CHAPTER 2:

FEASIBILITY ANALYSIS
Background

Sun Microsystems has defined three Java platforms, each of which addresses the needs of different computing environments:

- Java 2 Standard Edition (J2SE)
- Java 2 Enterprise Edition (J2EE)
- Java 2 Micro Edition (J2ME)

The inception of the J2ME platform arose from the need to define a computing platform that could accommodate consumer electronics and embedded devices.
These devices are sometimes referred to collectively as pervasive devices.

J2ME made its debut at the JavaOne Developers Conference in mid-1999 and is targeted to developers of intelligent wireless devices and small computing devices that need to incorporate cross-platform functionality in their products.

Consumers of mobile and small computing devices have high performance expectations for these devices. They demand quick response time, compatibility with companion services, and full-featured applications in a small computing device.

Consumers expect the same software and capabilities found on their desktop and laptop computers to be available on their cell phones and personal digital assistants.

To meet these expectations, developers have to rethink the way they build computer systems. Developers need to harness the power of existing front-end and back-end software found on business computers and transfer this power onto small, mobile, and wireless computing devices. J2ME enables this transformation to occur with minimal modifications, assuming that applications are scalable in design so that an application can be custom-fitted to resources available on a small computing device.

Developers seeking to build applications that run on cell phones, personal digital assistants, and various consumer and industrial appliances must strike a balance between a thick client and a thin client. A thick client is front-end software that contains the logic to handle a sizable amount of data processing for the system. A thin client is front-end software that depends on back-end software for much of the system processing.

Developers must determine the minimum client processing that will meet the end user’s expectations of quick response time that is feasible within the limited resources available on the small computing device.

The creators of the J2ME platform delineated pervasive devices into two distinct categories:

- Personal, mobile information devices

That are capable of intermittent networked communications—mobile phones, two-way pagers, personal digital assistants (PDAs), and organizers
Shared-connection information devices

connected by fixed, uninterrupted network connection—set-top boxes, Internet TVs, Internet-enabled screen phones, high-end communicators, and car entertainment/navigation systems. The first category describes devices that have a special purpose or are limited in function; they are not general-purpose computing machines. The second category describes devices that generally have greater capability for user interface (UI) facilities. Of course, devices with superior UI facilities typically have more computing power. Practically speaking, computing power is the primary attribute that distinguishes these two categories of devices. Nevertheless, this delineation is somewhat fuzzy, because technology continues to enable more and more power to be placed in smaller and smaller devices.

Like computing power, connectivity—the availability of media such as wireless networks—also affects the kinds of functionality and services that pervasive devices can support. The challenge—and the primary goal—for J2ME is to specify a platform that can support a reasonable set of services for a broad spectrum of devices that have a wide range of different capabilities.

The creators of J2ME identify modular design as the key mechanism that enables support for multiple types of devices. The J2ME designers use configurations and profiles to make J2ME modular.

Configurations and profiles are the main elements that comprise J2ME’s modular design. These two elements enable support for the plethora of devices that J2ME supports.

A J2ME configuration defines a minimum Java platform for a family of devices.

Members of a given family all have similar requirements for memory and processing power. A configuration is really a specification that identifies the system level facilities available, such as a set of Java language features, the characteristics and features of the virtual machine present, and the minimum Java libraries that are supported. Software developers can expect a
certain level of system support to be available for a family of devices that uses a particular configuration.

A configuration also specifies a minimum set of features for a category of devices. Device manufacturers implement profiles to provide a real platform for a family of devices that have the capabilities that a given configuration specifies. The other J2ME building block, the profile, specifies the application-level interface for a particular class of devices. A profile implementation consists of a set of Java class libraries that provide this application-level interface. Thus, a profile theoretically could specify all kinds of functionality and services.

This is not the intention of its creators, however. The creators of J2ME intend that a profile should address the needs of a specific device category or vertical market pertaining to that device category.

The idea is not to place a plethora of unrelated application level features in a profile. Rather, the main goal is to guarantee interoperability—which doesn’t necessarily imply compatibility between different manufacturers’ implementations—between all devices of the same category or vertical market family to define a standard platform for Java application development.

For example, a profile might support a network communication facility for the popular Short Message Service (SMS) standard widely used by mobile phones. Because the SMS standard is a ubiquitous feature of mobile telephony, it makes sense to define this service in a profile that targets mobile phones, rather than to build it into a configuration.

A profile is implemented on top of a configuration, one step closer to the implementation of real-world applications. Typically, a profile includes libraries that are more specific to the characteristics of the category of devices they represent than are the libraries that comprise configurations. Applications are then built on top of the configuration and profile; they can use only the class libraries provided by these two lower-level specifications. Profiles can be built on top of one another. A J2ME platform implementation, however, can contain only one configuration. Figure 2.1 shows the conceptual layers.
that comprise the J2ME platform.

So far, these notions of configurations, profiles, and platform definitions is somewhat abstract. The next section gives you a more concrete description of the characteristics of actual environments.

Java Platform Micro Edition or Java ME (formerly J2ME) refers to a collection of technologies and specifications to develop applications and games for (usually) mobile devices using Java Technology. Most of these technologies have been specified in open manner in Java Community Process by Nokia, Sun Microsystems, Motorola, Ericsson, Siemens, and others. There are many implementations available in different platforms. Nokia has internally developed Java ME implementations for both Series 40 and S60 platforms. These implementations are continually being refined with new APIs and other functionality.

Nokia phones have an extensive Java ME API set.

**Configurations and Profiles**

A configuration specifies three basic elements:

- a set of Java programming language features
- a set of Java virtual machine features
- a set of supported Java libraries and application programming interfaces (APIs)
Figure 2.1. The J2ME platform consists of a set of layers that support a basic runtime environment with core Java libraries and a Virtual Machine (VM), a set of system-level application programming interfaces (APIs) in a configuration, and a set of application-level APIs in a profile.

