Java™ Foundation Classes: Now and the Future

A White Paper

JavaSoft
2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
1-800-JAVASOFT
512-434-1591

April 1997

Java™ Foundation Classes: Now and the Future

A White Paper

JavaSoft
2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
1-800-JAVASOFT
512-434-1591

April 1997

Contents

Java™ Foundation Classes: Now and the Future

Executive Summary

In 1995, Java™ technology shook the World Wide Web with a network-centric, object-oriented language that provided client side processing for otherwise static pages. Many Web developers used Java to animate sites and loved it. Java's graphical user interface library, the Abstract Window Toolkit (AWT), fulfilled a key role in the creation of dynamic Web pages.

Early adopters of Java technology have recognized its significant productivity advantages. Java is an excellent implementation of object-oriented technology which makes it easier to learn. Java enables any application to be portable to any Java-enabled platform without the additional cost of development or maintenance—no extra effort is expended to create the port. This is a huge advantage for developers and system architects that use the Java development platform.

Java has evolved quickly and now transcends the browser, its initial runtime environment. Today, businesses worldwide are standardizing on Java as their primary development and deployment platform. The Java Virtual Machine (JVM) is rapidly being incorporated into operating systems and hardware devices to provide a common meta-platform for applications.

The Java Developer's Kit (JDK™) release 1.1 introduces the first installation of the most significant cross-platform graphical user interface technology since the advent of windowing systems: the Java Foundation Classes (JFC).

Java Foundation Classes extends the original AWT by adding a comprehensive set of graphical user interface class libraries that is completely portable and delivered as part of the Java platform. In addition, JFC will also include many of the key features found in Netscape's Internet Foundation Classes. Since the JFC is core to the Java platform, it eliminates the need to download special classes at runtime and is compatible with all AWT-based applications.JFC's continually evolving new features include a rich suite of high-level components and services that are fully and uniquely cross-platform compatible, and offer significant performance improvements. With the JFC, developers can create and deploy large scale, mission-critical intranet , Internet and Crossware applications. And because Java is an open, standard technology, a broad complement of third party tools and components are available to enhance application development.

The JFC raises the bar for GUI functionality in Java and delivers a rich API with a growing set of components and services to develop commercial-grade applications.

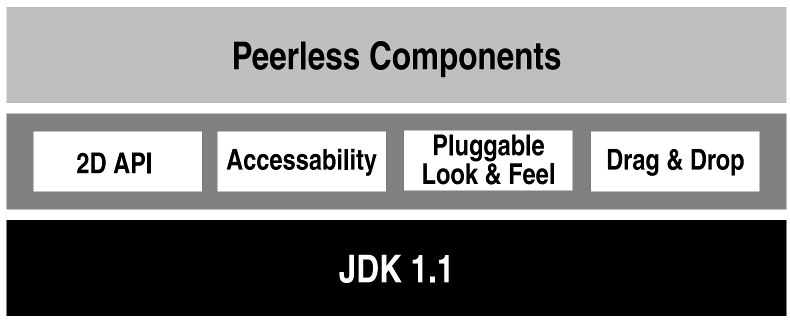

Figure 1 The Java Foundation Classes

The Java Foundation Classes includes many new, easy-to-use and sophisticated features that are designed together to offer the following key advantages over other frameworks:

- JFC is core to the Java platform and reduces the need for bundled classes.
- All new JFC components are JavaBeans™.
- No framework lock-in. Developers can easily bring in other third party components to enhance their JFC applications.
- Components are cross-platform.
- Enhanced services promote development of feature rich applications.
- JFC subclasses are fully customizable and fully extendible.
- JFC subclasses are fully customizable and fully extendible.

History of Java Foundation Classes

The Abstract Window Toolkit (AWT) provided developers with a rudimentary library for building applications and applets. Applets are Java applications that are executed from inside a browser instead of being launched and run from the native operating system. Designed for simple, web-centric tasks, developers encountered limitations with the AWT when attempting to create modern, sophisticated client applications.

Although the AWT was limited in scope, it did offer two important features for all applets and applications:

- 100% portability from a single set of source code
- Assumed a native look and feel on the deployment platforms

The AWT delivered on the promise of a standards-based platform that adapted to the user desktop. It also provided a good starting point for graphical Java development with room for improvements, some evolutionary and some revolutionary.

