

Animating Algorithms

```
public abstract class AlgorithmAnimator
    extends DBAnimationApplet {
    // the hook method
    abstract protected void algorithm();
    // the template method
    public void run() {
        algorithm();
    }
    final protected void pause() {
        if (Thread.currentThread() ==
            animationThread) {
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {}
            repaint();
        }
    }
}
```

Animating Sorting Algorithms

- ◆ A screen shot
- ◆ straightforward implementation
- ◆ single class
- ◆ support different algorithms

Animating Sorting Algorithms I

```
import java.awt.*;

public class Sort extends AlgorithmAnimator {

    <Method scramble()>
    <Method paintFrame()>
    <Method bubbleSort()>
    <Method quickSort()>
    <Method initAnimator()>
    <Method algorithm()>
    <Method swap()>
    // the array to be sorted
    protected int    arr[];
    // the name of the algorithm to be animated
    protected String algName;
}
```

Scramble

```
protected void scramble() {
    arr = new int[getSize().height / 2];
    for (int i = arr.length; --i >= 0; ) {
        arr[i] = i;
    }
    for (int i = arr.length; --i >= 0; ) {
        int j = (int)(i * Math.random());
        swap(arr, i, j);
    }
}

private void swap(int a[], int i, int j) {
    int T;
    T = a[i]; a[i] = a[j]; a[j] = T;
}
```

Visualize the Array

```
protected void paintFrame(Graphics g) {
    Dimension d = getSize();
    g.setColor(Color.white);
    g.fillRect(0, 0, d.width, d.height);
    g.setColor(Color.black);
    int y = d.height - 1;
    double f = d.width / (double) arr.length;
    for (int i = arr.length; --i >= 0; y -= 2) {
        g.drawLine(0, y, (int)(arr[i] * f), y);
    }
}
```

Sorting Algorithms

```
protected void bubbleSort(int a[]) {
    for (int i = a.length; --i >= 0; )
        for (int j = 0; j < i; j++) {
            if (a[j] > a[j+1]) {
                swap(a, j, j + 1);
            }
        }
    pause();
}

protected void quickSort(int a[]) { ... }
```

Let The Action Begin

```
protected void initAnimator() {
    algName = "BubbleSort";
    String at = getParameter("alg");
    if (at != null) {
        algName = at;
    }
    setDelay(20);
    scramble();
}

protected void algorithm() {
    if ("BubbleSort".equals(algName)) {
        bubbleSort(arr);
    } else if ("QuickSort".equals(algName)) {
        quickSort(arr, 0, arr.length - 1);
    } else {
        bubbleSort(arr);
    }
}
```

Separate the Algorithms



- Make different algorithms interchangeable components.
- Adding and changing algorithms will have little impact on the animation mechanism.

The Abstract Algorithm

```
abstract public class SortAlgorithm {
    abstract void sort(int a[]);
    private AlgorithmAnimator animator;
    protected SortAlgorithm(AlgorithmAnimator
                            animator) {
        this.animator = animator;
    }
    protected void pause() {
        if (animator != null)
            animator.pause();
    }
    protected static void swap(int a[],
                               int i, int j) {
        int T;
        T = a[i]; a[i] = a[j]; a[j] = T;
    }
}
```

A Concrete Algorithm

```
public class BubbleSortAlgorithm extends
SortAlgorithm {
    public void sort(int a[]) {
        for (int i = a.length; --i >= 0; ) {
            for (int j = 0; j < i; j++) {
                if (a[j] > a[j+1]) {
                    swap(a, j, j+1);
                }
                pause();
            }
        }
    }
    public BubbleSortAlgorithm(AlgorithmAnimator
                               animator) {
        super(animator);
    }
}
```

Using A Factory

The Abstract Algorithm Factory

```
public interface AlgorithmFactory {
    public SortAlgorithm
        makeSortAlgorithm(String algName);
}
```

