Chapter 6 Arrays

6.1 Many Objects of the Same Type

When a large number of variables of the same type is required, a numbered collection of variables called an *array* is often created.

Creating an array is a 2 step process. First one declares the type of the individual variables and a name:

```
int age[];       declares our intention to use:  age[0], age[1], age[2]…  each an int
double cost[];   declares our intention to use:  cost[0], cost[1], cost[2]…  each a double
Color tint[];    declares our intention to use:  tint[0], tint[1], tint[2]…  each a Color
```

One then creates the variables in each array.

```
age = new int[25];       creates 25 int variables:     age[0], age[1], …, age[24]
cost = new double[50];   creates 50 double variables:  cost[0], cost[1], …, cost[49]
tint = new Color[10];    creates 10 Color variables:   tint[0], tint[1], …, tint[9]
```

The following are examples of statements using array variables:

```
age[2] = 21;                     tint[3] = Color.white;
age[0] = age[2];                 tint[5] = new Color(50, 0, 200);
cost[1] = 39.95;                 tint[6] = tint[3].darker();
cost[3] = cost[1] + cost[2];     g.setColor(tint[4]);
```

An array avoids the tedious task of separately naming a large number of variables. However, the greatest power of the array comes from the fact that the number in the brackets can be a variable or formula.

Each of these three examples is equivalent (the first example requires 25 lines, '…' represents 20 lines).

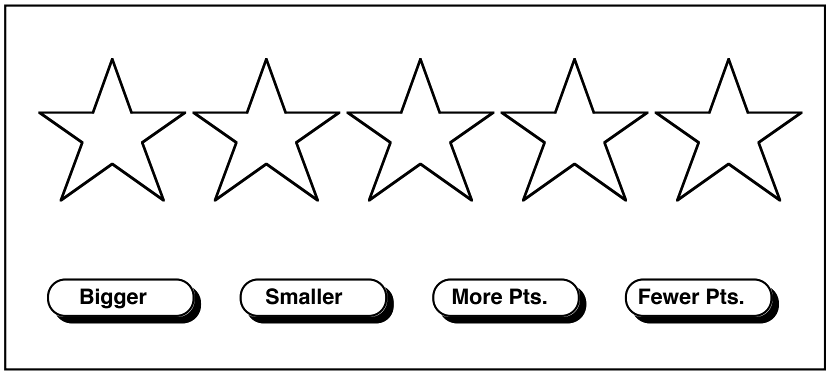
```
a. age[0] = 21;        b. int a = 0;          c. for (int a = 0; a < 25; a++)
   age[1] = 21;           while (a < 25)             age[a] = 21;
   age[2] = 21;              {
   age[3] = 21;              age[a] = 21;
   ...                       a++;
   age[24] = 21;             }
```

These two examples are equivalent; each creates 10 shades of blue-green.

```
a. tint[0] = new Color(0, 165, 165);
   tint[1] = new Color(0, 175, 175);
   ...
   tint[9] = new Color(0, 255, 255);

b. int brightness = 165;
   for (int s=0; s<10; s++)
      {
      tint[s] = new Color(0, brightness, brightness);
      brightness += 10;
      }

            or

   for (int s=0; s<10; s++)
      tint[s] = new Color(0, 165 + 10*s, 165 + 10*s);
```

Throughout this chapter, we will use the *for* form of a loop in preference to the *while* form. The two forms are very nearly equivalent. The *for* form is the most commonly used form with arrays. A description of the use of *for* and a comparison with *while* appear in section 2.6.

6.2 A Program Example with an Array



This program is a modification of the program in chapter 5, section 5.1
The number of stars is increased from two to five.
The buttons are relocated to match the figure above.
The *StarManager* class is changed by replacing each reference to *leftStar* and *rightStar* with a loop that refers to *star[0], star[1], …, star[4].*

The *StarManager* class with a loop

```
public class StarManager
   {
   private static final int HOW_MANY = 5, STAR_WIDTH = 70;

   private Star star[];

   public StarManager (int centerHorizontal, int centerVertical)
      {
      star = new Star[HOW_MANY];
      int left = centerHorizontal - HOW_MANY * STAR_WIDTH / 2;
      for (int s = 0; s < HOW_MANY; s++)
         {
         star[s] = new Star(left, centerVertical);
         left += STAR_WIDTH;
         }
      }

   public void paint(Graphics g)
      {
      for (int s = 0; s < HOW_MANY; s++)
         star[s].paint(g);
      }

   public void mark(int x, int y)
      {
      for (int s = 0; s < HOW_MANY; s++)
         if (star[s].contains(x, y))
            star[s].mark();
         else
            star[s].unmark();
      }

   public void bigger()
      {
      for (int s = 0; s < HOW_MANY; s++)
         star[s].bigger();
      }
   ... // The methods smaller, morePoints and fewerPoints are similar to bigger.

   }
```

Changes to the *Star* class and the Controller:

The initial radius of the stars must be made smaller to allow five to fit on one line, 30 is a good size.

Moving the buttons to the lower edge of the window requires changes in the constructor of the main program:

```
stars = new StarManager (150, 150);       // the area with the stars was 300 × 300 (center = 150)

          becomes

stars = new StarManager (200, 125);       // the area with the stars is 400 wide (center = 200)
                                          // and 250 high (center = 125)
```

The buttons need to be placed along the lower edge of the window.

```
bigger =  new AButton("Bigger",     320,  20);
smaller = new AButton("Smaller",    320,  50);
more =    new AButton("More Pts.",  320,  80);
fewer =   new AButton("Fewer Pts.", 320, 110);

          becomes

bigger =  new AButton("Bigger",      20, 250);
smaller = new AButton("Smaller",    110, 250);
more =    new AButton("More Pts.",  200, 250);
fewer =   new AButton("Fewer Pts.", 290, 250);
```

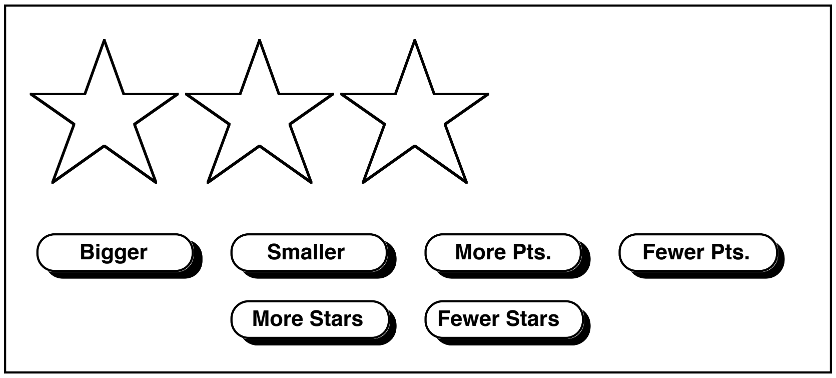
Moving the buttons to the lower edge of the window requires a change in the *mousePressed* method of the main program:

```
    if (x < 300)       // mark a star if the mouse is not near the right side of the window

          becomes

    if (y < 250)       // mark a star if the mouse is not near the bottom of the window
```

Exercises — 6.2

1. Put two rows of small graphical objects into the example. Use a single array, but place the objects in two rows. The positions of the objects are determined when the objects are created in the constructor. Use two loops in the constructor, the existing loop for the first five objects and a new, slightly different loop, that positions objects 5 through 9 in a second row.
2. On each floor of the house in chapter 5, use an array for the windows. Use loops to paint them.

