object, remember that it is usually appropriate to make a copy of nested objects as well. For example, suppose we have a simple `Date` class

```
class Date {
        private int day, month, year;
        .....................

        Date copy() { // copy me
```

```
            Date d = new Date();
            d.day = day; d.month = month; d.year = year;
            return d;
        }
}
```

and class `Person` has an instance variable of type `Date`:

```
class Person {
        String name;
        Date dob = new Date(); // nested object
        ......................

        Person copy() { // copy me
                Person p = new Person();
                p.name = name;
                p.dob = dob.copy(); // Not p.dob = dob!
                return p;
        }
}
```

Observe in method `copy`() in `Person` that the nested `Date` object is also copied. It might seem sufficient to write `p.dob = dob;` in place of `p.dob = dob.copy()`, but that would be to copy references only, leaving the original `Person` object and its duplicate sharing the same date-of-birth object. Usually, that's not what we want. (Aside: In `copy`(), the creation of `Person` object `p` has the side-effect of creating an empty `Date` object (on account of the initialisation in `Date dob = new Date()`) which is immediately orphaned by the assignment `p.dob = dob.copy()`. No harm follows from this other than a minor inefficiency.)

3 Large example: dating

For a more complex example, including use of nested objects, we write a simple dating program. We want to read a list of peoples' details, and match the first person on the list with the most compatible partner in the rest of the list. The details of each person are entered on a single line, and consist of forename, surname, the letter M or F to indicate male or female, respectively, and date of birth (year, month, and day). Each item is separated from the following one by a single space. A typical input looks like

```
Bill Smith M 15 7 1973
Mike Murphy M 21 6 1978
Mary Murphy F 13 9 1975
John Doe M 27 3 1972
Jane Sixpack F 10 6 1974
Sean Ryan M 17 12 1973
Jean Holland F 8 6 1971
Fred Stone M 23 8 1972
```

A partner is compatible if he/she is of the opposite sex and is no more than three years younger or older. Among compatible partners, younger ones are preferred. The output generated for the above input should be

```
Bill Smith aged 27 is partnered with Mary Murphy aged 25
```

For an object-oriented design, we first identify the important concepts in the problem statement. Two present themselves: the notion of dates, and the notion of persons. The appropriate data for a date is day, month, and year, and these will be represented by instance variables. The operations on dates that would seem to be useful are

(i) Read a date.
(ii) Compute the age in years of a date.
(iii) Compute the difference in years of age of two dates.

To compute ages, we need to know today's date. Java provides the means to discover this in a class called GregorianCalendar. To make it available in a program, the program must include the following line at the start:

```
import java.util.*;
```

The use of GregorianCalendar is explained below.

```
class Date {

    private int day, month, year;
```

```
    void get() { // read a date
        day = Console.readInt(); month = Console.readInt();
        year = Console.readInt();
    }

    int age() { // whole years elapsed since date
        GregorianCalendar c = new GregorianCalendar();
        int thisDay = c.get(Calendar.DATE);
        int thisMonth = c.get(Calendar.MONTH)+1;
        int thisYear = c.get(Calendar.YEAR);
        int years = thisYear-year;
        if (thisMonth<month || (thisMonth==month && thisDay<day))
           years--;
        return years;
    }

    int difference(Date d) { // absolute age difference with d
       int n = age()-d.age();
       if (n<0) n = -n;
       return n;
    }
}
```

In class GregorianCalendar, method get() takes an integer argument indicating whether the day of the month, the month, or the year is to be returned. For example, an invocation of get(1) returns the current year. You do not need to remember these integers, because Java supplies another class Calendar in which they are defined as the constants DATE, MONTH, and YEAR, respectively (a "constant" is a static variable whose value is never changed). For example, constant Year in Calendar is defined as 1. Note that GregorianCalendar numbers the months from 0 to 11 -- hence the "+1" in the assignment to thisMonth.

The data we need to represent a person is name, sex, and date of birth, and so we will have instance variables corresponding to each of these. Observe that the date of birth will be a nested object. The appropriate operations on persons would appear to be the following:

(i) Read a person's details.
(ii) Display a person's details in the form "Bill Smith aged 27".