**Architecture**

The creators of J2ME have defined only two configurations to avoid a fragmented landscape of incompatible platforms. The two configurations that exist currently represent the two categories of pervasive devices you saw earlier in this chapter, namely:

- personal, intermittently connected mobile devices—supported by the Connected, Limited Device Configuration (CLDC)

- constantly connected network devices—
Supported by the Connected Device Configuration (CDC)

Theoretically, a configuration could specify the very same support as the J2SE platform libraries. This is unlikely in the real world because, as you now know, J2ME is targeted at devices that are far less powerful than desktop computers.

Configuration specifications require that all Java classes adapted from J2SE be the same as or a proper subset of the original J2SE class. That is, a class cannot add methods not found in the J2SE version. Configurations can include additional classes in their specifications, however; configurations themselves are not necessarily proper subsets of J2SE. Both configurations that have been defined to date add classes not present in J2SE in order to address device attributes and constraints.

Its architecture includes:

Java Language

- System Libraries and Configurations (CLDC or CDC)
- Profiles and UI Libraries (MIDP or Personal Profile)
- Java Optional Packages

The J2ME Mindmap. MIDP 2.0 with CLDC is the common architecture found in the mass market. CDC with Personal Profile has more to offer to developers, but there aren't many devices yet with this configuration.

If you are using MIDP, you are developing MIDlets.
Java ME configurations

Connected Limited Device Configuration

The CLDC covers devices below 512k size of memory for the JVM environment and removes JNI, user-defined class loaders, some class verification items of the class verification process (the reason why we MIDlet Preverify MIDlets compiled to be deployed on CLDC devices) and reflection to fit into this small footprint. The JVM on CLDC devices is called KVM (K-VirtualMachine), nicknamed K-VirtualMachine for kilobyte. The application lifecycle concepts from javaTV and the installation concepts of javaPhone found their way into this Connected Limited Device Configuration.
The CLDC specification identifies devices in this category as having the following characteristics:

- 160 to 512 KB total memory available for the Java platform
- 16-bit or 32-bit processor
- Low power consumption, often battery powered
- Intermittent network connectivity (often wireless) with potentially limited bandwidth

The goal of the CLDC is to define a standard Java platform for these devices. Because of the wide variety of system software on various personal devices, the CLDC makes minimum assumptions about the environment in which it exists.

For example, one OS might support multiple concurrent processes, another might or might not support a file system, and so forth.

The CLDC is different from, yet also a subset of the CDC. The two configurations are independent of each other, however, so they should not be used together to define a platform.

Figure 2.3 shows the relationship between the two configurations and the J2SE platform.
Java Language Support. The CLDC specification omits support for the following features of the Java language:

- floating point calculations
- object finalization
- the java.lang.Error class hierarchy in its entirety

These features involve the VM ("Adherence to Java Virtual Machine Specification"). I address them here, however, because they have a language-level presence that affects programmers.

The lack of floating point support is the main language-level difference between a Java virtual machine that supports CLDC and a standard J2SE VM that is visible to programmers. This means that programs intended to run on the CLDC cannot use floating point literals,
types, or values. You can’t use the float built-in type, and the java.lang.Float class has been removed from CLDC libraries. This feature is not present because of the lack of floating-point hardware or software on most mobile devices.

Object finalization is also absent. This means that the Object.finalize() method has been removed from the CLDC libraries.

The java.lang.Error exception hierarchy has also been removed from the CLDC libraries and is therefore not available to applications. The primary reason that error handling is absent is memory constraints on mobile devices. This typically doesn’t create any disadvantages for applications development; after all, applications are not supposed to recover from error conditions. And the resource cost of implementing error handling is expensive, beyond the capabilities of today’s mobile devices. Moreover, error recovery is device-specific on embedded devices like mobile phones. In consequence, it doesn’t make sense to stipulate the recovery mechanism that devices should use.

This mechanism may well be outside the scope of an embedded VM.

Java Virtual Machine and Library Support. The CLDC specifies requirements for a Java virtual machine. It defines a VM that is highly portable and designed for resource-constrained small devices. Support for several features that exist in a standard J2SE VM have been omitted from the CLDC specification.

The following list describes the features that are not supported in a CLDC-compliant VM. The features in this list have been omitted because of either changes to libraries or security concerns:

- Java Native Interface (JNI)
- user-defined class loaders
- reflection
- thread groups and thread daemons
- finalization (no Object.finalize() method in CLDC libraries)
- weak references
- errors (a small subset of J2SE errors is supported)
- class file verification
Among these unsupported features, class file verification deserves further mention. The VM in the CLDC specification still performs this process, but it uses a two-step process and a different algorithm that requires fewer computation resources than the standard J2SE verifier. The VM that comes with the CLDC reference implementation is called the Kilobyte Virtual Machine (KVM), so named because it uses only a few KB of runtime memory. It is a reference implementation that adheres to the CLDC specification's description of a compliant VM. The KVM is not a full-featured J2SE VM.

The specification of the features that a VM supports includes a specification of the libraries that it supports. The CLDC specification details the libraries that an implementation must support.

As you know, a configuration is the basis for one or more profiles. The CLDC is a configuration on top of which one or more profiles are to be built in the same way that the Foundation Profile is built on top of the CDC. The intention is that the APIs in the CLDC profile support application development for the mass market of personal devices. The CLDC therefore targets third-party application developers.

This is somewhat different than the CDC, which targets OEM developers. Table 2.1 lists the packages of the CLDC. Notice that it is quite a bit smaller than the list of packages contained in the CDC, will show in Table 2.2.

The first three packages use the java. prefix in their name because each one contains a subset of the standard J2SE platform classes. The last one, however, must use the javax. prefix because it defines a new "standard extension" that is not part of the core Java platform.

<table>
<thead>
<tr>
<th>CLDC Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>Standard Java I/O classes and packages; subset of the J2SE package</td>
</tr>
<tr>
<td>java.lang</td>
<td>VM classes and interfaces; subset of the J2SE package</td>
</tr>
<tr>
<td>java.util</td>
<td>Standard utility classes and interfaces; subset of the J2SE package</td>
</tr>
<tr>
<td>javax.microedition.io</td>
<td>CLDC generic connection framework classes and interfaces</td>
</tr>
</tbody>
</table>

Table 2.1. Shows the CLDC Package
In the other hand, the Connected Limited Device Configuration (CLDC) defines the base set of application programming interfaces and a virtual machine for resource-constrained devices like mobile phones, pagers, and mainstream personal digital assistants. When coupled with a profile such as the Mobile Information Device Profile (MIDP), it provides a solid Java platform for developing applications to run on devices with limited memory, processing power, and graphical capabilities.

