



JAVAPOLIS

11 - 15 DECEMBER ■ ANTWERP ■ BELGIUM





Filthy Rich Clients

Enriching the User Experience

Romain Guy
Student, Author

Chet Haase
Sun Microsystems

www.javapolis.com



Speaker's Qualifications

- 👤 **Romain Guy** is a student, author, and has way too many things going on these days. Read his blog at <http://jroller.com/page/gfx>
- 👤 **Chet Haase** is a Client Architect in the Java SE group at Sun Microsystems. Chet focuses on issues of graphics, GUIs, and performance. Read his blog at <http://weblogs.java.net/blog/chet>



The Big Question

What do we mean by
“filthy rich clients”?

The Big Question

What do we mean by
“filthy rich clients”?



Rich Bair, SwingLabs Lead

The Big Question

What do we mean by
“filthy rich clients”?



Rich Bair

The Big Question

What do we mean by “filthy rich clients”?

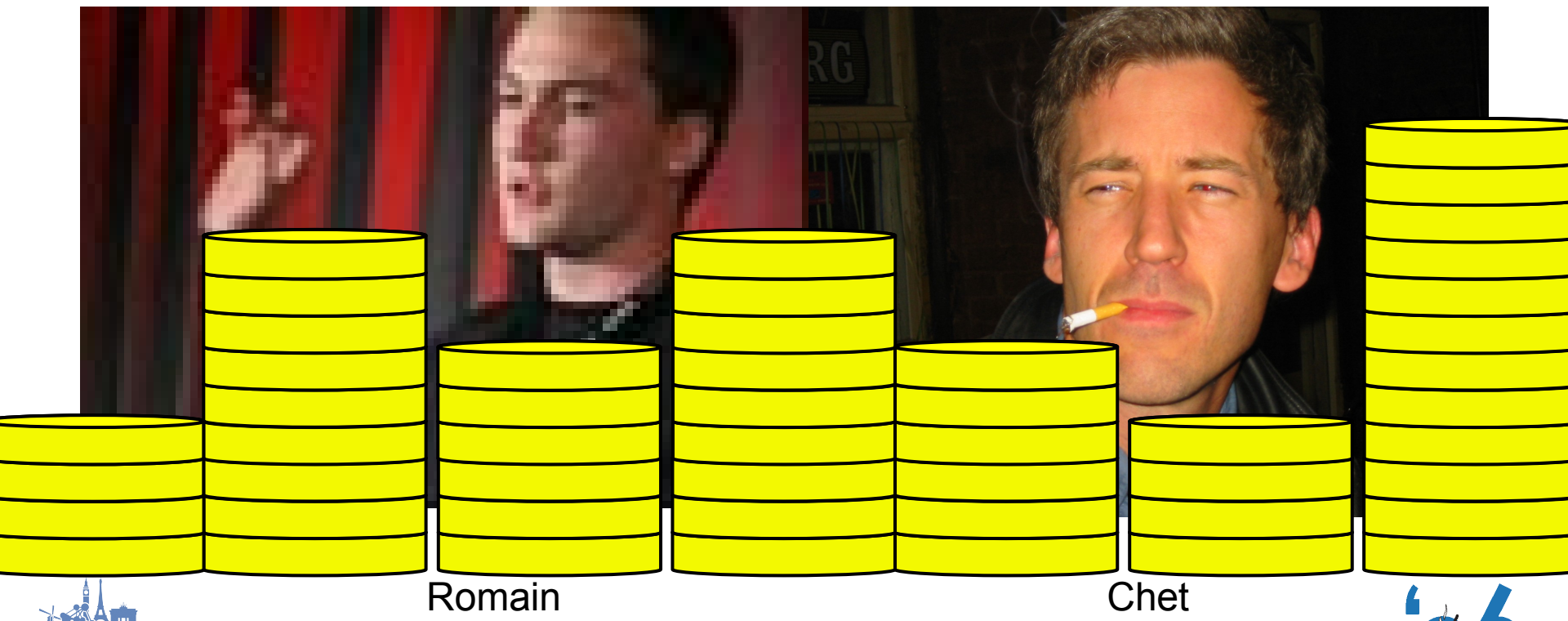


A Swing, .NET, and
Flex developer walk
into a bar....

Rich Bair, SwingLabs Lead

The Big Question

What do we mean by
“filthy rich clients”?



Romain

Chet



No, no, no






Filthy Rich Clients are applications so *graphically rich* that they *ooze cool*, they *suck the user in* from the outset and hang onto them with a *death grip of excitement*. They force the user tell their friends about the applications.

Presentation Goal






~~Hype the Book~~

Learn about Animation, 2D, and Swing
Techniques for writing applications your
users will love

Agenda

-  Graphics
-  Animation Fundamentals
-  Animated Transitions
-  3D
-  Filthy Rich Effects

