The Development of an Application in Java

Guy Lapalme
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, Québec
Canada H3C 3J7
e-mail:lapalme@iro.umontreal.ca

June 11, 2014

Abstract

This paper describes the development of a small application in Java. It illustrates the concepts of message passing, inheritance, static variables and methods, graphic animation and an approach to achieve a mouse and menu driven application.

1 Introduction

This paper[1] shows the development of a small application in Java that has the advantage of being easily understood and visualized, it is shown in figure 1. The best way to learn Java is by experimenting with the system; it is very hard to learn "by the books". Ideally this article should be read after having loaded the corresponding classes, available from the author, within an Integrated Development Environment (IDE) or within a text editor to inspect the code of the methods and experimenting sending messages to objects. But we try anyway to give a feeling of the development of an application for those who dot not have access to a Java environment. This application was developed in Java 7 using Eclipse on a Macintosh with OS X 10.9 and, but it was also tested on a PC and on Linux using the JDK 7 by Oracle. This shows that Java graphical applications are quite portable across platforms.

Java is a very rich programming environment that defines a whole gamut of classes for doing graphics and applications on the World-Wide Web. It is an object oriented language where most parts of the system are viewed as objects responding to messages. But, unlike Smalltalk[4], some entities are primitive (e.g. integers, floating point numbers and characters) and classes are not themselves objects; although Java defines some classes for defining these primitives in term of objects, in the usual applications, there are not considered as such.

We briefly review Java using the objects defined for our application. There are many books on Java, one that is both introductory and comprehensive is Horstmann [5] but the definitive reference is by Java Team [3]. For the principles behind interface building using Swing, standard since Java 1.2, one should consult the excellent book (in French) by Berthié and Briaud [1].

[1] this is a revision (June 2014) of the 2005 revision of a 1999 paper that was itself another *instantiation* of the same application that was originally developed in Smalltalk described in [6]

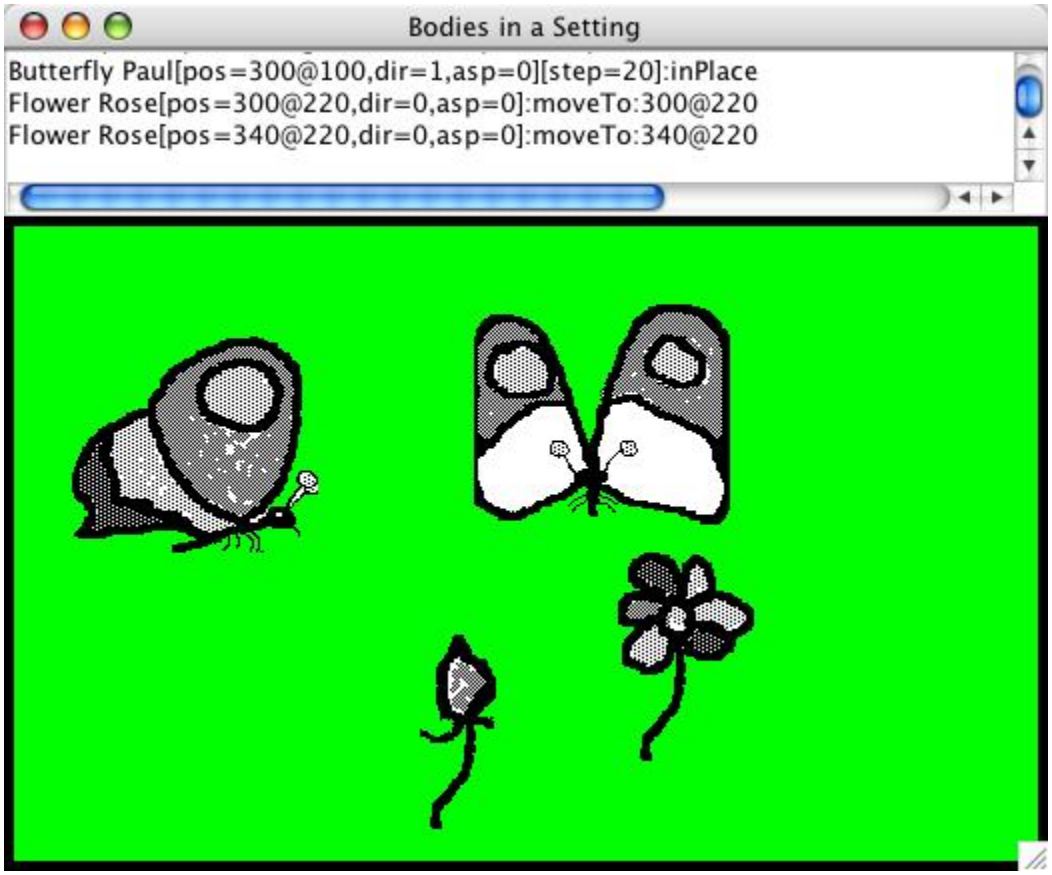

Figure 1: Snapshot of our application

This application describes the movements of graphical bodies that are displayed within a setting. Each body can *move* and display a sequence of pictures to simulate its movements. This graphical application is controlled with mouse clicks and questions that asks some information (such as the name to be given to an object) from the user. Following the Swing approach, actions of the user are handled by *event listeners* that modify the state of the application. In our case, this is the list of all animated bodies that have been created. After each change to the state of an animal, the display is updated. We will come back to the organization of the application later, but we will first give a very short introduction to object oriented programming in Java.

2 An introduction to object oriented concepts in Java

In an object oriented system, entities is described in terms of objects having an internal state and responding to messages. Objects are created from a mold (called a *class*). In our case, we have *animated bodies* moving within a *setting*. As most of the *animated bodies* will have share some behaviors, we define an abstract class `AnimatedBody` which we extend with concrete subclasses (`Butterfly` and `Flower`). From the concrete classes, we create objects, called instances that evolve within an instance of another class `Setting` which, not surprisingly, does not behave like an

animated body.

We use the following convention for identifiers: class names start with a capital letter while other identifiers are not capitalized. When an identifier is made up of more than one word, the words are not separated and each one (except possibly the first) is capitalized, for example, `theButterflyOverAFlower`.