(iii) Determine whether a given person is compatible with this person.

(iv) Determine which of two compatible persons is preferred by this person.

```
class Person {
        private String name;
        private boolean isMale;
        private Date dob = new Date(); // date of birth

        void get() { // read line of person data
                name = Console.readToken();
                name = name + " " + Console.readToken();
                char sex = Console.readChar();
                isMale = sex=='M' || sex=='m';
               dob.get();
        }

        boolean isCompatible(Person p) { // is p of opposite sex & close in age?
                return (isMale!=p.isMale && dob.difference(p.dob)<=3);
        }

        boolean isPreferredTo(Person p) {
        // Am I preferred over p?
           return (dob.age()<p.dob.age());
        }

        void put() {
           System.out.print(name + " aged " + dob.age());
        }
}
```

Class Person displays all the traits of a class with an instance variable of a class type. Study it carefully. Note that a Person object does not include the code to read the person's date of birth, but delegates the job to the relevant Date object. Note also the use of nested selection of components as in p.dob.age(). The main class is:

```
class Dating {
```

```
   public static void main(String[] args) {
      Person subject = new Person(); // subject person
      subject.get();
      Person partner = null; // current best partner for subject
      while (!Console.endOfFile()) {
         Person p = new Person(); // new prospect
         p.get();
         if (p.isCompatible(subject)) {
            if (partner==null || p.isPreferredTo(partner))
                  partner = p;
         }
      }
      if (partner==null) System.out.print("Sorry -- no partners!");
      else {
         subject.put();
         System.out.print(" is partnered with ");
         partner.put();
      }
   }
}
```

In main(), variable *partner* references an object representing the most attractive partner seen so far. It may seem tempting to initialise *partner* using new Person(), but that would be a mistake. Initially there is no partner and so *null* is the proper initial value. Variable *partner* remains null until the first compatible partner is read. At that moment, the new partner's details are encapsulated in an object referenced by p, and following on the assignment partner = p, *partner* will thereafter be non-null. Of course, it is possible that the input contains no compatible partner, in which case partner will remain null forever. That case has to be catered for in the final output phase.

4 Static variables

Variables declared at the top level in a class can be marked as `static` (note that variables declared inside a method are never marked `static`). Static variables have the special properties: that *they are created just once* when the program starts and live as long as the program. Here is a trivial example: Suppose we want to know how many times a particular method p(), say, was invoked during the execution of a program. We could do so as follows:

```
class MyClass {
        private static int pCount = 0; // records number of invocations of p()

        void p() {
                pCount++;
                ..............
        }

        public static void main(String[] args) {
                ..............
                System.out.print("p() was invoked " + pCount + " times");
        }

}
```

All the methods in a class can, instance or static, can refer freely to the static variables of the class. A static variable that is not marked as private can be accessed by methods in other classes, but only by using its full name. The full name of a static variable x defined in class MyClass is MyClass.x (note the period).

Here is another example. Suppose that a class contains a function

```
static double f(double x)
```

such that (i) invocations of f take a long time to execute, and (ii) it is frequently the case that f is called twice in succession with the same argument. Then we can employ static variables to remember the result of the most recent invocation of f, so that we have the opportunity to re-use it if possible.

```
class CC {
        static double x0 = 0; // arbitrary initial value
        static double res0 = f(x0); // always res0 equals f(x0)

        static double f(double x) {
                if (x==x0)  return res0; // no work needed
                else {
```

```
                    ... compute f(x), leaving result in variable y ...
                    x0 = x; res0 = y; // remember details of this invocation
                    return y;
                }
            }
            ........
        }
```

Consider for example, what happens when the following code is executed:

```
if (f(3.14)>0)
        System.out.println(f(3.14));
```