6.3 Variable Length Lists



In the preceding example there were five stars. To change the program to display the three stars shown above, one need only change the constant HOW_MANY from five to three. Since this constant appears in the *StarManager* class only, it is easy to change the number of stars. To make the number of stars variable, one changes the constant HOW_MANY to a variable *howMany* and provides methods for the user to change this variable.

This program is sneaky in the sense that all five stars are created when the program begins, even though only 3 are visible (since *paint* only paints the first 3). Adding stars increases the number that are painted and removing a star decreases the number that are painted, but all 5 stars always exist.

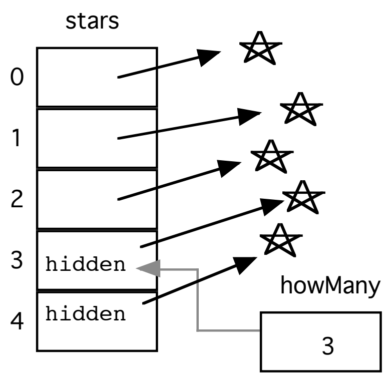
```
public class StarManager
    {
    private static final int HOW_MANY = 5, STAR_WIDTH = 70;

    private Star star[];
    private int howMany;

    public StarManager (int centerHorizontal, int centerVertical)
        {
        star = new Star[MAX];
        int left = centerHorizontal - HOW_MANY * STAR_WIDTH / 2;
        for (int s = 0; s < HOW_MANY; s++)
            {
            star[s] = new Star(left, centerVertical);
            left += STAR_WIDTH;
            }
        howMany = 3;
        }

    public void paint(Graphics g)
        {
        for (int s = 0; s < howMany; s++)
            star[s].paint(g);
        }

    public void mark(int x, int y)
        {
        for (int s = 0; s < howMany; s++)
            if (star[s].contains(x, y))
                star[s].mark();
            else
                star[s].unmark();
        }
```

The class *StarManager* continues on the next page.

```
public void bigger()
   {
   for (int s = 0; s < howMany; s++)
      star[s].bigger();
   }
public void addStar()
   {
   if (howMany < MAX)
      howMany++;
   }

public void removeStar()
   {
   if (howMany > 0)
      {
      howMany--;
      star[howMany].unmark();
      }
   }
}
```

// The methods *smaller*, *morePoints*
// and *fewerPoints* are similar to *bigger.*

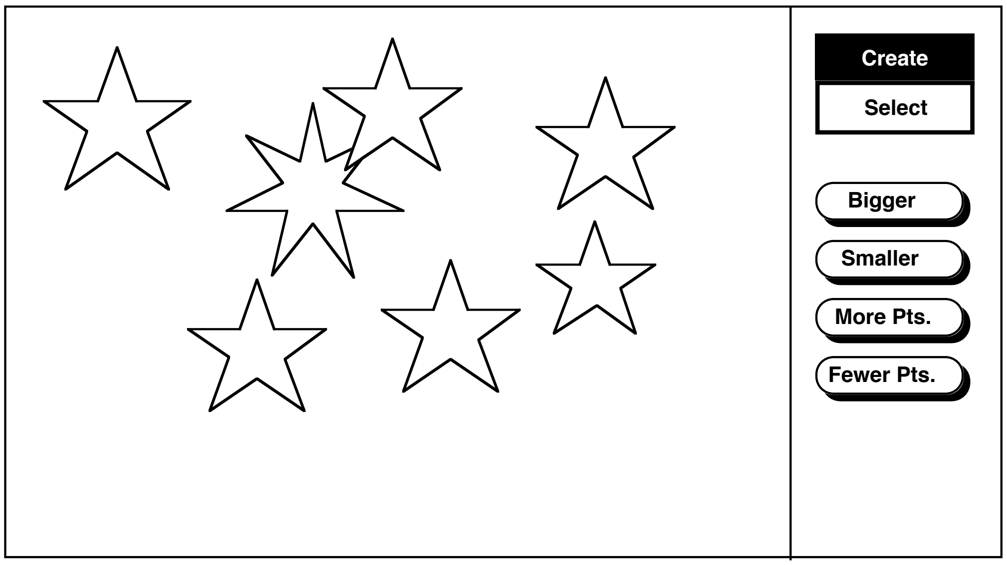
The method *removeStar* unmarks the star that it hides. This is done so that the star will not unexpectedly be marked when it later reappears (if and when the method *addStar* reveals it). The diagram is intended to help you understand why the value of *howMany* is decreased before the star is unmarked. Generally, the method *paint* displays each star whose position is less than *howMany*, thus displaying *howMany* stars. *HowMany* is also the position of the lowest numbered star that is hidden. If *howMany* has just been decreased, *howMany* is the number of the star that has just been hidden.



Exercises — 6.3

1. Modify the stars program if section 6.2 to include the *addStar* and *removeStar* methods of this section in the *StarManager* class. Change the controller to place the additional buttons in the window.
2. Put a maximum of 10 stars into this program. Use a single array, but place the stars in two rows. The positions of the stars are determined when the stars are created in the constructor. Use two loops in the constructor, the existing loop for the first five stars and a new, slightly different loop, that positions stars 5 through 9 in a second row.