Sun Java Wireless Toolkit 2.3 Beta is now available. This version of the toolkit is fully compatible with the Java Technology for the Wireless Industry (JTWI) specification (JSR 185). It also includes support for Wireless Messaging API (WMA) 2.0 (JSR 205), Mobile Media API (MMAPI) 1.1 (JSR 135), PDA Optional Packages (JSR 75), Java APIs for Bluetooth (JSR 82), Mobile 3D Graphics (JSR 184), and J2ME Web Services API (JSR 172).

Each Java technology has an API specification, a reference implementation (RI), and a technology compatibility kit (TCK) associated to it.

Specification:

- **CLDC 1.1 (JSR 139):** CLDC 1.1 is a revised version of the CLDC 1.0 specification, and includes new features such as floating point and weak reference support, in additional to other enhancements. CLDC 1.1 is backward compatible with CLDC 1.0, and continues to target small and resource-constrained devices with the objective of maintaining a tight footprint.

- **CLDC 1.0 (JSR 30)**

Reference Implementation:

- **CLDC 1.1 RI:** This version of the CLDC reference implementation is based on the CLDC 1.1 specification and is targeted at device manufacturers who want to port this J2ME configuration to another platform.
- CLDC 1.0.4 RI: This version of the CLDC reference implementation is based on the CLDC 1.0 specification and is targeted at device manufacturers who want to port this J2ME configuration to another platform.

Technology Compatibility Kit: The CLDC TCK can be licensed from Sun to certify a CLDC implementation on a particular platform.

Optimized Implementation: The CLDC HotSpot Implementation (tm) is a virtual machine that is highly optimized and provides superior performance compared to KVM. In addition to complying to the CLDC specification, the CLDC HotSpot Implementation includes patented features that propel faster application execution as well as more efficient resource management. CLDC HotSpot Implementation is targeted at devices with 16/32-bit RISC microprocessors/controllers (such as ARM-based processors) with as little as 192 KB of total available memory for the Java technology stack.
The CDC covers devices above the 512k size of memory for the JVM environment. The garbage collector is a separate pluggable item into the CVM deployed on CDC devices. The threads are implemented not as OS specific but as fully implemented threads within the CVM and are known as Green Threads. Threads can also be implemented as OS native threads. Class file verification takes place in the CVM on the device just as in Java SE java platform on desktop systems. The JVM on CDC devices is known as a CVM or the C-VirtualMachine.

The CVM.

Although the CVM supports the same features as the J2SE VM, it is designed for consumer and embedded devices. This means that the standard J2SE VM has been reengineered to suit the constraints of limited-resource devices.

The features of the resulting offspring CVM are:

- advanced memory system
- small average garbage collection pause times
- full separation of VM from memory system
- modularized garbage collectors
- generational garbage collection

In particular, the CVM has been engineered to offer the following features:

- portability
- fast synchronization
- execution of Java classes out of read-only memory (ROM)
- native thread support
- small class footprint
- provision of interfaces to and support for real-time operating system (RTOS) services
- mapping Java threads directly to native threads
- support for all Java 2, v1.3 VM features and libraries: security, weak references, Java Native Interface (JNI), Remote Method Invocation (RMI), Java Virtual Machine Debugging Interface (JVMDI)
CDC Class Libraries. The CDC specifies a minimal set of class libraries and APIs. It supports the following standard Java packages:

- `java.lang` — Java VM system classes
- `java.util` — underlying Java utilities
- `java.net` — Universal Datagram Protocol (UDP) datagram and input/output (I/O)
- `java.io` — Java file I/O
- `java.text` — very minimal support for internationalization
- `java.security` — minimal fine-grain security and encryption for object serialization

As you can see, these APIs do not include the full set of Java 2 software development kit (SDK) packages. In some cases, these packages and classes are subsets of the Java 2 SDK packages and classes. Resource constraints dictate removal of the remainder of the J2SE classes and APIs. Also, all deprecated J2SE APIs are removed. Table 2.2 lists the full set of packages supported by the CDC.

```
<table>
<thead>
<tr>
<th>CDC Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>Standard IO classes and interfaces</td>
</tr>
<tr>
<td>java.lang</td>
<td>VM classes</td>
</tr>
<tr>
<td>java.lang.ref</td>
<td>Reference classes</td>
</tr>
<tr>
<td>java.lang.reflect</td>
<td>Reflection classes and interfaces</td>
</tr>
<tr>
<td>java.math</td>
<td>Math package</td>
</tr>
<tr>
<td>java.net</td>
<td>Networking classes and interfaces</td>
</tr>
<tr>
<td>java.security</td>
<td>Security classes and interfaces</td>
</tr>
<tr>
<td>java.security.cert</td>
<td>Security certificate classes</td>
</tr>
<tr>
<td>java.text</td>
<td>Text package</td>
</tr>
<tr>
<td>java.util</td>
<td>Standard utility classes</td>
</tr>
<tr>
<td>java.util.jar</td>
<td>Java Archive (JAR) utility classes</td>
</tr>
<tr>
<td>java.util.zip</td>
<td>ZIP utility classes</td>
</tr>
<tr>
<td>javax.microedition.io</td>
<td>CDC generic connection framework classes and interfaces</td>
</tr>
</tbody>
</table>
```

Table 2.2. Shows the CDC full package name
The Foundation Profile. A configuration, together with a profile, creates a J2ME runtime environment. The system-level features and services supported by a configuration are more or less hidden from the application developer. In reality, the application developer is prohibited from accessing them directly. If this were not the case, the application would not be considered J2ME compliant.

From the programmer's perspective, a profile is required to do "useful" work. A profile defines the layer that contains the APIs that the programmer usually manipulates. The J2ME creators initially defined one CDC profile, the Foundation Profile, which is based on the J2SE v1.3 release. It was designed by standard committee through the Java Community Process, by an expert group of companies in the consumer electronics industry. The Foundation Profile contains the J2SE packages listed in Table 2.2.

