# 4 Classes: instance methods

## 1 Instance methods

Methods are either static or dynamic. Dynamic methods are also called *instance* or *non-static* methods. They are easily recognised – they don't have the keyword `static` in the header. Otherwise, they look similar. Dynamic methods can do whatever static methods can, with a little extra flexibility. Their extra flexibility comes at a small cost: dynamic methods are a little bit more computationally expensive than static methods.

Dynamic methods are difficult to grasp at first, and you will probably find you need to re-read this section several times. It is probably best to read briefly through this entire section first, and then make a second pass in more detail.

The additional flexibility of dynamic methods is this: *dynamic methods may refer directly to the instance variables of the class in which they are defined*. To understand what this means, consider the following:

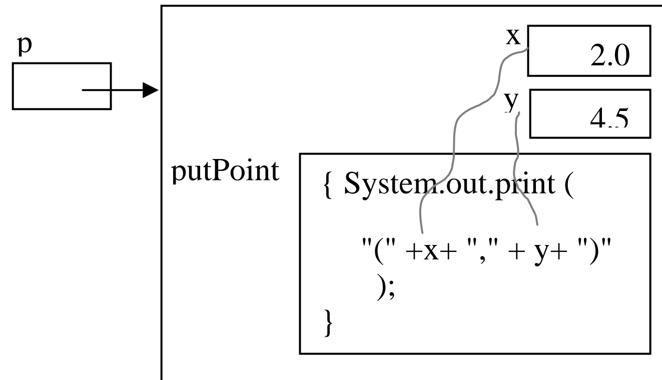
```
class Point {
    double x, y;

    void putPoint() { // display point
            System.out.print("(" + x + "," + y + ")");
    }

}
```

Method `putPoint` is a dynamic method (notice the absence of `static` in its header). Look at the body of `putPoint` and observe that it refers to instance variables simply as `x` and `y` – not

prefixed by any reference as in, say, `p.x` or `p.y`. Only dynamic methods may do this. If we create an object of type `Point`, as follows, say:

```
Point p = new Point(); p.x = 2.0; p.y = 4.5;
```

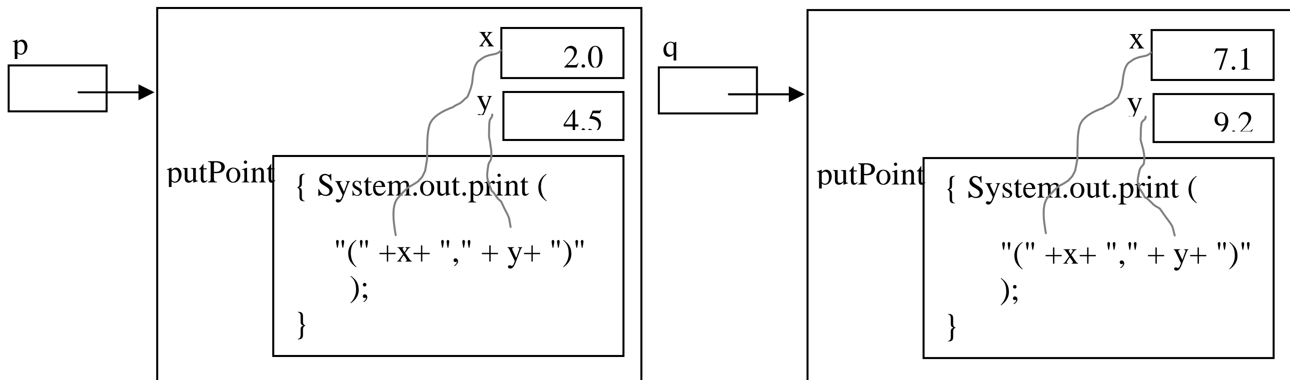
then we get:




Every object we create has a copy of each dynamic method written in the class. So every `Point` object has a version of `putPoint`. To emphasise that we really do get a copy of `putPoint` for each `Point` object we create, suppose we created two `Point` objects:

```
Point p = new Point(); p.x = 2.0; p.y = 4.5;
Point q = new Point(); q.x = 7.1; q.y = 9.2;
```

The situation created is:




Each copy of `putPoint` differs from the others in that each mention of `x` and `y` in the body of `putPoint` is locked onto the `x` and the `y` *in the object where the copy lives*. This is indicated by

the wavy lines in the diagram above. Every dynamic method is treated in the same way: when an object of its enclosing class is created: all mentions of instance variables in the body are locked on to the corresponding instance variables in the same object, and this remains so for the lifetime of the object.