6.4 A ‘Draw’ Program



Placing Stars where the Mouse is Pressed

A ‘draw’ program would have the user create stars by poking on the screen with the mouse — the stars being created where the user indicates. In itself, this is easy to implement. Alter the method *addStar* in the *StarManager* class to expect mouse coordinates and to use them to create a new *Star*. Also alter the *mousePressed* method in the controller to send mouse coordinates to *addStar*.

```
public void addStar(int h, int v)
    {
    if (howMany < MAX)
        {
        star[howMany] = new Star(h, v);
        howMany++;
        }
    }
```

Using the Mouse to Create and Select

The programs in sections 6.2 and 6.3 use the mouse to select stars (so they can be made bigger, smaller, etc.). We want to modify these programs to use the *addStar* method above without losing the ability to select stars. The standard solution is to have buttons (commonly called tools in this case) that choose the function of the mouse. Tool buttons, Create and Select in the illustration, should paint differently from other buttons and also optionally paint in a bold ‘chosen’ way. When the user clicks on a tool, it should appear ‘chosen’ and the other tool should appear ‘unchosen’. When the user clicks on the drawing area, call *select* or *addStar* depending on which tool has been chosen. The controller class *StarProg* now looks like:

```
public class StarProg extends EventPanel
    {
    private static final int GRAPHICS_BOUNDARY = 300;

    private StarManager stars;
    private ToolButton create, select;
    private AButton bigger, smaller, more, fewer;
```

The class *StarProg* continues on the next page.

```
public StarProg()
   {
   stars = new StarManager();
   create = new ToolButton("Create", GRAPHICS_BOUNDARY + 10, 20);
   create.choose();
   select = new ToolButton("Select", GRAPHICS_BOUNDARY + 10, 40);
   select.unchoose();
   bigger = new AButton("Bigger", GRAPHICS_BOUNDARY + 10, 70);
   smaller = new AButton("Smaller", GRAPHICS_BOUNDARY + 10, 100);
   more = ...
   }

public void paint(Graphics g)
   {
   ...
   }

public void mousePressed(MouseEvent e)
   {
   int x = e.getX(), y = e.getY();
   if (x < GRAPHICS_BOUNDARY)
      {
      if (create.isChosen())
         {
         stars.addStar(x, y);
         stars.mark(x, y);
         }
      else
         stars.mark(x, y);
      repaint();
      }
   else if (create.contains(x, y))
      {
      create.choose();
      select.unchoose();
      repaint();
      }
   else if (select.contains(x, y))
      {
      create.unchoose();
      select.choose();
      repaint();
      }
   else if (bigger.contains(x,y))
      { stars.bigger(); repaint(); }
   else if (smaller.contains(x,y))
      { stars.smaller(); repaint(); }
   ...
   }
}
```

Once the user can place stars anywhere on the window, it is a good idea to raise the maximum number of stars to a much higher number, 100 perhaps. Be sure to change the maximum both in the constructor of *StarManager* and in *addStar*.

In a situation like this (the same constant appears in more than one place), it is strongly recommended that the programmer give the constant a name. Declare a name for the constant (using the keyword *final*) at the beginning of the class. Then use the name for the constant in each place the constant appears. This makes sure that any change to the constant will affect every appropriate place . With this in mind, the *state variables*, *constructor* and *addStar* methods might contain:

```
public class StarManager
   {
   public static final int MAX = 100;

   private Star star[];
   private int howMany;

   public StarManager()
      {
      star = new Star[MAX];
      howMany = 0;
      }

   public void addStar(int h, int v)
      {
      if (howMany < MAX)
         {
         star[howMany] = new Star(h, v);
         howMany++;
         }
      ...
```

The constant *MAX* is declared to be *public* so that the controller can examine it — the controller refers to *MAX* as *StarManager.MAX*. This is safe in the sense that the controller can't tamper with a constant. It also allows the controller to display the maximum number of stars (for example, if the user attempts to exceed the maximum). Warning: if the controller uses *StarManager.MAX*, you must remember to recompile the controller whenever *MAX* is changed in *StarManager* even if no changes have been made to the controller.

A version of the *ToolButton* class is given below.

```
public class ToolButton
   {
   private final int WIDTH = 72, HEIGHT = 20;

   private String name;
   private int left, top;
   private boolean chosen;

   public ToolButton(String theName, int theLeft, int theTop)
      {
      name = theName;
      left = theLeft;
      top = theTop;
      chosen = false;
      }

   public void paint(Graphics g)
      {
      if (chosen)
         g.setColor(Color.black);
      else
         g.setColor(Color.white);
      g.fillRect(left, top, WIDTH, HEIGHT);
      g.setColor(Color.black);
      g.drawRect(left, top, WIDTH, HEIGHT);
      if (chosen)
         g.setColor(Color.white);
      else
         g.setColor(Color.black);
      g.drawString(name, left + 5, top + 14);
      }
```

The class *ToolButton* continues on the next page.

```
public boolean contains(int x, int y)
    {
    if (left <= x && x <= left + WIDTH && top <= y && y <= top + HEIGHT)
        return true;
    return false;
    }

public void choose()
    {
    chosen = true;
    }

public void unchoose()
    {
    chosen = false;
    }

public boolean isChosen()
    {
    return chosen;
    }
}
```

A Color Button Class

The buttons described in the *CButton* class below are small, colored squares many of which can fit on the window. *CButtons* can be used individually, but are especially convenient as elements of an array as they can then be checked in a loop. The *getColor* method is essential to writing a loop that responds to the buttons as it means the controller does not need to know the color of a button, it can ask the button.

```
public class CButton
    {
    private Color theColor;
    private int left, top;

    public CButton(Color aColor, int theLeft, int theTop)
        {
        theColor = aColor;
        left = theLeft;
        top = theTop;
        }

    public void paint(Graphics g)
        {
        g.setColor(theColor);
        g.fillRect(left, top, 30, 30);
        }

    public boolean contains(int x, int y)
        {
        if (left <= x && x <= left + 30 && top <= y && y <= top + 30)
            return true;
        return false;
        }

    public Color getColor()
        {
        return theColor;
        }

    public String toString()
        {
        return "CButton: <" + theColor + "> @ (" + left + ", " + top + ")";
        }
    }
```

Exercises 6.4

1. Modify the stars program above to display a different graphic.
2. Add to either the stars program or the program of exercise 1 to allow different types of changes to the objects.
3. Make several of the color buttons described on the previous page and have them change the color of the selected objects.
4. Make an array of color buttons and use a loop to paint them and a loop to respond to the mouse pressing them. The following lines of Java are intended as hint as to what to write in the *mousePressed* method.

```
for (int b=0; b < COLOR_BUTTONS; b++)
   if (colorButton[b].contains(mouseX, mouseY))
      {
      Color theColor = colorButton[b].getColor();
      stars.setColor(theColor);
      repaint();
      }
```

6.5 Names Without Associated Objects – the Keyword *null*

Variables to which we assign no values

The program in section 6.4 is the first one in which we have created variable names — *star[0]…star[49]* — without immediately assigning values to go with the names. We didn't assign values to these names because we didn't know what values to assign. We intend to use them as the user supplies them values.