We first create a `park`, an instance of `Setting`, in which instances of the classes `Butterfly` and `Flower` move. The `Setting` will eventually be represented on the screen by a window that delimits the visible area of activity of the animated bodies: if a body moves out of the `park`, it is not displayed. An `AnimatedBody` is represented by a picture on the screen. Given that we have already created a `Flower` named `daisy` and a `Butterfly` named `peter`, Table 2 shows messages that can be sent to these objects.

Object	message	action
`peter`	`turn()`	change its direction
`daisy`	`open()`	show the flower as open
`peter`	`moveTo(pos)`	change its position to `pos`

Table 1: Messages to instances of classes

In Java, messages are sent by means of procedure calls defined within objects that are called methods which can have parameters. The name of the called object is first given then a dot (`.`) and then the name of called method followed by its parameters in parentheses. Even if a method has no parameter, we put parentheses after the name of the called method. For example, `peter.turn()` or `daisy.setDirection(1)`.

Java provides standard control statements such as `if`, `for`, `while` statements whose syntax is closely modeled after the syntax of the C language. We will not elaborate further on these usual aspects of the language because we want to focus on the object oriented aspects of the language and on our application.

3 The `MyPoint` class

As a simple example of defining a new class, we define `MyPoint` an alternative to the standard `Point` class[2] to keep track and specify positions of our bodies in our setting. The `MyPoint` class that represents a location on a two-dimensional (x, y) coordinate space with two internal integer variables, called instance variables, `x` and `y`. The $(100, 200)$ position can thus be designated by creating a new point such as

```
MyPoint pos = new MyPoint(100,200);
```

As `x` and `y` represent the internal state that should not be changed from outside the class, we declare them as `private`. To get their values from the outside, we will define two public *accessor* functions `getX()` and `getY()`. Contrary to the standard `Point` class, a `MyPoint` instance cannot change so **we will not** define any *mutator* function such as `setX(newX)` or `setY(newY)`. But we

[2]we could have chosen to create a subclass of `java.awt.Point` but as we want to hide the internal state, we have decided to create an independent class with a different name even though no ambiguity would arise if we use the java package mechanism

will define arithmetic operations for `MyPoint` instances such as adding or subtracting points and finding the distance between two points.

We first declare a new class with two private `x` and `y` instance variables. To create an instance of `MyPoint`, we define a constructor to initialize the instance variables.

```
class MyPoint {
    private int x,y;
    MyPoint(int x, int y){
        this.x = x;
        this.y = y;
    }
    public int getX(){return x;}
    public int getY(){return y;}
}
```

`this.x` refers to the `x` private variable of the class and is necessary because `x` refers to the parameter of the `MyPoint` constructor which hides the `x` instance variable within the constructor. To avoid the name clash, we could have given a different name to the parameter, but this way of making explicit the initialization of instance variables with parameters of the constructor is conventional in Java code.

Now we define some new features[3] to manipulate `MyPoint` instances. We want to add, subtract and scale points but in order to combine them easily, the result should not change the internal values of the point (as do the `move` or `translate` methods of the standard Java `Point` class) but instead it creates a new instance of `MyPoint` with the result.

```
MyPoint add(MyPoint p){
   return new MyPoint(x+p.x,y+p.y);
}
MyPoint sub(MyPoint p){
   return new MyPoint(x-p.x,y-p.y);
}
MyPoint scale(double f){
   return new MyPoint((int)Math.round(f*x),(int)Math.round(f*y));
}
```

`add` and `sub` add and subtract the `x` and `y` coordinates. `scale` multiplies each coordinate by a floating point number but then rounds the result to an `int` value. So given two instances of `MyPoint`, `p1` and `p2` we can combine them as `p1.add(p2.scale(5))` which corresponds to $p1 + (p2 * 5)$. To find the distance, returned as a floating point number, between two points we define

```
double dist(MyPoint p){
   return Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
}
```

which can be called as `p1.dist(p2)`.

For any object, Java defines a standard string representation that can be used for identifying it in an output. By default, it just returns the name of the class followed by an internal hashcode

[3]These features follow closely the operations of the `Point` class of Smalltalk

(e.g. `MyPoint@687e7c`). This is not very useful, so one should define a more practical representation that returns the values of the instance variables. When Java needs to transform an object to string, it calls the `toString()` method of that object, so by defining this method in a new class, we get a printable version of our object such as `"100@200"` for an instance created by `new MyPoint(100,200)`.

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

It is also useful to go the other way and be able to create a new `MyPoint` instance from a string representation. By convention in Java this method is named `parse`. It separates the input string in three parts on the `@` character, using the `split()` function, and then parsing an integer on the first and second part. Should any part be missing or an integer not have the right syntax, an exception is raised that is handled within the method by returning a null value.

```
public static MyPoint parse(String s){
    //  return a MyPoint instance from a String as produced by toString()
    String[] ss = s.split("@");
    if(ss.length!=2)return null; // badly formed Point
    try {
        return new MyPoint(Integer.parseInt(ss[0]),Integer.parseInt(ss[1]));
    } catch (NumberFormatException e){
        return null;
    }
} //  method parse
```

The `static` keyword indicates that is a *class* method, i.e. it will called from the `MyPoint` class e.g. `MyPoint.parse(aString)`, instead than from one of its instances such as `aPoint.getX()`. As `parse(aString)` creates a new `MyPoint`, it can be considered as a special type of constructor, often called *factory methods* in the Java terminology.

Another kind of very useful static method is the public class method `main(String[])`[4] which is called when the class is directly called from the system prompt.

```
public static void main(String[] args){
    MyPoint p0 = new MyPoint(50,50);
    MyPoint p1 = new MyPoint(100,100);
    MyPoint p2 = new MyPoint(200,75);
    MyPoint p3 = new MyPoint(p0.getX(),p0.getY());

    System.out.println("p0 = "+p0+" vs p3 = "+p3);
    System.out.println(p1+" + "+p2+" = "+p1.add(p2));
    System.out.println(p1+" - "+p2+" = "+p1.sub(p2));
    System.out.println(p1+" * "+3+" = "+p1.scale(3));
```

[4]One should take the habit of defining a `main(String[])` class method for each important class that we define to make it easier to test the class separately. As can seen in the full code listing, we have done this for all non-abstract classes but we do not describe it the `main` methods are not shown in this paper.

```
        System.out.println("dist("+p1+","+p2+") = "+p1.dist(p2));
        System.out.println(MyPoint.parse("45@58"));
        System.out.println(MyPoint.parse("1@2@3"));
        System.out.println(MyPoint.parse("23.4@45"));
    }
```

The following is an excerpt of an interaction at a Unix prompt. We first compile the class using `javac` and then we call the Java interpreter on `MyPoint` which calls the `main` class method of `MyPoint`. What follows the prompt is the trace output which gives us some confidence that our class is behaving as we want.

```
=> javac MyPoint.java
=> java MyPoint
p0 = 50@50 vs p3 = 50@50
100@100 + 200@75 = 300@175
100@100 - 200@75 = -100@25
100@100 * 3 = 300@300
dist(100@100,200@75) = 103.07764064044152
45@58
null
null
```

4 The `AnimatedBody` class

This class describes the general behavior of the bodies that move around a setting. It defines instance variables representing the state of each body and it implements messages describing the actions performed by all bodies.