The Challenge of a Portable GUI Library

The technical issues surrounding cross-platform compatibility have plagued software developers for years. Many vendors have attempted to solve this seemingly simple problem by making applications independent of their windowing systems. Most implementations involved "lowest common denominator" approachesand used proprietary libraries. None offered an open solution. With the AWT, Java solved this portable GUI environment problem in the most elegant and simple way possible.

There are many different approaches to implementing a portable GUI class library or environment. A common approach is to layer an API or class library on top of the native toolkit. Alternatively, other portable GUI environments contain a single toolkit and then emulate the native look and feel on each platform. The first approach has the desired look and feel, but applications often do not work consistently across platforms. The later approach enables applications to work consistently but their emulation never fully captures the correct look and feel on all platforms.

The AWT uses the layered toolkit model where each Java component creates a native component. The JDK refers to this as the “peer” model. Each Java component class “wraps” the peer or native implementation. This can increase complexity because many native components maintain their own state. More challenges arise at the toolkit level. Each native toolkit has a completely different event model and contains a distinct set of native components.

For users, differences in native appearance can have a radical and negative effect on the look and behavior of an application. Problems can arise when applications designed and developed for one platform are deployed on a different windowing system. Components such as the scrollbar can exhibit inconsistent behavior across platforms. Changes to minimum sizes, shading and colors, and fonts can result in unusable user interfaces for applications that are otherwise healthy. Since cross platform capabilities are one of the compelling reasons for developing with Java, the cross platform capabilities of the GUI had to be improved.

AWT 1.0 — Write Once, Run Anywhere™

AWT 1.0 had several strengths and weaknesses. It’s biggest strength was that with it, an application/applet could be written once and run anywhere—a single set of source code, 100% portability. No other toolkit or language has been able to match AWT’s breadth of deployment platforms while supporting native look and feel.

Many seasoned GUI developers have complained that the AWT’s peer model was too restrictive in rendering and event handling, and that because of this, it was difficult to extend or override different aspects of the component. It was also prone to problems if the idiosyncrasies of a platform were not implemented properly. Generally speaking, the peer model was viewed as unacceptable if Java was to truly act as a metaplatform that would guarantee cross platform support.

Nevertheless, the peer model played an important role in the acceptance of the AWT. It enabled Java’s AWT to be brought to market quickly and provided true native look and feel. In fact, browsers are moving users toward a more universal look and feel on all platforms and soon, users will expect this uniformity. However, in 1995 the market was much more dogmatic about the appearance of applications and would have almost certainly rejected the AWT if it had not provided native look and feel.

In retrospect, AWT 1.0 was a good initial step that enabled applet development. It provided the initial foundation upon which further releases of the AWT could build and eventually improved. The API was simple and clean, and enabled developers to come up to speed quickly. The AWT shielded developers from each platform's idiosyncrasies and yet was 100% portable with a native look and feel. Finally, the AWT enabled the early Java adopters to deliver applets and applications on any Java enabled platform with one set of source code and no "porting".

Internet Foundation Classes

To address the needs of developers who wanted to create commercial applications that behave and appear identical across all platforms, Netscape created the Internet Foundation Classes (IFC). While IFC was a robust framework which delivered many important functionalities to the development community, Java developers demanded a single industry standard solution.

As a result, Sun, Netscape™, and IBM joined force to create Java Foundation Classes which provides a single GUI component and services solution. Java Foundation Classes builds on top of the existing AWT by integrating the best of new Sun technologies and Netscape's Internet Foundation Classes.

JFC — A Comprehensive Set of Classes and Services

With JDK 1.1, the web-centric AWT GUI toolkit becomes part of a comprehensive set of GUI classes and services called The Java Foundation Classes. The Java Foundation Classes, a superset of the AWT, provides a robust infrastructure for creating commercial quality intranet and Internet applets and applications. The JFC is the result of developer and end user feedback and provides a significant advance in the development of client applications written in Java.

The JFC released in JDK 1.1 provides a wide range of new features that enable Java developers to be more productive while delivering applications that are fully integrated into the desktop environment. The JFC contains a powerful, mature delegation event model, printing, clipboard support, a lightweight UI framework and is 100% JavaBeans compliant. The new capabilities of JFC make

Java a platform that will support the most sophisticated enterprise applications. Development of commercial applications using Java is now both possible and attractive.