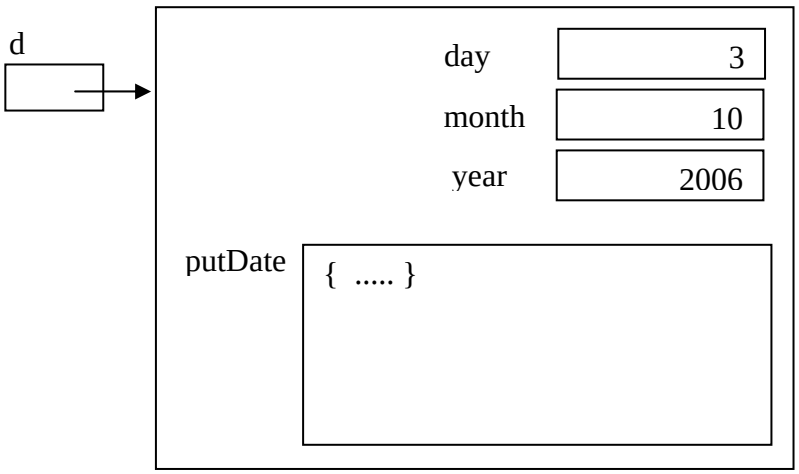
Even though f(3.14) needs to be computed twice, in fact almost no machine effort is required for the second invocation because the result of the first invocation is re-used. Of course this coding trick is only useful when we know that it is likely that f will be called twice in succession with the same argument.

No matter how many instances of a class are created, the static variables in the class exist independently and are not created a fresh for each instantiation of the class. For example, consider the following class:

```
class Date {

  private static int daysInYear = 365;

  private int day; private int month; private int year;

  Date(int d, int m, int y) {
       day = d; month = m; year = y;
  }

  private static String dig2(int n) { // rightmost two digits of n
       return("" +  n/10 + n%10);
  }

  void putDate() {
```

```
        System.out.println(dig2(day)+"-"+dig2(month)+"-"+dig2(year%100));
    }

}
```

Variable `daysInYear` is static and so it exists just once, rather than a version of it being created at each instantiation of `Date`. For example, an execution of `Date d = new Date(3,10,2006)` creates:



– note there is no occurrence of `daysInYear` in the created object.

5 toString() and equals()

toString()

Java allows us to supply an object wherever a string is expected. The run-time system will automatically apply a conversion function which creates a string representing the state of the object. For example, the statement `System.out.print(new Point())` is legal (assuming class Point has been defined).

However, the string supplied by the automatic conversion function is neither elegant nor informative. Java allows us to provide a tailored conversion routine for each class, if we wish, as a public dynamic string-valued function called toString() defined in the class. This is illustrated below:

```
class Point {
    int x = 3; int y = 4;
```

```
   public String toString() {
      return "(" + x + "," + y + ")";
   }
}

class ToStringTest {
        public static void main(String[] args) {
                Point p = new Point();
                System.out.println("My point is " + p);
        }
}
```

When the println() statement is executed, a string representing the object referenced by p is computed by invoking p.toStringt() implicitly. Hence the following message appears:

```
My point is (3,4)
```

The only requirement on the definition of toString() is that it return a string and be explicitly marked public. Note that toString() must include `public` explicitly in its declaration.

equals()

Testing for equality of classes using == is just testing for equality of their references, which is almost certainly *not* what you want. Consider the following program:

```
class Point {
    double x, y;

    Point(double xval, double yval) {
        x = xval; y = yval;
    }
}

class Equality {

    public static void main(String args[] ) {
        Point p = new Point(3,4);
```

```
        Point q = new Point(3,4);
        if (p==q)  System.out.println("They're equal!");
        else System.out.println("They're not equal!");
    }
}
```

This produces the somewhat surprising output `They're not equal`. Java evaluates p==q in the usual way: by comparing the contents of p with the contents of q, and hence it compares *references*, not objects! If you need to test for equality among objects (of a given class), include a boolean-valued method in the class definition. It is usual to call the method "equals".

```
class Point {
        double x, y;

        Point(double xval, double yval) {
                x = xval; y = yval;
        }

        boolean equals(Point p) {
        // Points are equal if their respective components are equal
           return (x == p.x && y == p.y);
        }
}

class Equality2 {

        public static void main(String args[] ) {
                Point p = new Point(3,4);
                Point q = new Point(3,4);
                if (p.equals(q))  System.out.println("They're equal!");
                else System.out.println("They're not equal!");
        }
}
```

When the above program is run, `They're equal!` will be output, as we would wish.
Later we will see that Java provides generic methods that operate on objects of any type. Some of these require that the objects include a boolean-valued method called *equals* that is explicitly

marked as public. In anticipation of that, we will get into the habit of making equals() explicitly public.

