Java™ Media Programming Code Camp

Bill Day
bill.day@sun.com

Technology Evangelist
Sun Microsystems

Sun microsystems
We make the net work.

Sun microsystems

Agenda

- Overview of the Java™ 2 Platform
- Introduction to the Media APIs
- In Depth: Java 2D™ API
- In Depth: Java Media Framework
- In Depth: Java 3D™ API
- Summary and Resources

Note: This tutorial assumes you will be deploying on the Java™ 2 Platform, Standard Edition (J2SE™ Platform). Examples were developed and tested under J2SE SDK version 1.4.0.

Sun microsystems

Overview of the Java™ 2 Platform

Overview of the Java™ 2 Platform

- Java 2 Platform technology is developed via the Java Community Process (JCP): www.jcp.org
- The JCP is used to develop specifications for Java programming APIs and related technologies
- JCP specifications are developed starting from a Java Specification Request (JSR)
- Before final acceptance, a JSR expert group must provide:
  - Specification
  - Reference implementation
  - Testing Compatibility Kit (TCK) used by implementers to verify compatibility with specification

Overview of the Java™ 2 Platform

- Java programming language specification (grammar, keywords)
- Virtual machine spec (including bytecode)
- Core APIs
  - Specified for each Java 2 Platform edition: Java 2 Platform, Enterprise Edition (J2EE™ Platform), Java 2 Platform, Standard Edition (J2SE™ Platform), and Java 2 Platform, Micro Edition (J2ME™ Platform)
- Optional Packages
  - Examples include JMF and Java 3D API
- Related tools
  - Compiler, RMI registry, javadoc, etc.

Differences Between Core APIs and Optional Packages

- All Java technology licensees must implement the core APIs for a given edition
  - For the J2SE™ Platform, this includes the `java.*` packages plus a few related APIs such as `javax.swing`, etc.
  - J2SE version 1.4 includes Java 2D, Image I/O, and Sound APIs built-in
- Licensees may choose to implement an Optional Package
  - If they do, they must implement the entire package according to specification
  - Provided in `javax` packages
  - Examples: JMF and Java 3D APIs

Java™ Platform Media Support

- Java 1.0 and 1.1 technology
  - Primitive core support for AWT-based 2D graphics, limited audio (applets only)
- Java 2 version 1.2 technology
  - J2SE™ Platform: Java 2D, Java Sound Engine
  - Optional packages: Java 3D, JMF, Java Speech
- Java 2 version 1.3 technology
  - J2SE Platform: New Java 2D features, Sound API
  - Optional packages: Java 3D and JMF updates
- Java 2 version 1.4 technology
  - J2SE Platform: Java 2D performance enhancements, new Image I/O framework
  - Revisions to optional packages: updated Java 3D API, new open source Java Speech implementation

Sun microsystems

Introduction to the Media APIs

Introduction to the Java™ Media APIs

- Java 2D
- Java Image I/O
- Java Sound
- Java Media Framework
- Java 3D
- Java Speech
- Java Advanced Imaging
- Java Shared Data Toolkit

Note: "*Java Media*" properly refers to the entire set of Java Media APIs. "*JMF*" refers to a specific API, the Java Media Framework.

Introduction to the Java™ Media APIs

- Java 2D™ API
  - 2D graphics and image manipulation
  - `Graphics` capabilities extended in `Graphics2D`
- Java Image I/O
  - Framework for image input and output
  - Handles transcoding between image formats, accessing individual images in multi-image files, various other image I/O operations
- Java Sound API
  - Software sound processor and MIDI synthesizer
  - Sound engine (Java 2 SDK 1.2) and sound API (beginning in Java 2 SDK 1.3)

Introduction to the Java™ Media APIs

- Java™ Media Framework API
  - Playback of synchronized media in 1.0 API
  - 2.0 API adds support for media capture and streaming of audio and video
- Java 3D™ API
  - Object-based 3D graphics runtime
  - Optimized for fast 3D rendering for simulations, interactive graphics, gaming, and similar uses
- Java Speech API
  - Speech recognition and synthesis

Introduction to the Java™ Media APIs

- Java Advanced Imaging API
  - Advanced 2D image processing
  - Implements many Java 2D API interfaces
- Java Shared Data Toolkit
  - Free toolkit for adding collaborative features to Java technology-based applications
  - Objects share data via a Session object, JSDT URLs, and a JSDT registry

Availability of J2SE™ Platform Core Media APIs

API	*Type*	*Spec*	*Related JSRs*	*FAQ*	*Reference Impl.*	*Mailing list*
Java 2D	Core Java 2	Yes (part of Java 2 specs)	JSR 59	Yes	Yes (part of J2SE v1.4 RI)	Yes
Java Image I/O	Core Java 2	Yes (part of Java 2 specs)	JSRs 15 and 59	Yes	Yes (part of J2SE v1.4 RI)	Yes
Java Sound	Core Java 2	Yes (part of Java 2 specs)	JSR 59	Yes	Yes (part of J2SE v1.4 RI)	Yes

Availability of Optional Package Media APIs

API	*Type*	*Spec*	*Related JSRs*	*FAQ*	*Reference Impl.*	*Mailing list*
JMF	Optional Package	Yes (2.0)	JSRs 908, 135	Yes	Yes (2.1.1a)	Yes
Java 3D	Optional Package	Yes (1.3 Beta 2)	JSRs 912, 148, 184	Yes	Yes (1.3 Beta 2)	Yes
Java Speech	Optional Package	Yes (1.0)	2.0 API via JSR 113	Yes	No, but FreeTTS and other impls	Yes
Java Advanced Imaging	Optional Package	Yes (1.1)	JSR 34	Yes	Yes (1.1.1_01)	Yes
Java Shared Data Toolkit	Optional Package	Yes (2.0 released)	N/A	Yes	Yes	Yes

Potentially Competing Technologies

- OpenGL
  - Procedural, low-level graphics language
  - Java technology-to-OpenGL bindings also available from various third party vendors
  - Many Java 3D API implementations (including Solaris™, Win32, Linux, and IRIX) build on OpenGL
  - Competes with and complementary to Java 3D API
- X3D (XML compliant update to VRML)
  - X3D primarily provides a file format for 3D models
  - Sun joined the Web 3D Consortium in mid-1998 and contributed its source code to start the Xj3D Toolkit (an X3D and VRML97 browser written using Java 3D)
  - Complementary to Java 3D API

Potentially Competing Technologies

- QuickTime for Java™
  - Apple has released Java platform bindings to its QuickTime multimedia architecture
  - Targets established QuickTime market (good for existing QT users, bad if need other formats)
  - Primarily competes with JMF
- RealSystem and Windows Media
  - Real Network's and Microsoft's streaming media systems, respectively
  - Microsoft SDK completely Win32-reliant
  - RealSystem tends to be too content creator centric, not developer centric enough
  - Primary competition is JMF

Sun microsystems

In Depth: Java 2D™ API

Java 2D™ API

- Treats all forms of 2D visual information (text, primitive shapes, polygons, Bezier curves, images, etc.) alike
- Enables consistent compositing, color manipulation, other 2D operations
- Included in Java™ 2 Platform, Standard Edition
  - All J2SE (and thereby J2EE) platform implementations, including Java Plug-in technology, are required to support Java 2D API
  - As of J2SE v1.4 release, includes Image I/O, too

Java 2D API Package Summary

- Java 2D API is specified in the following packages:
  - java.awt (portions 2D related)
  - java.awt.color
  - java.awt.font
  - java.awt.geom
  - java.awt.image (portions 2D related), java.awt.image.renderable
  - java.awt.print
- Sun implementations of the J2SE™ Platform provide support for JPEG via package:
  - com.sun.image.codec.jpeg
- J2SE v1.4 release and beyond also includes Image I/O Framework via packages under:
  - javax.imageio

