

Abstract Classes

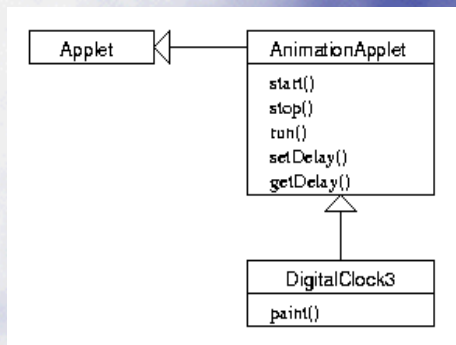
- An *abstract class* is a class with partial implementation.
- It implements behaviors that are common to all subclasses, but defers to the subclasses to implement others (*abstract methods*).
- Abstract methods must be implemented in each non-abstract subclasses.
- Instance of an abstract class is not allowed.

Interfaces

- Interfaces are classes with no implementation.
- Interfaces represent *pure design*.
- Abstract classes represent mixed design and implementation.
- An interface consists of only *abstract methods* and *constants*, i.e., static and final.
- All methods and constants are public.
- No static methods.
- No instance allowed.
- *Multiple inheritance* for interfaces is allowed.
- An interface can only extend interface, not classes.
- There is no single root for interfaces.

A Generic Animator

Abstract classes can be used to design *generic* components.



A Generic Animator (cont'd)

```
public abstract class AnimationApplet
    extends java.applet.Applet
    implements java.lang.Runnable {
    protected Thread animationThread;
    protected int delay = 100;
    final public void setDelay(int delay) {
        this.delay = delay;
    }
    final public int getDelay() {
        return delay;
    }
}
```

A Generic Animator (cont'd)

```
public void start() {
    animationThread = new Thread(this);
    animationThread.start();
}
public void stop() {
    animationThread = null;
}
public void run() {
    while (Thread.currentThread() ==
        animationThread) {
        try {
            Thread.currentThread().sleep(delay);
        } catch (InterruptedException e){}
        repaint();
    }
}
```

Digital Clock Using the Generic Animator

```
import java.awt.*;
import java.util.Calendar;

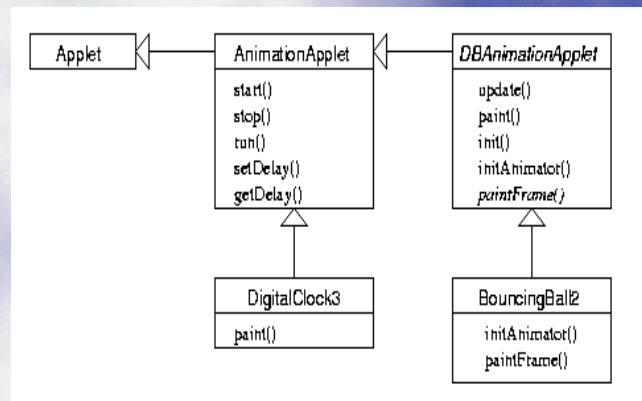
public class DigitalClock3
    extends AnimationApplet {
    public DigitalClock3() {
        setDelay(1000);
    }
    public void paint(Graphics g) {
        // identical to the paint() method in DigitalClock
    }
    protected Font font =
        new Font("Monospaced", Font.BOLD, 48);
    protected Color color = Color.green;
}
```

A Double-Buffered Generic Animator

Key design issues

- Accommodate various sizes of the view area.
- Allow the subclasses to decide whether to use double-buffering or not.
- Factorize common code segments that contain variable parts.