Scalar Variables to which we assign no values

When an array of simple (scalar) variables is created, as in *ageList = new int[50]*, the individual variables are given the value 0 (or *false* if the variables are *boolean*).

Object References to which we assign no values

When an array of object references is created, as in *list = new String[50]*, the individual variables are given the special value *null*. This value is used to indicate that a variable declared to refer to an object has no object associated with it. All of the variables *star[0]…star[49]* initially have the value *null*. As the program creates objects and these are assigned to the variables, these variables refer to the newly created objects, losing the value *null*.

It is occasionally appropriate to have a variable that sometimes refers to an object and at other times refers to no object. To have a variable refer to no object, we assign the value *null* to the variable.

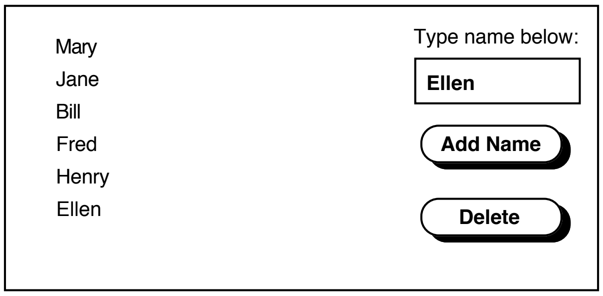
In the following fragment of a game program, sometimes one of the players is 'king', at other times no one is 'king'. The variable *king* intermittently refers to an object.

```
private Player player1, player2, player3;
private Player king;

    player1 = new Player("Henry");
    player2 = new Player("Richard");
    player3 = new Player("Elvis");
    king = null;

    if (player1 should be crowned king)
        king = player1;

    if (no one should be king)
        king = null;

    if (king != null)
        g.drawString(king.getName() + " is king", 100, 20);
    else
        g.drawString("Nobody is king", 100, 20);
```

Runtime errors from attempts to use a *null* object reference

If an object reference is unassigned, and thus has the value *null*, any attempt to use it to call a method will result in a *NullPointerException* reported on the console window. For example, if *list[10]* has had no object assigned to it, them attempting *list[10].paint(g)* will result in a *NullPointerException*. A *NullPointerException* is almost always the result of not initializing an object reference, though sometimes it is the result of an improperly written loop attempting to call a method using a reference that was intentionally not initialized.

6.6 Lists of Names (or …)



This program keeps a list of strings (which can be names or …) and allows adding to the list (at the end) and removing from the list (from the end). We will build this program up through much of the rest of this chapter. Eventually, we will move the names around in the list and change or delete any name we want.

Three classes are used: a controller class, a button class, and a list class. The names are strings (created from the built-in class *String*).

The controller *StringProg* is listed here.

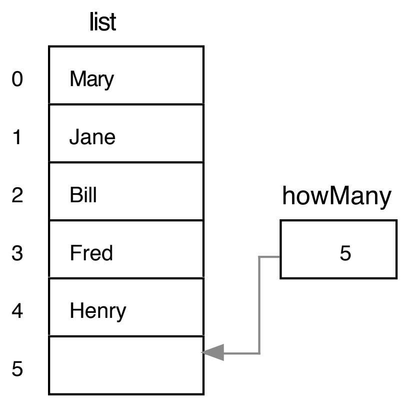
The list class *StringList* is listed after a diagram and a few paragraphs describing commands to add and remove names from the list.

```
public class StringProg extends EventPanel
   {
   private TextField nameField;
   private StringList list;
   private AButton addName, delete;

   public StringProg()
      {
      setLayout(null);
      nameField = new TextField();
      nameField.setLocation(300, 40);
      nameField.setSize(80, 25);
      add(nameField);

      list = new StringList();
      addName = new AButton("Add Name", 300, 100);
      delete = new AButton("Delete", 300, 150);
      }

   public void paint(Graphics g)
      {
      list.paint(g);
      addName.paint(g);
      delete.paint(g);
      g.drawString("Type name below:", 300, 20);
      }

   public void mousePressed(MouseEvent e)
      {
      int x = e.getX(), y = e.getY();
      if (addName.contains(x, y))
         {
         list.addName(nameField.getText());
         repaint();
         }
      else if (delete.contains(x, y))
         {
         list.delete();
         repaint();
         }
      }
   }
```

Since the strings don't contain any information but the names, the *paint* method in *StringList* positions them on the screen. All names are drawn 15 pixels from the left edge of the window. The first name is drawn with its base line 30 pixels from the top of the window. Each subsequent name in the list is drawn 20 pixels lower than the previous one.

The diagram at the right is provided to illustrate the logic behind the code in the method *addName*. Notice that the value of *howMany* accurately reflects the number of names in the list and is also the number of the first unused position in the list. Thus, a name is saved by *list[howMany] = name* and then the value of *howMany* is changed to reflect the additional name.

The method delete decreases the value of *howMany*. This is sufficient since it causes the method *paint* to ignore the last name in the list, and the user won't see it.



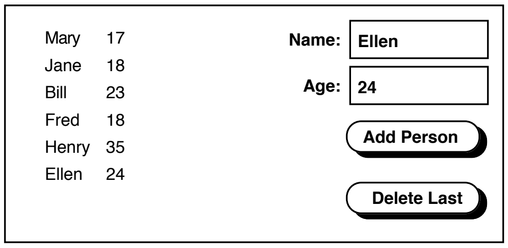
```
public class StringList
   {
   public static final int MAX = 50;

   private String list[];
   private int howMany;

   public StringList()
      {
      list = new String[MAX];
      howMany = 0;
      }

   public void paint(Graphics g)
      {
      for (int count=0; count<howMany; count++)
         g.drawString(list[count], 15, 30 + 20*count);
      }

   public void addName(String name)
      {
      if (howMany < MAX)
         {
         list[howMany] = name;
         howMany++;
         }
      }

   public void delete()
      {
      if (howMany > 0)
         howMany--;
      }
   }
```

The complete program uses, the classes *StringProg*, *StringList* and *AButton*.

Exercises — 6.6

1. Change the program so that it makes a list of numbers instead of names. Change variables of type *String* (for the names) to *int* or *double*. You will need either the method *convertToInt* or the method *convertToDouble* from section 3.8 to convert the text in the *TextField* into a number.
2. Add to the program of exercise 1. In the list class, put a public method that computes of the sum of the numbers and returns the sum. Have the controller display the sum. In the list class, include an accessor method for the number of items so that the controller can avoid displaying the sum when there are no numbers in the list.
3. Have the controller display the average of the numbers. Use the methods from exercise 2.