4.1 Instance variables

A body is defined by a name and a position in a setting. As we want to simulate movement, each body is defined by an array of bitmaps picture (`Image`s in the Java terminology) giving different aspects. Movements of the bodies can be simulated by displaying different aspects cyclicly. A body is represented by different pictures to indicate where it is heading; for example, it can be seen from the front, back or side. Figure 2 shows in a window the four directions (numbered 0 to 3) along the horizontal axis and the two aspects (numbered 0 and 1) for each on the vertical axis for a butterfly. A flower has only two "directions" (numbered 0 and 1) and one aspect (numbered 0). As every `AnimatedBody` has this behavior, it is defined in this class but the exact picture to display is left to each kind of `AnimatedBody`. We will see in the next section how the pictures are kept but each `AnimatedBody` has its own direction and aspect. Table 2 shows the four instance variables of an `AnimatedBody` instance.

4.2 Instance Methods

Like we did in section 3, one should always try to hide the internal state of an object and avoid the `public` declaration of variables as much as possible. We instead define some `public` methods that

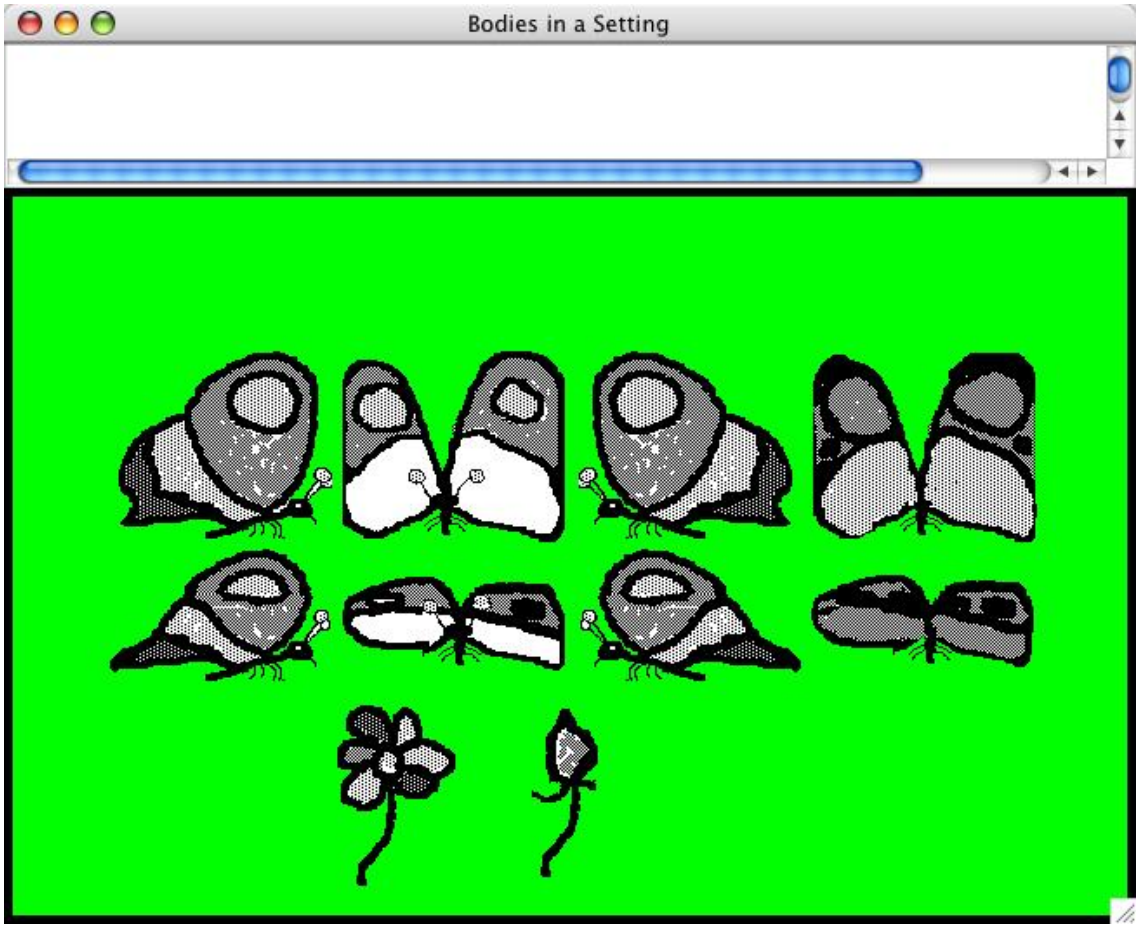

Figure 2: Directions and aspects of `AnimatedBody` instances. There are four directions and two aspects for a `Butterfly`. There are two directions and one aspect for a `Flower`.

give a controlled access to these variables. These methods are quite "trivial" and are often called *accessor* methods.

By convention, to give access to the variable, we give it the name formed by `get` followed by the name of the internal variable (but with an initial capital). To change the value of the internal variable, we define a method whose name is `set` followed by its name and having as parameter the new value to assign to the internal variable. For example, to access the variable `direction`, `getDirection()` will give its current value and `setDirection(v)` will change its value to `v`[5].

We might argue that it would be much simpler to simply declare the internal variable as `public` instead of defining two new methods for each variable, but bear in mind that the methods do not have to be so simple and could easily implement a control over the access to this variable. But, more important, it defines an access to this variable that can be eventually redefined further down in a hierarchy of objects. We give here some code excerpts that define these instance variables and their accessor methods.