A Double-Buffered Generic Animator (cont'd)



A Double-Buffered Generic Animator (cont'd)

```
public abstract class DBAnimationApplet
    extends AnimationApplet {
    protected DBAnimationApplet(boolean
        doubleBuffered) {
        this.doubleBuffered = doubleBuffered;
    }
    protected DBAnimationApplet() {
        this.doubleBuffered = true;
    }

    // other methods

    protected boolean doubleBuffered;
    protected Image im;
    protected Graphics offscreen;
    protected Dimension d;
}
```

A Double-Buffered Generic Animator (cont'd)

```
protected void initAnimator() {}

final public void init() {
    d = getSize();
    initAnimator();
}

abstract protected void paintFrame(Graphics g);

final public void paint(Graphics g) {
    paintFrame(g);
}
```

A Double-Buffered Generic Animator (cont'd)

```
final public void update(Graphics g) {
    if (doubleBuffered) {
        if (im == null) {
            im = createImage(d.width, d.height);
            offscreen = im.getGraphics();
        }
        paintFrame(offscreen);
        g.drawImage(im, 0, 0, this);
    } else {
        super.update(g);
    }
}
```

Bouncing Ball Using the Generic Animator

```
public class BouncingBall2
    extends DBAnimationApplet {
    public BouncingBall2() {
        super(true); // double buffering
    }

    // methods

    protected int x, y;
    protected int dx = -2, dy = -4;
    protected int radius = 20;
    protected Color color = Color.green;
}
```

Bouncing Ball Using the Generic Animator (cont'd)

```
protected void initAnimator() {
    String att = getParameter("delay");
    if (att != null)
        setDelay(Integer.parseInt(att));
    x = d.width * 2 / 3 ;
    y = d.height - radius;
}
```

Bouncing Ball Using the Generic Animator (cont'd)

```
protected void paintFrame(Graphics g) {
    g.setColor(Color.white);
    g.fillRect(0,0,d.width,d.height);
    if (x < radius || x > d.width - radius) {
        dx = -dx;
    }
    if (y < radius || y > d.height - radius) {
        dy = -dy;
    }
    x += dx; y += dy;
    g.setColor(color);
    g.fillOval(x - radius, y - radius,
              radius * 2, radius * 2);
}
```

Design Pattern: Template Method

Category

Behavioral design pattern.

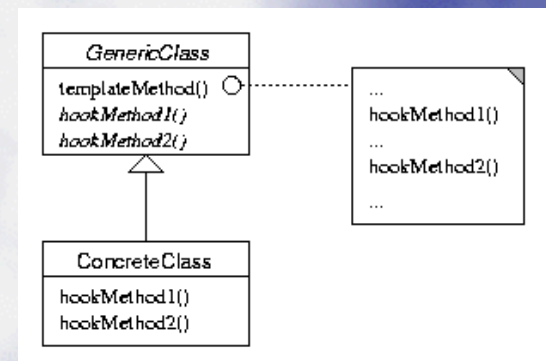
Intent

Define the skeleton of an algorithm in a method, deferring some steps to subclasses, thus allowing the subclasses to redefine certain steps of the algorithm.

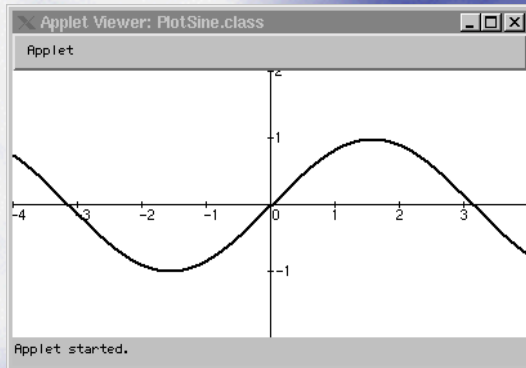
Applicability

- to implement the invariant parts of an algorithm once and leave it up to the subclasses to implement the behavior that can vary.
- to factorize and localize the common behavior among subclasses to avoid code duplication.