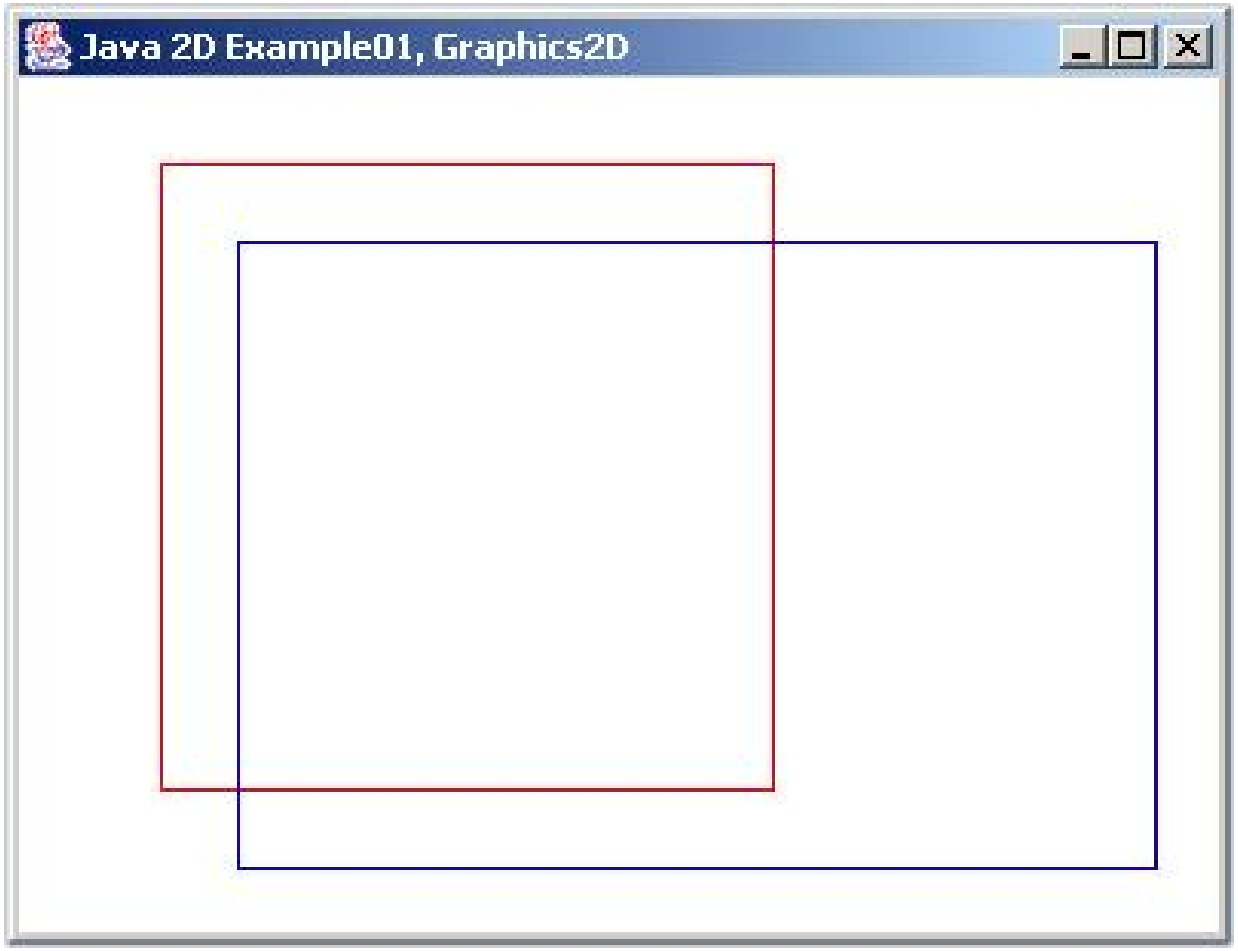
Graphics2D: A Better Graphics Class

- `java.awt.Graphics2D` is the rendering engine for the Java 2D API
- `Graphics2D` extends abstract class `Graphics`, maintaining backwards compatibility
- `Example01` illustrates using `Graphics2D` by casting a `Graphics` reference to a `Graphics2D` reference

Java 2D API: Example01, Graphics2D

```
051     * The paint method provides the real magic.  Here we
052     * cast the Graphics object to Graphics2D to illustrate
053     * that we may use the same old graphics capabilities with
054     * Graphics2D that we are used to using with Graphics.
055     **/
056    public void paint(Graphics g) {
057      //Here is how we used to draw a square with width
058      //of 200, height of 200, and starting at x=50, y=50.
059      g.setColor(Color.red);
060      g.drawRect(50,50,200,200);
061
062      //Let's set the Color to blue and then use the Graphics2D
063      //object to draw a rectangle, offset from the square.
064      //So far, we've not done anything using Graphics2D that
065      //we could not also do using Graphics.  (We are actually
066      //using Graphics2D methods inherited from Graphics.)
067      Graphics2D g2d = (Graphics2D)g;
068      g2d.setColor(Color.blue);
069      g2d.drawRect(75,75,300,200);
070    }
```

Java 2D API: Example01 Output



Sun microsystems

Java 2D API: Shapes and GeneralPaths

- `Shapes` are used to create arbitrarily shaped 2D graphics
- `GeneralPaths` are the most general implementation of Shape
- `GeneralPath` interiors are specified using winding rules
- All `Shapes`, including `GeneralPaths`, are manipulated using matrices in `AffineTransforms`

Java 2D API: Example02, GeneralPaths and Transforms

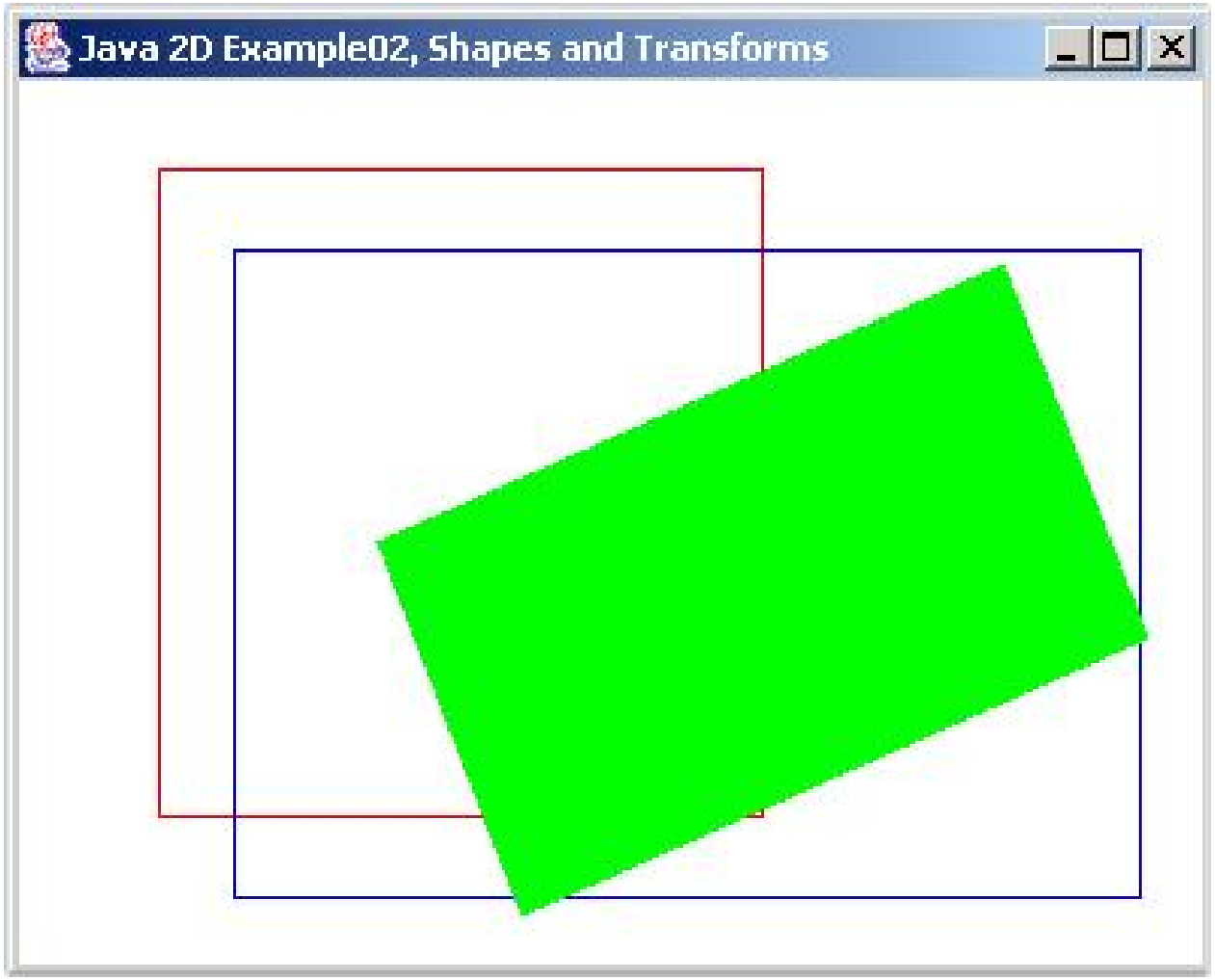
- Example02 illustrates `GeneralPath`, winding rules, and `AffineTransform`

```
073     //Now, let's draw another rectangle, but this time, let's
074     //use a GeneralPath to specify it segment by segment.
075     //Furthermore, we're going to translate and rotate this
076     //rectangle relative to the Device Space (and thus, to
077     //the first two quadrilaterals) using an AffineTransform.
078     //We also will change its color.
079     GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
080     path.moveTo(0.0f,0.0f);
081     path.lineTo(0.0f,125.0f);
082     path.lineTo(225.0f,125.0f);
083     path.lineTo(225.0f,0.0f);
084     path.closePath();
```