The list of packages above looks exactly like the list that comprises the CDC. In fact, they are the same. To say that the Foundation Profile contains these packages really means that they are available to the Foundation Profile. The intention is that the Foundation Profile be used with the CDC. The delineation between the profile and the configuration is a conceptual one, not a physical one.

Notice that the whole java.awt Abstract Window Toolkit (AWT) andjavax.swing Swing package hierarchies that define the J2SE graphical user interface (GUI) APIs are absent from the supported packages. If an application needs a GUI, an additional profile would be required. Profiles can be built on top of one another. An implementation of the J2ME platform, however, can contain only one configuration.

The lack of GUI support in the Foundation Profile has less impact for the family of shared, constantly connected network devices such as TV set-top boxes than it does for personal, mobile devices, which are served by the second J2ME configuration, the CLDC.

In general, the decision to include or omit features and libraries from a configuration or profile is based on their footprints, static and dynamic resource requirements, and security requirements.
Foundation Profile

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang</td>
<td>Rounds out full java.lang, * J2SE package support for the Java language (Compiler, UnknownError)</td>
</tr>
<tr>
<td>java.util</td>
<td>Adds full zip support and other J2SE utilities (java.util.Timer)</td>
</tr>
<tr>
<td>java.net</td>
<td>Adds TCP/IP Socket and HTTP connections</td>
</tr>
<tr>
<td>java.io</td>
<td>Rounds out full java.io, * J2SE package support for Java language input/output (Reader and Writer classes)</td>
</tr>
<tr>
<td>java.text</td>
<td>Rounds out full java.text, * J2SE package support for internationalization (BASN): Annotation, Collator, Iterator</td>
</tr>
<tr>
<td>java.security</td>
<td>Adds code signing and certificates</td>
</tr>
</tbody>
</table>

Table 2.3. Shows the Foundation Profile Package

Personal Profile. The Personal Profile specification was created through the Java Community Process, resulting in JSR-62. The Personal Profile provides an environment with full AWT support. The intention of its creators is to provide a platform suitable for Web applets. It also provides a J2ME migration path for Personal Java applications.

Personal Profile version 1.0 requires an implementation of the Foundation Profile version 1.0. It is a superset of the Personal Basis Profile version 1.0. Personal Profile is a subset of the J2SE version 1.3.1 platform, however, which makes Personal Profile applications upward compatible with J2SE version 1.3.1. Table 2.3 lists the packages that comprise Personal Profile version 1.0.

RMI Profile. The RMI Profile is a profile designed for platforms that support the CDC configuration. It has been defined by JSR-66 by various companies participating through the Java Community Process.

The RMI Profile requires an implementation of the Foundation Profile and is built on top of it. RMI Profile implementations must support the following features:

* full RMI call semantics
* marshaled object support
<table>
<thead>
<tr>
<th>Personal Profile Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.applet</td>
<td>Classes needed to create applets and those used by applets</td>
</tr>
<tr>
<td>java.awt</td>
<td>Classes for creating AWT UI programs</td>
</tr>
<tr>
<td>java.awt.datatransfer</td>
<td>Classes and interfaces for transferring data within and between applications</td>
</tr>
<tr>
<td>java.awt.event</td>
<td>Classes and interfaces for AWT event handling</td>
</tr>
<tr>
<td>java.awt.font</td>
<td>Classes and interface for font manipulation</td>
</tr>
<tr>
<td>java.awt.im</td>
<td>Classes and interfaces for defining input method editors</td>
</tr>
<tr>
<td>java.awt.im.spi</td>
<td>Interfaces that aid in the development of input method editors for any Java runtime environment</td>
</tr>
<tr>
<td>java.awt.image</td>
<td>Classes for creating and modifying images</td>
</tr>
<tr>
<td>java.beans</td>
<td>Classes that support JavaBean development</td>
</tr>
<tr>
<td>j2me.microedition.xlet</td>
<td>Interfaces used by J2ME Personal Profile applications and application managers for communication</td>
</tr>
</tbody>
</table>

Table 2.4. Shows the Personal Profile Package

- RMI wire protocol
- export of remote objects through the UnicastRemoteObject API
- distributed garbage collection and garbage collector interfaces for both client and server side
- the activator interface and the client side activation protocol
- RMI registry interfaces and export of a registry remote object

The RMI profile supports a subset of the J2SE v1.3 RMI API. The following interfaces and features are part of the J2SE v1.3 RMI specification and public API, but support for these interfaces and functionality is omitted from the RMI profile specification because of limitations on device processing power, network performance, and throughput:

- RMI through firewalls and proxies
- RMI multiplexing protocol
- implementation model for an “active” remote object
- deprecated methods, classes, and interfaces
Support for the following J2SE RMI v1.3 properties is omitted:

- java.rmi.server.disableHttp
- java.rmi.activation.port
- java.rmi.loader.packagePrefix
- java.rmi.registry.packagePrefix
- java.rmi.server.packagePrefix

The Connected Device Configuration (CDC), developed within the Java Community Process (JCP), is a framework for using Java technology to build and deliver applications that can be shared across a range of network-connected consumer and embedded devices, including smart communicators, high-end personal digital assistants (PDAs), and set-top boxes.

Connected Device Configuration (CDC) is a specification of Java ME that defines basic functionality that a connected device must have. It can be used in mobile phones (smartphones), car GPSs, DVR recorders or any device that connects to the Internet.

The CDC was developed under the Java Community Process as JSR 36 (CDC 1.0) and JSR 218 (CDC 1.1).

Typically, a CDC Virtual Machines requires 2Mb RAM and 2.5Mb of ROM. It's more powerful than the CLDC configuration, most common until now in Java ME phone devices. Each Java technology has an API specification, a reference implementation (RI), and a technology compatibility kit (TCK) associated to it.

Specification:

- CDC 1.0 (JSR 36): defines the basis for the Java ME devices that have a sufficient 32-bit microprocessor and ample memory.
- **CDC 1.1.2 (JSR 218):** defines a revision to the Java ME CDC specification. This JSR provides updates (based on J2SE, v1.4.2) to the existing core, non-graphical Java APIs for small electronic devices.