Like instance variables, an instance method only come to life when an object of its enclosing class is created. For example, there will be precisely as many versions of `putPoint` in existence as there are objects of type `Point`. And each version will be slightly different, because each is locked onto the instance variables in its own object.

When we invoke a dynamic method we must indicate which copy we are invoking – by prefixing the method with an object reference. In the example above, we might invoke `p.putPoint()` – which prints (2.0,4.5) – or `q.putPoint()` which prints (7.1,9.2). For example, the following trivial program displays (2.0,4.5):

```
class Point {
	double x, y; // cooordinates

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

	void putPoint() { // display point
		System.out.print("(" + x + "," + y + ")");
	}

}

class PointDemo2 {
	public static void main(String args[] ) {
		Point p = new Point(2.0,4.5);
		p.putPoint();
	}
}
```

Could we have made `putPoint` a static method in the above program, by including `static` in its header? No we couldn't, because `putPoint` uses the freedom to refer to instance variables nakedly (in writing `x` rather than `p.x`, say), and that can only happen in dynamic methods.

It is very important to recognise that `putPoint` has no parameters. The point it displays is not passed as an argument when the method is invoked, but is accessed directly within the object to which this version of `putPoint` belongs. This is a major distinction between dynamic and static methods.

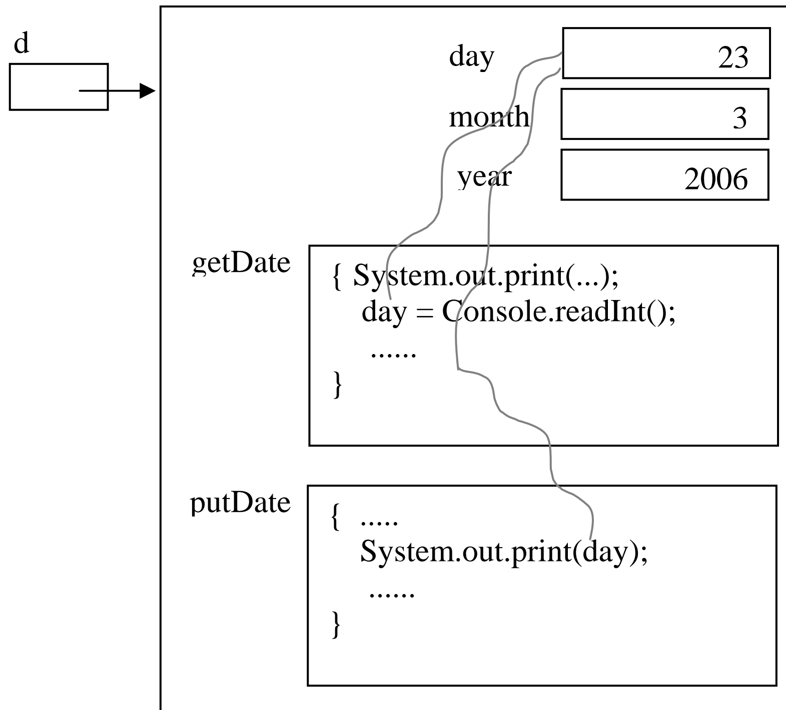
## 2 Object-oriented programming

The following program reads a date typified by 23 3 2006 and prints it in the form 23rd March 2006.

```
class Date {

  int day, month, year;

   void getDate() {
        System.out.print("Enter day month year: ");
        day = Console.readInt();
        month = Console.readInt();
        year = Console.readInt();
   }

   void putDate() {  // in form 23rd March 2006
        String[] ord = {"th", "st", "nd", "rd", "th" , "th" , "th" , "th" , "th" , "th"};
        String [] months = {"January", "February", ..., "December"};
        System.out.print(day);
        System.out.print(ord[day%10]);
        System.out.print("  " + months[month-1]);
        System.out.print("  " + year);
   }

}

class PrintDate {
   public static void main(String[] args) {
        Date d = new Date();
```

```
            d.getDate();
            d.putDate();
        }
    }
```

Both methods in class `Date` are dynamic – observe that they refer to `day`, `month`, and `year` nakedly. After `d.getDate()` has completed, and supposing the user enters 23 3 2006, we have




Observe that when we invoke a dynamic method such as `getDate` to initialise an object with values entered at the keyboard, we have to create the object before invocation – otherwise the necessary dynamic method doesn't exist. Observe also `getDate` and `putDate` have no parameters. In this case all the variables they need reside in the object to which they belong; this is often the case with dynamic methods, It might be tempting to write `getDate` so that it returns a `Date` object, but this is not done because `getDate` assigns to the coordinate variables in its own object.