6.7 A List with both Names and Numbers



Mary	17
Jane	18
Bill	23
Fred	18
Henry	35
Ellen	24

Name: Ellen

Age: 24

This program keeps a list of names and numbers. It allows adding to the list (at the end) and removing from the list (from the end). The easiest way to create this program is to make a class *Person* to hold the data on a person. Then we can use this class in the same way that the *String* class was used for the name in the program of section 6.4.

```
public class Person
   {
   private String name;
   private int age;

   public Person(String theName, int theAge)
      {
      name = theName;
      age = theAge;
      }

   public void setName(String theName)
      {
      name = theName;
      }

   public void setAge(int theAge)
      {
      age = theAge;
      }

   public String getName()
      {
      return name;
      }

   public int getAge()
      {
      return age;
      }

   public String toString()
      {
      return "Person: " + name + " age = " + age;
      }
   }
```

The class *Person* does not contain instance variables for a screen position. The *paint* method in the list class decides where to place the data for each person. Programs that display lists allow the users to rearrange the order of the items. If the items in the list know their positions, many items may have to be altered to effect a simple rearrangement. It is less work to rearrange a list if the elements are ignorant of their positions on the screen.

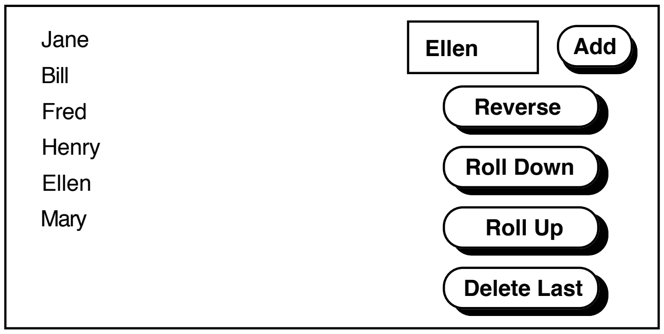
The class *PersonList* is little different from the class *StringList*:

```
public class PersonList
   {
   public static final int MAX = 50;

   private Person list[];
   private int howMany;

   public PersonList()
      {
      list = new Person[MAX];
      howMany = 0;
      }

   public void paint(Graphics g)
      {
      for (int count=0; count<howMany; count++)
         {
         g.drawString(list[count].getName(), 15, 30 + 20*count);
         g.drawString("" + list[count].getAge(), 95, 30 + 20*count);
         }
      }

   public void addPerson(Person aPerson)
      {
      if (howMany < MAX)
         {
         list[howMany] = aPerson;
         howMany++;
         }
      }

   public void delete()
      {
      if (howMany > 0)
         howMany--;
      }
   }

public class PersonProg extends EventPanel
   {
   private PersonList list;
   private AButton addName, delete;
   private TextField nameField, ageField;

   public PersonProg()
      {
      list = new PersonList();
      addName = new AButton("Add Person", 300, 90);
      delete = new AButton("Delete", 300, 120);

      setLayout(null);

      nameField = new TextField();
      nameField.setLocation(300, 20);
      nameField.setSize(80, 25);
      nameField.setText("Joe");
      add(nameField);

      ageField = new TextField();
      ageField.setLocation(300, 50);
      ageField.setSize(80, 25);
      ageField.setText("21");
      add(ageField);
      }
public void paint(Graphics g)
    {
    list.paint(g);
    addName.paint(g);
    delete.paint(g);
    }

public void mousePressed(MouseEvent e)
    {
    int x = e.getX(), y = e.getY();
    if (addName.contains(x, y))
        {
        String name = nameField.getText();
        int age = convertToInt(ageField.getText());
        list.addPerson(new Person(name, age));
        }
    else if (delete.contains(x, y))
        list.delete();
    repaint();
    }

private int convertToInt(String s)          // This is copied from section 3.8
    {
    s = s.trim();
    try
        {
        return (new Integer(s)).intValue();
        }
    catch (NumberFormatException err)
        {
        return -9999;
        }
    }
}
```

Exercises — 6.7

1. Write a program that keeps a list of names, ages and grade point averages. Use a *Person* class with a *String*, an *int* and a *double* as instance variables. In addition to the method convertToInt above, you will need the method *convertToDouble* from section 3.8.
2. Write a program that keeps a list of first and last names and ages. Use three text fields for the first name, the last name, and the age. Use a *Person* class with two *Strings*, and an *int* as instance variables.

6.8 Rearranging Items in a List



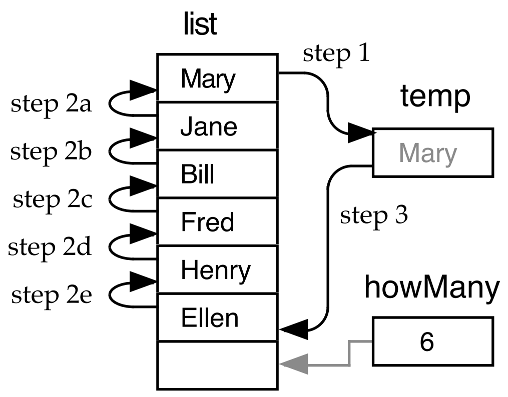
Example 1, *rollUp*

We would like to allow the user to rearrange the names in a list. We will start with moving every name in the list up one place, except for the first one, which we will place at the end of the list.

If the list were as illustrated, we would move Mary aside, move Jane, Bill, Fred, Henry and Ellen up one and then move Mary to the end.

The method below which works in this particular case is illustrated at the right.

```
public void rollUp()
   {
   String temp = list[0];      ← step 1
   list[0] = list[1];          ← step 2a
   list[1] = list[2];          ← step 2b
   list[2] = list[3];          ← step 2c
   list[3] = list[4];          ← step 2d
   list[4] = list[5];          ← step 2e
   list[5] = temp;             ← step 3
   }
```



The first name is copied to a temporary location. The second name is copied to the first position, the third name is copied to the second position…, the last name is copied to the next to last position. Finally, the first name is copied from the temporary location to the last position. It is necessary that no position in the list is copied to until the name there has been moved somewhere else — hence the use of the temporary location to get things started.