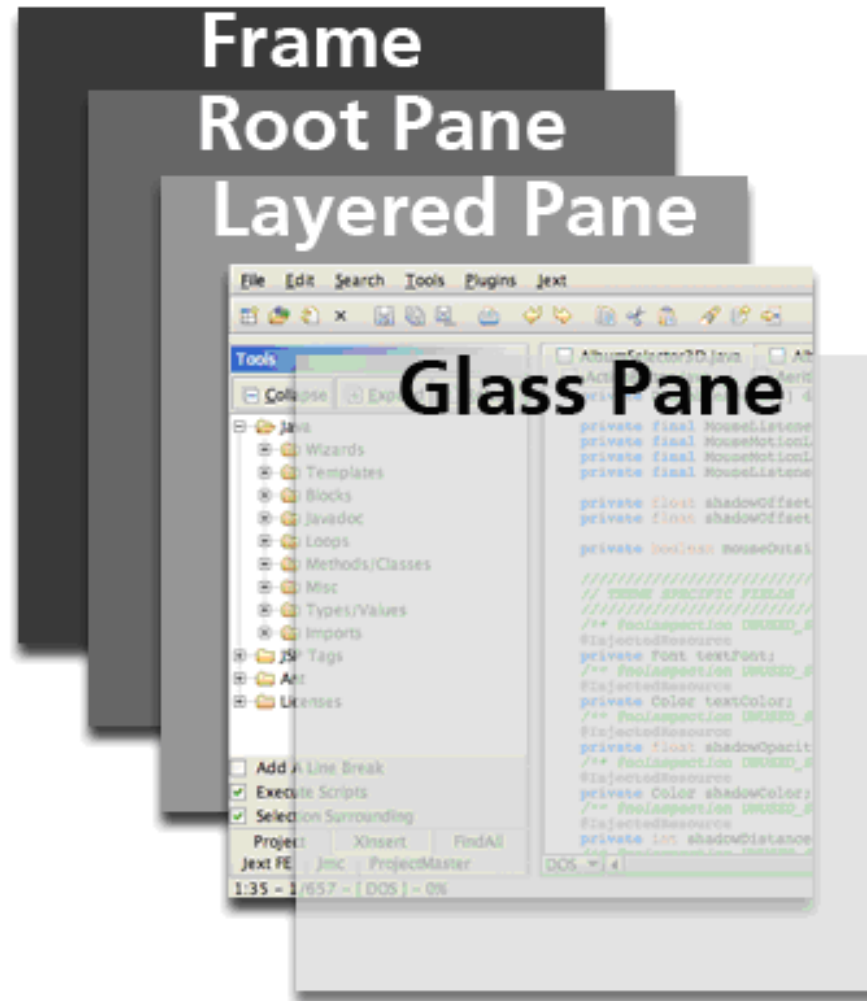
Agenda

-  **Graphics**
-  Animation Fundamentals
-  Animated Transitions
-  3D
-  Filthy Rich Effects

Graphics

- ☕ GlassPane
- ☕ Gradients
- ☕ Alpha Composite

GlassPane: Not just a pane in the glass



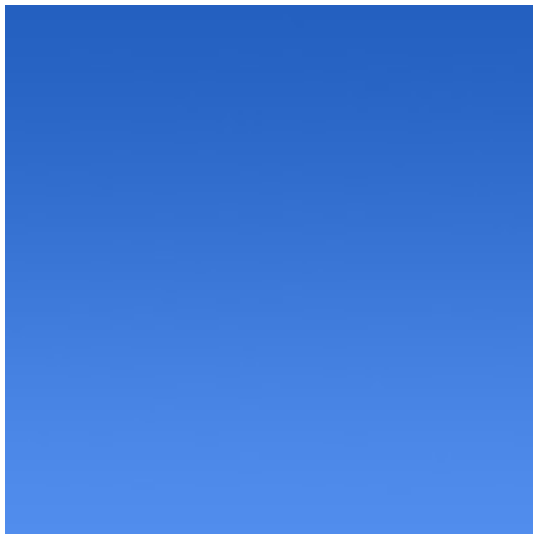
GlassPane: Example

```
class MyGlassPane extends JComponent {  
    @Override  
    protected void paintComponent(Graphics g) {  
        // painting jobs  
    }  
}
```

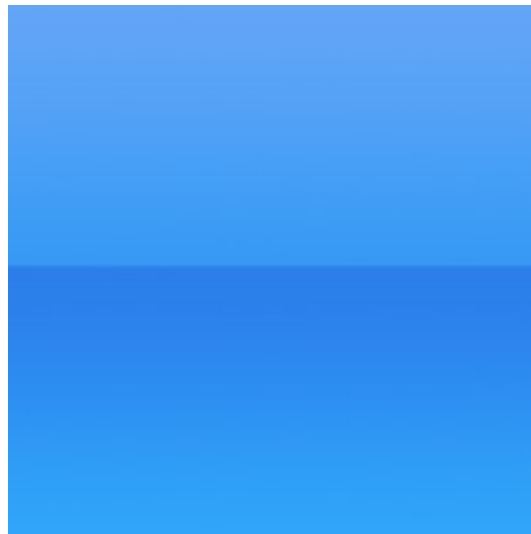
```
JFrame f = new JFrame();  
f.setGlassPane(new MyGlassPane());  
  
f.getGlassPane().setVisible(true);
```

Gradients

J2SE 1.2+



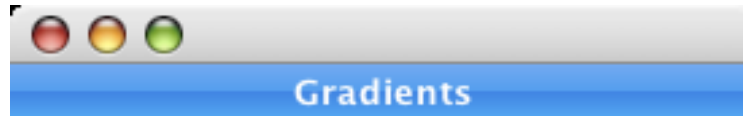
Java SE 6



Java SE 6



Gradients



Gradients: Example

```
GradientPaint p;
```

```
p = new GradientPaint(0, 0, new Color(0x63a5f7),  
    0, 10, new Color(0x3799f4));
```

```
g2.setPaint(p);
```

```
g2.fillRect(0, 0, getWidth(), 10);
```

```
p = new GradientPaint(0, 10, new Color(0x2d7eeb),  
    0, 20, new Color(0x30a5f9));
```

```
g2.setPaint(p);
```

```
g2.fillRect(0, 10, getWidth(), 10);
```