[5]This convention is also used for defining Java Beans

`name`	identifies a body
`position`	where it is in the setting
`direction`	the current direction
`aspect`	the current aspect

Table 2: Instance variables of an `AnimatedBody` instance

```
private String name;
private MyPoint position;
private int direction,aspect;

public String getName(){
   return name;
}
public void setName(String name){
   this.name=name;
}

public int getDirection(){
   return direction;
}

public int getAspect(){
   return aspect;
}

public MyPoint getPosition(){
   return position;
}
public void setPosition(MyPoint position){
   this.position = position;
}
```

But changing the `aspect` and `direction` is a bit more complicated because the value has to lie within an acceptable number of possible values returned by the methods `getNbAspects()` and `getNbDirections()` (to be defined in section 4.3) `%` is the modulo operator. As a body should not care about the aspect and direction numbers, we build on these methods two other methods `nextAspect()` and `nextDirection()` that simply change to the next one.

```
public void setDirection(int d){
   direction = d % getNbDirections();
}
public void nextDirection(){
   setDirection(direction+1);
}
```

```
public void setAspect(int a){
   aspect = a % getNbAspects();
}
public void nextAspect(){
   setAspect(aspect+1);
}
```

Moving a body is obtained by changing its position and aspect by calling some of these methods. Note that an object can use some of its own methods. Although the object could change its position directly by assigning a new value to the instance variable, it is more appropriate to call the `setPosition(MyPoint)`[6] method.

```
public void moveTo(MyPoint newPos){
   nextAspect();
   setPosition(newPos);
}
```

As we do not want that an object changes its name during its life, we do not define a method to change the name. The initialization of the name is done within the constructor of the class which also initializes the other instance variables using the previously defined methods.

```
AnimatedBody(String name){
   this.name = name;
   position(new MyPoint(0,0));
   setDirection(0);
   setAspect(0);
}
```

For a body to appear on the screen, we use a collection of images corresponding to each direction and aspect. These images will be displayed on the screen. The access to the current image is a bit more complicated (and interesting...). As we would like all instances of the same subclass of an `AnimatedBody` to share the same array of images, they cannot be instance variables because in this case, they would be repeated in each class. So we make them `static` variables of the class instead. `static` is (an unfortunate) term used in Java to designate variables and methods that in other object oriented languages are called *class variables* and *class methods*; in this document, we will use this latter term to designate these shared variables between all instances of a class. Like we did in the `MyPoint` class, a class method is designated by the name of the class itself and not an instance.

But note that although these images are global to all instances of the class, they should not be shared among all instances of `AnimatedBody` but only among instances of its subclasses, i.e. all instances of `Butterfly` share the images of the `Butterfly` images and instances of `Flower` share instances of `Flower` images. So these class variables will be defined within each subclass, but we we need some access to the values of these variables... so how can we define, once in a superclass, actions that will be shared among all its subclasses but that depend on the behavior of subclasses.

The way out of this dilemma is to declare, in the superclass, `abstract` methods to be defined in the subclasses but that can be called from the superclass methods as if they were defined. Declaring

[6] from this point on, when we refer to a method, we give its name followed by the types of its parameters in parentheses

an `abstract` method in a class makes it also `abstract` so now the definition of an `AnimatedBody` is as follows:

```
abstract class AnimatedBody {
   ...
}
```

which makes it impossible to create instances of `AnimatedBody` but only of its subclasses. This makes senses because abstract classes lack some functionalities.

4.3 `abstract` methods

These methods have to be defined by the subclasses and will return an access to the array of all images for a body, the number of its aspects and directions.

```
abstract Image[][] getFilm();
abstract int getNbAspects();
abstract int getNbDirections();
```

The display of an image on the screen is done using Graphic Interchange Format (GIF) `Image`s and we will explain later how they are used. These images are built using an external graphic editor and saving the result in GIF format. Images are of rectangular shapes, but it is possible to define some parts of the image as being transparent so that, when they are displayed, they appear as being of an irregular shape.

The `film` class variable will be initialized with an array of dimension `getNbAspects()` each element being an array of `getNbDirections()` images. As the reading algorithm is the same for all subclasses of `AnimatedBody`, it is defined in the class as a class method. Reading is done using Java methods that can get images anywhere on the World Wide Web but in this case, we take for granted that the images are within a directory called `images` located in the same directory as the application. The name of the image files are of the form

class_name aspect_number direction_number `.GIF`

```
protected static Image [][] initFilm(String className,int nbA,int nbD){
    Image [][] f = new Image[nbA][nbD];
    String baseFileName = "images/"+className;
    for (int a=0;a<nbA;a++){
        for(int d=0;d<nbD;d++){
            f[a][d] = new ImageIcon(baseFileName+a+d+".GIF").getImage();
        }
    }
    return f;
}
```

4.4 Instance methods (continued)

When an `AnimatedBody` instance needs to access the current image corresponding to its current aspect and direction, it indexes the array of images initialized in `initFilm`. We also define a method for checking is a point `x,y` is within the current image of an animal and another to display the current image in a graphical context

```
public Image image(){
   return getFilm()[getAspect()][getDirection()];
}
public boolean contains(int x, int y){
   Image im = image();
   return new Rectangle(position.getX() - im.getWidth(setting)/2,
                        position.getY() - im.getHeight(setting)/2,
                        im.getWidth(setting),im.getHeight(setting)).contains(x,y);
}
public void display(Graphics g){
    Image im = image();
    g.drawImage(im,getPosition().getX() - im.getWidth(setting)/2,
                   getPosition().getY() - im.getHeight(setting)/2,setting);
}
```

Like we did for `MyPoint`, we redefine the `toString()` method to get a readable version of an `AnimatedBody` i.e. the values of the instance variables with appropriate labels. To get the name of the class, we call the `getClass()` method which returns the class of an object to which we ask its name, to which we add the name of the instance, we then add the list of the variable names with their values enclosed in square brackets.

```
public String toString(){
   return getClass().getName()+" "+name+
          "[pos="+position+",dir="+direction+",asp="+aspect+"]";
}
```

4.5 Graphical user interface

The animated bodies will move within a setting (an instance of `Setting` defined in section 5) and leave a written trace of their action in a `JTextArea`, a Java class that implements a stream of characters whose contents can be displayed in a scrollable window. As all animated bodies share a single setting and trace window, we will keep a single reference to these as class variables and define a class method to initialize them. Note that `this` cannot be used in a class method, so we have to use another variable name as parameter of `setSettingTrace`.

```
private static JTextArea traceTA;
protected static Setting setting;

public static void setSettingTrace(Setting s,JTextArea t){
   setting=s;
   traceTA=t;
}
```