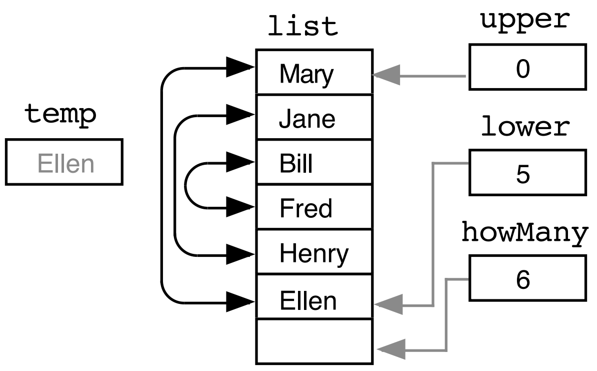
The method above works when there are 6 names in the list. To make the method work with any number of names in the list we use a loop. The group of nearly identical lines above starts by copying to position 0 and ends by copying to position *howMany* – 2. This observation yields the range of values for *n* in the loop below.

```
public void rollUp()
   {
   String temp = list[0];              ← step 1
   for (int n=0; n<=howMany-2; n++)    ← step 2
      list[n] = list[n+1];             ← step 2 continued
   list[howMany - 1] = temp;           ← step 3
   }
```

Example 2, *reverse*

Similarly we would like to reverse the order of the names in the list. To do this we trade Mary and Ellen, then Jane and Henry, and finally Bill and Fred. As above, the method must work for lists of any length. The variable *temp* is used to store Ellen (so she won't get lost when Mary is copied), then to store Henry, and finally to store Fred.

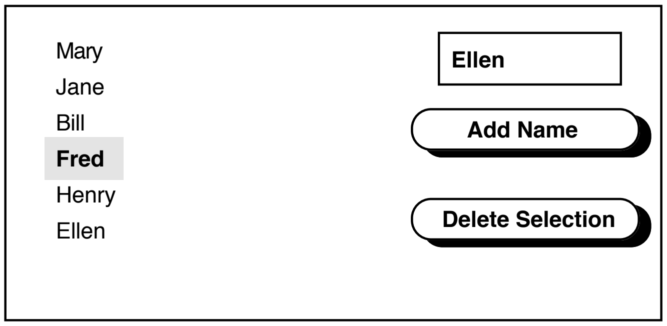
```
public void reverse()
   {
   int upper = 0;
   int lower = howMany - 1;
   while (lower < upper)
      {
      String temp = list[lower];
      list[lower] = list[upper];
      list[upper] = temp;
      lower--; upper++;
      }
   }
```



Exercises — 6.8

1. Use the two preceding methods in a program.
2. Explain the reason for the use of the variable *temp* in the methods *rollUp* and *reverse*.
3. Write the method *rollDown* which moves all of the names down one place, except for the last which is moved to the beginning of the list.

6.9 Choosing Items in a List



The best way to choose a name from the list is to interpret mouse presses in the portion of the screen containing the names as attempts to chose a name. Since the names are evenly spaced, a bit of arithmetic can figure out which name the user wishes to select.

The controller needs to tell the list that the user is trying to select a name and where the mouse was pressed. Adding the two lines just before *repaint()* below, to the method *mousePressed* will suffice. The value 250 ensures that the user is not trying to press a button.

```
public void mousePressed(MouseEvent e)
   . . .
   else if (x < 250)
      list.select(y);
   repaint();
   }
```

Selecting with the Mouse

The selected name will be painted in a distinctive way and the list should keep track of its position. An additional state variable should be used. The new state variable (which we will call *selection*) is changed by the *select* method and used by the *paint* method to render the appropriate name in a distinctive way.

To write the *select* method, we need a formula to turn mouse positions on the screen into positions in the list. The names are listed vertically and the *y* value of the mouse position is used to decide on a position in the list. The names are listed with their baselines 20 pixels apart. If we divide the *y* value by 20, the result will change by 1 each time *y* changes by 20 — this is appropriate. If the list started right at the top of the window, *listPosition = y/20* would be the appropriate formula. The list does not start at the top since the bottom of the first name is at 30 and the names are only 20 pixels apart. The formula *listPosition = (y – 10)/20* works ok. (This formula isn't ideal; it ignores the fact that some letters extend below the baseline.)

```
public class StringList
   {
   public final int MAX = 50;
   public final int FIRST_ROW = 30;
   public final int ROW_HEIGHT = 20;

   private String list[];
   private int howMany;
   private int selection;

   public StringList()
      {
      list = new String[MAX];
      howMany = 0;
      selection = -1;
      }

   public void paint(Graphics g)
      {
      int baseLine = FIRST_ROW;
      for (int count=0; count<howMany; count++)
         {
         if (count == selection)
            {
            g.setColor(Color.lightGray);
            g.fillRect(13, baseLine - ROW_HEIGHT + 3, 80, ROW_HEIGHT);
            }
         g.setColor(Color.black);
         g.drawString(list[count], 15, baseLine);
         baseLine += ROW_HEIGHT;
         }
      }

   public void select(int y)
      {
      selection = (y - FIRST_ROW + ROW_HEIGHT)/ROW_HEIGHT;
      if (selection < 0 || selection >= howMany)
         selection = -1;
      }
   ...
   }
```

Exercises — 6.9

1. Move the selected name to the beginning of the list. Roll down the names that appear before the selected name to make room at the top. Change the selection to 0 so that the name moved to the beginning is still selected.
2. Move the selected name to the end of the list. Roll up the names that appear after the selected one to make room for the selected name at the end. Change the selection so that the name moved to the end is still selected.
3. Have the delete method remove the selected name (instead of the name at the end of the list). Move the selected name to the end (exercise. 2) and then delete it. Change the selection to –1 since the selected name has been removed.
4. Write a method that moves the selected name down one place in the list. Change the selection appropriately.
5. Write a method that moves the selected name up one place in the list. Change the selection appropriately.
6. Add to the *rollUp* method so that the selection moves with the selected name.
7. Add to the *rollDown* method so that the selection moves with the selected name.
8. Add to the *reverse* method so that the selection moves with the selected name.
9. Select a name; then allow the user to change (edit) the name. This is more challenging than the preceding problems. One could do this by putting each name into the *TextField* when it is selected and providing a button to replace the selected name with the one in the *TextField*. Two methods in the list are required, one to obtain the selected name and a second to replace the selected name. The method in the list class that returns the selected name should return *null* when no name is selected. The method to replace the selected name should do nothing when no name is selected.

