

» [search tips](#)

[APIs](#) [Downloads](#) [Technologies](#) [Products](#) [Support](#) [Sun.com](#)

Article

Personal Basis Profile vs. Personal Profile: What's the Difference?

 [Print-friendly Version](#)

by Eric Giguere

May 2003

For a long while, all the excitement in the Java 2 Platform, Micro Edition (J2ME) was centered on the Connected Limited Device Configuration (CLDC), released in final form in May, 2000. The first CLDC-based profile, the Mobile Information Device Profile (MIDP), followed two months later. MIDP garnered much interest in the Java community because it defined a new application model, as well as classes for user interface and persistence. In other words, it provided a complete environment for deploying and running interactive applications.

J2ME's other configuration, the Connected Device Configuration (CDC), was not finalized through the Java Community Process (JCP) until almost a year later, in March, 2001. The first CDC-based profile, the Foundation Profile (FP), was released at the same time. Unlike MIDP, however, FP does not provide classes for building interactive applications. Consequently, the CDC/FP combination has not generated as much excitement in the Java community.

Recently, however, two new CDC-based profiles were finalized. These profiles, the Personal Basis Profile (PBP) and the Personal Profile (PP), replace the PersonalJava platform, which is no longer under active development. Like PersonalJava, they provide the classes necessary for building interactive applications. This article describes the two profiles, and compares them to each other and to the PersonalJava platform.

A Common Foundation

To understand the new profiles properly, you must first understand what the Connected Device Configuration and the Foundation Profile bring to the J2ME platform, because both PBP and PP are based on the CDC/FP combination.

CDC Quick Summary

The Java runtime environment CDC (JSR 36) defines is much closer to J2SE's than the one CLDC defines. For one thing, CDC-based devices support the complete Java 2 Platform language specification and the complete Java 2 Platform virtual

machine specification. For another, they include a much larger subset of the J2SE 1.3 application programming interfaces (APIs), with classes drawn from the following packages:

- `java.io`
- `java.lang`
- `java.lang.ref`
- `java.lang.reflect`
- `java.lang.math`
- `java.net`
- `java.security`
- `java.security.cert`
- `java.text`
- `java.util`
- `java.util.jar`
- `java.util.zip`

Note, however, that in general CDC includes only a subset of the classes in each package. For example, from the `java.net` package it includes only the classes related to datagram and URL handling - stream-based socket support is not part of CDC.

CDC is a superset of CLDC, so it also includes the classes in the `javax.microedition.io` package that make up the Generic Connection Framework (GCF). CDC-based devices can use GCF to read and write files and to send and receive datagrams, in addition to the APIs in the `java.io` and `java.net` packages.

Foundation Profile Quick Summary

The Foundation Profile (JSR 46) mostly adds J2SE APIs omitted from CDC. In particular, FP augments the following packages:

- `java.io`
- `java.lang`
- `java.net`
- `java.security`
- `java.security.cert`
- `java.text`
- `java.util`
- `java.util.jar`
- `java.util.zip`

FP also includes classes from three packages not found in CDC:

- `java.security.acl`
- `java.security.interfaces`

- `ava.security.spec`

Note that FP includes the `java.net` classes required for stream-based socket and HTTP connections. These connections are also available through GCF - in fact, FP includes MIDP's `javax.microedition.io.HttpConnection` interface.

As you can see, the CDC/FP combination defines a powerful environment for general-purpose programming. Missing, however, are the classes needed to build interactive applications.

The Personal Basis Profile

The Personal Basis Profile (JSR 129) is a superset of the Foundation Profile, with three new features:

- The Xlet application model, adapted from the Java TV APIs
- A subset of the J2SE Abstract Windowing Toolkit (AWT) for building user interfaces based exclusively on lightweight components
- Inter-Xlet communication (IXC), using a subset of the Remote Method Invocation (RMI) API

These features enable the creation of interactive applications with well-defined lifecycles.

Included Packages

To support its new features, PBP includes subsets of the following J2SE 1.3 packages in addition to those already in CDC/FP:

- `java.awt`
- `java.awt.color`
- `java.awt.event`
- `java.awt.image`
- `java.beans`
- `java.rmi`
- `java.rmi.registry`

Note that PBP does *not* support RMI: The `java.rmi` and `java.rmi.registry` packages are included to support inter-Xlet communication, not full-fledged RMI. A PBP-based device *can* support RMI, but only if it implements the RMI Optional Package (RMIOP), a separate specification that can be included on any CDC-based device.