Java 2D API: Example02 (Cont.)

```
086     AffineTransform at = new AffineTransform();
087     at.setToRotation(-Math.PI/8.0);
088     g2d.transform(at);
089     at.setToTranslation(50.0f,200.0f);
090     g2d.transform(at);
091
092     g2d.setColor(Color.green);
093     g2d.fill(path);
```

Java 2D API: Example02 Output



Sun microsystems

Java 2D API: Curves, Text, and Antialiasing

- Example03 introduces GeneralPath's `quadto()` and `curveto()` methods, adds text into the mix, and illustrates how to request antialiased rendering

Java 2D API: Example03, Curves and Antialiasing

```
061   public void paint(Graphics g) {
062     Graphics2D g2d = (Graphics2D) g;
063
064     //This time, we want to use anti-aliasing if possible
065     //to avoid the jagged edges that were so prominent in
066     //our last example.  We ask the Java 2D rendering
067     //engine (Graphics2D) to do this using a "rendering hint".
068     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
069        RenderingHints.VALUE_ANTIALIAS_ON);
070
071     //We reuse our GeneralPath filled shape.  We translate
072     //and rotate this shape as we did before.
073     GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
074     path.moveTo(0.0f,0.0f);
075     path.lineTo(0.0f,125.0f);
076     path.quadTo(100.0f,100.0f,225.0f,125.0f);
077     path.curveTo(260.0f,100.0f,130.0f,50.0f,225.0f,0.0f);
078     path.closePath();
```

Java 2D API: Example03 (Cont.)

```
080     AffineTransform at = new AffineTransform();
081     at.setToRotation(-Math.PI/8.0);
082     g2d.transform(at);
083     at.setToTranslation(0.0f,150.0f);
084     g2d.transform(at);
085
086     g2d.setColor(Color.green);
087     g2d.fill(path);
088
089     //Now, let's use some of the Java font and text support.
090     //Note that you need to be sure you have the same fonts I
091     //use in the example (Times New Roman True Type) if you
092     //execute this example code.
093     Font exFont = new Font("TimesRoman",Font.PLAIN,40);

104     g2d.setFont(exFont);
105     g2d.setColor(Color.black);
106     g2d.drawString("Hello Camp",0.0f,0.0f);
107   }
```

Java 2D API: Example03 Output

Java 2D API: Image Processing

- Java 2D API presents a new model for image processing, the buffered image model
- Example04, aka “ImageDicer”, makes use of the buffered image model to blur, sharpen, and otherwise manipulate user specified images

Sun microsystems

ImageDicer Source Image

Lady Agnew of Locknaw, by John Singer Sargent

Sun microsystems

Java 2D API: Color Inversion With ImageDicer

- Java 2D API provides lookup table support for use in color-related image manipulations
- Perform color inversion by inverting each of the red, blue, and green (RGB) color values for each pixel in an image

```
001 short[] invert = new short[256];
002 for (int i = 0; i < 256; i++)
003   invert[i] = (short)(255 - i);
004 BufferedImageOp invertOp = new LookupOp(
005     new ShortLookupTable(0, invert), null);
```

Sun microsystems

Example04 “ImageDicer”: Inverted Image

Inverting all three RGB channels gives a *negative* image

Java 2D API Tip: Snapshot Your Components

- Use Sun's JPEG support classes to save snapshots of Components
- The basic steps are:
  - Create a `BufferedImage` with the same dimensions as your `Component`
  - Draw the `Component` into the `BufferedImage`
  - Save the `BufferedImage` into a file using the JPEG package and `FileOutputStream`

Note: Requires the use of `com.sun.image.codec.jpeg`, which may not be available in all Java 2 runtimes

Java 2D API: General Case
SaveComponentAsJPEG method

```
001   public void saveComponentAsJPEG(Component myComponent,
002                                   String filename) {
003     Dimension size = myComponent.getSize();
004     BufferedImage myImage =
005         new BufferedImage(size.width, size.height,
006         BufferedImage.TYPE_INT_RGB);
007     Graphics2D g2 = myImage.createGraphics();
008     myComponent.paint(g2);
009     try {
010       OutputStream out = new FileOutputStream(filename);
011       JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
012       encoder.encode(myImage);
013       out.close();
014     }
015     catch (Exception e) { System.out.println(e); }
016   }
```

Java 2D API: Snapshot method in Example04 “ImageDicer”

- We can further simplify the snapshot method if we already have a BufferedImage available, as in Example04 “ImageDicer”:

```
317   public void saveImage(String filename) {
318     try {
319       OutputStream out = new FileOutputStream(filename);
320       JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
321       encoder.encode(mBufferedImage);
322       out.close();
323     }
324     catch (Exception e) { System.out.println(e); }
325   }
```

Java 2D API: New Features in J2SE SDK 1.4

- Pluggable Image I/O framework
- New 2D pipeline architecture for better performance (details in J2SE 1.4 SDK docs)
- Hardware acceleration for offscreen images
- Public Unicode Bidirectional Algorithm used to order and arrange bidi text
- Introduced in 1.3 release: support for PNG image format and multiple monitors
- Learn more from the Java 2D documentation: java.sun.com/products/java-media/2D/

Sun microsystems

In Depth:
Java™ Media Framework

Java™ Media Framework API (JMF)

- JMF delivers and renders synchronized multimedia
- Specified and implemented in phases:
  - JMF 1.0 supports Java Media Players to play audio and video from both push and pull sources
  - JMF 2.0 adds support for audio and video capture from input devices, on-the-fly manipulations of media data, plugable codecs

JMF Implementations

- Sun provides an all-Java technology version
  - Implements JMF using Java programming language code (no native methods) for maximum portability
  - Runs on any compliant J2SE™ Platform implementation (Solaris, Linux, Win32, AIX, HP-UX, any other OS with J2SE runtime)
- Sun also provides “performance packs” optimized for Solaris and Win32
- Blackdown.org provides Linux implementation

JMF: Supported Content Types

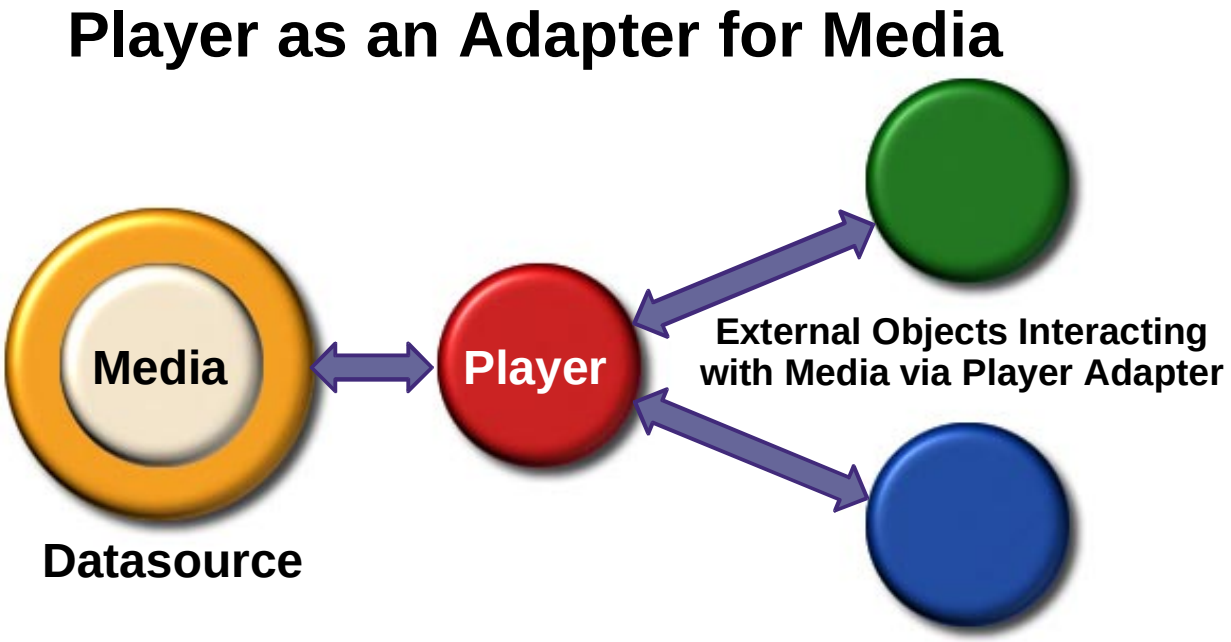
- Supported media content types include:
  - QuickTime, AVI video
  - MPEG-1
  - WAV, AU audio
  - MIDI
  - Sun supports MPEG-1 Layer 3 (MP3) audio in its JMF 2.0 implementation
  - H.261, H.263 video and G.723 audio low bitrate ITU protocols
- Details on content types supported in Sun implementations: java.sun.com/products/java-media/jmf/2.1.1/formats.html