**Reference Implementation:**

- CDC reference implementation 1.0.2 corresponding to JSR 36
- CDC reference implementation 1.1.2 corresponding to JSR 218

**Technology Compatibility Kit:** The CDC TCK can be licensed from Sun to certify a CDC implementation (JSR 36 or JSR 218) on a particular platform.

**Optimized Implementation:**

- **Cutting edge performance:** Sun Java Connected Device Configuration HotSpot Implementation

(CDC HotSpot Implementation) is highly optimized for resource-constrained devices like consumer products and embedded devices, which combines excellent performance and reliability with a low memory footprint to meet the needs of a broad range of product scenarios. Because product designs vary, CDC HotSpot Implementation allows device-friendly tradeoffs between performance and constrained resources. This product achieves best-of-class performance with a modern dynamic compiler and solid reliability for multi-threaded and low-memory conditions. In addition, CDC HotSpot Implementation's portability interfaces enable rapid modification to support new target CPUs and operating systems while maintaining excellent performance.

- **Multitasking capabilities:** Leveraging the strengths of CDC HotSpot Implementation

Sun has been working on enabling support for concurrent applications and reliable termination of applications. The design philosophy behind this approach is illustrated in the CDC Application Management System (CDC AMS).
With CDC we can have these profiles:

- Foundation Profile: Has a Java SE-like API and no graphical user interface.
- Personal Profile (PP): It's the most common in mobile devices with CDC. It has full AWT and applet support.
Profiles provide APIs that focus on a specific group of devices. These devices typically have the same or similar user interfaces (screen and inputs), how the device connects to the network, how the device stores data, and etc. Currently, the Profile Name indicates the Family name of this group of devices or a common functionality with this group of devices. For example, the MIDP (MID Profile) is for the group of Mobile Information Devices. Java ME profiles are:

Mobile Information Device Profile. Because the category served by the CLDC encompasses so many different types of personal devices, potentially many different profiles are necessary to support them all. The most popular and well known of these is the Mobile Information Device Profile (MIDP), sometimes called the MID Profile. The MIDP layers atop the CLDC and defines a set of user interface (UI) APIs designed for contemporary wireless devices.

Following in the tradition of Java parlance, MIDP applications are called MIDlets. A MIDlet is a Java application that uses the MIDP profile and the CLDC configuration.

The MIDP specification, like the CDC’s Foundation Profile, was produced by an expert group, in this case, the Mobile Information Device Profile Expert Group, which is an international forum that includes representatives from several companies in the mobile device arena. The MIDP targets mobile information devices (MIDs), such as mobile phones, two-way pagers, and so forth, which have roughly the following characteristics:

- screen size of approximately (at least) 96x54 pixels
- display depth of 1 bit
- one- or two-handed keyboard, touchscreen input device
- 128 KB nonvolatile memory for MIDP components
- 8 KB nonvolatile memory for application-persistent data
- 32 KB volatile runtime memory for Java heap
- two-way wireless connectivity

Because the range of MID capabilities is so broad, the MIDP established a goal to address the least common denominator of device capabilities. The MIDP, therefore, specifies the following APIs:
- application (MIDP application semantics and control)
- user interface
- persistent storage
- networking
- timers

Table 2.5 lists the packages that comprise the MIDP.

<table>
<thead>
<tr>
<th>MIDP Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.microedition.io.ui</td>
<td>UI classes and interfaces</td>
</tr>
<tr>
<td>javax.microedition.rmss</td>
<td>Record management system (RMS) supporting persistent device storage</td>
</tr>
<tr>
<td>javax.microedition.midlet</td>
<td>MIDP application definition support class types</td>
</tr>
<tr>
<td>javax.microedition.io</td>
<td>MIDP generic connection framework classes and interfaces</td>
</tr>
<tr>
<td>java.io</td>
<td>Standard Java IO classes and interfaces</td>
</tr>
<tr>
<td>java.lang</td>
<td>VM classes and interfaces</td>
</tr>
<tr>
<td>java.util</td>
<td>Standard utility classes and interfaces</td>
</tr>
</tbody>
</table>

Table 2.5. Shows the MIDP Package Name

A MIDP implementation must consist of the packages and classes specified in the MIDP specification. Additionally, it can have implementation-dependent classes for accessing native system software and hardware.

Figure 2.6 juxtaposes the CDC and CLDC platform stacks. There is nothing inherent in either the CDC or CLDC that prohibits a manufacturer from porting either platform to a given family of devices. Nevertheless, the platform stacks—specifically, the configuration and profile features—have been specified to address practical limitations of the different families of hardware devices.
The Mobile Information Device Profile (MIDP) extends and enhances the CLDC to provide a Java environment to develop applications on Mobile information appliances and voice communication devices. The profile's are:

- MIDP 1.0
- MIDP 2.0
- MIDP 3.0

The Foundation Profile is what can be called a vertical specification profile. A vertical specification profile specifies the targeted family of devices by memory size and the optional profiles that may plug-in more functionality in addition to this profile. The GUI APIs are often Profiles plugged in on top of this Profile. Typical rich network devices include set top boxes (iTV), auto-computing (telematics), and etc.
This Personal Profile provides high web fidelity to device with a rich network connection and can use the heavy weight applet classes that were part of the PersonalJava spec. While Personal Profile, JSR62, supports PersonalJava1.1.x and PersonalJava1.2.x of the PersonalJava Application Environment; Personal Profile 1.1 supports the adoption of the subset of J2SE1.4 APIs as Personal Profile 1.0 was based on the subset of J2SE1.3 APIs.

This Personal Basis Profile provides a basic graphical user interface, i.e. the AWT lightweight APIs, on top of the CDC and Foundation Profile and is considered as a subset to the Personal Profiles. This profile also provides the xlet classes required for JavaTV.

This Information Module Profile supports embedded network connected devices without a graphical user interface. This profile is a subset of the MIDP1.0 profile without the LCDUI classes of MIDP1.0. The typical devices targets are modems, household devices, and industrial metering devices. While IMP is a subset of MIDP1.0, IMP-NG is a subset of MIDP2.0.