Java and the AWT freed developers from being tied to a specific or proprietary platform for GUI application development and deployment. Today, the JFC provides the core set of classes and services enabling applications to scale from the desktop to the enterprise and across multiple platforms. Java developers will find the additions and improvements to the toolkit greatly enhance their ability to deliver 100% Java applications quickly and provide reliability and cross-platform consistency.

The release of JFC with JDK 1.1 is only a starting point. JFC will be extended to encompass an even wider range of components and services which will support the development of even richer Java applications. The JFC strategic roadmap intends to build on this solid foundation with new functionality that will continue to deliver a new generation of applications to a new era of computing.

Current JFC Features

GUI developers expect baseline functionality to create professional quality applications. The AWT, while best suited for applet development, provided little integration into the desktop environment and even less functionality for creating large scale applications. JFC, introduced in JDK 1.1, delivers a more robust framework for GUI development. It also delivers the baseline components and frameworks that developers expect from the Java platform.

With JFC, JavaSoft placed a great emphasis on quality, fixing many of the platform inconsistencies which existed in the AWT. While it is recognized that the peer model of the AWT was limited in its scope, it was necessary to correct existing problems and to help customers deploy existing applications reliably. In particular, the Win32 native implementation was 100% re-written in order to provide a more robust base for the JFC going forward. Although newly introduced, JFC is based on a mature, consistent, and widely known set of classes and services.

Current features of the Java Foundation Classes are:

- JavaBeans Compliant
- Lightweight UI Framework

- Delegation Event Model
- Printing
- Data Transfer/Clipboard
- Desktop Colors Integration
- Graphics & Image Enhancements
- Mouseless Operation
- Popup Menu
- ScrollPane Container

JavaBeans has played an important role in the JFC. The JavaBeans specification defines a standard component architecture for visual and non-visual components. All JFC components in JDK release 1.1 are JavaBean compliant, which means they have a consistent API and a standard event handling mechanism. This standard property and event model lends itself well to component reuse and interoperability. JavaBeans also ensures easy integration of standard components and frameworks with RAD tools and GUI builders.

As described earlier, the peer model has certain limitations. While native look and feel is still important, a universal look and feel is also becoming increasingly more desirable. The Lightweight UI Framework enables components to be peerless, or lightweight, and completely written in Java. This new framework will enable component developers to create and deliver a wide range of third party components that will work consistently across platforms while remaining completely portable. The lightweight UI also enables Java developers to easily create stunning user interfaces.

The new event model is especially well suited to visual development environments and development of distributed and multicast applications. The new delegation event model in the JFC extends the previous single method implementation of the AWT. Before large conditional statements were required to determine an event's type or origin. Now, events are class specific and can be "sent to" or "delegated" directly to the object that can handle the request. This provides a large degree of flexibility when handling the many different types of events generated by a GUI application.

Operating system vendors are currently integrating Java's Virtual Machine directly into the OS which will shift deployment from webcentric browsers to applications, provide a common metaplatform from IBM MVS to Microsoft

Windows, and provide a more consistent, flexible, and reliable operating environment than a browser. Once this is accomplished and applications based on release 1.1 are in operation, users will no longer have to view the future of the desktop windowing environment through a single vendor's eyes. Innovation will expand as the time-to-market for application decreases.

In summary, JavaBeans, lightweight components, and the delegation event model provide a stronger infrastructure which enables larger and more complex applications to be built more easily and in less time. The remaining features such as popup menus, desktop colors, mouseless operation, and printing all allow Java applications to be seamlessly integrated into the native windowing environment. Collectively, these additions to the core Java GUI toolkit are designed to result in feature-rich, high-performance applications.

100% Pure Java™ — Ensuring Portability

It is important that both applications and applets should be written completely in Java to preserve cross-platform capability. Today, as customers mix and match different desktop systems developers cannot assume a single deployment platform. It's also important that Java applications and applets do not rely on any proprietary technologies. Some operating system vendors attempt to "capture" their developers by providing proprietary technologies as a part of the Java development tools, thus limiting the deployment of created Java applications or applets to a single platform. Developing exclusively using the native toolkit, or "100% Pure Java" frees the developer from these concerns and lets them take advantage of the full benefits of the Java platform. Of course, the native toolkit must support all the capabilities that developers require and take full advantage of whatever native environment they are on.