JMF: Supported Protocols

- Supported JMF 1.0 protocols:
  - HTTP, FILE, FTP, RTP receive-only
- JMF 2.0 adds support for RTP send
  - RTP send support enables JMF-based audio and video servers
- Details on protocols supported in the Sun implementations: java.sun.com/products/java-media/jmf/2.1.1/formats.html

JMF: Player Basics

- Players extend MediaHandler and serve as an adapter for time-based media
- Media itself is encapsulated by a DataSource object



JMF Player API: `javax.media`

- Players, content handling, and core synchronization are in `javax.media` package
  - `Clock`, `Controller`, `Player` interfaces
  - `MediaHandler`, `MediaProxy` interfaces
  - various `Control` and `Listener` interfaces
  - `Manager`, `PackageManager` classes
  - All events, errors, and exceptions to support state machine model
- Note: JMF events extend `java.util.EventObject`, consistent with standard Java™ 2 Platform event mechanisms

JMF Player API: `javax.media.protocol`

- Data source and protocol resolution specified in `javax.media.protocol`
  - Interfaces for source stream configuration and source controls
  - Classes to support push and pull `DataSources`
- Most JMF developers will not need to use this package directly
  - `Manager` automatically creates `Player` and `DataSource` and hooks the two together
  - `Player` methods called by programmer automatically use `DataSource` when need be

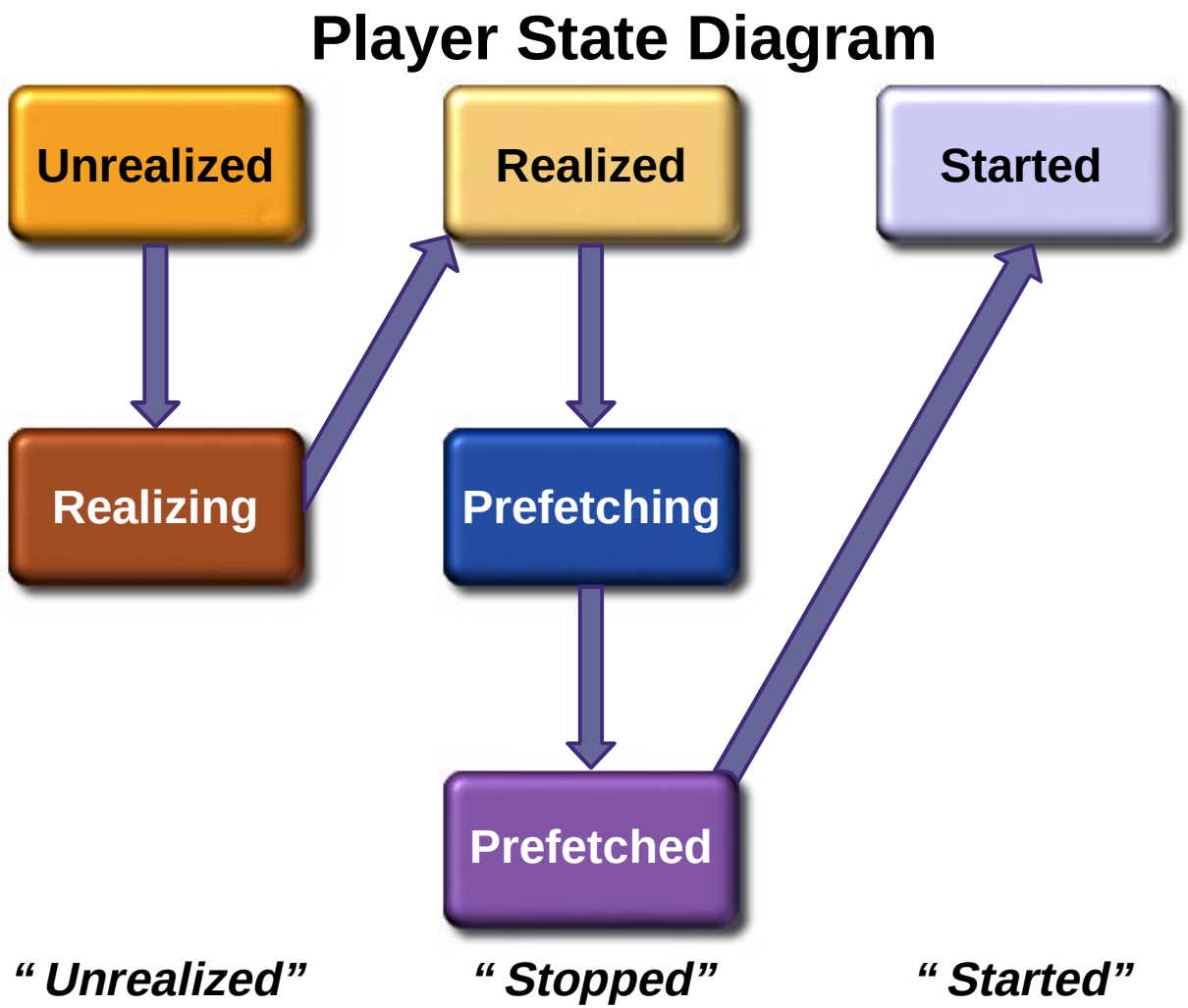
Sun microsystems

JMF Player API: Player States

- Players behave as state machines
- JMF specification details both legal and illegal state transitions and the corresponding events and exceptions

JMF Player API: State Model

JMF 2.0 supports access to internals of JMF state model



Example05 "JMFApplet"

- JMFApplet illustrates several key JMF Player concepts, including how to:
  - Use `Manager` to request a `Player` instance
  - Register `ControllerListener` for JMF callbacks
  - Catch `RealizeCompleteEvent` in `controllerUpdate()` method to finish setting up `Player`
  - Stop and properly deallocate `Player` to free up any exclusive resources

Example05 “JMFApplet”

```
024 public class Example05 extends Applet
                           implements ControllerListener {
025     private URL myURL = null;
026     private Player myPlayer = null;
027     private Component myVisual = null;
028     private Component myControls = null;
029     private Panel visualPanel = null;
030
031     /**
032      * Initialize JMFApplet.  We lay out the interface and
033      * create our player in the init().
034     **/
035     public void init() {
036         super.init();
037
038         // Specify AWT Layout Manager.
039         setLayout (new BorderLayout());
040
041         // Load URL from the web page JMFApplet is embedded in.
042         String asset = getParameter("ASSET");
```

Example05 “JMFApplet” (Cont.)

```
044         // Check the URL and create a URL object to hold it.
045         if (asset.equals("")) {
046            //we haven't entered an asset in the applet.
047         } else {
048            try {
049               myURL = new URL(getDocumentBase(),asset);
050            } catch (MalformedURLException e) {
051               //We entered an incomplete asset or built incorrect URL.
052               //More robust applet should handle this gracefully.
053            }
054         }
```

Example05 “JMFApplet” (Cont.)

```
055         try {
056            //Here's an interesting bit.  Manager is used to
057            //create the actual player for this URL.  We then
058            //add JMFApplet as a ControllerListener for myPlayer.
059            //This lets us respond to RealizeCompleteEvents.
060            myPlayer = Manager.createPlayer(myURL);
061            myPlayer.addControllerListener(this);
062         } catch (IOException e) {
063            // Encountered some problem with I/O; exit.
064            System.out.println("I/O problem attempting to
                                  create player...exiting");
065            System.exit(1);
066         } catch (NoPlayerException e) {
067            // Unable to return a usable Player; exit.
068            System.out.println("No usable Player returned...exiting");
069            System.exit(1);
070         }
071      }
```