The Xlet Application Model

The traditional Java application model is quite simple: load a class, invoke its `main()` method, and wait until all non-background threads terminate or `System.exit()` is called. For many applications, this model allows too little control over the application's behavior. This is why MIDP defines a MIDlet application model that allows the device to start, stop, and pause applications in response to external events, and why browser-based applications - applets - use a different application model.

PBP defines its own application model, similar in many ways to the MIDlet model. The Xlet model has been borrowed from the Java TV API, where it's used to control application lifecycles in set-top boxes. The model's two key elements are the Xlet and XletContext interfaces, both found in the `javax.microedition.xlet` package. The application's main class implements the Xlet interface, which defines event methods for the system to invoke. The XletContext interface defines callback methods through which an application can obtain information about its operating environment. Here's a trivial Xlet:

```
// Example of a trivial Xlet

import javax.microedition.xlet.*;

public class TrivialXlet implements Xlet {
    private XletContext context;

    public TrivialXlet(){
    }

    public void destroyXlet( boolean unconditional )
        throws XletStateChangeException {
    }

    public void initXlet( XletContext context )
        throws XletStateChangeException {
        this.context = context;
    }

    public void pauseXlet(){
    }

    public void startXlet() throws XletStateChangeException {
        context.notifyDestroyed(); // immediately quit
    }
}
```

More details on Xlets and PBP can be found in these articles:

- [Understanding J2ME Application Models](#)

Note that PBP supports the traditional application model as well as the Xlet model.

The AWT Subset

Unlike MIDP, PBP does not define an entirely new set of user-interface classes, but instead subsets the standard AWT classes. Developers experienced in writing interactive J2SE applications will find PBP (and, as you'll see, the Personal Profile), a much more familiar environment than MIDP.

The classes in PBP's AWT subset are those required to define and use lightweight user-interface components. Originally, AWT consisted of *heavyweight* components, where each component instance was really just a proxy for a UI component (called a *peer* component) native to the underlying operating system. An application built with heavyweight components looks and feels like any other application on that platform, but the Java runtime has limited control over user-interface behavior or appearance. Java 1.1 therefore introduced the concept of lightweight components, whose look and feel are entirely under the runtime's control. There are no peer components in the lightweight model, except for top-level windows, as each component draws itself directly in its parent's drawing area and responds only to the events delegated by the parent. Top-level windows are normally heavyweight components, however, because the runtime system needs a way to interact with the operating system.

To support lightweight components, PBP specifically includes:

- `java.awt.Component` - the base class for all user-interface components
- `java.awt.Container` - the base class for AWT components that contain and manage other components
- `java.awt.Window` - the base class for top-level windows
- `java.awt.Frame` - a top-level window with a title bar and a border

These are the core component classes required to build a complete user interface. `Component` and `Container` are used to create lightweight components, while `Window` and `Frame` are used to create the top-level windows that hold the lightweight components. All the other AWT classes in PBP are included to support these four.

Note that PBP specifically excludes all heavyweight components other than `Window` and `Frame`. There are no buttons, lists, menus, or other basic user-interface components. In J2SE, the lightweight versions of these basic components are defined by the Swing user-interface toolkit, but Swing is *not* included in the PBP.

If the heavyweight components are missing and Swing is unavailable, you might wonder how you create a user interface. There are two possibilities.

If your application is simple enough, you can create your own components:

```
import java.awt.*;

/**
 * A very simple UI component that draws a
 * centered text string.
 */

public class SimpleTextLabel extends Component {
    private String text;

    public SimpleTextLabel( String text ){
        this.text = text;
    }

    public void paint( Graphics g ){
        int h = getHeight();
        int w = getWidth();

        g.setColor( getBackground() );
        g.fillRect( 0, 0, w, h );

        g.setColor( getForeground() );
        FontMetrics fm = g.getFontMetrics();
        int textWidth = fm.stringWidth( text );
        int textHeight = fm.getHeight();
        int x = ( w - textWidth ) / 2;
        int y = ( h - 2 * textHeight ) / 2;
        g.drawString( text, x, y );
    }
}
```