Gradients: Example

```
LinearGradientPaint p;
```

```
p = new LinearGradientPaint(0.0f, 0.0f, 0.0f, 20.0f,  
    new float[] { 0.0f, 0.499f, 0.50f, 1.0f },  
    new Color[] { new Color(0x63a5f7),  
                  new Color(0x3799f4),  
                  new Color(0x2d7eeb),  
                  new Color(0x30a5f9) } );  
  
g2.setPaint(p);  
g2.fillRect(0, 0, getWidth(), 20);
```



AlphaComposite

 12 Rules

 Learn Just 2

⇒ AlphaComposite.SRC_OVER

⇒ AlphaComposite.DST_IN



AlphaComposite: SrcOver



Alpha Composite: SrcOver

```
c = AlphaComposite.getInstance(  
    AlphaComposite.SRC_OVER, 0.5f);  
g2.setComposite(c);  
g2.drawImage(picture, x, y, null);
```



AlphaComposite: DstIn



DstIn: Example

```
Image subject = ...;
BufferedImage alphaMask = createGradientMask(
    subjectWidth, subjectHeight);
BufferedImage buffer = createReflection(
    subjectWidth, subjectHeight);

Graphics2D g2 = buffer.createGraphics();
g2.setComposite(AlphaComposite.DstIn);
g2.drawImage(alphaMask, null, 0, subjectHeight);
g2.dispose();
```





DEMO

Graphics

www.javapolis.com



Agenda

- ☕ Graphics
- ☕ **Animation Fundamentals**
- ☕ Animated Transitions
- ☕ 3D
- ☕ Filthy Rich Effects

Animation: It's About Time

- ☕ Animations should be based on time
 - ➔ Not successive steps
 - ➔ Accounts for variable machine performance
 - ➔ Works the same across all environments
- ☕ Determine appropriate speed for animation
- ☕ For every animation frame
 - ➔ Calculate time delta from last time
 - ➔ Calculate change to object from time & speed
 - ➔ Render object with appropriate change

Timers: Wakeup Calls

- ☕ Timers are utilities for knowing when to render the next frame
- ☕ Create Timer with “resolution”
 - ➔ determines frame rate (frames per second)
- ☕ Timer will call your code at this frame rate
 - ➔ assuming your frame rate is achievable
- ☕ Timers in core JDK
 - ➔ **java.util.Timer**: for general usage
 - ➔ **javax.swing.Timer**: GUI specific



javax.swing.Timer

```
// Create and start timer
startTime = System.currentTimeMillis();
timer = new Timer(msBetweenCallbacks, listener);
timer.start();

// timer callbacks in actionPerformed() method
public void actionPerformed(ActionEvent ae) {
    // Get elapsed time
    long currentTime = ae.getWhen();
    long elapsedTime = currentTime - startTime;
    // Now do whatever you want with this information
    // ...
}
```



Beyond the Built-in Timers

- ☕ Key functionality lacking in core timers for typical animation requirements
 - ➔ duration
 - nothing lasts forever (besides acne)
 - ➔ elapsed time:
 - have the system tell you how much animation has elapsed
 - ➔ repeat:
 - repeating, reversing animations common
 - ➔ advanced:
 - many other requirements for typical animations, like non-linear interpolation

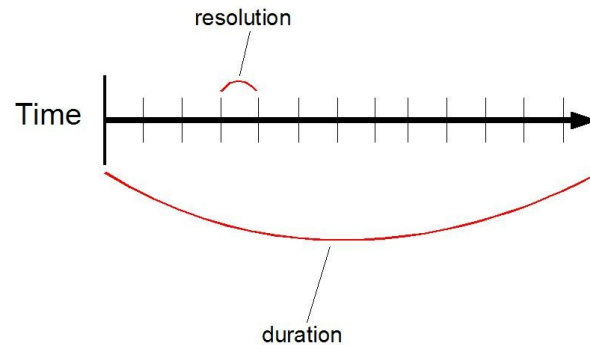


Timing Framework

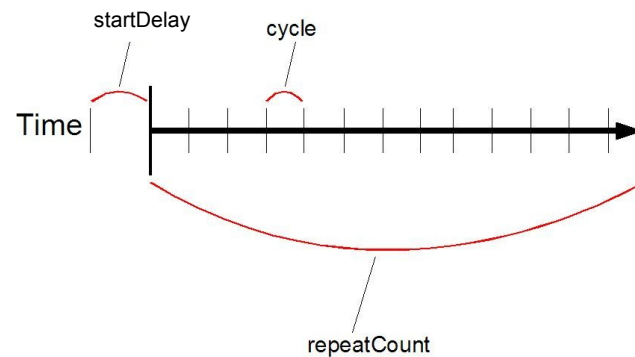
- ☕ <http://timingframework.dev.java.net>
 - ➔ Project in development for the last two years
- ☕ Core concepts:
 - ➔ Cycle: basic animation loop
 - duration, resolution
 - ➔ Envelope: contains one or more Cycles
 - number of cycles, start delay, repeat behavior, end behavior
 - ➔ TimingTarget: callback target
 - begin, end, repeat, timingEvent(fraction)
 - ➔ Animator:
 - Cycle and envelope properties, one or more TimingTargets