We previously solved the preceding problem using static methods, so which version is best? Both versions are perfectly correct programs, but they represent different styles of programming. The style that favours the use of dynamic methods is called *object-oriented programming* and is almost universally used by professional Java programmers. The style that favours static methods is used in some other languages, such as C. We will adopt the object-oriented style. We will see that even

in the object-oriented style, there are plenty of situations in which static methods are employed,

## 3 Instance methods with parameters

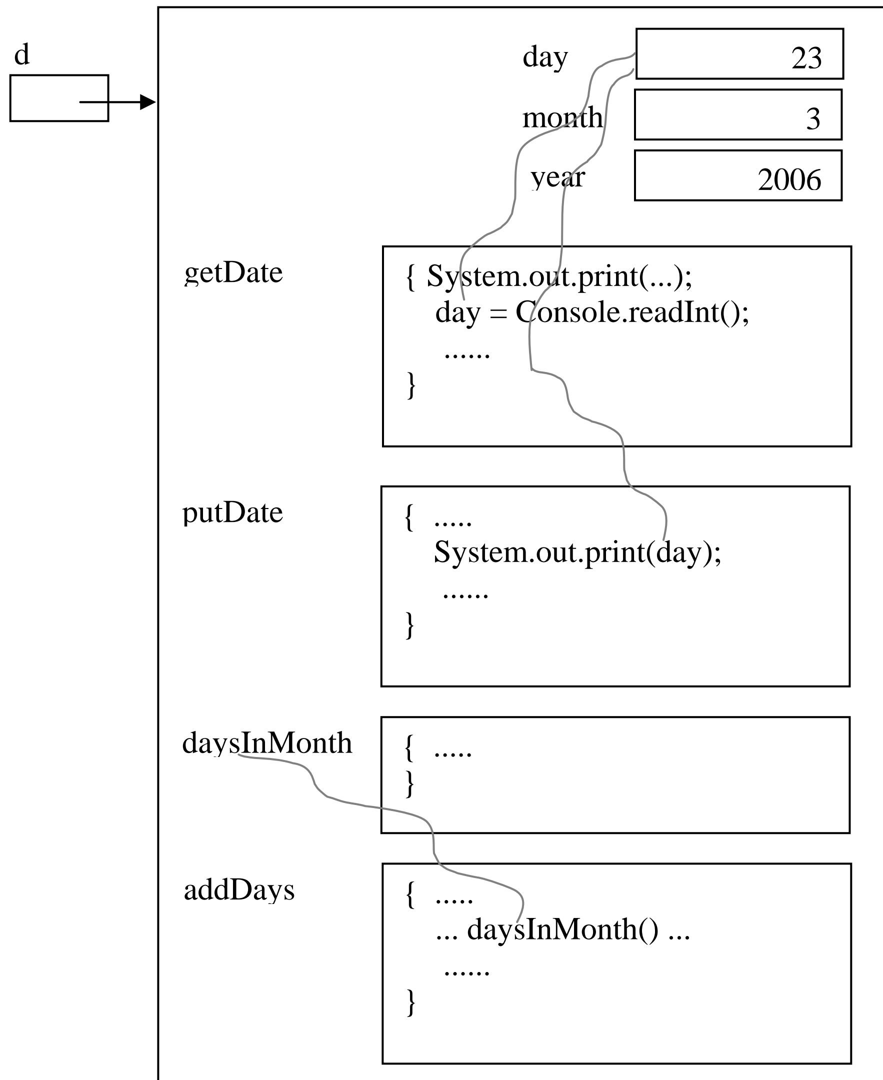
Instance methods may have parameters and may return values, just like static methods. We illustrate with an extension of the preceding example. The program reads a date, followed by a number of days, and displays the date after that number of days have elapsed. The following is a typical input/output dialogue:

```
Enter day month year: 28 5 2006
Enter duration: 7
4th  June  2006
```

```
class Date {

  int day, month, year;

   void getDate() {
        System.out.print("Enter day month year: ");
        day = Console.readInt();
        month = Console.readInt();
        year = Console.readInt();
   }

   void putDate() {  // in form 23rd March 2006
        String[] ord = {"th", "st", "nd", "rd", "th" , "th" , "th" , "th" , "th" , "th"};
        String[] months = {"January", "February","March", "April", "May", "June",
                "July", "August", "September", "October", "November", "December"};
        System.out.print(day);
        System.out.print(ord[day%10]);
        System.out.print("  " + months[month-1]);
        System.out.print("  " + year);
   }

   int daysInMonth() { // Number of days in month
        if (month==9 || month==4 || month==6 || month==11)
                return(30);
        else if (month==2) {
                if (year%4==0 && year!=1900) return(29);
```

```
            else return(28);
        }
        else return(31);
    }

    void addDays(int n) { // add n days to date, n>=0
        int d = n;
        while (d>0) {
            if (day+d<=daysInMonth()) {
                day = day+d; d = 0;
            }
            else {
                d = d - (daysInMonth()-day+1);
                day = 1;
                if (month<12) month++;
                else {
                    month = 1; year++;
                }
            }
        }
    }
}

class Duration {
    public static void main(String[] args) {
        Date d = new Date();
        d.getDate();
        System.out.print("Enter duration: ");
        int period = Console.readInt();
        d.addDays(period);
        d.putDate();
    }
}
```