"100% Pure Java" means the application is written completely in Java and does not rely on any external class libraries or native code. Third party technologies can be used but their technology must be part of the application or applet download in order for the software to be "100% Pure Java." This works well for third party components that are typically small in size with few classes. It becomes a problem with large third party frameworks.

Although there is a growing third party market for additions to the Java platform, an obvious need arose in the Java development community. Initially, AWT was intended to support only a limited set of GUI components. It was expected that the market would provide the higher-level components. The Java development community has overwhelmingly asked for the additional components to be delivered as part of the JDK. Why? So applications can more easily be “100% Pure Java.” Java developers do not want to deliver these additional components along with their applets and applications. With JFC, much of this additional functionality has been incorporated into the Java platform, thus improving overall performance and appeal of Java applications and applets.

Java — A Commitment To Open Standards

Recently, other companies have announced or delivered specifications and class libraries to the Java development community. Although their apparent enthusiasm for the Java platform is certainly understandable, such offerings can cause confusion in the marketplace. JavaSoft has endeavored to cultivate relationships with other technology providers in order to provide an open standard for all Java developers. This collaboration is evident in many of the specifications now available for review. JFC represents the demands of Java stakeholders and is a high quality development environment that makes high performance, network-centric applications a reality.

It is important to remember that all specifications from JavaSoft have been made available to the larger Java development community for comment and do not reflect the views or position of a single vendor. Once these specifications have been implemented the technology will become part of the JDK and will exist on every Java platform. This is why it is important for Java applications and applets remain “100% Pure Java.”

JFC — Expanding On A Solid Foundation

The JFC represents a significant introduction of new functionality to the Java development environment. The release of these classes with JDK 1.1 marks the beginning of major enhancements that will continue to improve the completeness and functionality of Java GUI applications.

Additions to the Java Foundation Classes will soon incorporate several features that further enhance a developer’s ability to deliver scaleable, commercial-grade applications. These new features will be made available to developers as they are completed over the next few months and then rolled into the next release of the JDK. New features will include:

- Drag and Drop

- New High-Level Components
- Pluggable Look and Feel
- 2D API
- Accessibility Features for the Physically Challenged
- Additional new features continually being added

Drag and Drop

The Drag and Drop specification is available in the JDK 1.1 documentation and will soon be released as an update to the JFC. Drag and Drop (D&D) functionality as part of the core JFC class library means it will be available on all Java platforms in a consistent and supported fashion. It will significantly improve application interoperability by enabling D&D between Java applications and non-Java applications.

The generic, platform-independent implementation of D&D will enable objects to be dragged and dropped from one application to another. This update will support D&D between two Java applications, but more importantly, it will support D&D between a Java application and a native application. This means a user running the application on UNIX will be able to D&D to a Motif application and then run the same code on Microsoft-Windows and D&D to native application running there. D&D support makes it much easier to introduce Java applications to the enterprise.

New Java Foundation Classes Components

Working closely with Netscape and IBM, JavaSoft will soon release an expanded set of components written in 100% Pure Java that will enable applications to be more fully integrated into the native environments. These new high-level components will be available on all platforms as part of the standard Java platform, thereby eliminating the need to download extra classes. In addition, new JFC components will carry two important attributes: all components will be both lightweight and peerless, thus facilitating a customizable look and feel without relying on native windowing systems or adding system overhead, and improving the deployment of applications will be simplified.

This comprehensive set of new high-level components will allow developers to create sophisticated professional-quality applets and applications using core JDK 1.1 functionality. The following components will be included with JFC:

- Tree View
- List View
- Table View
- Toolbar
- Pane Splitter
- Tabbed Folder
- List
  - Multi-column
  - Supports Images
- Progress Bar
- Slider
- Styled Text
  - Support import/export of HTML and Rich Text Format
- Font Chooser
- Color Chooser
- File Chooser
- Custom Cursors
- Tool Tips
- Generic Button and MenuItem
  - Support Images, Check marks
- StatusBar
- SpinBox
- ComboBox
- Drop down ComboBox
- Drop down ListBox
- Composable Button

- Multimedia Controls
  - MovieSlider
  - AudioController
  - MovieController
  - Properties
  - AudioGauge
  - MidiPanel

Pluggable Look and Feel

Native look and feel was once the mantra of cross-platform developers. Since all applications on a platform had a common look, all new applications were assumed to require that user interface. Or so it seemed until World Wide Web came along. Now, users are familiar with and excited by the rich and unique user interfaces delivered through browsers. The rigor with which user interfaces were designed in the past has been replaced with the requirement for effective and reliable clients.