To add information to the trace, we define the following instance method which adds a line to the JTextArea comprising the name of current object followed by the string given as parameter:

```
public void trace(String s){
   traceTA.append(this+":"+s+"\n");
}
```

Animal movements are defined by a pop-up menu that will be displayed by the setting when it detects that the user clicked over an animal (see figure 3). It will be the job of an animal to display the menu and to perform the appropriate action on the current animal. The items of the pop-up menu will be listened by a single listener that will call the `perform(String)` method of the current animal.

```
// listener for all popmenu actions
public static ActionListener AniBodyMenuListener = new ActionListener(){
    public void actionPerformed(ActionEvent e){
        thisBody.perform(e.getActionCommand());
    }
};
```

As all animals will share many of these behaviors, we define methods that will be called by subclasses in order to build their own action menu.

```
public static JPopupMenu appendMenu(JPopupMenu menu,String[] labels){
   for(int i=0;i<labels.length;i++){
      JMenuItem item = new JMenuItem(labels[i]);
      item.addActionListener(AniBodyMenuListener);
      menu.add(item);
   }
   return menu;
}
```

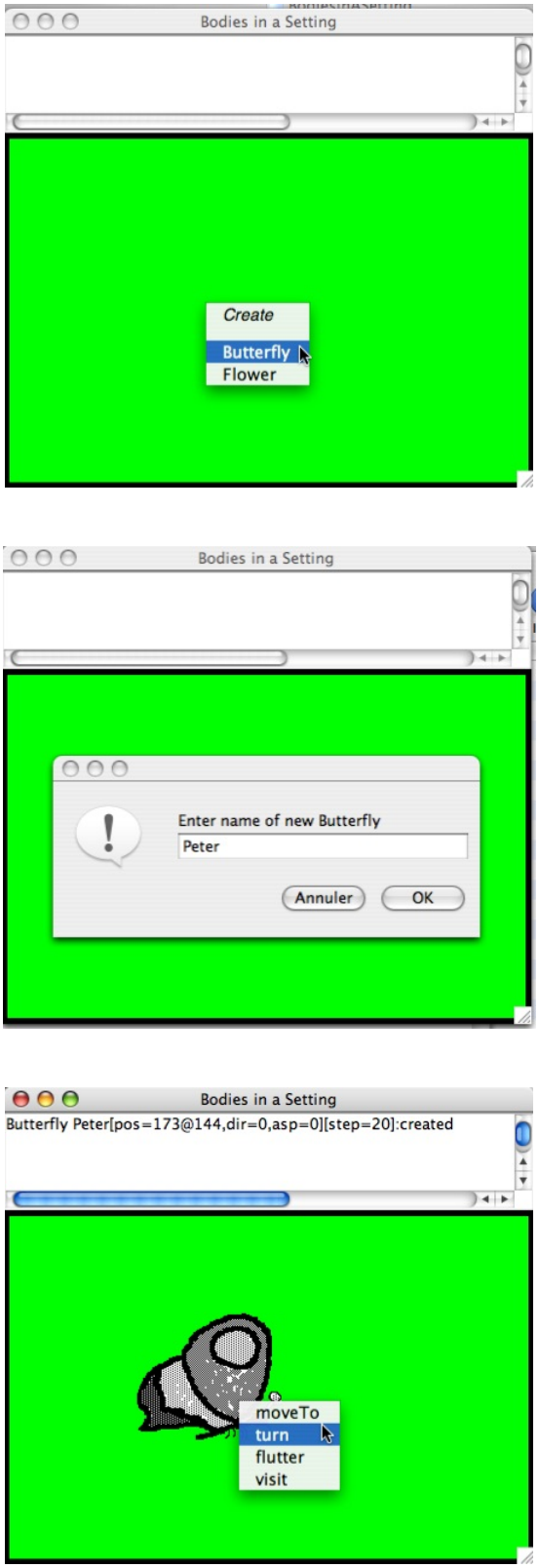

Figure 3: Three steps in creating a new animal and asking it to turn: display the creation menu, ask for the name of new animal and making it turn

Like we did for the film, all instances of a subclass of `AnimatedBody` will share the same menu. Thus this menu will be defined in the subclass, so the method `showMenu` that uses it must also be defined by all subclasses of `AnimatedBody`. This is why the declare `showMenu(int,int)` as `abstract`.

```
abstract public void showMenu(int xMouse,int yMouse);
```

As movement is defined for all animated bodies, we declare the following

```
protected static String[] aniBodyMenuLabels = {"moveTo"};
```

for defining the content of the menu to be displayed to the user. Subclasses will add their own action labels after this value.

`perform(String)` first asks the user for the new position of this animated body and then calls `moveTo(MyPoint)` and traces this action:

```
public void perform(String action){
    if(action.equals("moveTo")){
        MyPoint pt = setting.askPosition("New position (x@y) for "+name);
        moveTo(pt);
        trace("moveTo:"+pt);
    }
}
```

Now that we have defined all the necessary methods of a general `AnimatedBody`, we can look at the specific behaviour of its subclasses.

4.6 Subclasses of `AnimatedBody`

4.6.1 The `Butterfly` class

We define a `Butterfly` as a special `AnimatedBody` that can do everything an `AnimatedBody` can do and more. A `Butterfly` can appear to be fast or slow depending on the value of an instance variable `step` that defines the distance a `Butterfly` travels before being displayed with a new aspect. `step` is accessed and updated by the following *accessor* and *mutator* methods:

```
int step;
public int getStep(){
   return step;
}
public void setStep(int s){
   step = s;
}
```

We define a new movement method `turn()` which is merely another name for `nextDirection()` but one that is more appropriate for an animal.

```
public void turn(){
   nextDirection();
}
```

The string version of a `Butterfly` is obtained by redefining the `toString()` method which starts by calling the one defined in `AnimatedBody` and adding the value of the `step` instance variable.

```
public String toString(){
   return super.toString()+"[step="+step+"]";
}
```

The creation of a Butterfly is defined with the following constructor which makes a `Butterfly` fly to a position when it is created.

```
private static final int INITSTEP = 10;
Butterfly(String name, MyPoint pos){
   super(name);
   step = INITSTEP;
   setPosition(pos);
}
```