This Digital Set Top Box Profile is a subset of JavaTV and is targeted for the smaller set top boxes that deploy OCAP. Thus, this profile differs from the javaTV(CDC, PP PBP) setup in that its on top of CLDC not CDC to support a smaller device memory foot print. However, several concepts are borrowed from the JavaTV items in Personal Basis Profile such as the xlet lifecycle.
All J2ME applications—MIDlets and others—are real Java applications that run under the control of a Java VM. But what controls the Java VM, for instance on a mobile phone? There’s no command shell from which you can invoke your favorite Java applications like you do on your workstation. Starting, stopping, and managing the execution of J2ME applications is controlled by application management software (AMS) that resides on the device. In fact, the AMS controls the entire application lifecycle, from installation, upgrade and version management, to removal of application software.

The device manufacturer typically provides the AMS software. This is the most logical scenario because AMS software must work in conjunction with the device’s native system software, which, presumably, the manufacturer knows best. Nevertheless, third parties can also develop AMS systems for specific devices. AMS software could be written, for example, in Java or in some native language such as C.

Understanding the issues surrounding application management is important for the J2ME developer. You must be aware of the ramifications of your choices regarding packaging, licensing, charging for use, and so forth, and how these decisions will affect the usability and viability of your software.

![Figure 2.7. CDC structure](image1)

![Figure 2.8. CLDC structure](image2)
You should download a Java ME IDE (like Eclipse, Carbide or NetBeans) and start creating an application. You can compile it and test it in an emulator or on a real device.

Java ME APIs in Nokia devices

As follows, you can see Additional APIs available for Java ME and Mobile Service Architecture for more information on APIs

† (JSR-75 Optional package) FileConnection API

* `javax.microedition.io.file`

† (JSR-75 Optional package) PIM API

* `javax.microedition.pim`

† (JSR-82) Bluetooth API

* `javax.bluetooth`
* `javax.obex`

† (JSR-118) MIDP 2.0

* `java.lang`
* `java.util`
* `javax.microedition.io`
* `javax.microedition.lcdui`
* `javax.microedition.lcdui.game`
* `javax.microedition.media`
* `javax.microedition.media.control`
* `javax.microedition.midlet`
* `javax.microedition.pdu`
* `javax.microedition.rms`
+ (JSR 135) MMAPI

- javax.microedition.media
- javax.microedition.media.control
- javax.microedition.media.protocol

+ (JSR-139) CLDC 1.1

- java.io
- java.lang
- java.lang.ref
- java.util
- java.microedition.io

+ (JSR-172) WSA (RPC Package)

- java.rmi
- javax.microedition.xml.rpc
- javax.xml.namespace
- java.xml rpc

+ (JSR-172) WSA (Parser Package)

- javax.xml.parsers
- org.xml.sax
- org.xml.sax.helpers

+ (JSR-177) SATSA API

- java.lang
- java.rmi
- java.security
- java.security.spec
- javax.crypto
- javax.crypto.spec
- javax.microedition.apdu
+ (JSR-179) Location API

* javax.microedition.location

+ (JSR-180) SIP API

* javax.microedition.sip

+ (JSR-184) M3G API

* javax.microedition.m3g

+ (JSR-205) WMAPI

* javax.wireless.messaging

+ (JSR-226) M2G API

* javax.microedition.m2g
* org.w3c.dom
* org.w3c.dom.events
* org.w3c.dom.svg
Mobile Information Device Profile (MIDP) is a specification published for the use of Java on embedded devices such as mobile phones and PDAs. MIDP is part of the Java Platform, Micro Edition (Java ME) framework and sits on top of Connected Limited Device Configuration (CLDC), a set of lower level programming interfaces.

MIDP applications are called as MIDlets. A MIDlet (or Midlet) is an a Java ME program using MIDP as the profile.

A MIDlet has to fulfill the following requirements in order to run on a mobile phone:

- The "main" class of the project needs to be a subclass of javax.microedition.midlet.MIDlet
- The MIDlet needs to be packed inside a JAR file (e.g. by using the jar-tool or automatically using an IDE)
- The JAR file needs to be preverified by using a preverifier.

There are two versions in the market:

- MIDP 1.0
- MIDP 2.0

In the near future a new implementation will be released as MIDP 3.0
Figure 2.9. MID and MIDP structure
MidP (Mobile Information Device Profile) is a specification/profile designed for mobile devices and PDAs (JSR 37). MIDP 1.0 is the initial version which contains all the features required for java coding for Java ME. It works over CLDC 1.0 or 1.1.

In order to overcome the limitations of MIDP 1.0 most of the mobile vendors have added implementation of proprietary API, like Nokia UI API.

First Java powered mobile phones includes this profile and it is compatible with MIDP 2.0 phones. Every Nokia Series 40, S60 and Series 80 and 90s supports this profile.

- MIDP 1.0 has no active rendering APIs
- MIDP 1.0 has no support for direct access to image pixels (RGB data)
- MIDP 1.0 has no support for full screen mode without a proprietary API, like Nokia UI API.
- MIDP 1.0 has no support for audio directly without a proprietary API, like Nokia UI API or MMAPI.
- MIDP 1.0 only supports HTTP support.
- MIDP 1.0 cannot query key status when you want (although key events are supported)

MIDP 1.0 includes these packages:

- `javax.microedition.io`
- `javax.microedition.lcdui` (see LCDUI)
- `javax.microedition.rms`
- `javax.microedition.midlet`
MIDP 2.0 (JSR 118) is an extension and enhancement of the MIDP 1.0. All the mobile phones that are coming today are MIDP 2.0 specific phones. This device also supports MIDP 1.0 applications and games.

This profile usually run over CLDC 1.1. Older ones run over CLDC 1.0. It is included in JTWI and MSA standards.

Enhancements from the MIDP 1.0 are:

- Security
- Application signing
- Supports of audio playback
- Supports of full screen mode in LCDUI
- Includes Game API for game programming
- Supports HTTPS for secure connections
- Can work better with RGB images
- New controls and features for high level UI
- Push Architecture


Usually, we can find additional APIs installed on MIDP 2.0 devices, like MMAPI for Multimedia or WMA for messaging.