Pluggable look and feel will increase the reliability and consistency of applications and applets deployed across platforms. The "pluggable" look and feel will provide an easy yet powerful mechanism for individualizing an application's visual personality without having to subclass the entire component set. This will increase user acceptance of network computer (NC) replacement of PCs as users will select an operating environment they are comfortable with.

To further enhance a user's visual experience, an entire application's GUI will be able to switch from one look and feel to a different one at runtime. This feature will give users the ability to switch without restarting the application. This is significant since users will want to work in a GUI that is familiar to them, even if they are using a network computer.

2D Graphics API

The existing graphic capabilities of Java will be extended with new classes. New graphical capabilities will promote the development of visually rich applications. The Java 2D API extends the JFC's strengths by enabling developers to incorporate high-quality graphics into their applications and applets. The Java 2D API will extend the java.awt and java.awt.image core class libraries. By extending the existing classes, the Java 2D API maintains compatibility for existing programs and allows programs to seamlessly integrate the features provided by JFC. This is one of many evolutionary additions to the Java platform which ensures that existing code co-exists and benefits from these new enhancements.

The Java 2D API provides a subclass of the AWT Graphics class, Graphics2D, with additional features for specifying fancy paint styles, defining complex shapes, and controlling the rendering process. The Java 2D API treats paths, text, and images uniformly; they can all be rotated, scaled, skewed, and composited using the new Graphics2D class.

The Java 2D API will:

- Enable the path to be specified to define complex shapes.
- Enable text to be drawn, transformed, used as a clipping path, and composited just like any other graphic element.
- Provide text layout facilities that handle most common cases, including text strings with mixed fonts, mixed languages, and bidirectional text.
- Provide a full range of features for handling images with several new classes including: BufferedImage, Tile, Channel, ComponentColorModel and ColorSpace.
- Enable the control of how graphics primitives are rendered by specifying a set of attributes that allows characteristics such as the stroke width, join types, and color and texture fills.

This means better performing, more sophisticated visual applications for scientific, engineering, and business users.

Accessibility Features for the Physically Challenged

JFC introduces new capabilities to the Java platform that deliver benefits to end users. Many physically challenged users are prevented from using applications because developers did not integrate with the necessary accessibility functionality. The JFC Accessibility API enables Java applications to scale to encompass the physically challenged users. Two of the most important features are Screen Readers and Screen Magnifiers.

The Screen Reader creates an off screen representation of the GUI components enabling the information to be provided via a text to speech and/or a Braille terminal. The user can then interact with the GUI through the alternate computer input and output device.

The Screen Magnifier enables the user to adjust the magnification of the screen from 1 to 16 times the normal size. Event tracking and screen content knowledge enable the user to more easily navigate the interface. Font smoothing and text highlighting act in concert to create a clearer picture which renders the content more easily read and understood.

Accessibility features will be easily integrated into existing Java applications and will ensure an even broader audience for information in a networked age.

Current JFC Features Delivered in JDK 1.1

An In depth Look

JFC provides significant new capabilities for Java developers. With the JDK 1.1 release, JFC embraces the new JavaBeans component model, introduces a new high-performance UI framework, improves event handling, enables printing support, supports keyboard navigation, and introduces popup menu and scroll pane components. JFC delivers a wish list of features that make Java the most compelling development platform available.

JavaBeans Compliant

JavaBeans is an architecture and platform neutral API for creating and using dynamic Java components. JavaBeans enhances the Java platform by allowing richer, more dynamic interaction. JavaBeans allow developers to define independent components that can be used and re-used in a variety of combinations to compose new applications. JavaBeans will be supported inside browsers and in standalone applications.

JavaBeans components can be GUI widgets, non-visual functions and services, applets and or scale applications. Each of these components can be built by different developers at separate times. JavaBeans components do not need to be part of the same application build. Instead, they communicate dynamically.

By delivering JFC as JavaBeans, developers will be able to create Java applications and applets that interoperate with legacy applications, can be created with visual development environments, and can be distributed easily across the network.