The preceding example illustrates that instance methods may invoke instance methods in the same class nakedly, i.e. without explicitly identifying which version of the instance method is being invoked – notice the invocation of daysInMonth within addDays. When an object is created, all invocations of instance methods within the object are locked into calls within the same object, just like references to instance variables. For example, Date d = new Date(23.3.2006) gives rise to:




### The instance method trap

Remember that dynamic methods belong to objects, and so if no objects of the class have been created, then there are no copies of the method. You cannot use a dynamic method without having first created at least one instance of the class where it is defined. Hence the following code contains errors:

```
class MyClass {
        int x;
        void  inc() { x = x+1;}
        ...;
}
```

```
class MyClassTest {
        public static void main(String[] args) {
                MyClass t = new MyClass();
                ....
                inc();                  // Error – must prefix inc() with an object
                MyClass.inc();          // Error –inc() prefixed with class name
                t.inc();                // ok
                .....
        }
}
```

## 4 Instance methods returning objects

An instance method in class `MyClass` may return a reference to an object, even an object of type `MyClass`. Suppose that in the duration example of the preceding section, we wanted to improve the output so that it looks like

```
Enter day month year: 28 5 2006
Enter duration: 7
28th May 2006 plus 7 = 4th  June  2006
```

We might mistakenly rewrite `main` as follows:

```
public static void main(String[] args) {
     Date d = new Date();
     d.getDate();
     System.out.print("Enter duration: ");
     int period = Console.readInt();
     Date first = d; // remember old date
     d.addDays(period);
     first.putDate(); System.out.print(" plus " + period + " = ");
     d.putDate();
}
```

This will won’t behave as expected. We actually see;

```
Enter day month year: 28 5 2006
Enter duration: 7
4th  June  2006 plus 7 = 4th  June  2006
```

The problem is that the assignment `first=d;` introduces aliasing, and so any change to `d`'s date is also a change to `first`'s. To resolve the problem, we re-design `addDays` so that it leaves the current date object unchanged, and instead returns a fresh date object containing the later date. The changed parts are emphasised below:

```
Date addDays(int n) { // add n days to date, n>=0
    Date f = new Date();
    f.day = day; f.month = month; f.year = year;
    int d = n;
    while (d>0) {
        if (f.day+d<=f.daysInMonth()) {
            f.day = f.day+d; d = 0;
        }
        else {
            d = d - (f.daysInMonth()-f.day+1);
            f.day = 1;
            if (f.month<12) f.month++;
            else {
                f.month = 1; f.year++;
            }
        }
    };
    return f;
}
```