Example05 “JMFApplet” (Cont.)

```
074      * Override the default applet start method to call Player's
075      * realize().  This will first do the realization, which in turn
076      * triggers the final bits of GUI building in the controllerUpdate()
077      * method.  We do not automatically start playback:  The user needs
078      * to click on the "play" button in our applet to start playing the
079      * media sample.
080     **/
081     public void start() {
082        myPlayer.realize();
083     }
084
085
086     /**
087      * Override the default applet stop method to call myPlayer.stop()
088      * and myPlayer.deallocate() so that we properly free up resources
089      * if someone exits this page in their browser.
090     **/
091     public void stop() {
092        myPlayer.stop();
093        myPlayer.deallocate();
094     }
```

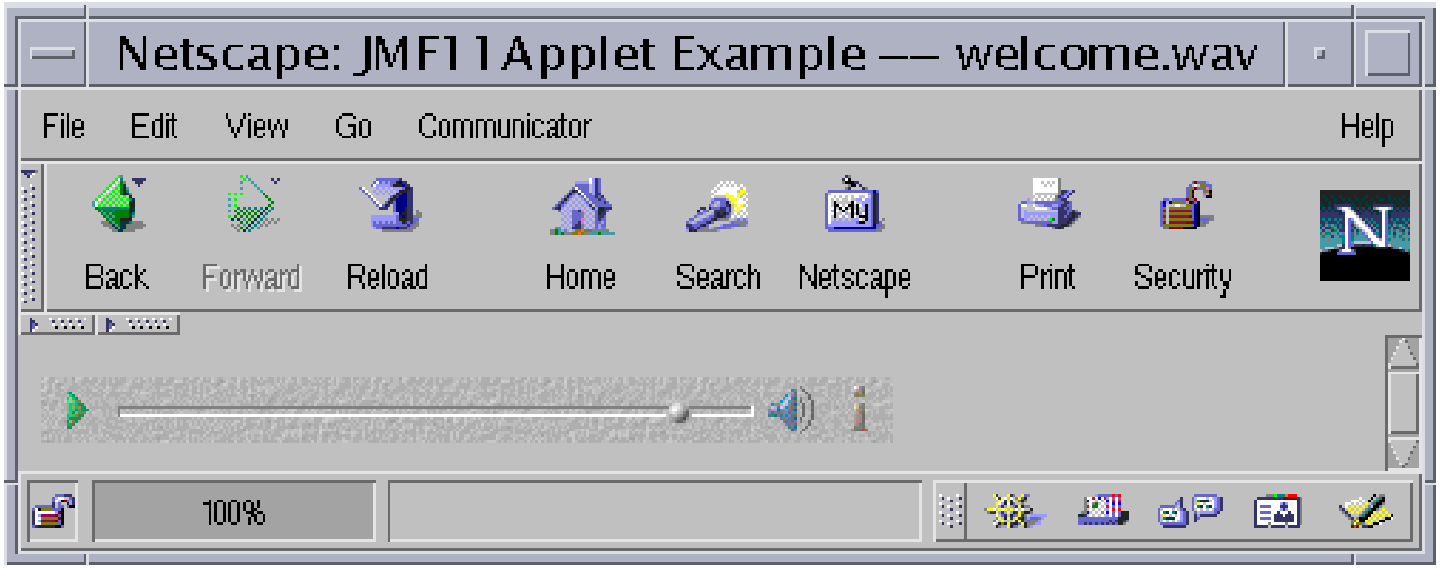
Example05 "JMFApplet" (Cont.)

```
097       * Since we must know when realize completes, we use
098       * controllerUpdate() to handle RealizeCompleteEvents.
099       * When we receive the RealizeCompleteEvent, we layout
100       * and display the video component and controls in our
101       * applet GUI.
102      **/
103     public void controllerUpdate(ControllerEvent event) {
104        if (event instanceof RealizeCompleteEvent) {
105           //System.out.println("Received RCE...");
106           // Now that we have a Realized player, we can get the
107           // VisualComponent and ControlPanelComponent and pack
108           // them into our applet.
109           myVisual = myPlayer.getVisualComponent();
110           if (myVisual != null) {
111              // In order to ensure that the VisualComponent
112              // is not resized by BorderLayout, I nest it
113              // within visualPanel using FlowLayout.
114              visualPanel = new Panel();
115              visualPanel.setLayout(new FlowLayout());
116              visualPanel.add(myVisual);
```

Example05 “JMFApplet” (Cont.)

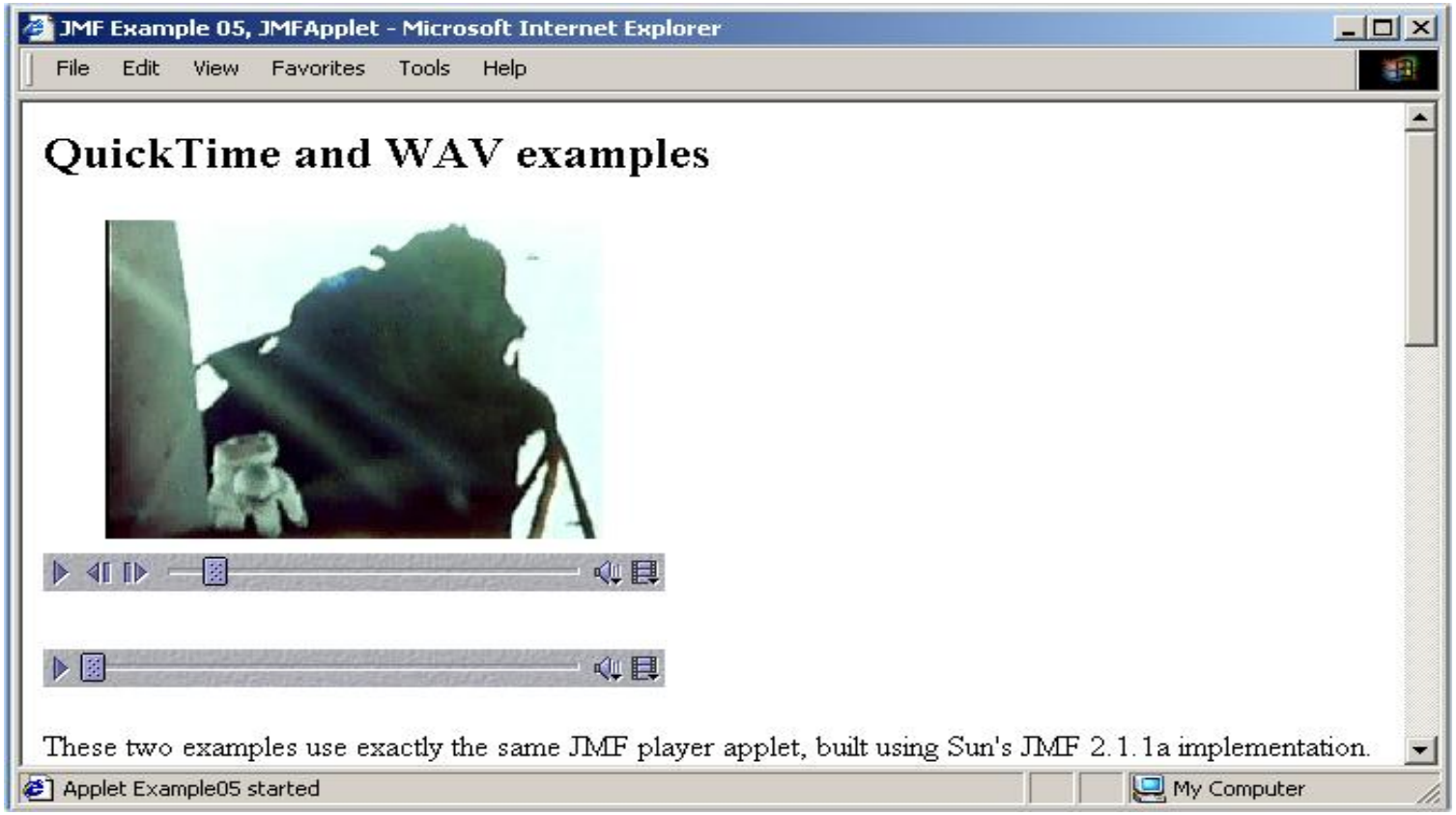
```
117                 add(visualPanel,BorderLayout.CENTER);
118                 //System.out.println("Added VisualComponent...");
119               }
120               myControls = myPlayer.getControlPanelComponent();
121               if (myControls != null) {
122                  add(myControls,BorderLayout.SOUTH);
123                  //System.out.println("Added controls...");
124               }
125               //invalidate();
126               validate();
127           }
128           // Else we simply consume the event.
129       }
130 }
```

Example05 “JMFApplet” Output



- Example05 “JMFApplet” playing welcome.wav audio file using the Sun JMF 1.1, Java platform-based player on Solaris™ 7 software within Netscape Communicator™ 4.51