Lightweight UI Framework

In AWT release 1.0, peer-based components (Button, Label, etc.) could not be easily extended or have their look and feel overridden. Many developers created their own components by deriving new ones from java.awt.Canvas and specialized containers were derived from java.awt.Panel. The problem with this approach was that each Canvas or Panel derived component was “heavy-weight”, meaning it required a native window which is rendered opaquely. Opaque windows can cause problems for those wanting to layer components on top of each other, since they completely cover the components behind them. Additional overhead occurs when developers create their own classes—the applet or application must download additional classes to perform what should be a standard task. JFC solves these problems by introducing a lightweight framework that radically improves the performance and appearance of Java applications.

Lightweight components are completely transparent because they do not require a native window. This means that they can be written entirely in Java and do not carry with them any overhead from the native windowing system. Lightweight components also provide a consistent look and feel across all platforms. It also allows for a new class of GUI components such as tool tips without significant overhead.

It is remarkably easy to take advantage of this new framework in existing applications and applets. To convert existing Canvas and Panel based components into Lightweight components, a developer needs only to change the superclass from Canvas to Component and from Panel to Container. When the paint() method is called, all rendering will be performed within the parent’s Graphic context and transparent areas can be left unrendered thus enabling the background to show through.

Advantages of the new lightweight UI framework are:

- improved runtime performance for GUI applications
- improved “time to load” for applets with fewer classes to download

- easy conversion of existing AWT components to JFC lightweight components

Delegation Event Model

The event model in AWT 1.0 was quite simple. All events were passed up the inheritance hierarchy forcing objects to catch and process events by subclassing and overriding either the handleEvent() or action() methods. Complex if-then-else conditional logic was required in the top level object to determine which object triggered an event. This was not scalable and was ill-suited to high-performance distributed applications.

With JFC, the callback-style delegation event model is elegant yet powerful. Events are identified by their class instead of their ID and are either propagated or delegated from an event "Source" to and event "Listener." Only objects interested in a particular event need deal with the event and no super-event handler is required. This is also a better way of passing events to distributed objects.

The new event hierarchy is defined by the package java.awt.event. Each derived class from java.util.EventObject is now unique because of the data it holds. There are no longer public data fields, all event data is accessed through a standard set/get JavaBeans compliant interface. However, some derived Event classes still contain an ID to help objects distinguish between events within a specific group of events.

The real power behind the new model is the enabling of an object to delegate or "fire" an event to a specific listener or set of listeners. When a source object wishes to delegate a specific event type, it must first define a set of methods that enable the listener object(s) to register with the source. These methods take the form of set<EventType>Listener for single-cast and/or add<EventType>Listener for multi-cast delegation.

Any object desiring to be a "Listener" of a specific event type must implement the <EventType>Listener interface for that event type. The "Listener" interface is typically defined by only a few methods, which makes it easy to implement the interface. For example, a new Lightweight component may wish to send an "ActionEvent" when the user clicks on the component. The creator of the component would define an event registration method call addActionListener() which enables any object implementing the ActionListener interface to register as a consumer of ActionEvents.

Additionally, a new language feature called “Inner Classes” has been added in JDK 1.1 which makes creating listener objects much more convenient. With Inner Classes, you can create a listener class on the fly within the body of another class.

The following example uses Inner classes to hook the action of pushing a button to closing a Frame:

```
public class MyFrame extends Frame {
   public MyFrame(String title) {
    super(title);

    Button button = new Button("Close");
    ActionListener action = new ActionListener() {
     public actionPerformed(ActionEvent e) { close(); }
    };
    button.addActionListener(action);

    add("Center", button);
    pack();
   }

   public void close() {
     hide();
   }
  }
```

Printing

Until now Java applications have not been able to send text or graphics to a printer. Native code could be used to print text but then the application/applet was no longer portable, which did not solve the problem of printing graphics. This limitation has made it difficult for Java applications to be truly integrated into the native environment. JFC eliminates these barriers by introducing printer support that can be easily integrated into existing applications and makes the Java platform ready for a wide range of commercial software applications.

JFC has made it simple to add printing to any Java application or applet. To send content or graphics to the printer, a call is made to a Frame object's paint() method with a new type of Graphics context. This special context is derived from the Graphics class and implements the java.awt.PrintGraphics interface. It is obtained from the newly implemented PrintJob class.