At runtime, place these components directly on a Frame component:

```

...
Frame frame = ... // some frame you created
SimpleTextLabel label;

// Now build our user interface

label = new SimpleTextLabel( "Press a key to exit" );
label.setBackground( Color.blue );
label.setForeground( Color.yellow );
label.addKeyListener( new KeyAdapter(){
    public void keyTyped( KeyEvent e ){
        exit();
    }
}
);

frame.add( label );

```

An application running under the new Xlet model uses the root container supplied by the system (available through the `XletContext` object passed to each Xlet on initialization) instead of creating its own frame.

The second way to create user interfaces is to use third-party lightweight user-interface toolkits. These toolkits may also be augmented by vendor-supplied classes.

Besides the lack of heavyweight components, the PBP AWT subset includes specific restrictions on what an application's user interface can do, so that PBP applications can run on systems with relatively limited UI capabilities. The most important restriction is that applications may use only a single instance of the `Frame` class. PBP applications built using the traditional model are responsible for creating the frame themselves. By contrast, in the Xlet model the system creates the frame on the application's behalf, and makes it (or a child container) available by way of the `XletContext.getContainer()` method. Other frame behavior is optional. The system is free to restrict the size, state, and position of a frame and to hide its title. There are also restrictions on cursor use and other minor details.

Inter-Xlet Communication

Inter-Xlet Communication (IXC) allows two or more Xlets running in the same virtual machine to exchange objects and to execute code across class-loader boundaries. IXC is based on the Remote Method Invocation (RMI) capabilities of J2SE, although the latter is actually meant for communication *within* a VM rather than *between* VMs. It's important to remember that by itself inter-Xlet communication does *not* imply that full RMI capabilities are available. To perform RMI in a PBP application, you must use RMIOP.

An Xlet can use the IXC *registry* to make an object available for use by other Xlets. This is known as *binding* or *exporting* the object. All Xlets share the same registry, a singleton instance of the `javax.microedition.xlet.ixc.IxcRegistry` class. The

getRegistry() factory method returns the singleton:

```
...
import javax.microedition.xlet.*;
import javax.microedition.xlet.ixc.*;

XletContext context = ..... // get the Xlet's context
IxcRegistry registry = IxcRegistry.getRegistry( context );
...
```

The registry is like a shared hash table. An Xlet binds a name to the object using the `bind()` method. Only objects that implement a *remote interface* can be bound to the registry. A remote interface is an interface that extends `java.rmi.Remote` such that each method throws `java.rmi.RemoteException`, and all method parameters and return types are primitive types, serializable objects, or other remote interfaces.

Other Xlets find a bound object using the `lookup()` method, which returns a *stub* for the actual object. The stub is automatically generated by the runtime system (unlike RMI, where the stubs must be generated manually using the `rmic` tool). The stub implements the same remote interface as the original object and acts as a proxy for that object. When an application invokes a method of the stub, the system invokes the same method on the original object, although perhaps on a different thread. Method arguments are handled mostly as you would expect: Primitive types are copied, serializable objects are serialized in the caller and deserialized in the callee, and stubs are returned for remote objects (except in the Xlet that bound the object, which gets a reference to the actual object).