6.10 Comparing Objects and their Contents

Comparing the values of simple (scalar) variables is familiar:

```
int x, y;...   if (x == y)...  if (x != y)...  if (x > y)...
```

Comparing objects is not so straightforward. The classes *Color* and *String* have special comparison methods. Both have an *equals* comparison method which is used as in these examples:

```
Color c,d;...  if (c.equals(d))...  if (!c.equals(d)) ...
```

Note the use of '!' in the right-hand example. In this case '!' is read *not* and acts to change true into false and vice-versa.

Strings have a special *equals* method: *equalsIgnoreCase*. This method is usually more appropriate than equals.

```
String a,b;... if (a.equalsIgnoreCase(b))...  if (!a.equalsIgnoreCase(b)) ...
```

WARNING: While the symbols == and != can be placed between *Colors* or *Strings*, they often do not give the desired results.

The class *String* has a *compareTo* method which is used to sort alphabetically. Values that are earlier in the alphabet compare as smaller values. You must use the *compareTo* method to sort strings. The symbols <, <=, >, and >= cannot be placed between *Strings*.

```
String a,b;... if (a.compareTo(b) < 0)...  if (a.compareTo(b) <= 0)...
               if (a.compareTo(b) > 0)...  if (a.compareTo(b) >= 0)...
```

The first example above asks if *a* is earlier in the alphabet than *b*. The last asks if *a* is later or the same as *b*.

All but rather old versions of Java also have a *compareToIgnoreCase* method. As with *equals* and *equalsIgnoreCase*, *compareToIgnoreCase* is usually more appropriate than *compareTo*.

6.11 A Method to Select the (Alphabetically) First Name in a List

The strategy is to guess that the first item (item number zero) is the alphabetically first. Then the remaining items are examined, comparing them with the current guess, and changing the guess when appropriate.

```
public void selectAlphaFirst()
   {
   if (howMany > 0)
      {
      int guessForPositionOfFirst = 0;
      for (int examinee=1; examinee<howMany; examinee++)
         if (list[examinee].compareTo(list[guessForPositionOfFirst]) < 0)
                    guessForPositionOfFirst = examinee;
      selection = guessForPositionOfFirst;
      }
   }
```

Exercises — 6.11

1. Put the method *selectAlphaFirst* into a program.
2. Write a method to select the alphabetically last element in the list and use it in a program.

6.12 Sorting a List

Putting one element in the right place

The method outlined in the previous section is a good first step toward a sorting method. It is more convenient, however, to base a sorting procedure on a method for finding the last item in the list (alphabetically) and then placing it at the end of the list.

```
private void putAlphaLastAtEnd(int itemsInList)
    {
    if (itemsInList > 1)
        {
        int guessForPositionOfLast = 0;
        for (int n=1; n<itemsInList; n++)
            if (list[n].compareTo(list[guessForPositionOfLast]) > 0)
                guessForPositionOfLast = n;
        String temp = list[guessForPositionOfLast];
        list[guessForPositionOfLast] = list[itemsInList - 1];
        list[itemsInList - 1] = temp;
        }
    }
```

- The method *putAlphaLastAtEnd* trades two names to put the appropriate item at the end of the list. It doesn't worry about the positions of other names since they are about to be sorted anyway.
- We deliberately used a parameter for the length of the list. The sorting technique we intend to use depends upon applying the method *putAlphaLastAtEnd* to less than the complete list.

Putting every element in the right place

We now write a method that uses the previous one to sort the list. We call *putAlphaLastAtEnd* using the actual value for the number of items in the list — thus putting the appropriate name at the end of the list. We then call *putAlphaLastAtEnd* with one less than the number of items — thus putting the appropriate name at the position immediately preceding the end of the list. We then call *putAlphaLastAtEnd* with two less than the number of items — thus putting the appropriate name at the position immediately preceding …

This particular method of sorting is commonly known as sorting by selection. There are whole books on sorting methods, this example was chosen because:

- How the sort works is easy to understand.
- The technique used is very similar to others in this chapter.
- A selection sort is somewhat more efficient than other elementary sorting techniques (on thoroughly mixed up lists at any rate).

```
public void selectionSort()
    {
    for (int numberUnsorted=howMany; numberUnsorted>1; numberUnsorted--)
        putAlphaLastAtEnd(numberUnsorted);
    }
```

Maintaining a Sorted List

As an alternate to sorting the list, it may be convenient to maintain the list with the elements in the correct order. To do this we determine where the new element should be placed and then insert it in that place. The *addInOrder* method below shows the steps needed to keep the list in sorted order as integer elements are added.

```
public void addInOrder(int number)
    {
    int index = 0;       // index will be the position to place the the number

    if (howMany>0)
        {
        for (int n=0; n<howMany; n++)           // move index past smaller numbers
            if (list[n] < number)
                index++;
        for (int n=howMany; n>index; n--)       // move larger numbers down to make room
            list[n] = list[n-1];
        }

    list[index] = number;
    howMany++;
    }
```

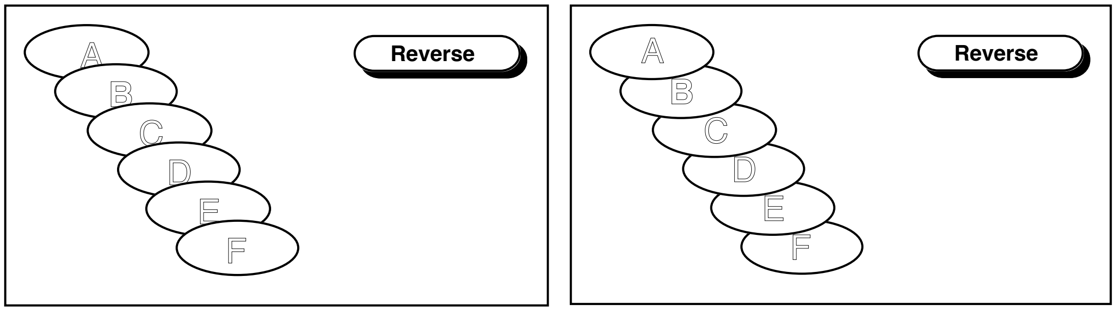
Exercises — 6.12

1. Put the sort methods above into a list program to sort names.
2. Put the sort methods above into a list program to sort integers.
3. Put the *addInOrder* method above into a program to build a list in sorted order.
4. Modify the *addInOrder* method to build a list of names in sorted order and put it into a program.

6.13 Reordering and Removing objects in the ‘Draw’ Program

Reversing the order

Reversing the order of the items in the list reverses their order from front to back. The *reverse* method for a list of names from section 6.8 will work with very little modification.