The PrintJob class encapsulates all the printing functionality. A call method call to getPrintJob in the java.awt.Toolkit package displays a platform specific dialog, allowing the user to enter typical printing information such as, print name, number of pages, etc.

The returned PrintJob object contains the Graphics context which can be used to call a component's paint() method. This sends all the rendering information to the printer. The underlying JFC implementation takes care of translating those calls to the appropriate print device. Sometimes it is necessary to print the entire GUI hierarchy; this can be accomplished with a call to printAll().

The following code is an example of a method that handles a print request:

```
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("print")) {
      PrintJob pjob = getToolkit().getPrintJob(this,
                "Printing Test", null);

      if (pjob != null) {
        Graphics pg = pjob.getGraphics();

        if (pg != null) {
          canvas.printAll(pg);
          pg.dispose(); // flush page
        }
        pjob.end();

      }
    }
  }
```

Data Transfer/Clipboard

JDK release 1.1 now has full clipboard support. It enables dynamic data types to be created, registered, and transferred from both within and across process space boundaries. This means that any object can be transferred within an application and also to other applications. It also supports data transfer between Java and non-Java applications by enabling access to the “system clipboard.” Java applications can now be easily rolled out to endusers with a high degree of interoperability with existing applications. This increases the overall acceptance of the Java platform.

The new API is centered around the “transferable” objects. A transferable object must be able to provide a list of formats, called “data flavors”. It must also be able to return the data (in the form of an Object reference) when requested in a particular flavor. The data transfer architecture introduced in 1.1 is a key enabling technology for the upcoming drag and drop functionality.

Desktop Colors

An important aspect of an application’s integration into the desktop environment is its ability to adhere to the system color scheme, such as those offered by Windows95 or CDE. Although some applications achieve a unique look or personality through a distinct set of colors, most applications change their colors when a user selects a new desktop scheme. Developers will need to implement a new class in their applications, but Java applications will now be further indistinguishable from native applications.

In previous releases of the AWT this was not possible. JFC in JDK release 1.1 introduced a new class java.awt.SystemColor to handle the dynamic changes in desktop color scheme. The SystemColor class is derived from java.awt.Color and defines “symbolic” colors for backgrounds highlights, shadows, etc. The following code demonstrates how a lightweight component can use the symbolic colors to render itself:

```
// draw highlight left and top
g.setColor(SystemColor.controlLtHighlight);
g.drawLine(0, 0, 0, mySize.height);
g.drawLine(0, 0, mySize.width, 0);

// drawbackground
g.setColor(SystemColor.control);
g.fillRect(1,1, mySize.width-2, mySize.height-2);
```

```
// draw shadow bottom and right
g.setColor(SystemColor.controlDkShadow);
g.drawLine(0, mySize.height-1, mySize.width-1, mySize.height-1);
g.drawLine(mySize.width-1, 0, mySize.width-1, mySize.height-1);
```

Graphics & Image Enhancements

The JFC enhances the graphical capabilities available for creating truly high-performance financial, business, and engineering vertical applications.

While certain problems did exist with graphics handling in AWT, these have been corrected with the JFC. For example, in AWT 1.0 the graphics clipping area could only be set to an ever decreasing area. Developers found themselves creating a temporary graphics object, setting a clipping area, calling the drawing methods and then disposing of the object. The JFC adds a new class "Shape" and several new clipping methods, most important of which is setClip. It allows developers to set the clipping area to any size and does not maintain a relationship to the previously set clipping area.

The AWT 1.0 did not offer an easy way to render a subset of an image or a mechanism for flipping the image horizontally or vertically, a typical requirement when working with labels. JFC corrects this with addition of two new drawImage methods. Each drawImage method requires a source and destination rectangle definition and uses the pixels from the original unscaled image to draw into the Graphics object. The following line of code demonstrates the scaling the 100x100 pixel area down to a 50x50 area:

```
g.drawImage(myImage, 10, 10, 110, 110, 0, 0, 50, 50, this);
```

Creating images on the fly is much more flexible and offers better performance than retrieving images from files. In release 1.0 an image could be created from an array of pixels. The pixels were converted into a screen representation and as long as the image was drawn at the same scale it ignored any changes to the original array. Several new methods have been added to the MemoryImageSource class giving the developer a finer grain of control and easier animation capability.