Timing Framework: The Basics

Cycle



Envelope



TimingFramework: The Basics

```
class MyTarget implements TimingTarget {
    public void begin() {...}
    public void end() {...}
    public void repeat() {...}
    public void timingEvent(float fraction) {...}
}
TimingTarget target = new MyTarget();

// animate once for 5 seconds, then stop
Animator singleRun = new Animator(5000, target);
singleRun.start();

// animate for 5 cycles of 2 seconds, reversing each time
Animator oscillator = new Animator(2000, 5,
                                   RepeatBehavior.REVERSE,
                                   target);

oscillator.start();
```





DEMO

BasicRace

www.javapolis.com



BasicRace: The Code

```
public class BasicRace extends TimingTargetAdapter
    implements ActionListener {
    // Starts/stops timer based on Go/Stop action events
    public void actionPerformed(ActionEvent ae) {
        if (ae.getActionCommand().equals("Go")) {
            timer = new Animator(RACE_TIME, this);
            timer.start();
        } else if (ae.getActionCommand().equals("Stop")) {
            timer.stop();
        }
    }
    // Callback: Linearly interpolate car position according
    // to fraction of animation elapsed thus far
    public void timingEvent(float fraction) {
        current.x = (int)(start.x + (end.x-start.x) * fraction);
        current.y = (int)(start.y + (end.y-start.y) * fraction);
        track.setCarPosition(current);
    }
}
```



TimingFramework: Advanced

- ☕ Non-Linear Interpolation
 - ➔ More realistic motion
- ☕ Property Setters
 - ➔ TimingTargets that animate JavaBean properties

Non-Linear Interpolation

- ☕ When was the last time you saw someone gliding smoothly along the street?
 - ➔ Not counting bad movie camera effects
- ☕ We live in a non-linear world
 - ➔ Gravity, acceleration, deceleration, friction
 - ... as well as tripping, stumbling, falling, crashing, settling
- ☕ ... so our eyes expect to see non-linear movement
- ☕ Linear movement results in bad animations
 - ➔ Looks unnatural
 - ➔ Emphasizes rendering artifacts
 - Easy to track mistakes and hiccups when we are tracking linear movement



Interpolation: Acceleration/Deceleration

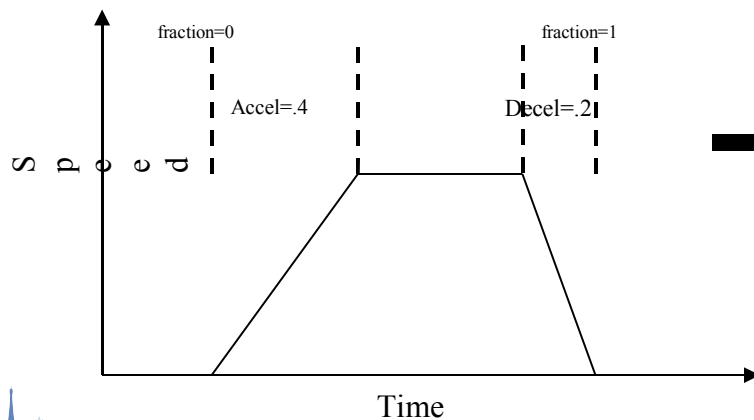
- Simplest approach for simple situations

```
Animator.setAcceleration(float);
```

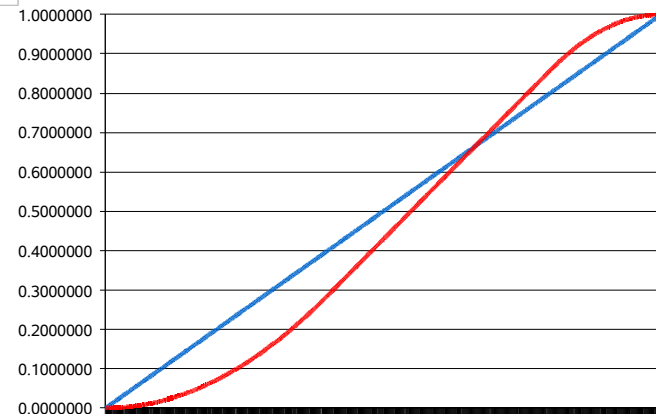
```
Animator.setDeceleration(float);
```

- Fraction of cycle speeding up, slowing down

Speed vs. Time



Interpolated t vs. t



Interpolation: Interpolator interface

- ☕ Flexible way to define arbitrary interpolation

```
public interface Interpolator {  
    public float interpolate(float fraction);  
}
```

- ☕ Set on Animator

```
Animator.setInterpolator(Interpolator)
```

- ☕ Pre-defined implementations:

- ➔ LinearInterpolator
- ➔ DiscreteInterpolator
- ➔ SplineInterpolator

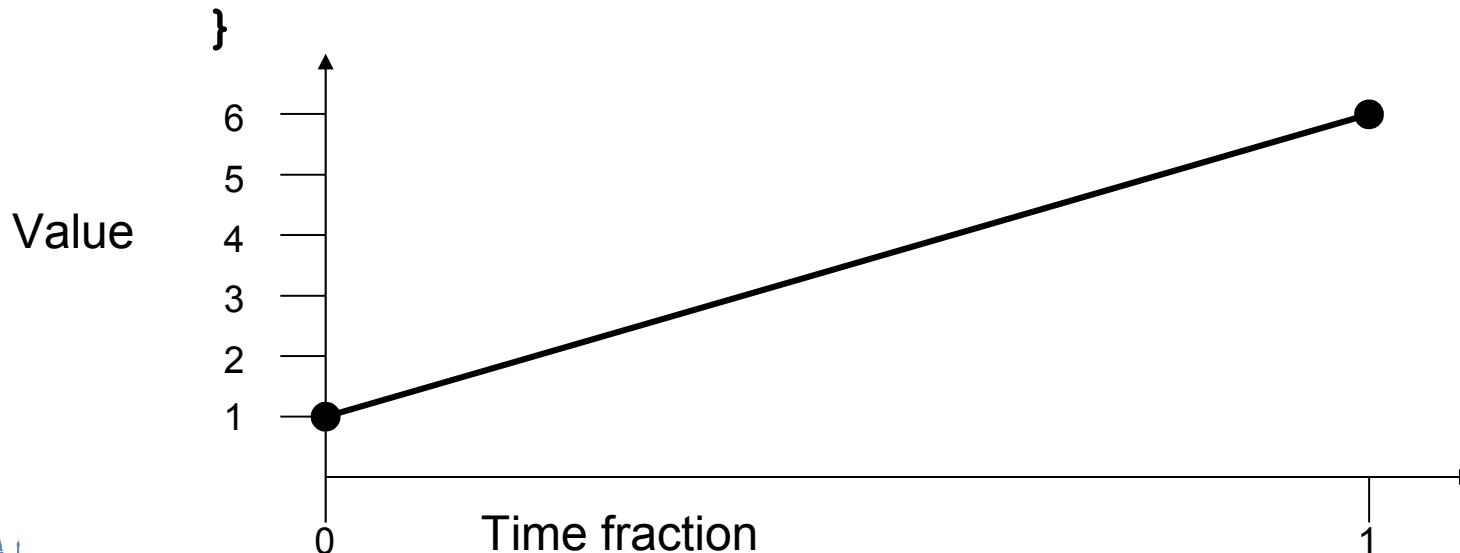
- ☕ Or build your own



LinearInterpolator

- ☕ LinearInterpolator: Animator's default
- ☕ Just uses the current timing fraction

```
public void interpolate(float fraction) {  
    return fraction;  
}
```

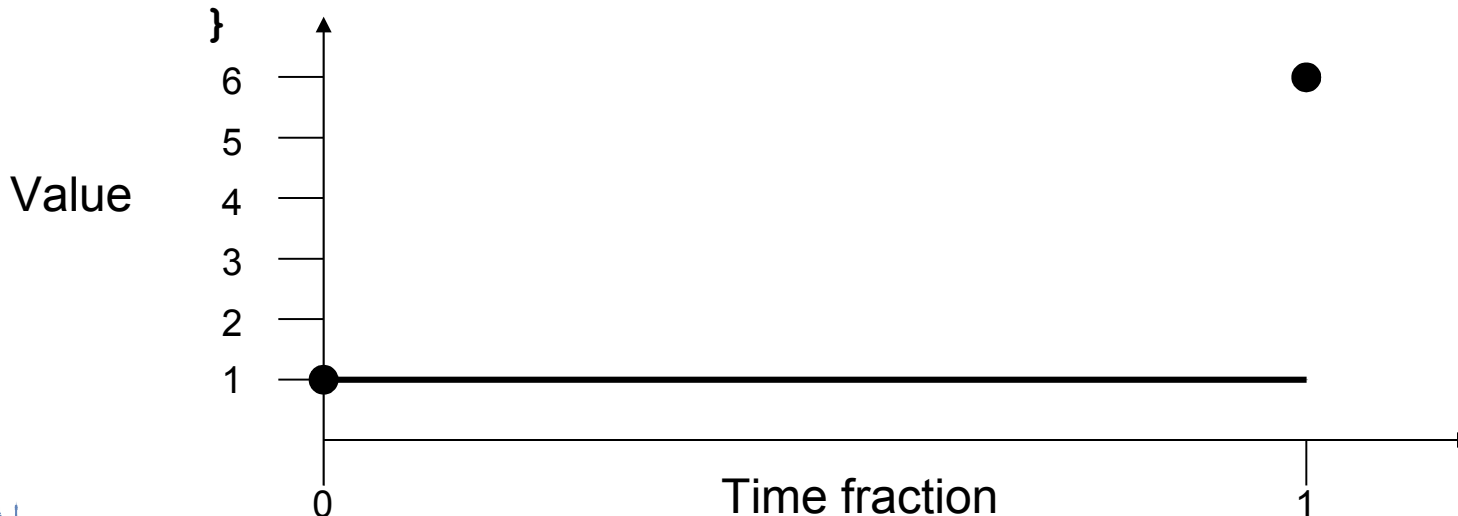


DiscreteInterpolator

☞ Useful for discrete values

☞ e.g., index values

```
public float interpolate(float fraction) {  
    if (fraction < 1.0f) { return 0; }  
    return 1.0f;  
}
```