Limiting Selections to One Object at a Time

Reorderings that are dependent upon which objects are selected are made much easier by allowing only one item to be selected at a time. This following version of the mark method in *StarManager* will select one star. The idea is to deselect all stars and then find the top item that was clicked on by the mouse and select this item.

```
public void mark(int x, int y)
   {
   for (int s = 0; s < howMany; s++)
      star[s].unmark();

   int selection = -1;
   for (int s = 0; s < howMany; s++)
      if (star[s].contains(x, y))
         selection = s;

   if (selection != -1)
      star[selection].mark();
   }
```

Exercises 6.13

1. Move the selected item back one place.

   - Determine which star is selected (modify the second loop in the *mark* method above).

     It is tempting to have the *selection* variable in the *mark* method be a state variable so you wouldn‘t need the loop. However, having the selection as a state variable creates a need to be sure that the selection variable always agrees with which star is actually selected. Once one allows the user to rearrange the order of things in the list, this is a moving target. Generally it is safer to keep information in only one place rather than trying to keep two records of the same information coordinated.

   - If the selection is found to be one or more, trade the objects at positions *selection* and *selection* – 1.

2. Move the selected item forward one place.
   This is very similar to moving the item back one place. Determine which item is selected and then if it is not the last one (at position *howMany – 1*) trade *selection* with *selection* + 1.

3. Move the selected item all the way forward.
   Determine the position of the selected item as though you were going to move it forward just one place. Then use a method similar the method *rollUp* in section 6.8 that uses the value of *selection* instead of zero for the position to move to the end of the list.

4. Move the selected item all the way back.
   Determine the position of the selected item as though you were going to move it forward just one place. The use a method similar the method *rollDown* which is mentioned in the exercises for section 6.8 . Use the value of *selection* instead of *howMany* – 1 for the position to move to the beginning of the list.

5. Delete the selected item.
   Move the selected item all the way forward (to the end of the list), then decrease *howMany* so it disappears.

6. Delete selected items (even though there may be more than one).
   You want to move non-selected items to the beginning of the list covering up some selected ones and then decrease *howMany* by the number of selected items. To move the appropriate non-selected items and determine how many items were selected, write a while loop that starts examining the list at position zero and maintains two numbers: the place you are currently examining and the number of selected items you have encountered. As soon as the number of selected items encountered is greater than zero, copy each non-selected item you examine back in the list by the number of selected items you have encountered.

7. Move the selected items all the way forward (even though there may be more than one).
   Make a second, temporary array of items with *howMany* entries so that it is sure to be large enough to hold all of the selected items. Write a loop like the one for exercise 6 which additionally moves the selected items into the temporary array as it encounters them. After this loop is finished, copy the selected items from the temporary array to the appropriate positions at the end of the original array.

8. Move the selected items all the way back (even though there may be more than one).
   This very similar to exercise 7. Start the first loop at *howMany* – 1 and …

9. Move the selected items one place back (even though there may be more than one)
   You need a temporary variable to hold a non-selected item while you move selected items past it.

```
Initialize the temporary variable to null.
Write a while loop that starts examining the list at position zero.
    When a non-selected item is encountered:
        If the temporary is not null, it is copied to the preceding position.
        The newly encountered item is copied to the temporary.
    When a selected item is encountered:
        If the temporary is not null, the newly encountered item is copied back one position.
After the loop is finished, if the temporary is not null, it is copied to the last position in the list.
```

10. Move the selected items one place forward (even though there may be more than one)
    Similar to exercise 9 except that the loop starts by examining the last position in the list, the copying of selected items is one place forward and the final copy is to position zero.

6.14 Moving objects in the 'Draw' Program

Moving Selected Objects Around

The program in 6.4 can be improved by allowing the user to move items around. This could be accomplished by sending coordinates to the items. We need methods to move an item in both *Star* and *StarManager*.

in class *Star*:

```
public void setLocation(int h, int v)
   {
   if (marked)
      {
      x = h; y = v;
      }
   }
```

in class *StarManager*:

```
public void setLocation(int h, int v)
   {
   for (int s = 0; s < howMany; s++)
      star[s].setLocation(h, v);
   }
```

Using the Methods *mouseDragged* and *mouseReleased*

When moving items, it is also appropriate to use the event handler *mouseDragged*. Although we don't need it here, the method *mouseReleased* is also required in some similar situations. These are syntactically just like *mousePressed*.

- *mouseDragged* is called when the mouse is moved with the button pressed.
- *mouseReleased* is called when the button is released.

```
public void mouseDragged(MouseEvent e)
   {
   int x = e.getX(), y = e.getY();
   if (x < GRAPHICS_BOUNDARY)
      {
      stars.setLocation(x, y);
      repaint();
      }
   }
```

Avoiding surprises while dragging

The *mouseDragged* method above will allow you to drag a single star around. It will work as expected provided you never drag the mouse over to the drawing area when you have pressed the button down in the button area. If you do press a button and then (holding the button down) move the mouse to the drawing area, the selected star will suddenly start to follow the mouse. To prevent this behavior, have the controller keep track as to whether the mouse button came down over the drawing area or the button area. Then be sure that the controller only requests that a star be moved when the mouse button came down over the drawing area.

Use a *boolean* state variable (*clickedOnDrawingArea*) as follows:

```
public class StarProg extends EventPanel
   {
   private static final int GRAPHICS_BOUNDARY = 300;

   private Stars stars;
   private boolean clickedOnDrawingArea;
   private boolean creating;
   private ToolButton ...
   private AButton ...

   public StarProg()
      {
      stars = new Stars();
      clickedOnDrawingArea = false;
      creating = true;
      ...
      }

   public void mousePressed(MouseEvent e)
      {
      int x = e.getX(), y = e.getY();
      clickedOnDrawingArea = false;
      if (x < GRAPHICS_BOUNDARY)
         {
         if (creating)
            ...
         else
            {
            stars.select(x, y);
            clickedOnDrawingArea = true;
            ...
            }
         }
      else if (bigger.contains(x, y))
         ...

   public void mouseDragged(MouseEvent e)
      {
      int x = e.getX(), y = e.getY();
      if (clickedOnDrawingArea && x < GRAPHICS_BOUNDARY)
         {
         stars.setLocation(x, y);
         repaint();
         }
      }

   public void mouseReleased(MouseEvent e)
      {
      clickedOnDrawingArea = false; // This method is not really required for this program
      }
   }
```

It would be really nice to have several types of objects in your program. The right way to do this is to keep a single list containing objects of various types. To do this properly you must use *interfaces* (or the more complex *subclasses*), features of the Java language that are discussed in chapter 8, in part 2 of the book.