The final area improved in the JFC with respect to image handling came directly from the request of developers input. Several enhancements were made to the PixelGrabber class. This class now enables developers to get the original ColorModel for the image, retrieve the size of the buffer needed for the image, and start and stop grab operations before the image is loaded.

Mouseless Operation

Two important aspects of a GUI are second nature to most users: keyboard navigation and accelerators on menus. For data entry applications, keyboard traversal is a requirement. It is common to use the "tab" key to move from component to component in a standard user interface, and the "control" keys to perform accelerated actions or short cuts, like <control>C to copy text. JFC based applications now take advantage of keyboard traversal transparently.

Components can now be "tabbable", they can request and transfer focus, and can be notified when they gain or lose focus. To better support menu short cuts a new class, java.awt.MenuShortcut has been added. Additional methods and constructors have been added to the MenuItem and MenuBar classes to handle the new class. The short cut keys apply to the "command" key on the Macintosh and the <control> keys on UNIX® and Microsoft-Windows.

The benefits of mouseless operation are:

- Ideally suited for data entry and operational applications typical of terminal-based applications
- Ensures higher user acceptance of Java applications
- Supports power user navigation

Popup Menu

Popup menus (also known as "context menus") are now a very common UI element in modern GUIs (Win95, CDE, etc.). The JFC introduces a new component that makes it easy to enable the creation and display of popup menus.

Popup menus are created by simply allocating a new PopupMenu object and adding MenuItem objects to it, much like you would create a MenuBar menu. When the popup event is fired (right-button click on MS Windows, middlebutton click on UNIX) a single method is called to assign the popup menu to a component and display it. This enables a single popup menu to be used by many different components. Here is a simple one item popup menu:

```
// Create and add the popup
popup = new PopupMenu("Edit");

mi = new MenuItem("Cut");
mi.addActionListener(this);
popup.add(mi);

add(popup); // add popup menu to applet

enableEvents(AWTEvent.MOUSE_EVENT_MASK);
.
.
.
// The activation code
public void processMouseEvent(MouseEvent e) {

if (e.isPopupTrigger()) {
popup.show(e.getComponent(), e.getX(), e.getY());
}
super.processMouseEvent(e);
}
```

Popup Menus will enable :

- user control of object attributes at runtime
- easier application navigation
- better enduser acceptance of applications

ScrollPane Container

In the AWT 1.0, the task of implementing all scrolling behavior is left to the developer. Only a basic Scrollbar class is provided which must be managed by the application, meaning the application must catch the scrollbar events and then take the appropriate action to update the contents being scrolled. This made applications slow and unreliable across platforms. This was a problem for virtually any scrolling function.

Not only is this a general burden to developers who are accustomed to better support for this in toolkits, but it is also a serious performance problem, since there is a round trip (native->java->native) for each individual scroll event that occurs, and the application must respond to the event and move its contents using slower Java draw/move operations. This is particularly noticeable during the event-intensive scroll-drag operations. The result was slow applications prone to flickering and unreliable actions.

To resolve this problem, a ScrollPane class has been added to the JFC. ScrollPane provides a container that implements automatic scrolling for a single component child and supports three modes for its scrollbars: “when needed”, “always”, and “never”. The introduction of the ScrollPane container greatly simplifies the task of displaying information in a fixed area.

Summary

The Java Foundations Classes delivered with JDK 1.1 casts Java GUI development in a new light. The JFC is a complete GUI toolkit that dramatically extends the original AWT with a comprehensive set of classes and services.

The JFC is a scalable, robust and open technology that enables developers to create and deploy commercial-grade intranet and Internet applications. Even as components and services grow, JFC promotes ease of use and facilitates rapid application development. JFC is delivered as core Java technology, which means it is available on all Java platforms. This results in faster application downloads, more reliable applications, and simplified application deployment.

The JFC will continue to expand with plans in-process to include a rich complement of high-level components that will enhance the user’s visual experience and improve productivity. New application services are slated for JFC that will further integrate Java applications into the desktop environment.

- Drag and Drop
- New High-Level Components
- Pluggable Look and Feel
- The Java 2D API
- Accessibility Features for the Physically Challenged

The Java Foundation Classes empowers developers on all platforms, and brings the future of portable GUI development here today. Write Once, Run Anywhere.

JavaSoft
2550 Garcia Avenue
Mountain View, CA 94043
1-800-JAVASOFT
512-434-1591

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000

WPR-0002-01

Printed in USA