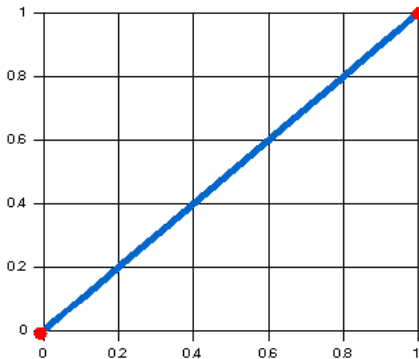


SplineInterpolator

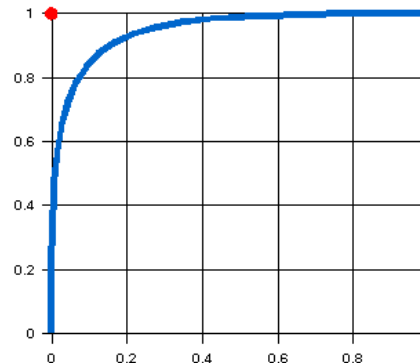
- ☞ Bezier curve that defines interpolation used between endpoints of animation
- ☞ Common technique in animation tools

```
public SplineInterpolator(x1, y1, x2, y2)
```

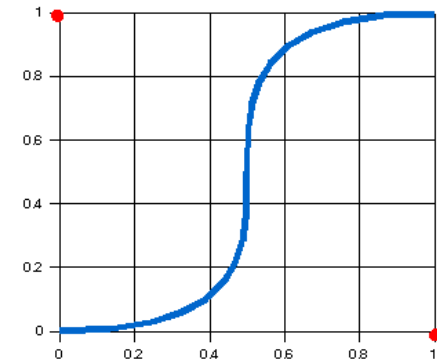
Linear



Quick Start



Ease-In, Ease-Out



- ☞ Play with Romain's SplineEditor demo on the [timingframework](#) site to see how these things work



PropertySetter

- ☕ Built-in facility to animate JavaBean properties of Objects
 - ➔ e.g., “location” of a button, “bounds” of a label
- ☕ Works for any property name (“prop”) that has related setter (“setProp”)
 - ➔ For example: Component.size, Component.foreground, Component.location, ...
- ☕ Custom components or delegators when no appropriate property exists
 - ➔ e.g., opacity, rotation, scale
- ☕ PropertySetter implements TimingTarget
 - ➔ Hand it to Animator and it all just works...

PropertySetter Example

- ☕ To move “button” between Points ‘from’ and ‘to’ over 2 seconds:

```
PropertySetter ps =  
    new PropertySetter(button, "location", from, to);  
Animator mover = new Animator(2000, ps);  
mover.start();
```

- ☕ Or, even easier:

```
PropertySetter.  
    createAnimator(2000, button, "location", from, to).  
    start();
```

SetterRace: The Code

- No more need to implement TimingTarget
 - ➔ Just use a PropertySetter instead

```
Animator racer = PropertySetter.createAnimator(  
    RACE_TIME, track, "carPosition", start, end);
```



But Wait, There's More!

Multi-Step Animations

- ➔ Support more complex animations than simple from/to
- ➔ Key frames define times/values/interpolation for multiple intervals

Triggers

- ➔ Simple wrappers around EventListeners
- ➔ Simpler sequencing of animations based on GUI events and other animations

More properties in Animator

- ➔ `initialFraction`, `direction`, `EndBehavior`





DEMO

Final Race

www.javapolis.com








Animation: Summary

- ☕ Animation is about varying values over time
- ☕ Possible with existing Java classes
 - ➔ `java.util.Timer`
 - ➔ `javax.swing.Timer`
- ☕ ... But easier with Timing Framework
 - ➔ Callbacks give more information (elapsed fraction)
 - ➔ Desirable animation behaviors built into framework
 - Repetition, duration, non-linear interpolation, multi-step
 - ➔ Natural tie-in to GUI animations
 - PropertySetters, Triggers
- ☕ API still in-development; use it and let me know what changes you would like to see
 - ➔ Wouldn't you like to see something like this in the next version of Java SE?

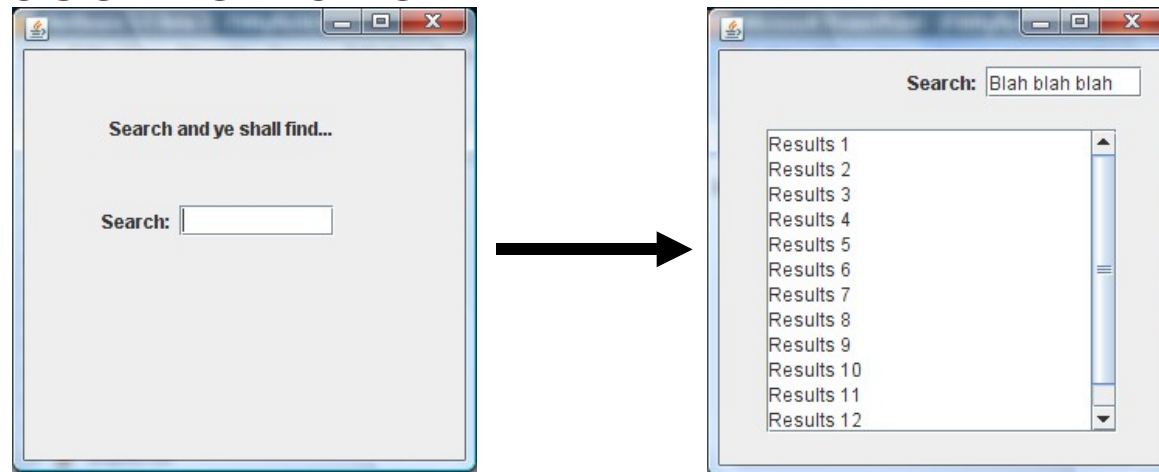


Agenda

-  Graphics
-  Animation Fundamentals
-  **Animated Transitions**
-  3D
-  Filthy Rich Effects