A Concrete Algorithm Factory

```
public class StaticAlgoFactory
    implements AlgorithmFactory {
    public StaticAlgoFactory(AlgorithmAnimator
        animator) {
        this.animator = animator;
    }
    public SortAlgorithm makeSortAlgorithm(
        String algName) {
        if ("BubbleSort".equals(algName)) {
            return new BubbleSortAlgorithm(animator);
        } else if ("QuickSort".equals(algName)) {
            return new QuickSortAlgorithm(animator);
        } else {
            return new BubbleSortAlgorithm(animator);
        }
    }
    protected AlgorithmAnimator animator;
}
```

Sorting Algorithm Animation II

```
public class Sort2 extends AlgorithmAnimator {
    protected SortAlgorithm theAlgorithm;
    protected AlgorithmFactory algorithmFactory;

    protected void initAnimator() {
        algName = "BubbleSort";
        String at = getParameter("alg");
        if (at != null) {
            algName = at;
        }
        algorithmFactory =
            new StaticAlgoFactory(this);
        theAlgorithm =
            algorithmFactory.makeSortAlgorithm(algName);
        setDelay(20);
        scramble();
    }
}
```

Sorting Algorithm Animation II (cont'd)

(class Sort2 continued.)

```
protected void algorithm() {
    if (theAlgorithm != null)
        theAlgorithm.sort(arr);
}

protected void scramble() {
    <same as scramble() in Sort>
}

protected void paintFrame(Graphics g) {
    <same as paintFrame() in Sort>
}

protected int    arr[];
protected String algName;
}
```

Separate the Display Component

SortDisplay Design Option 1

- ♦ Very small interface

```
public interface SortDisplay {
    public void display(int a[], Graphics g,
                       Dimension d);
}
```

- ♦ Minor changes to the main class.

The `paintFrame()` method becomes:

```
public void paintFrame(Graphics g) {
    theDisplay.display(arr, g, getSize());
}
```

SortDisplay Design Option 1 (cont'd)

Problems:

- ♦ Strong *coupling* between the main class and the implementations of `SortDisplay`.
- ♦ Limited flexibility.
- ♦ What about drawing lines vertically?
- ♦ What about drawing lines of 3 pixels wide?

SortDisplay Design Option 2

- ♦ Include `scramble()` in the `SortDisplay`

```
public interface SortDisplay {
    public void scramble(int a[], Dimension d);
    public void display(int a[], Graphics g,
                       Dimension d);
}
```

- ♦ The `scramble()` method in the main class becomes:

```
void scramble() {
    theDisplay.scramble(arr, getSize());
}
```

SortDisplay Design Option 2 (cont'd)

- ♦ The coupling between the main class and the implementations of `SortDisplay` is reduced.
- ♦ The methods of `SortDisplay` are not very *cohesive*.
- ♦ Scrambling is quite independent from displaying.
- ♦ The only interdependence between the two methods are --- the array size.

SortDisplay Design Option 3

- ◆ The SortDisplay interface

```
public interface SortDisplay {
    public int getArraySize(Dimension d);
    public void display(int a[], Graphics g,
        Dimension d);
}
```

- ◆ The scramble() method in the main class becomes:

```
void scramble() {
    int n = theDisplay.getArraySize(getSize());
    arr = new int[n];
    // scramble the numbers in arr[0..(n-1)]
}
```

Animating Sorting Algorithms III

```
public class Sort3 extends Sort2 {

    protected SortDisplay theDisplay;
    protected SortDisplayFactory displayFactory;

    protected void initAnimator() {
        String att = getParameter("dis");
        displayFactory =
            new StaticSortDisplayFactory();
        theDisplay =
            displayFactory.makeSortDisplay(att);
        super.initAnimator();
    }
}
```

Animating Sorting Algorithms III (cont'd)

(class Sort3 continued)

```
protected void scramble() {
    int n = theDisplay.getArraySize(getSize());
    arr = new int[n];
    for (int i = arr.length; --i >= 0; )
        arr[i] = i;
    for (int i = arr.length; --i >= 0; ) {
        int j = (int)(i * Math.random());
        SortAlgorithm.swap(arr, i, j);
    }
}

protected void paintFrame(Graphics g) {
    theDisplay.display(arr, g, getSize());
}
}
```



Display Strategies