Example05 "JMFApplet" Output



- The same Example05, with two instances playing example audio and video under Sun JMF 2.1.1a performance pack for Win32 and the Java™ 2 Plug-in in Microsoft Internet Explorer 5.5 browser

Sun microsystems

JMF Player API: Other Capabilities

- JMF also allows the programmer to:
  - Control multiple `Players` with a single `Controller` object, perhaps a `Player` itself
  - Integrate `Players` with other Java technology-based software. Same language, same tools, etc.
  - Integrate `Players` with `JavaBeans`™ architecture-based components for reusable streaming media components that can be quickly connected together using visual builder tools

Sun microsystems

New in the JMF 2.0 API: Media Capture

- Once you are comfortable with JMF Player basics, learn more about JMF 2.0 API's media capture and streaming capabilities
- Excellent resource: Simon Ritter's article, “Image Capture From Webcams Using the Java Media Framework API”, available with example source code from:
  sun.com/developers/evangcentral/totallytech/jmf.html
- More in depth information from the JMF API docs and guide:
  java.sun.com/products/java-media/jmf/

Sun microsystems

In Depth: Java 3D™ API

Sun microsystems

Java 3D™ API

- Java 3D API is an optional package API specifying a scene graph based 3D graphics runtime
- Java 3D API is optimized for display speed (interactive graphics and games) rather than image quality (render farms)

Java 3D™ Implementations

- Sun provides Win32 and Solaris™ operating environment implementations
- Other implementations available today:
  - Blackdown.org for Linux
  - SGI IRIX
  - Hewlett Packard HP-UX
  - IBM AIX

Java 3D API: Requirements

- Sun's Win32 implementation requires:
  - Java 3D implementation itself, available from Sun's web site
  - J2SE™ Platform, available from Sun
  - OpenGL 1.1 (bundled with WinNT 4.0, Win98,and newer Win32 flavors)
  - Optional: Sun also provides a Java 3D implementation for Win32 that uses DirectX rather than OpenGL

Java 3D API: Strengths

- High level, object oriented view of 3D graphics
- Optimized for speed using compiled branch groups, capability bits, etc.
- Large number of 3D loaders are available to import content into Java 3D runtime
  - Currently 20+ loader packages available, supporting formats varying from X3D and VRML97 to DXF to Protein DataBank
  - Detailed list of loaders and supported formats: www.j3d.org/utilities/loaders.html

Java 3D API: Strengths

- Sun and the Web3D Consortium also provide an X3D/VRML97 browser written entirely using Java technology and based upon the Java 3D API
- Ongoing work is being driven as an open source project by the Web3D Consortium's source task group, via its Xj3D Toolkit
- For more information, refer to: www.web3d.org/TaskGroups/source/xj3d.html

Java 3D API: Other Strengths

- Java 3D API supports exotic input and control devices
  - Data gloves
  - Wands
  - Heads-up displays (HUDs)
  - Virtual environments such as the NCSA C.A.V.E.
- Java 3D API specifies spatialized vector math support not available elsewhere in the Java™ 2 Platform

Java 3D API: Potential Weaknesses

- Java 3D API hides rendering pipeline details from the developer, a “feature” with sometimes negative consequences
  - Java platform-to-OpenGL bindings may be a better choice for developers needing direct access to the rendering pipeline
- Java 3D API components are heavyweight, which can complicate Swing-based GUI development

Sun microsystems

Java 3D API Package Summary

- Java 3D API is specified in:
  - javax.media.j3d
  - javax.vecmath
- Sun provides some very useful supporting classes and utilities under:
  - com.sun.j3d

Sun microsystems

Java 3D API: Scene Graph Basics

- Java 3D API programs create a tree-like structure of Nodes to represent the world to be rendered and rendering instructions
- Java 3D API scene graphs contain two major branches
  - Content branch: describes the objects to render (how to draw them, color them, arrange them in 3D space, how they should behave)
  - View branch: everything else (placement of user's view in 3D space, ability to move this view interactively, manipulations for stereo viewing, HUDs, etc.)

Java 3D API: Scene Graph Basics

- Java 3D API view branches are typically quite small compared to content branches
- View branches will often contain only a few nodes, while content branches may contain thousands for complicated 3D worlds
- Consequently, many Java 3D API optimizations focus on the content branch

Java 3D API: View Branch Example

- Example06 creates a very simple Java technology-based 3D world
- This world illustrates
  - Using the heavyweight `Canvas3D` component within a `Frame` container
  - Creating the view branch of the scene graph
  - Attaching a `View` to the view branch

Java 3D API: Example06, Canvas3D

```
046     //Title our frame and set its size.
047     super("Java 3D Example06, Basics");
048     setSize(400,300);
049
050     //Here is our first Java 3D-specific code.  We add a
051     //Canvas3D to our Frame so that we can render our 3D
052     //graphics.  Java 3D requires a heavyweight component
053     //Canvas3D into which to render, and in order to instantiate
054     //this component, we first have to access a 3D GraphicsConfiguration
055     //using GraphicsConfigTemplate3D and java.awt GraphicsEnvironment
056     //and related classes.  Note how we use GraphicsEnvironment's static
057     //method getLocalGraphicsEnvironment() to return reference to the
058     //systems GraphicsEnvironment.
059     GraphicsConfigTemplate3D myGraphicsConfigTemplate3D
                        = new GraphicsConfigTemplate3D();
060     GraphicsEnvironment myGraphicsEnvironment
                        = GraphicsEnvironment.getLocalGraphicsEnvironment();
061     GraphicsDevice myGraphicsDevice
                        = myGraphicsEnvironment.getDefaultScreenDevice();
062     GraphicsConfiguration myGraphicsConfiguration
          = myGraphicsDevice.getBestConfiguration(myGraphicsConfigTemplate3D);
063     Canvas3D myCanvas3D = new Canvas3D(myGraphicsConfiguration);
064     add(myCanvas3D,BorderLayout.CENTER);
065
066     //Turn on the visibility of our frame.
067     setVisible(true);
```

Java 3D API: Example06, Constructing the View

```
089     * constructView() takes a Canvas3D reference and constructs
090     * a View to display in that Canvas3D.  It uses the default
091     * PhysicalBody and PhysicalEnvironment (both required to be
092     * set or else the 3D runtime will throw exceptions).  The
093     * returned View is used by constructViewBranch() to attach
094     * the scene graph's ViewPlatform to a Canvas3D for rendering.
095     *
096     * @see constructViewBranch(View)
097    **/
098    private View constructView(Canvas3D myCanvas3D) {
099      View myView = new View();
100      myView.addCanvas3D(myCanvas3D);
101      myView.setPhysicalBody(new PhysicalBody());
102      myView.setPhysicalEnvironment(new PhysicalEnvironment());
103      return(myView);
104    }
```