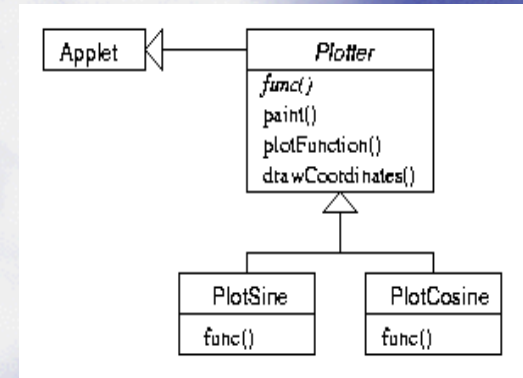
Template Method



A Generic Function Plotter



A Generic Function Plotter: The Design



A Generic Function Plotter (cont'd)

```
public abstract class Plotter extends Applet {
    //hook method
    public abstract double func(double x);

    public void paint(Graphics g) {
        drawCoordinates(g);
        plotFunction(g);
    }

    public void init() {
        <get the parameters and initialize the fields>
    }

    protected void drawCoordinates(Graphics g) {
        <draw the X and Y axis and the tick marks>
    }
}
```

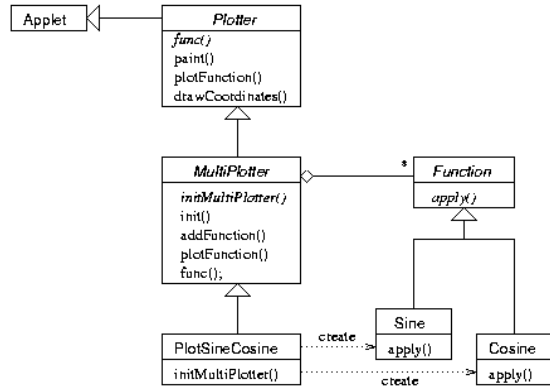
A Generic Function Plotter (cont'd)

(class Plotter continued)

```
protected void plotFunction(Graphics g) {
    for (int px = 0; px < dim.width; px++) {
        try {
            double x = (double)(px - xorigin) /
                (double)xratio;
            double y = func(x);
            int py = yorigin - (int) (y * yratio);
            g.fillOval(px - 1, py - 1, 3, 3);
        } catch (Exception e) {}
    }
}

<fields>
}
```

Generic Multiple Function Plotter



Interface Function

```

interface Function {
    double apply(double x);
}

public class Sine implements Function {
    public double apply(double x) {
        return Math.sin(x);
    }
}

public class Cosine implements Function {
    public double apply(double x) {
        return Math.cos(x);
    }
}

```

Multiple Function Plotter Class

```

public abstract class MultiPlotter extends Plotter {

    protected static int MAX_FUNCTIONS = 5;
    protected int numOfFunctions = 0;
    protected Function functions[] = new
        Function[MAX_FUNCTIONS];
    protected Color colors[] = new
        Color[MAX_FUNCTIONS];

    abstract public void initMultiPlotter();

    public void init() {
        super.init();
        initMultiPlotter();
    }
}

```

Multiple Function Plotter Class (cont'd)

```

    final public void addFunction(Function f,
        Color c) {
        if (numOfFunctions < MAX_FUNCTIONS &&
            f != null) {
            functions[numOfFunctions] = f;
            colors[numOfFunctions++] = c;
        }
    }
}

```

Multiple Function Plotter Class (cont'd)

```
protected void plotFunction(Graphics g) {
    for (int i = 0; i < numOfFunctions; i++)
        if (functions[i] != null) {
            Color c = colors[i];
            if (c != null)
                g.setColor(c);
            else
                g.setColor(Color.black);
            for (int px = 0; px < d.width; px++)
                try {
                    double x = (double) (px - xorigin) /
                        (double) xratio;
                    double y = functions[i].apply(x);
                    int py = yorigin - (int) (y * yratio);
                    g.fillOval(px - 1, py - 1, 3, 3);
                } catch (Exception e) {}
        }
}
```

A Concrete Multiple Function Plotter

```
public class PlotSineCosine
    extends MultiPlotter {

    public void initMultiPlotter() {
        addFunction(new Sine(), Color.green);
        addFunction(new Cosine(), Color.blue);
    }
}
```

Design Pattern: Strategy

Category

Behavioral design pattern.

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable.

Applicability

- many related classes differ only in their behavior.
- you need different variants of an algorithm.
- an algorithm uses data that clients shouldn't know about.
- a class defines many behaviors, and these appear as multiple conditional statements in its methods.

Strategy