Observe that now `addDays` must necessarily refer to the instance variables of two `Date` objects: the exiting one and the new one. The variables in the current date are as always referred to nakedly (e.g. `day`), while the variables in the copied object must be prefixed as in `f.day` etc. The enhanced version of `main` can now be written correctly:

```
class Duration {
  public static void main(String[] args) {
        Date first = new Date();
```

```
                first.getDate();
                System.out.print("Enter duration: ");
                int period = Console.readInt();
                Date second = first.addDays(period);
                first.putDate(); System.out.print(" plus " + period + " = ");
                second.putDate();
    }
}
```

## 5 Class parameters

If `MyClass`, say, is the name of some class, `MyClass` may be used as a type anywhere in the program, even within `MyClass` itself. It is especially common for dynamic methods in a class `MyClass` to have parameters of type `MyClass`. For example:

```
class Point {
        double x, y;   // cooordinates

        double distance(Point p) { // distance of this point from point p
                return(Math.sqrt((x-p.x)*(x-p.x)+ (y-p.y)*(y-p.y)));
        }

}
```

If `pt1` and `pt2` are each variables of type `Point`, then the distance between them is obtained by invoking either `pt1.distance(pt2)` or `pt2.distance(pt1)`. Observe that `distance` takes a parameter of type `Point`. In the body of `distance`, `x` unadorned with a prefix denotes, as usual, that `x` in the same object as the current incarnation of `distance`, and `p.x` denotes that `x` in the object referenced by parameter `p`.

### The redundant parameter trap

Remember that a dynamic method belongs to an object, and has free access to the object's instance variables. It is very important to grasp the significance of this: *every dynamic method has free access to the instance variables of the object to which it belongs*. Many beginners forget this and try to pass a redundant object as a parameter. For example, they may write method `distance2` as follows:

```
class Point {
	double x, y; // cooordinates

	double distance2(Point p, Point q) { // distance between two points, but BAD!
		return(Math.sqrt((p.x-q.x)*( p.x-q.x)+ (p.y-q.y)*( p.y-q.y)));
	}

}
```

Method `distance2` above determines the distance between two points, but it is poor object-oriented style in that both points are passed as parameters. `distance2` is a dynamic method and so has free access to a point, i.e. to the `x` and `y` variables in the object in which it resides. Therefore it only needs access to one other point, and that can be passed as a single parameter. Method `distance` above exhibits the correct object-oriented style of coding.

In general, dynamic methods tend to have one less parameter than you might think at first, because every invocation of a dynamic method has free access to everything in the object to which it belongs.

Comments in dynamic methods often refer to the object to which it belongs as "me" or "I" or "this point", "this employee", "this date" etc., or "the point", "the employee", "the date" etc. For example, the behaviour of distance2(Point p) may be described by any of the following comments

- // distance from point P
- // my distance from point P
- // how far am I from point P
- // distance of this point from point P
- // distance of the point from point P

### Example: least name

Let us write a program to read a list of persons' names, one name per line of input. Each name consists of a forename followed by a space followed by a surname. The program should display the name which comes first in the usual phone-book ordering of names, output as surname followed by forename separated by a comma. For example, the input

```
Charles Darwin
Marie Curie
Neils Bohr
Albert Einstein
```

should give rise to the output

```
Bohr, Neils
```

The program is

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

   String forename, surname; // first and second names

   void get() {  // Read name from keyboard
      forename = Console.readToken(); // forename is one word
      surname = Console.readString(); // surname is rest of line
   }

   void put() { // Write name in form "Smith, Fred"
      System.out.println(surname + ", " + forename);
   }

   boolean lt(Name s) { // My name precedes s.name?
      return(surname.compareTo(s.surname)<0 ||
         (surname.equals(s.surname) && forename.compareTo(s.forename)<0));
   }

}

class LeastName {

   public static void main(String[] args) {
      Name least = new Name();
      least.get(); // least name so far
      while(! Console.endOfFile()) {
```