As we can find from above is in version 1.0 of the Mobile Information Device Profile (MIDP) provides a standard API for application development. It includes APIs for application lifecycle, HTTP network connectivity, user interface, and persistent storage. MIDP 2.0 includes many enhancements and additions. Most of the action is in the user interface packages, but the enhancements are broad and comprehensive.
Although the MIDP 2.0 specification and tools are available in the near term, it will be some months before MIDP 2.0 handsets will be available in volume. The process works this way:

- The specification becomes final after a public review. (As I write this, the specification is "Proposed Final").
- The reference implementation and Technology Compatibility Kit are released.
- Device manufacturers implement MIDP 2.0.
- Device manufacturers seek device approval from government regulators.
- Devices are released.

MIDP 2.0 devices are available in large volumes from Summer 2003.

The only network protocol the MIDP 1.0 specification requires is HTTP. MIDP 2.0 also requires HTTPS, which is basically HTTP over the Secure Sockets Layer (SSL). SSL is a socket protocol that encrypts data sent over the network and provides authentication for the socket endpoints.

Although many MIDP 1.0 implementations support HTTPS, application developers cannot rely on its availability. MIDP 2.0 provides a stable, consistent foundation for wireless applications that deal with money or sensitive information.

HTTPS support is provided through the CLDC's Generic Connection Framework in the javax.microedition.io package. Here's how you obtain a regular HTTP connection:

```java
String url = "http://www.omid-it.tk/";
HttpConnection hc = null;
hc = (HttpConnection)Connector.open(url);
```

In MIDP 2.0, it's just as easy to make an HTTPS connection:
String url = "https://www.orflic-it.tk/";

HttpsConnection hc = null;

c = (HttpsConnection)Connector.open(url);

javax.microedition.io.HttpsConnection is only one of several new interfaces that support secure networking. MIDP 2.0 also includes javax.microedition.io.SecurityInfo, which contains information about a secure connection, and javax.microedition.pki.Certificate, which represents a cryptographic certificate.

The HttpsExample MIDlet in the source code bundle for this article is a complete running example that demonstrates retrieving security information from an HttpsConnection.

One of the most exciting aspects of MIDP 2.0 is its set of media APIs. These APIs are an audio-only subset of the Mobile Media API (MMAPI).

The simplest foray you can make into multimedia is to generate simple tones using the playTone() method in javax.microedition.media.Manager. All you have to do is supply a note number (60 is middle C), a duration (in milliseconds), and a volume (0 is silent, 100 is loudest). This method throws a MediaException if the tone cannot be played. The following code plays middle C for half a second at the loudest volume.

```java
try {
    Manager.playTone(60, 500, 100);
}

catch (MediaException me) {
    // Handle the exception.
}
```
The API also supports playing sequences of tones. You’ll need to dive a little deeper and use the `javax.microedition.media.control.ToneControl` interface. The API documentation for `ToneControl` has a simple example that illustrates this technique.

Finally, MIDP 2.0’s media APIs support playback of sampled audio. Implementations must be able to play WAV files and are free to support additional audio formats.

The code to play audio is surprisingly simple. First you obtain a `Player` implementation for your audio data; then you set the `Player` running. The following code demonstrates how to play a WAV file stored as a MIDlet resource:

```java
InputStream in =
    getClass().getResourceAsStream("/signs_in.wav");

Player p = Manager.createPlayer(in, "audio/x-wav");
p.start();
```

Various goodies have been added to the `javax.microedition.lcdui` APIs in MIDP 2.0, but the largest changes (excluding the game APIs, discussed later) are in `Form` and the `Item` family.

First, form layout is considerably more sophisticated than it was in MIDP 1.0. The API documentation for `Form` describes the new layout algorithm in detail. Stated briefly, items are laid out left to right in rows that stack from top to bottom, just like ordinary text. You can modify this layout, but keep in mind that your application only requests behavior from the implementation. In the end, it’s the implementation that decides exactly where the items go and how big they are.

Items now have a minimum size and a preferred size that can be set by the application. If you don’t specify these sizes the implementation computes them. The `Item` class also includes
layout constants that previously were present only in the ImageItem class. You can specify a
horizontal layout, a vertical layout, newlines before or after an item, and other constraints.

Besides the new layout features, the available toolbox of form Items has been expanded.
New in MIDP 2.0 is Spacer, a non-navigable Item that represents empty space. You can use it to fine-tune form layout. Next, the ChoiceGroup item has a new type, POPUP. A POPUP
ChoiceGroup is essentially a combo box. It shows a current selection and some visual cue (a triangle pointing down, for instance) to indicate that there are more options. Select or activate it and the whole list of choices pops up, ready for a new selection.

MIDP 2.0 also extends command handling. In MIDP 1.0, Commands are added to Displayables and a single listener object receives all command events. MIDP 2.0 extends this model by allowing you to add Commands to individual Items. From the application programmer's point of view, this added flexibility is very easy to understand. The implementation has a somewhat harder job of figuring out how to show commands appropriately. The programming paradigm is very similar to adding Commands to a Displayable. Just pass a command to the Item's addItemCommand() method. To register a listener, implement the ItemCommandListener interface and register the listener with Item's setItemCommandListener() method.

Items also have a default command, a command that can be invoked using a particular user interface gesture. This varies from device to device; it could be a special button or a stylus gesture. You can set a default command with Item's setDefaultCommand() method.

