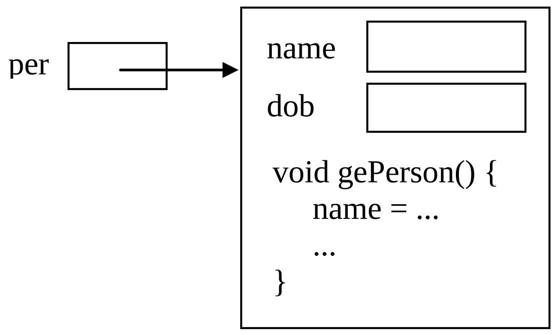
17 Nested Objects

1 Nested objects

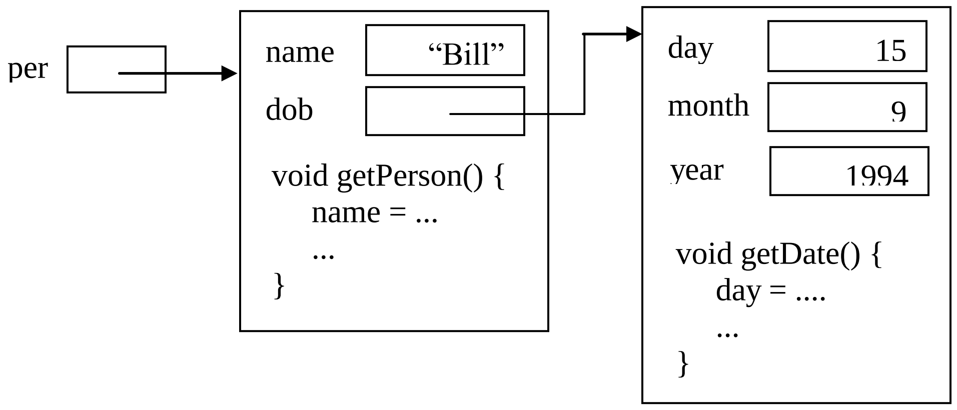
Classes may contain instance variables of a class type, and so an object of such a class will contain (references to) other objects within it. The inner objects are said to be “nested” in the outer objects. Consider the following example:

```
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 1994`):



2 Nested objects traps

Not creating the nested object

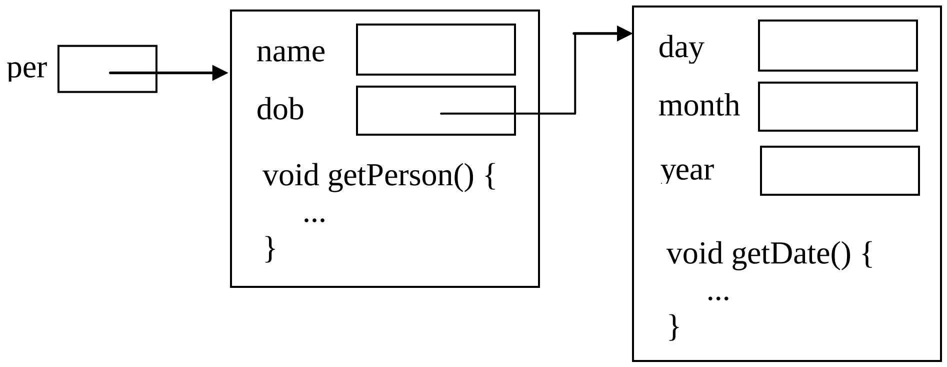
You must always ensure in the case of a nested object, that the object exists before you operate on it. For example, many beginners forget to include the statement `dob = new Date()` in method `getPerson()` above. A useful strategy to avoid this pitfall is to code the creation of an object into the variable's declaration:

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

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

}
```

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:



Very important: The failure-to-delegate trap

This trap is extremely important to understand and avoid! Although falling into it may still result in a working program, the program is likely to be very badly structured.

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 called `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 `getPerson` is as we wrote it originally – any other way of writing is either technically or stylistically wrong.

You will be protected from this trap if you mark instance variables as private, and keep them that way. If the compiler objects that you are accessing private variables illegally, don't fix the problem by removing `private`. The problem is that you are failing to delegate, and that's what you should fix up.

Never deal with compiler errors by removing* `private` *attributes

3 Example: dating

We write a simple dating program to read a list of peoples' details, and find a compatible partner for the first person on the list, the candidates being the persons in the rest of the list. We also count the total number of compatible partners for him/her. A typical input looks like

```
Bill Smith M 15 7 1992
Mike Murphy M 21 6 1994
Mary Murphy F 13 9 1991
John Doe M 27 3 1988
Jane Kinnear F 10 6 1990
Sean Ryan M 17 12 1989
Jean Holland F 8 6 1990
Fred Stone M 23 8 1988
```

Each person's details 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 on a line is separated from the following one by a single space.

A partner is compatible if he/she is of the opposite sex and is no more than three years younger or older. The output generated for the above input might be

```
3 compatible partners for Bill Smith aged 20 including Jean Holland aged 22
```

If there is more than one compatible partner, any one will do. It is guaranteed that there are at least two persons in the input.

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 using library class GregorianCalendar as explained below (access to the class requires importing java.util.*; as shown).

```
import java.util.*;

class Date {

        private int day, month, year;

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

        private int age() { // whole years elapsed since date
                GregorianCalendar c = new GregorianCalendar();
                int thisDay = c.get(GregorianCalendar.DATE);
                int thisMonth = c.get(GregorianCalendar.MONTH)+1;
                int thisYear = c.get(GregorianCalendar.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;
        }
}
```

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.

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

Class Person is typical of a class containing 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. The main class is:

```
class Dating {

    public static void main(String[] args) {
        Person subject = new Person(); // subject person
        subject.get();
        Person partner = null; // the partner
        int count = 0; //  number of compatible partners
        while (!Console.endOfFile()) {
                Person p = new Person(); // candidate
                p.get();
                if (p.isCompatible(subject)) {
                        partner = p; count++;
                }
        }
        System.out.print(count + " compatible partners for ");
        subject.put();
        if (count>0) {
                System.out.print(" including ");
                partner.put();
        }
    }
}
```