```
      Name nn = new Name();
      nn.get();
      if (nn.lt(least))
        least = nn;
    }
    least.put();
  }
}
```

In the program, we only ever retain access to two at most objects, one accessed via `least` and the other accessed via `nn`, and yet we create an object each time a name is read. Hence the program creates lots of orphans, but this is not a problem. You may be tempted to reduce the computational effort by avoiding the creation of a name object for each name read. If so, you may try to effect this placing the declaration `Name nn = new Name();` alongside the declaration of `least` (removing it from the body of the loop). Can you see why this will fail (hint: the aliasing trap!)?

When a program is comprised of several classes, the name of the file should be the same as the class which contains the main() you want to be run when the program is executed. The above program, for example, should be placed in a file called `LeastName.java`.

## 6 Access control

Each method, constructor, and variable within a class, whether static or dynamic, may be marked as either `public` or `private`. If neither is stated, then the default is public (but Java requires an explicit writing of `public` in the declaration of main(), and in some other situations which we will explain when we meet them). These keywords do not bring any new power, in the sense that they do not enable us to write programs that we couldn't write without them. They role is is a stylistic one.

The basic rule is that private entities cannot be accessed outside the class in which they are defined:

```
class Pair {
  int x;            // x is public by default
  private int y;    //  y is private
```

```
    private void meth1(Pair p) {
        .....
    }
    .....
}

class UsePair {
    public static void main(String args[] ) {
        Pair r = new Pair (); Pair q = new Pair();
        r.x = 1;          // legal as x public
        r.y = 2;          //illegal as y private
        r.meth1(p);       // illegal as meth1 private
    }
}
```

Code may refer to private entities defined in the same class:

```
class Pair {
    int x;                // x is public by default
    private int y;        //  y is private

    private void meth1(Pair p) {
        x = p.x;          // legal
        y = p.y;          // legal
    }

    void meth2(Pair p) {
        meth1(p);         // legal
    }
}
```

It is good programming practice to make all the instance variables in a class private. Methods should be marked private if they are written solely to help us to write another method in the same class. In other words, they were not part of the formal requirements for the class. For example, in a problem tackled above we made a class `Date`; it is reproduced below with its variables and methods marked as private where appropriate:

```
class Date {

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

    void getDate() { .....
    }

    void putDate() { .....
    }

    private int daysInMonth() { .....
     }

    void addDays(int n) { .....
   }
}
```

The instance variables `day`, `month`, and `year` are marked private as is usually the case. In addition, method `daysInMonth` is marked private because it was introduced only during the coding process to help us write `addDays`.

It is not easy for beginners to see the value of `public` and `private`. Indeed, if we replace all occurrences of `private` in a program with `public`, then the program will compute the same result. However, if you earn your living updating other peoples programs, then you will pray that they have employed private variables and methods as much as possible. Professional programmers always privatise when possible. Regular changes in large working programs are the norm, and it is easier to change programs where the internal details of classes are kept hidden (by making them private). If hidden internal details of a class are changed, it will have no impact on code outside the class. If visible details are changed, however, it is likely that other parts of the program will have to be changed also, and that can be time-consuming and difficult in large programs.

### The private/public trap

Only variables declared at the outermost level in a class may be qualified as private or public. Local variables (those declared inside methods) are never so marked:

```
    class Duration {
```

```
    public static void main(String[] args) {
        private Date d = new Date();     // Wrong!
        .....
    }
}
```

## 7 Getters and setters

When instance variables are private, code outside the class cannot access them directly either to inspect them or to change them. We may include for some or all instance variables a function which simply returns its value; such functions are called “getters”. We may also include functions which assign a new value to an instance variable; these functions are called “setters”. In the following example, `getYear` is a getter and `setYear` is a setter.

```
class Date {

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

   int getYear() { return day;}            // getter

   void setYear(int y) { year = y;}        // setter

   .....
}
```

You should hesitate a moment before you write a getter or setter, and you should use them sparingly and only for very simple purposes (such as displaying the value of an instance variable on a screen).

### The getter/setter trap

The use of setters and getters sometimes reveals a poor programming style, where the programmer is using them to avoid a correct object-oriented design. They should never be used to code an action outside a class that should properly be written as a method within it. For example, suppose we are writing a program about dates, for which we are using the `Date` class as described above. Suppose in the main program we want to determine whether some date `p` is a leap year in the 21st century; we might be tempted to write:

```
    int y = p.getYear();
```

```
if ((2000<=y && y<2100 && y%4==0) ....
```

While this code is correct it is not good style. The object-oriented style insists that we code all the operations on dates within the `Date` class. So we should add the following method to the `Date` class:

```
boolean isLeap21() {
    return (2000<=year && year<2100 && year%4==0);
}
```

and change the code in the main program to

```
if (p.isLeap21()) ....
```