This first calls the constructor of the superclass (i.e. `AnimatedBody(String)` defined in section 4.2) and then initializes the `step` to a default value and then moves to the position given in the constructor call. So to declare a new Butterfly called *Peter* at position `(100,100)`, we can use:

```
Butterfly peter = new Butterfly("Peter",new MyPoint(100,100));
```

We also define the *default* constructor, i.e. one with no parameters which merely initializes the `step`, it will be used by the `setting` for creating new `Butterfly` dynamically.

```
Butterfly(){
   super("");
   step = INITSTEP;
}
```

We redefine `moveTo(MyPoint)` so that a `Butterfly` turns in the right direction before starting to go at a new position. This behavior is build upon the `moveTo(MyPoint)` of an `AnimatedBody`. But a `Butterfly` does not *jump* directly to the new position changing its aspect once, it goes there gradually. If the distance Δ between the current position p and the destination d is greater than the `step`, a new position p' is computed by the following formula:

$$p' = p + (d - p) \times \frac{\texttt{step}}{\Delta}$$

using the arithmetic functions we defined in the `MyPoint` class

```
public void moveTo(MyPoint destination){
   if (getPosition().x < destination.x)
      setDirection(0);
   else setDirection(2);
   MyPoint pos = getPosition();
   double distance = destination.dist(pos);
   while(step<distance){
      super.moveTo(pos.add(destination.sub(pos).scale((float)step/distance)));
```

```
      pos=getPosition();
      if(setting!=null)setting.paintImmediately();
      distance = destination.dist(pos);
   }
   super.moveTo(destination);
}
```

This method uses `moveTo(MyPoint)` defined in `AnimatedBody` using `super` as receiver of the message. `super` designates the current object but the search for methods starts at the superclass of the class where the `super` appears. If the `setting` is defined, we force its redisplay.

A `Butterfly` can also perform other specific actions:

- *flutter in place* for a certain number of times, changing aspect each time
- *visit a flower* by going to the flower, if the flower is *open* then the `Butterfly` flutters over the flower otherwise it goes slightly below.

```
public void inPlace(int nb){
   if(nb>0){
      for(int i=0;i<nb;i++){
         nextAspect();
         setting.paintImmediately();
      }
   }
}

public void visit(Flower f){
   moveTo(f.getPosition());
   if(f.isOpen()){
      turn();
      inPlace(5);
   } else
      moveTo(f.getPosition().add(new MyPoint(0,100)));
}
```

As we described earlier, each subclass of `AnimatedBody` needs five class variables described in table 3. For the `Butterfly` class, the `static` (class) variables are declared as:

`film`	collection of images
`nbAspects`	number of aspects
`nbDirections`	number of directions
`bodyMenu`	pop-up menu of its actions
`bodyMenuLabels`	labels displayed in the menu

Table 3: Class variables of an `AnimatedBody` subclass

```
private static int nbAspects = 2;
private static int nbDirections = 4;
private static Image[][] film;
private static JPopupMenu butterflyMenu;
private static String[] butterflyMenuLabels = {"turn","flutter","visit","new step"};
```

For initializing the class, we define the following class method which

- *registers* the `Butterfly` class for the setting, as we will see in section 5, this is for enabling the setting to call the `Butterfly` constructor
- *initializes* the `film` class variable using the `initFilm` class method defined in `AnimatedBody` now that the number of aspects and directions are known
- creates the pop-up menu of the `Butterfly` actions using `appendMenu` class method also defined in `AnimatedBody`

```
public static void initialize(Applet applet){
   setting.registerAnimatedBody(Butterfly.class);
   film = initFilm("Butterfly",nbAspects,nbDirections);
   butterflyMenu = appendMenu(appendMenu(new JPopupMenu(),aniBodyMenuLabels),
                              butterflyMenuLabels);
}
```

`Butterfly` is a subclass of the `AnimatedBody` abstract class so to make it `concrete` and be able to create instances of this class, we define its abstract methods which merely return the value of a class variable. Note that although these methods access class variables, we make them instance methods which are usually more convenient to use. We also define the `showMenu` function, which keeps track of the current `AnimatedBody` which is needed by the pop-up menu listener so that it calls the appropriate `perform` function.

```
public int getNbAspects(){
   return nbAspects;
}
public int getNbDirections(){
   return nbDirections;
}
public Image[][] getFilm(){
   return film;
}
public void showMenu(int xMouse,int yMouse){
   thisBody=this;
   butterflyMenu.show(setting,xMouse,yMouse);
}
```

`perform` checks that the `action` is one defined by the `Butterfly` otherwise it calls the `perform` of the superclass. For each action, if necessary it asks some information from the user, it then calls the appropriate instance method and adds a textual trace of the action performed. Finally it updates the display of the `setting`.

```
public void perform(String action){
   if(action.equals("turn")){
      turn();
      trace("turn");
   } else if (action.equals("flutter")){
      inPlace(5);
      trace("inPlace");
   } else if (action.equals("visit")){
      AnimatedBody b = setting.askBody("Name a flower to visit");
      if(b instanceof Flower){
         visit((Flower)b);
         trace("visit:"+b.getName());
      }
   } else if(action.equals("new step")){
      int newStep = setting.askInt("new positive integer value for step "+
                                   "(current="+step+")");
      if(newStep>0 && newStep<100){
         setStep(newStep);
         trace("setStep:"+step);
      }
   } else
      super.perform(action);
   setting.repaint();
}
```

4.6.2 The `Flower` class

The structure of the `Flower` class is basically the same as the `Butterfly` one except that it is much simpler because a `Flower` can only `open()` or `close()` itself. The fact that it is open or closed depends on its current `direction` (defined in `AnimatedBody`). It also has only one `aspect` so it needs only two bitmaps. Like `Butterfly`, it inherits the `moveTo(MyPoint)`, `getAspect()`, etc. from `AnimatedBody`. It defines:

```
public boolean isOpen(){
   return getDirection() == 0;
}
public void open(){
   setDirection(0);
}
public void close(){
   setDirection(1);
}
```

The creation of an instance of the `Flower` class is done as for the `Butterfly` class and the same five class variables are also defined.

```
Flower(String name, MyPoint pos){
```

```
   super(name);
   position(pos);
}
Flower(){
   super("");
}

private static int nbAspects = 1;
private static int nbDirections = 2;
private static Image[][] film;
private static JPopupMenu flowerMenu;
private static String[] flowerMenuLabels = {"open","close"};
```