Animated Transitions

- ☕ Don't make the user work to understand the GUI
- ☕ Lead them logically through state changes
- ☕ Suppose we have:



- ☕ We would like to transition smoothly from one screen to the next

AnimTrans

- ☕ Project built on top of Timing Framework
- ☕ Not yet released
 - ➔ Planned to release with the book
- ☕ Simple to use, possible to extend
- ☕ Idea:
 - ➔ Hand a container (JComponent) to the system
 - ➔ Configure the Animator (e.g., duration)
 - ➔ start()
 - ➔ Handle callback to set up next screen
 - ➔ animtrans handles the details



AnimTrans: Example

```
Animator animator = new Animator(1000);
ScreenTransition transition = new ScreenTransition(
    transitionContainer, this, animator);
animator.setDeceleration(.4f);
transition.start();

// TransitionTarget implementation
public void setupNextScreen() {
    // rearrange GUI for next screen
}
```

AnimTrans: How it Works

- ☕ ScreenTransition determines state for components in container for current screen
- ☕ Callback into user code to `setupNextScreen()`
- ☕ ScreenTransition determines component state for next screen
- ☕ Appropriate “effects” chosen
 - ➔ e.g., Move, Scale, Fade
- ☕ Animator started, animtrans uses effects to render in-between states for components



AnimTrans: Tips and Tricks

- ❏ How can the GUI change in `setupNextScreen()` without causing the user to see the changes?
 - ➔ `ScreenTransition` renders an image of the current screen into the `GlassPane` to cover
 - ➔ Just like a curtain between scene changes in a play
- ❏ How does animtrans get good performance?
 - ➔ Most Effects only need to copy images around
 - ➔ Scale requires re-rendering components since layout and text may change during scale
- ❏ What if I want a different effect?
 - ➔ Effects are pluggable; create one and tell animtrans which component/transition to use it on