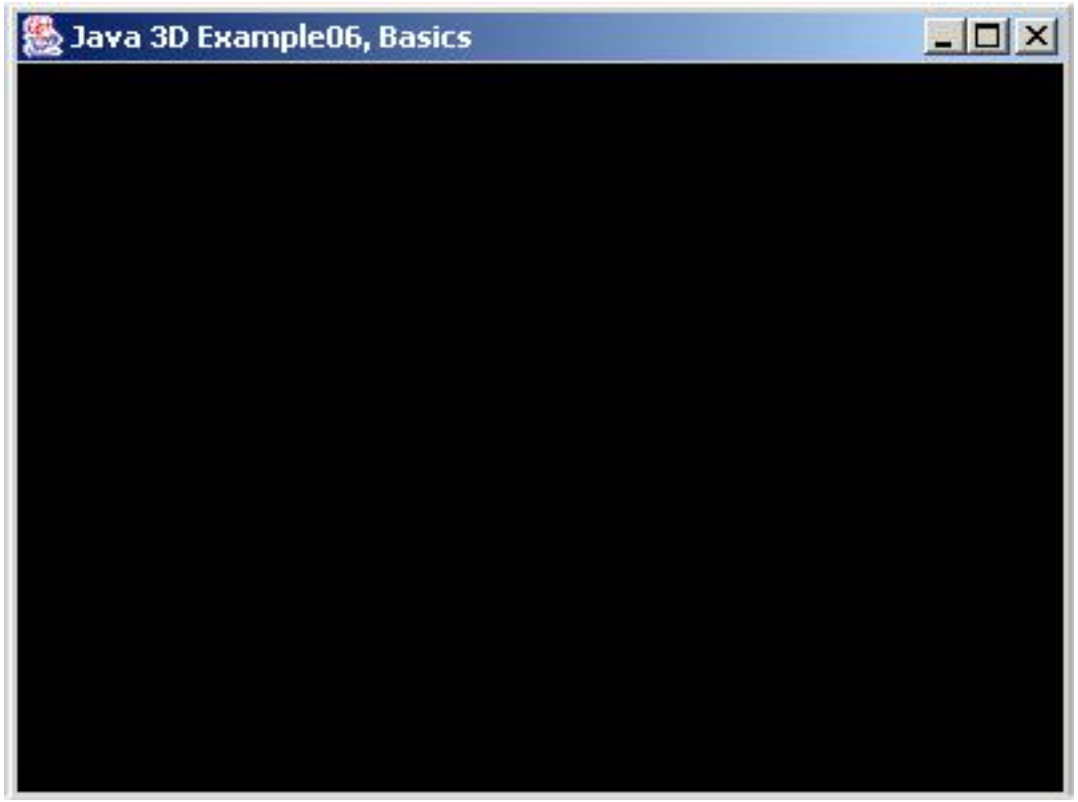
Java 3D API: Example06, Finishing the `View`

```
108     * constructViewBranch() takes as input a View which we
109     * attached to our Canvas3D in constructView().  It constructs
110     * a default view branch for the scene graph, attaches
111     * the View to the ViewPlatform, and returns a reference to
112     * our Locale for use by constructContentBranch()
113     * in creating content for our scene graph.
114     *
115     * @see constructView(Canvas3D)
116     * @see constructContentBranch(Locale)
117    **/
118    private Locale constructViewBranch(View myView) {
119
120      //First, we create the necessary coordinate systems
121      //(VirtualUniverse, Locale), container nodes
122      //(BranchGroup, TransformGroup), and platform which
123      //determines our viewing position and direction (ViewPlatform).
124      VirtualUniverse myUniverse = new VirtualUniverse();
125      Locale myLocale = new Locale(myUniverse);
126      BranchGroup myBranchGroup = new BranchGroup();
127      TransformGroup myTransformGroup = new TransformGroup();
128      ViewPlatform myViewPlatform = new ViewPlatform();
```

Java 3D API: Example06, Finishing the `View` (Cont.)

```
130     //Next, we insert the platform into the transform group,
131     //the transform group into the branch group, and the branch
132     //group into the locale's branch graph portion of the
133     //scene graph.
134     myTransformGroup.addChild(myViewPlatform);
135     myBranchGroup.addChild(myTransformGroup);
136     myLocale.addBranchGraph(myBranchGroup);
137
138     //Finally, we attach our view to the view platform and we
139     //return a reference to our new universe.  We are ready to
140     //render 3D content!
141     myView.attachViewPlatform(myViewPlatform);
142     return(myLocale);
143   }
```

Java 3D API: Example06 Output



- Calling no-op `constructContentBranch()` turns the Java 3D renderer on (sets the scene graph to be live), which renders empty universe

Java 3D API: Content Branch Example

- Example07 adds a more interesting body to the `constructContentBranch()` method of our previous example
  - Uses the heavyweight `Canvas3D` component within a `Frame` container
  - Creates the view branch of the scene graph
  - Attaches a `View` to the view branch

Java 3D API: Example07

```
152   private void constructContentBranch(Locale myLocale) {
153     //We first create a regular 2D font, then from that a
154     //3D font, 3D text, and 3D shape, in succession.  We use
155     //the default constructors for FontExtrusion and Appearance.
156     Font myFont = new Font("TimesRoman",Font.PLAIN,10);
157     Font3D myFont3D = new Font3D(myFont,new FontExtrusion());
158     Text3D myText3D = new Text3D(myFont3D, "Hello Camp");
159     Shape3D myShape3D = new Shape3D(myText3D, new Appearance());
160
161     //We created a new branch group and transform group to hold
162     //our content.  This time when we create the transform group,
163     //however, we pass in a Transform3D to the transform group's
164     //constructor.  This 3D transform has been manipulated
165     //to perform the transformations we desire, which results
166     //in those manipulations being carried out on all children
167     //of the given transform group, in this case, our content.
168     BranchGroup contentBranchGroup = new BranchGroup();
```

Java 3D API: Example07 (Cont.)

```
169     Transform3D myTransform3D = new Transform3D();
170     myTransform3D.setTranslation(new Vector3f(-1.0f,0.0f,-4.0f));
171     myTransform3D.setScale(0.1);
172     Transform3D tempTransform3D = new Transform3D();
173     tempTransform3D.rotY(Math.PI/4.0d);
174     myTransform3D.mul(tempTransform3D);
175     TransformGroup contentTransformGroup
                       = new TransformGroup(myTransform3D);
176
177     //We add our child nodes and insert the branch group into
178     //the live scene graph.  This results in Java 3D rendering
179     //the content.
180     contentTransformGroup.addChild(myShape3D);
181     contentBranchGroup.addChild(contentTransformGroup);
182     myLocale.addBranchGraph(contentBranchGroup);
183   }
```

Java 3D API: Example07 Output

- Our universe is empty no more!

Java 3D API: Utilities Can Make Your Code Simpler

- You may have looked at the set-up code in the first two examples and wondered "Why do we have to make so many redundant calls each time we use the Java 3D API?"
- We do not, if we are willing to use Sun utility classes (or write our own)
- Example08 makes use of Sun's `SimpleUniverse` and `ColorCube`

Java 3D API: Example08, Using SimpleUniverse

```
047     //First, we use SimpleUniverse's static getPreferredConfiguration()
048     //method to set up our Canvas3D and add it to our Frame.
049     GraphicsConfiguration myGraphicsConfiguration
                              = SimpleUniverse.getPreferredConfiguration();
050     Canvas3D myCanvas3D = new Canvas3D(myGraphicsConfiguration);
051     add(myCanvas3D,BorderLayout.CENTER);
052
053     //Then, we instantiate a SimpleUniverse using our Canvas3D,
054     //create our content branch, and add it into the SimpleUniverse.
055     SimpleUniverse myUniverse = new SimpleUniverse(myCanvas3D);
056     BranchGroup contentBranchGroup = constructContentBranch();
057     myUniverse.addBranchGraph(contentBranchGroup);
```

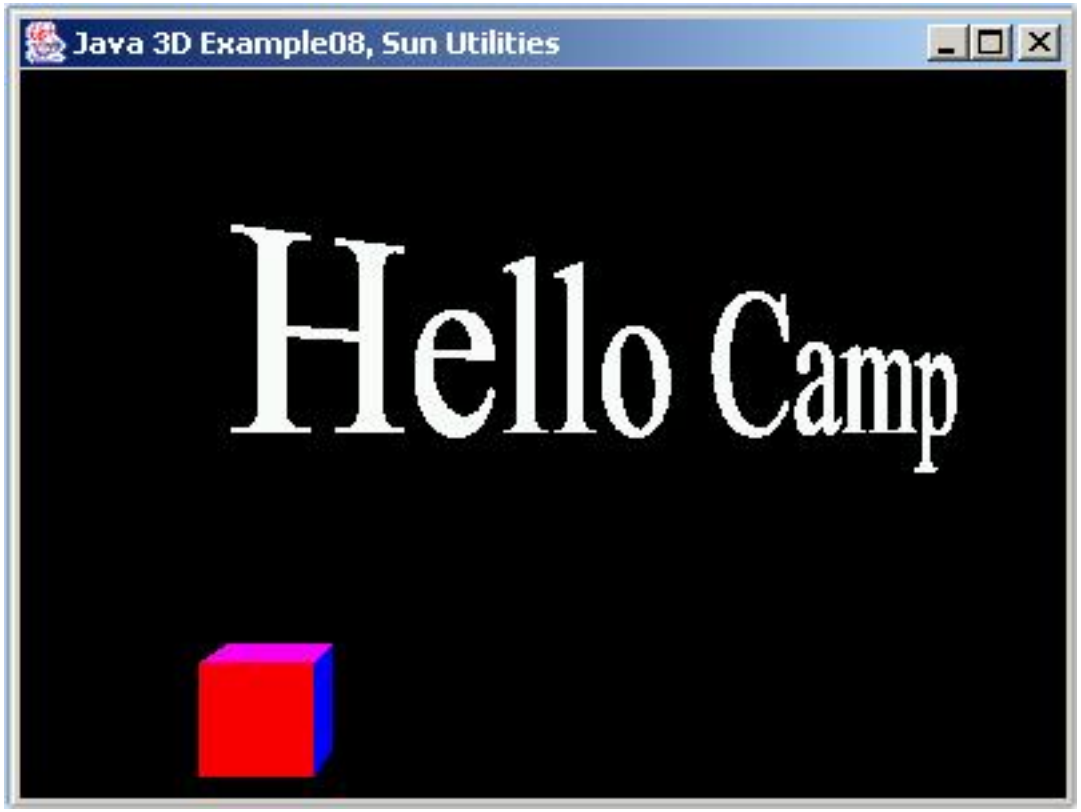
Java 3D API: Example08, ColorCube

```
073     * constructContentBranch() is where we specify the 3D graphics
074     * content to be rendered.  We return the content branch group
075     * for use with our SimpleUniverse.  We also demonstrate the
076     * use of com.sun.j3d.utils.geometry.ColorCube to build more
077     * complicated 3D shapes.
078     *
079    **/
080    private BranchGroup constructContentBranch() {
081      Font myFont = new Font("TimesRoman",Font.PLAIN,10);
082      Font3D myFont3D = new Font3D(myFont,new FontExtrusion());
083      Text3D myText3D = new Text3D(myFont3D, "Hello Camp");
084      Shape3D myShape3D = new Shape3D(myText3D, new Appearance());
085      Shape3D myCube = new ColorCube();
086
087      BranchGroup contentBranchGroup = new BranchGroup();
088      Transform3D myTransform3D = new Transform3D();
089      myTransform3D.setTranslation(new Vector3f(-1.0f,0.0f,-4.0f));
090      myTransform3D.setScale(0.1);
091      Transform3D tempTransform3D = new Transform3D();
092      tempTransform3D.rotY(Math.PI/4.0d);
093      myTransform3D.mul(tempTransform3D);
```