6 Testing classes

When you have designed a solution as a collection of classes, it is wise to test each class individually. A good order in which to test the classes is "bottom-up". This means that you first test those classes at the bottom of the hierarchy, i.e. those which rely on no other classes (in simple programs, all the classes will be of this kind). Then test the classes at the next level up, i.e. those classes which rely only on those you have already tested. And so on. For example, for the dating program above, we would test class Date before class Person, because Person makes use of Date. In this way, if you discover an error you can be sure it is the class you are currently testing. Fix any errors as you meet them.

To test a class, run it against some well-chosen test data to confirm the correct behaviour of all the constituent methods. The best way to organise this is to include a main() method in the class which exercises all the methods. Consider as an example, class Name above. A professional programmer would actually include a main() in the class to test the class methods, as follows:

```
class Name {  // The name of a person

   private String forename, surname; // first and second names

   void get() {  .... }

   void put() { .... }

   boolean lt(Name s) { .... }

   public static void main(String args[]) { // test routine
      Name p, q;
      p = new Name();
      System.out.print("Type a name e.g. Bill Smith ");
      p.get(); // tests get(). Key in "Bill Smith"
      p.put(); // tests put()
      q = new Name();
      System.out.print("Type a name e.g. Bill Smith ");
```

```
        q.get(); // for this, key in "Claude van Damme"
        q.put();
        System.out.println(p.lt(q)); // expect to "true" to be displayed
    }
}
```

The class can then be tested by compiling it and running it from the command line (just type `java Name`). The system will run main() from Name, which executes the test routine. When testing is successfully done, there is no need to remove main() -- let it sit there in case you change the class and want to test it again.

7 Changing the world via object parameters

The behaviour of object parameters has a subtlety which we now explain. The point is a little subtle. A parameter is a variable which is local to a method (or constructor), and is special only in that its initial value is set by the invoker each time the method is called. Assignments to a parameter can never affect the associated argument:

```
static void bump(int n) {
        n = n+1;
}
...
int k = 3;
bump(k);
System.out.print(k);
```

What value appears on the screen – 3 or 4? The answer is 3, because n is a local variable in bump(), and an assignment to it cannot have any effect on the value in k. A similar story holds when the parameter is of a class type:

```
class Single {
        int x =  3;
}
.......
static void nullify(Single p) {
        p = null; // has no external effect
}
```

```
...
Single r = new Single(); // r.x = 3
nullify(r);
System.out.print(r.x);
```

Does the print command fail (on account of a null reference), or does 3 appear on the screen? The answer is that 3 appears. However, when a method takes an object parameter, the state of the object can indeed be permanently altered by the method. Consider the following example:

```
class Single {
    int x = 3;
}

class ObjectParam {

    static void bump(Single p) {
            p.x = p.x+1;
    }

   public static void main(String[] args) {
      Single r = new Single(); //r.x = 3
      bump(r);
      System.out.print(r.x);
   }
}
```

Does the print command cause 3 or 4 to appear? The answer this time is 4!. Remember that at the point of invocation of bump(r), the argument (here the reference contained in variable r) is copied into the formal parameter p. Although bump(r) can never change the contents of r, it has access to the object referred to by p, and this of course is the same object as referenced by r. Hence it can change the state of the object referenced by r, and it does so.

Two rules summarise the effect of assignments to parameters. Let ff be a formal parameter in some method, and let variable aa be the corresponding actual parameter.

(i) An assignment to ff has no effect on aa.

(ii) If ff and hence aa are of a class type, then an assignment to a *component* of the object referenced by ff is in fact an assignment to that object referenced by aa.