Here's a sample application that uses IXC to ensure that two or more instances of the application don't run simultaneously:

```
import java.rmi.*;
import javax.microedition.xlet.*;
import javax.microedition.xlet.ixc.*;

/**
 * An Xlet that uses inter-Xlet communication (IXC)
 * to ensure that only once instance of the Xlet is
 * ever running.
 */

public class RunOnceXlet extends BasicXlet {
    private static final String NAME = "RunOnceXlet.activator";

    private boolean removeBinding = false;

    public RunOnceXlet(){
```

```
    }

    // The first thing the Xlet does is check to see if an
    // instance of itself is already running.

    public void initXlet( XletContext context )
        throws XletStateChangeException {
        try {
            // Get the IXC registry and create an instance of
            // our own remote interface in case we're the first
            // instance running

            IxcRegistry registry =
                IxcRegistry.getRegistry( context );
            RemoteInterface remote = new RemoteInterfaceImpl();

            while( true ){
                try {
                    // First we try to bind our remote interface
                    // to the registry. If two instances are
                    // started simultaneously, only
                    // one of them will win.

                    registry.bind( NAME, remote );
                    removeBinding = true;
                    break;
                }
                catch( AlreadyBoundException abe ){
                    try {
                        // Somebody else has bound an interface,
                        // so we get it and call its
                        // activateAgain method before
                        // terminating.
                        remote =
                            (RemoteInterface) registry.lookup( NAME );
                        String[] args = (String[])
                            context.getXletProperty( XletContext.ARGS );
                        remote.activateAgain( args );
                        throw new
                            XletStateChangeException( "Already running" );
                    }
                    catch( NotBoundException nbe ){
                        // Whoops, no object was found, the other
                        // instance might have just terminated
                    }
                }
            }
        }
    }
}
```

```

        // so try to register ourselves again....
    }
}
}
}
catch( RemoteException e ){
    // If we can't connect, just start up...
    System.out.println( "Error dealing with registry:" );
    e.printStackTrace();
}

..... // perform normal initialization
}

// Deregister the instance from the IXC registry. This
// will happen automatically when the Xlet is destroyed,
// but it's better to do it explicitly.

public void exit(){
    if( removeBinding ){
        try {
            IxcRegistry registry =
                IxcRegistry.getRegistry( getContext() );
            registry.unbind( NAME );
        }
        catch( NotBoundException e ){
        }
        catch( RemoteException e ){
        }
    }

    ..... // do other cleanup stuff
}

/**
 * The remote interface instances of RunOnceXlet use to
 * communicate with each other.
 */

public interface RemoteInterface extends Remote {
    void activateAgain( String[] args )
        throws RemoteException;
}

```

```

/**
 * The implementation of the remote interface. One difference
 * between IXC and regular RMI is that in regular RMI this
 * class would extend java.rmi.server.UnicastRemoteObject
 * and you would have to run rmic to generate the appropriate
 * stub classes. With IXC the stubs are generated
 * automatically.
 */

private class RemoteInterfaceImpl
        implements RemoteInterface {
    public RemoteInterfaceImpl() throws RemoteException {
    }

    // Our activation method. All it does is print out a
    // message, but it could easily do other
    // things by invoking methods on its containing class.
    // methods on its containing class.

    public void activateAgain( String[] args )
                                throws RemoteException {
        System.out.println(
            "[RemoteInterfaceImpl] Activation request" );
    }
}
}

```

Note that classes are not loaded remotely using IXC. Although the stubs are generated automatically, the actual interfaces that those stubs implement must already be packaged with the application.

The Personal Profile

The Personal Profile (JSR 62) is a superset of the Personal Basis Profile that adds the following features:

- Applet support for building browser-based applications
- A more comprehensive AWT subset that includes support for heavyweight components, JDK 1.02-style events, and deprecated APIs
- Clipboard support using a subset of the `java.awt.datatransfer` package

As you can see, PP provides an even richer, more J2SE-like environment, as well as offering the new PBP features like the Xlet application model and inter-Xlet communication.

Included Packages

To the classes of PBP, PP adds classes from three J2SE 1.3 packages:

- `java.applet`
- `java.awt`
- `java.awt.datatransfer`

Unlike PBP, the Personal Profile defines no additional non-J2SE classes.

Applet Support

The applet model is another alternative to the traditional and Xlet application models that PBP supports. Applets are browser-based applications that run in a controlled environment (often referred to as a *sandbox*) created by a web browser. Like MIDlets and Xlets, applets are notified by the system when they should be started, paused, or stopped, generally in response to events like the user moving from one web page to another with the browser. The applet support in PP is essentially the same as provided since version 1.1 of Java and should be familiar to most readers.

The AWT Subset

Unlike PBP's AWT subset, PP's includes support for the standard heavyweight UI components, classes like `java.awt.Button` and `java.awt.List`. These classes are necessary to support the applet model properly, because the `java.applet.Applet` class directly extends `java.awt.Panel`, and many applets use the standard components to build their user interfaces. The AWT subset also supports the original Java 1.0.2 event model based on the `java.awt.Event` class and the `Component.handleEvent()` method, again to support the applet model. PP also reintroduces deprecated AWT methods removed by the PBP - like `Component.enable()` - solely for backwards compatibility with existing applets.