The initialization is also on the same lines as for a `Butterfly`:

```
public static void initialize(Applet applet){
   setting.registerAnimatedBody(Flower.class);
   film = initFilm("Flower",nbAspects,nbDirections);
   flowerMenu = appendMenu(appendMenu(new JPopupMenu(),aniBodyMenuLabels),
                           flowerMenuLabels);
}
```

The abstract methods are defined analogously:

```
public int getNbAspects(){
   return nbAspects;
}
public int getNbDirections(){
   return nbDirections;
}
public Image[][] getFilm(){
   return film;
}
```

The user interface methods are also defined similarly as for a `Butterfly`:

```
public void showMenu(int xMouse,int yMouse){
    thisBody=this;
    flowerMenu.show(setting,xMouse,yMouse);
}
public void perform(String action){
     if(action.equals("open")){
         open();
         trace("open");
     } else if (action.equals("close")){
         close();
         trace("close");
     } else super.perform(action);
```

```
        setting.repaint();
    }
```

5 The `Setting` class

The setting is our basic entry point and the *dispatcher* of messages for all bodies that evolve within it. It is also responsible for displaying all bodies at their current position, direction and aspect.

To react to mouse events, we could define a class that implements the `MouseListener` interface and by defining the following methods: `mouseClicked(MouseEvent)`, `mouseEntered(MouseEvent)`, `mouseExited(MouseEvent)`, `mousePressed(MouseEvent)` and also `mouseReleased(MouseEvent)`. But in our case, we only want to deal with mouse clicks so it is more practical to use a subclass of the `MouseAdapter` class which defines dummy methods for all mouse events so that we only need to define the one that is of interest to us.

The setting is conceptually only an object (more precisely a `JPanel` instance) that knows about *AnimatedBodies* that evolve within its borders. It also deals with mouse clicks within its border: this means determining if the last mouse click was over a displayed animated body or not. If it was, then it calls the `showMenu` menu of this body, otherwise it presents a menu for creating a new animated body at the position of the click (see figure 3). This behavior is initialized in the constructor.

A `Setting` has four instance variables:

- `bodies` is a `Map<String,AnimatedBody>` (i.e. a hash table) of `AnimatedBody` that evolve within the setting; the name of the `AnimatedBody` serves a the key.
- `creationMenu` a pop-up comprising the name of each `AnimatedBody` subclass, the items of this menu are added when the `AnimatedBody` subclasses call the `registerAnimatedBody(Class)` class method.
- `xMouse` and `yMouse` keep track of the position of the last click, they are used as a means of communication between the `MouseListener` and the item menu listener defined in `registerAnimatedBody(Class)`

```
public class Setting extends JPanel{
private Map<String,AnimatedBody> bodies;
   private JPopupMenu creationMenu;
   private int xMouse,yMouse;

   Setting (){
      setBackground(Color.green);
      bodies = new HashMap<String,AnimatedBody>();
      creationMenu = new JPopupMenu();
      creationMenu.add(new JMenuItem("<html><i>Create</i></html>"));
      creationMenu.addSeparator();
      addMouseListener(new MouseAdapter(){
         public void mousePressed(MouseEvent e){
            xMouse = e.getX();
```

```
            yMouse = e.getY();
            AnimatedBody b = findBody(xMouse,yMouse);
            if(b==null)
               creationMenu.show(Setting.this,xMouse,yMouse);
            else
               b.showMenu(xMouse,yMouse);
         }
      });
   }
```

The setting often has to find a given `AnimatedBody` by its name or to determine which `AnimatedBody` is displayed at a certain position, so we define two methods for searching the `Map`.

```
    public AnimatedBody findBody(int x, int y){
        for(AnimatedBody b: bodies.values())
            if(b.contains(xMouse,yMouse))return b;
        return null;
    }

   public AnimatedBody findBody(String name){
      return bodies.get(name);
   }
```

In principle, the setting should not have to know the name of the subclasses of `AnimatedBody` that evolve within it except for the fact that it has to create them by calling their constructor. So normally, it would be necessary to change the menu and the call to the corresponding constructor each time a new subclass of `AnimatedBody` is written.

It is possible to circumvent this problem by using a simple form of *reflection* by getting access to the class itself and calling the default constructor via the `newInstance` method of the class. The new object is then initialized with mutator functions for setting its name and position. An indication of the creation is also added to the trace and the setting is redisplayed.

But this implies that each new class makes itself known to the setting in some way: this is why we have defined the following method which keeps track of each class and uses its name to create a new menu item label and the corresponding listener which is then added to the `creationMenu`.

```
public void registerAnimatedBody(final Class<?> c){
  JMenuItem item = new JMenuItem(c.getName());
  item.addActionListener(new ActionListener(){
     public void actionPerformed(ActionEvent e){
        String bodyName = = askNewBodyName("Enter name of new "+c.getName());
        MyPoint pos = new MyPoint(xMouse,yMouse);
        try {
           AnimatedBody b = (AnimatedBody) c.newInstance();// creation of a new body
           b.setName(bodyName);// initialisation of fields
           b.setPosition(pos);
           bodies.put(bodyName,b);// add to the known bodies
           b.trace("created");
```

```
            repaint();
         } catch (Exception ie){
            System.out.println(ie+"** could not create "+c.getName()+
                               "("+bodyName+","+pos+")");
         }
      }
   });
   creationMenu.add(item);
}
```

The setting will often need to ask some information from the user using a modal dialog box, so we define some auxiliary procedures that do some primitive validation on the form of the input given typed by the user.

```
   public String askString(String prompt){
      return JOptionPane.showInputDialog(this,prompt,"",JOptionPane.QUESTION_MESSAGE);
   }

   public int askInt(String prompt){
      while (true)
         try{
            String s = askString(prompt);
            return Integer.parseInt(s);
         } catch (NumberFormatException e){}
   }

   public MyPoint askPosition(String prompt){
      MyPoint pt = null;
      while (pt==null){
         String s = askString(prompt);
         pt=MyPoint.parse(s);
      }
      return pt;
   }

   public AnimatedBody askBody(String prompt){
      return findBody(askString(prompt));
   }

   public String askNewBodyName(String prompt){
      String bodyName = null;
      while(bodyName==null || bodyName.length()==0 || bodies.get(bodyName)!=null)
         bodyName = JOptionPane.showInputDialog(Setting.this,prompt,"",
                                                JOptionPane.QUESTION_MESSAGE);
      return bodyName;
   }
```