Java 3D API: Example08, ColorCube (Cont.)

```
094     TransformGroup contentTransformGroup
                       = new TransformGroup(myTransform3D);
095
096     contentTransformGroup.addChild(myShape3D);
097     contentBranchGroup.addChild(contentTransformGroup);
098
099     myTransform3D.setIdentity();
100     myTransform3D.setTranslation(new Vector3f(-0.5f,-0.5f,-2.3f));
101     myTransform3D.setScale(0.1);
102     TransformGroup cubeTransformGroup
                       = new TransformGroup(myTransform3D);
103
104     cubeTransformGroup.addChild(myCube);
105     contentBranchGroup.addChild(cubeTransformGroup);
106
107     return(contentBranchGroup);
108   }
```

Java 3D API: Example08 Output



- Note the multi-colored `ColorCube`, and that it is offset in space from the text

Java 3D API: Behaviors

- With the Java 3D API, behaviors are scheduled when the view platform crosses the stimulus bounds, a region of space defined by the programmer
- Bounds are used by the Java 3D runtime to avoid computations for non-visible or inaudible `Nodes`
- Both sounds and behaviors have bounds
- Example09 illustrates basic behavior, adding rotation to previous examples

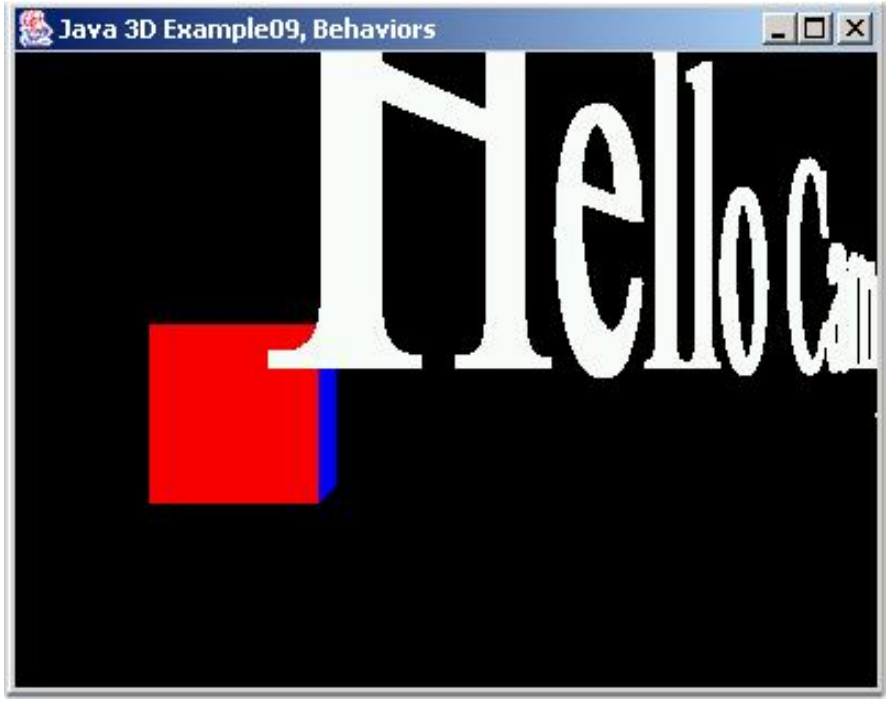
Java 3D API: Example09

```
083     * constructContentBranch() is where we specify the 3D graphics
084     * content to be rendered.  We return the content branch group
085     * for use with our SimpleUniverse.  We have added a RotationInterpolator
086     * to Example03 so that in this case, our "Hello Camp" text rotates
087     * about the origin.  We have also removed the scaling and static
088     * rotation from the text, and the scaling from our ColorCube.
089    **/
090    private BranchGroup constructContentBranch() {
091      Font myFont = new Font("TimesRoman",Font.PLAIN,10);
092      Font3D myFont3D = new Font3D(myFont,new FontExtrusion());
093      Text3D myText3D = new Text3D(myFont3D, "Hello Camp");
094      Shape3D myShape3D = new Shape3D(myText3D, new Appearance());
095      Shape3D myCube = new ColorCube();
096
097      BranchGroup contentBranchGroup = new BranchGroup();
098      Transform3D myTransform3D = new Transform3D();
099      TransformGroup contentTransformGroup = new TransformGroup(myTransform3D);
100      contentTransformGroup.addChild(myShape3D);
101
102      Alpha myAlpha = new Alpha();
103      myAlpha.setIncreasingAlphaDuration(10000);
```

Java 3D API: Example09 (Cont.)

```
104     myAlpha.setLoopCount(-1);
105     RotationInterpolator myRotater =
106                         new RotationInterpolator(myAlpha,contentTransformGroup);
107     myRotater.setTransformAxis(myTransform3D);
108     myRotater.setMinimumAngle(0.0f);
109     myRotater.setMaximumAngle((float)(Math.PI*2.0));
110     BoundingSphere myBounds = new BoundingSphere();
111     myRotater.setSchedulingBounds(myBounds);
112     contentTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
113     contentTransformGroup.addChild(myRotater);
114
115     contentBranchGroup.addChild(contentTransformGroup);
116
117     myTransform3D.setTranslation(new Vector3f(-0.5f,-0.5f,-2.3f));
118     TransformGroup cubeTransformGroup = new TransformGroup(myTransform3D);
119     cubeTransformGroup.addChild(myCube);
120     contentBranchGroup.addChild(cubeTransformGroup);
121
122     return(contentBranchGroup);
123   }
```

Java 3D API: Example09 Output



- The text rotates around in 3D space (rotates counter-clockwise from viewer's perspective, around the left bar of “H”)

Sun microsystems

Summary and Resources

Sun microsystems

What You Should Do Next

- Install the J2SE™ Platform 1.4 SDK and the JMF and Java 3D™ APIs and try out the included examples
- Experiment with 2D, JMF, and 3D to solve your own problems, trying out the other Media APIs as needed
- Enjoy the freedom and possibilities of cross platform media programming by getting started with the Java™ Media APIs today!

Resources

- The Java™ Media APIs homepage links to more information (including specs and more examples) for each API: java.sun.com/products/java-media
- jGuru Java Media APIs FAQ: www.jguru.com/faq/Media
- The Java 3D™ Community site: www.j3d.org
- More Java Media APIs information, howtos, tools, etc., are available now as part of the Sun™ ONE Starter Kit: www.sun.com/sunone/starterkit

Sun microsystems

For more, please visit us at:

www.sun.com/developers/evangcentral

In pursuit of the best software in the universe

- All presentations
- Audiocasts
- Codecamp materials
- Technology briefings
- Code/articles/links/chats/resources

Sun microsystems

Software License

IMPORTANT: The source code and examples listed in these materials are offered under the license at:
http://wireless.java.sun.com/berkeley_license.html

Bill's Brief Bio

Bill Day is a Technology Evangelist at Sun Microsystems.

Bill moderates jGuru's *Java™ Media APIs FAQ* and speaks frequently on wireless technology, system security, and multimedia programming. Bill also writes about software development for numerous publications and teaches Java and Wireless development as an extension instructor for the University of California Berkeley.

More information is available from Bill's site:
www.billday.com