PP includes most of the AWT classes in J2SE 1.3. So what's missing? The two-dimensional graphics classes (like `java.awt.Paint` and `java.awt.Stroke`), the printing classes (like `java.awt.PrintJob`), the accessibility classes (like `java.awt.Component.AccessibleAWTComponent`), and the `java.awt.Robot` class.

The Personal Profile relaxes some of the user-interface restrictions imposed by the Personal Basis Profile. In particular, PP does not restrict the number of frames or dialogs that an application creates. The system is free to restrict the size, state, or position of top-level windows, however, as before. Again, these restrictions are to support systems with more limited UI capabilities.

Migrating from PersonalJava

Applications written to the PersonalJava specification have a J2ME migration path to the Personal Profile (or, in some cases, the Personal Basis Profile). The migration is not seamless, because there are differences between the two Java environments. For an experienced Java programmer, however, the transition is not hard to make, and you end up with an application that uses the more familiar J2SE 1.3 APIs instead of the older (now quite dated) Java 1.1 APIs.

The following packages defined by the PersonalJava 1.2a specification are the same as, or subsets of, the packages of the

same name in the Personal Profile:

- `java.applet`
- `java.awt.datatransfer`
- `java.awt.event`
- `java.awt.image`
- `java.io`
- `java.lang`
- `java.reflect`
- `java.net`
- `java.security`
- `java.security.cert`
- `java.security.interfaces`
- `java.security.spec`
- `java.text`
- `java.util`
- `java.util.jar`
- `java.util.zip`

Note that `java.awt` is not in this list. PersonalJava includes the `PrintGraphics` and `PrintJob` classes from this package, while the Personal Profile does not. With these two exceptions, however, the PersonalJava `java.awt` package is a subset of the Personal Profile `java.awt` package.

If your PersonalJava application limits itself to using the packages listed above (including `java.awt` without the printer classes) it should work unchanged. The only place that you might run into difficulty is with deprecated methods, because the Personal Profile does not include the deprecated methods of Java 1.1.8 other than those in the AWT classes.

The other packages included in PersonalJava are not found in their entirety in the Personal Profile. In particular:

- The design-time classes are missing from the `java.beans` package.
- PP does not support RMI unless the RMI Optional Package is present. (Note that RMI is optional in PersonalJava.)
- PP does not include any JDBC classes from the `java.sql` package unless the JDBC Optional Package for CDC/Foundation Profile (JSR 169) is present. (Note that JDBC is optional in PersonalJava.)
- PP does not support the PersonalJava-specific APIs in the `com.sun.awt.com`, `com.sun.lang`, and `com.sun.util` packages. Code that uses these must be rewritten to use the nearest Personal Profile equivalent.

These differences are the reasons that the transition from PersonalJava to the Personal Profile may not be seamless. Some of the changes are quite minor, though. For example, no method in the Personal Profile will throw `com.sun.lang.UnsupportedOperationException`. Others entail more work, such as replacing use of `com.sun.util.PTimer` with use of `java.util.Timer`.

Conclusion

As you can see, both the Personal Basis Profile and the Personal Profile provide complete environments for building interactive Java applications on devices that are too constrained to support full J2SE implementations. They also provide a migration path for PersonalJava applications to the Java 2 Platform.

For more information on these profiles, see the [Personal Profile section of the Wireless Java portal](#).

About the Author: [Eric Giguere](#) is a software developer for iAnywhere Solutions, a subsidiary of Sybase, where he works on Java technologies for handheld and wireless computing. He holds BMath and MMath degrees in Computer Science from the University of Waterloo and has written extensively on computing topics.

Reader Feedback

Excellent Good Fair Poor

If you have other comments or ideas for future technical tips, please type them here:

Comments:

If you would like a reply to your comment, please submit your email address:

Note: We may not respond to all submitted comments.

Have a question about Java programming? Use [Java Online Support](#).

[Back To Top](#)



[About Sun](#) | [About This Site](#) | [Newsletters](#) | [Contact Us](#) |
[Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) |
[Trademarks](#)

Copyright 1994-2006 Sun Microsystems, Inc.

[A Sun Developer Network Site](#)

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this [License](#).

[Content Feeds](#)

XML