DEMO

Animated Transitions

www.javapolis.com



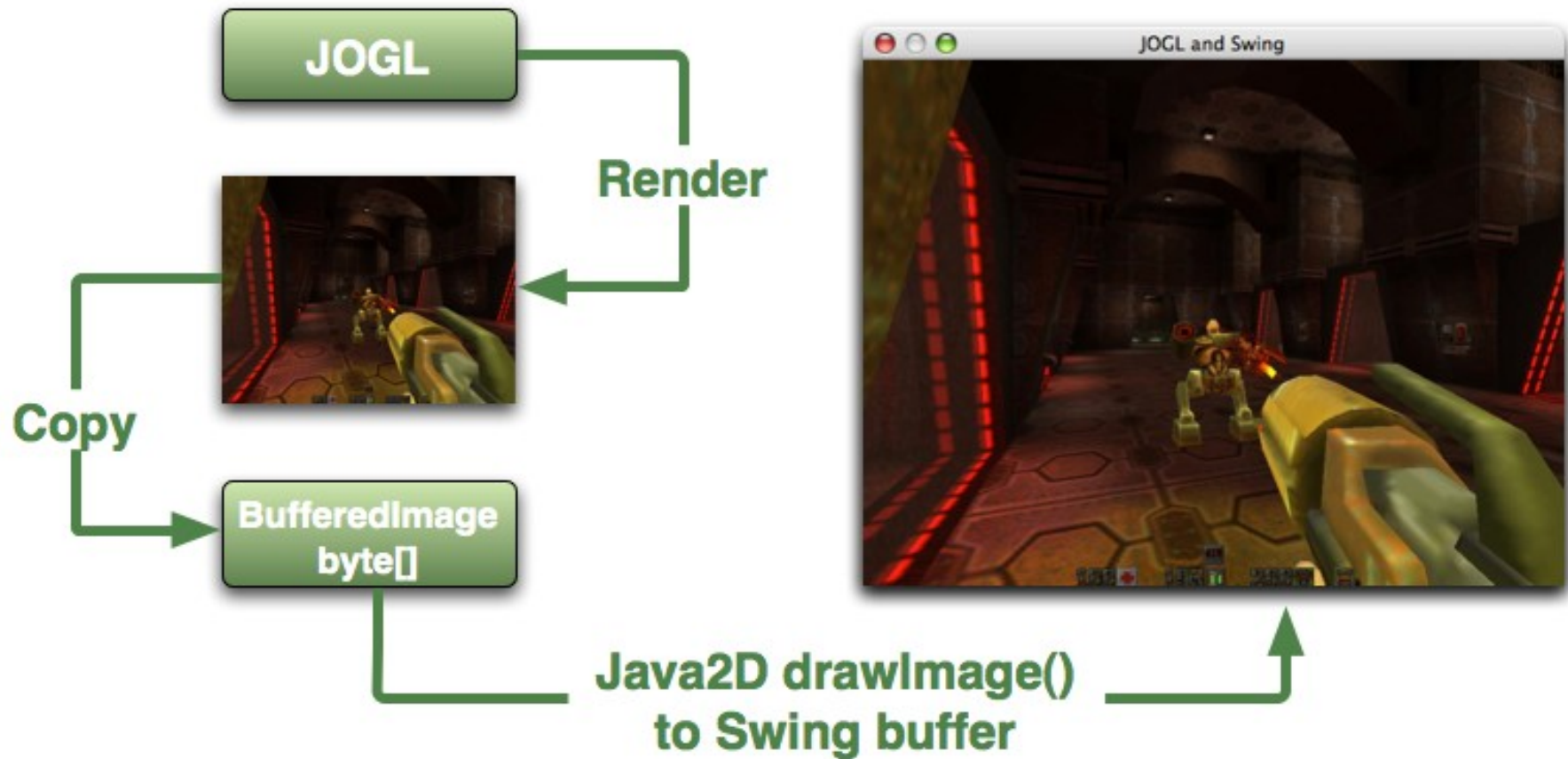
Agenda

- ☕ Graphics
- ☕ Animation Fundamentals
- ☕ Animated Transitions
- ☕ 3D
- ☕ Filthy Rich Effects

JOGL

- ☉ <http://jogl.dev.java.net>
- ☉ Swing and 3D interaction

Historical 3D/Swing Interaction



New in JavaSE 6: GLJPanel

JOGL

Render



Agenda

- ☕ Graphics
- ☕ Animation Fundamentals
- ☕ Animated Transitions
- ☕ 3D
- ☕ **Filthy Rich Effects**

Bring Life to Your Applications

- ☕ Life is restless
 - ⇒ Transitions
 - ⇒ Highlights
 - ⇒ Progress Indicators
 - ⇒ Motion
- ☕ Life is not flat
 - ⇒ 3D visualization



Transitions

- ☕ When a value is changed
 - ➔ A label's text
 - ➔ A screen
- ☕ Fade in/out
 - ➔ Fade from/to a color (e.g. Fade to black)
 - ➔ Opacity change
- ☕ Cross-fade
 - ➔ Current value fades out
 - ➔ New value fades in



Fade to Black, Timing Code

```
timer = PropertySetter.createAnimator(  
    1200, this, "fadeOut", 0.0f, 1.0f);  
timer.setAcceleration(0.7f);  
timer.setDeceleration(0.3f);  
timer.start();
```



Fade to Black, Painting Code

```
public void setFadeOut(float fadeOut) {  
    this.fadeOut = fadeOut;  
    repaint();  
}
```

```
protected void paintComponent(Graphics g) {  
    g.setColor(new Color(0.0f, 0.0f, 0.0f, fadeOut));  
    Rectangle r = g.getClipBounds();  
    g.fillRect(r.x, r.y, r.width, r.height);  
    // ...  
}
```



Highlights

- ☉ Outline an interactive element
- ☉ Canonical effects
 - ⇒ Brightness increase
 - ⇒ Glow, Pulse
 - ⇒ Spring
- ☉ Usually triggered by a user input



Highlights, Timing Code

```
public void mouseEntered(MouseEvent e) {  
    if (timer != null && timer.isRunning()) {  
        timer.stop();  
    }  
    timer = PropertySetter.createAnimator(duration,  
        button, "foreground", Color.WHITE);  
    timer.start();  
}
```



Progress Indicators

- ☕ Show that your UI is still alive
 - ⇒ Otherwise the user might poke it with a stick
- ☕ Indeterminate progress indicators
 - ⇒ Rotation
 - ⇒ Glow/Pulse
- ☕ Short, repeated animations



Progress, Timing Code

```
PropertySetter setter = new PropertySetter(  
    this, "glow", 0.0f, 1.0f);  
timer = new Animator(800, Animator.INFINITE,  
    RepeatBehavior.REVERSE, setter);  
timer.start();
```



Progress, Painting Code

```
protected void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;

    Composite composite = g2.getComposite();
    g2.setComposite(
        AlphaComposite.SrcOver.derive(glow));
    g2.drawImage(javaCupGlow, x, y, null);
    g2.setComposite(composite);

    g2.drawImage(javaCup, x, y, null);
}
```



Motion

- ☕ Helps the user understand what happened
 - ➔ When I miss a drop, the item goes back to its origin
 - ➔ When items change location, it is obvious
- ☕ No more "undo/redo" syndrome
- ☕ Realistic motion
 - ➔ Non-linear movements
- ☕ Implementation is simple
 - ➔ Use property setters
 - ➔ Use acceleration/deceleration



DEMO

Putting it All Together

www.javapolis.com



Animated User Interfaces

- ☕ Use built-in properties
 - ➡ Foreground/background colors
 - ➡ Location, size
- ☕ Use advanced components
 - ➡ JPanel from the SwingLabs project exposes an alpha property for easy fade in/out
- ☕ Keep animations short and simple
 - ➡ Do not bore the user!

Summary: Make Your Clients Rich

Swing
+
Java 2D
+
Animation
+
3D

Filthy Rich Clients


Did We Mention There's a Book?

 And the surprise title is:

“Filthy Rich Clients”

 ETA: JavaOne 2007

➔ Wish us luck finishing it!

 ISBN: 0132413930

 Early access:

➔ http://groups.yahoo.com/group/filthy_rich_clients/

For More Information

 Romain's blog

⇒ <http://jroller.com/page/gfx>

 Chet's blog

⇒ <http://weblogs.java.net/blog/chet>

 Java Desktop content in general

⇒ <http://javadesktop.org>





Q&A

www.javapolis.com



Thank you for your attention!