The display of a `JPanel` is done by redefining the `paintComponent` method which first paints the background in green with a black border around, it then asks to each body to display itself. `paintImmediately` is a simplified call to the `JPanel` `paintImmediately(int,int,int,int)` method giving the dimension of the whole `JPanel`; it is used for simulating animation in movement.

```
  public void paintComponent(Graphics g) {
      super.paintComponent(g);
      int width = getWidth();
      int height = getHeight();
      // draw border
      g.setColor(Color.black);
      g.fillRect(0,0, width, height);
      g.setColor(getBackground());
      g.fillRect(5,5,width - 10, height - 10);
      for(AnimatedBody b:bodies.values())
          b.display(g);
      // cheap trick to slow the animations...
      try{Thread.sleep(100);}catch (InterruptedException e){};
   } // paint method

   public void paintImmediately(){
      paintImmediately(0,0,getWidth(),getHeight());
   }
```

6 `BodiesInASetting` class

Now that we have our setting in which the animated bodies can evolve, we can now build a graphical application containing both the `Setting` instance and the textual trace. As we want our program to be used as a stand-alone application, we will define our main class as a subclass of `JPanel`. When this class is called from the command line, it will call the `main` method which will first create a `JFrame` in which the `JPanel` will be put. It first creates the objects and then does their layout.

`createObjects()` first creates the `JTextArea` for the trace and then the `Setting` instance which are added to the content pane of the applet. The class variables of `AnimatedBody` are then set and the `Butterfly` and `Flower` classes are also initialized for reading their images and registering to the setting. If we want to add new subclasses of `AnimatedBody`, we would have to initialize them here also.

In `layoutObjects()`, the `JTextArea` is embedded in a `JScrollPane` in order to get the scroll bars and the instance of the `Setting` is added in the center of the `JPanel`.

```
public class BodiesInASetting extends JPanel{
    // elements of the display
    private JTextArea traceTA;
    private Setting park;

    private void createObjects(){
```

```
        // create the interface elements
        traceTA = new JTextArea(5,60);
        park = new Setting();
        // initialize the classes
        AnimatedBody.setSettingTrace(park,traceTA);
        Butterfly.initialize();
        Flower.initialize();
    }

    public void layoutObjects(){
        this.setLayout(new BorderLayout());
        this.add(new JScrollPane(traceTA),BorderLayout.NORTH);
        this.add(park,BorderLayout.CENTER);
        this.validate();
    }

    public static void main(String[] args){
        JFrame f = new JFrame("Bodies in a Setting");
        f.setSize(700,600);
        BodiesInASetting a = new BodiesInASetting();
        a.createObjects();
        a.layoutObjects();
        f.getContentPane().add(a);
        f.setLocation(200,100);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

7 Ideas for extending the system

Now we have all the elements for a functional system but it would be fun to add new kinds of bodies, for example: cows that eat flowers, crabs that only move in right angles, rabbits that jump, turtles that leave a path when they move, etc. We leave these as an exercise to the reader who would need to draw images and redefine the appropriate movement methods, menus and messages.

Another interesting extension would be to define groups of bodies that move in parallel using the quasi-parallelism available through the `Thread` class. This might involve a bit more work to get it working smoothly with a good user interface.

Another kind of extension that is really the "essence" of object oriented programming: subclassing would be to define a `StickyButterfly` so that when it `visit`s a `Flower`, it follows it if the flower moves. Should the `StickyButterfly` decide to move elsewhere without `visit()`ing then the `Flower` can move without dragging the butterfly any more. The idea being that a `StickyButterfly` is now becomes linked to a flower when it visits one and so when the `Flower` moves, the `StickyButterfly` moves with it. "Stickiness" could also be defined for flowers.

8 Conclusion

We have shown how to develop an interactive application in Java. We first defined mouse-driven `AnimatedBodies` to move around in a window. This gave us the opportunity to see examples of instance and class variables and methods, subclasses and the Java event driven approach to graphical user interfaces. We finally gave many suggestions for extending the system.

References

[1] Valérie Berthié, Jean-Baptiste Briaud, *Swing, la synthèse*, Dunod, 2001.

[2] Xavier Briffault, Stéphane Ducasse *Squeak: programmation*, Eyrolles, 2002.`http://www.squeak.org`

[3] James Gosling, Bill Joy, Guy Steele, Gilad Bracha *The Java Language Specification*,2nd ed, Addison-Wesley, 2000.

[4] Trevor Hopkins, Bernard Horan, *Smalltalk: an introduction to application development using VisualWorks*, Prentice-Hall, 1995.

[5] Cay Horstmann, *Big Java*, 25th ed, Wiley, 2014.

[6] Lapalme, G. *The development of an application in Smalltalk*, internal report available at `http://www.iro.umontreal.ca/ lapalme/BodiesInASetting/DevelopmentInST80.pdf`, 1997.

9 Acknowledgements

We thank Anne Bergeron who wrote the first version of this program after having been given only a few hours of introduction to Smalltalk. Her work was "spectacular" and used as a demo at the Université de Montréal. It prompted us to revise and extend the system. The bitmaps of the flowers and the butterflies were designed by her. As her work was initially done in the eighties, it explains why the images are still in black and white.

We thank Pierre Cointe whose idea it was to use video animals moving on the screen to illustrate object-oriented programming. We also had many fruitful discussions with him, Isabelle Borne, Jean-François Perrot and Jean Bézivin that gave us a better understanding of Smalltalk. We thank Jean Vignolle, Patrick Sallé and Henri Farreny of IRIT in Toulouse for providing a stimulating working environment where a Smalltalk-80 version of this program was first developped. The VisualWorks version was redesigned during a stay at the Ecole des Mines de Nantes in March 1998 where discussions with Frédéric Rivard were most fruitful. Frédéric was very patient with me for explaining the "new" way of using the Smalltalk environment which was then used as a base for this Java version.

Finally we acknowledge the contribution of Jean Vaucher who introduced Simula-67 to us in the early seventies. Since then, we have always considered that object oriented programming is the "right" way to go...

Page numbers of method names