The big new fish in Form's waters is CustomItem, a class that allows you to create your own Items. Similar conceptually to a Canvas, CustomItem allows you to do your own drawing and to respond to user interface events. Creating a custom item is a matter of subclassing CustomItem and implementing its abstract methods. The following example is essentially a toggle button with a hardcoded minimum size.
import javax.microedition.lcdui.*;

public class DiamondItem extends CustomItem {
    private boolean mState;

    public DiamondItem(String title) {
        super(title);
        mState = false;
    }

    public void toggle() {
        mState = !mState;
        repaint();
    }

    // CustomItem abstract methods.

    public int getMinContentWidth() { return 80; }
    public int getMinContentHeight() { return 40; }

    public int getPreferredWidth(int width) {
public int getPrefContentHeight(int height) {
    return getMinContentHeight();
}

public void paint(Graphics g, int w, int h) {
    g.drawRect(0, 0, w - 1, h - 1);
    int stepx = 8, stepy = 16;
    for (int y = 0; y < h; y += stepy) {
        for (int x = 0; x < w; x += stepx) {
            g.drawLine(x, y, x + stepx, y);
            g.drawLine(x, y, x + stepy, y);
            if (mState == true) {
                int midx = x + stepx / 2;
                int midy = y + stepy / 2;
                g.fillTriangle(x, y, x + stepx, y, midx, midy);
                g.fillTriangle(midx, midy, x, y + stepy,
                                x + stepx, y + stepy);
            }
        }
    }
}
The source code bundle contains both the `DiamondItem` source code and a MIDlet that demonstrates it, `CustomItemExample`. Here's a screen shot of the running MIDlet:
The release of MIDP 2.0 is good news for game developers. Many of the release's enhancements can be found in the Game API, in `javax.microedition.lcdui.game`. The five classes in the Game API extend MIDP's graphics capabilities in powerful ways. The basic concept is that the contents of the screen can be composed of different layers. One layer might contain the background. Another could show a spaceship, or a hedgehog. Another layer could show fog, or clouds, or water, or anything else.

The basic class in the Game API is `Layer`, essentially just a drawing surface with a location and size. Specialized subclasses implement more specific functionality. `TiledLayer` is a `Layer` whose contents are drawn from tiles contained in a source image. `Sprite` is a `Layer` that animates frames contained in a source image.

A `LayerManager` makes it easy to compose and render multiple layers.
GameCanvas is a subclass of Canvas. In addition to Canvas's capabilities, it provides an offscreen buffer for fast, flicker-free rendering and the ability to poll the states of the device's game keys.

Another exciting advance for MIDP developers is the representation of images as integer arrays, which allows MIDlets to manipulate image data directly. Image data is represented as one int per pixel, with 8 bits each for alpha (opacity), red, green, and blue values. The color components are packed into the int as 0xARGBBB. For example, the value 0xff00ff00 is fully opaque green, while 0x80ff0000 is half-transparent red.

MIDP 2.0's Graphics class supports RGB images with the following method:

```java
public void drawRGB(int[] rgbData, int offset, int scanlength,
int x, int y, int width, int height, boolean processAlpha)
```

The rgbData array should have at least width * height elements, starting at offset. The scanLength parameter describes the number of pixels between consecutive rows in the integer array. The x, y, width and height parameters describe where the integer data should be rendered on the Graphics' drawing surface. Finally, processAlpha is a flag that indicates whether alpha compositing should be used. If it's false, all pixels in the integer array are treated as opaque.

The following example, SnowCrash, is a Canvas that has the dubious distinction of simulating video snow using an integer array. In the source code bundle, the RGBImageExample MIDlet demonstrates this Canvas.

```java
import java.util.Random;
import javax.microedition.khronos.*;
import javax.microedition.midlet.*;
```
public class SnowCrash
    
    extends Canvas
    
    implements Runnable {

    private boolean mTrucking;
    private int[] mRGB;
    private Random mRandom;

    public SnowCrash() {
        mTrucking = true;
        mRandom = new Random();

        Thread t = new Thread(this);
        t.start();
    }

    protected void randomize() {
        if (mRGB == null) return;
        int bitCounter = 0;
        int r = 0;
        for (int i = 0; i < mRGB.length; i++) {
            
        }
// Get the next random int if necessary.

if (bitCounter == 0) {
    r = mRandom.nextInt();
    bitCounter = 32;
}

// Get the next bit.

int bit = r % 2;

r = (r >> 1);
bitCounter--;

// Set the color to black or white.

mRGB[i] = (bit == 0) ? 0x000000 : 0xffffffff;

} 

public void stop() { mTracking = false; }

// Canvas abstract method

public void paint(Graphics g) {
    int w = getWidth();
    int h = getHeight();
    int rw = 50;
    int rh = 50;
    int rx = (w - rw) / 2;
```java
        int ry = (h - rh) / 2;
        
        if (mRGB == null) mRGB = new int[rw * rh];

        // Clear the screen.
        g.setColor(0xfff00000);
        g.fillRect(0, 0, w, h);

        // Draw the outline.
        g.setColor(0xfff00000);
        g.drawRect(rx, ry, rw + 1, rh + 1);

        // Draw the snow.
        g.drawRGB(mRGB, 0, rw, rx + 1, ry + 1, rw, rh, false);
    }

    // Runnable method
    public void run() {
        // Attempt 12 fps.
        int interval = 1000 / 12;

        while (mTrucking) {
            randomize();
            repaint();
            try {
                Thread.sleep(interval);
            } catch (InterruptedException ie) {}  
        }
    }
```
The following screen shot shows SnowCrash in action:

![SnowCrash Screen shot](image)

**Figure 2.11.** Shows the SnowCrash Screen shot

### Permissions and Code Signing

MIDlets that use the Generic Connection Framework may cost users money (in the case of network connections) or involve a security risk (in the case of serial port access). In light of the sensitivity of the connection framework, the MIDP 2.0 specification recognizes the concepts of trusted and untrusted code and permissions. Untrusted code cannot make connections at will; it must receive permission from the user. Code can be designated as trusted if the developer digitally signs it and the user's device can verify the signature.
The exact mechanics of asking and granting permission depend on the security policies of the device and those defined by the user. As an example, consider a game that connects to a server to maintain a community list of high scores. If the game is not signed, or is signed by an entity the device doesn't recognize, the code is untrusted. When the game attempts to connect to the high-score server, the MIDP 2.0 implementation denies the action (by way of a runtime SecurityException) or, more likely, prompts the user to allow or deny the network connection. If the game is signed and the device verifies the signature, the code can be designated as trusted and allowed to make its network connection.

Special entries in the manifest file allow MIDlet suites to indicate which permissions they require in order to run smoothly. The MIDP 2.0 specification defines permissions for various types of network connections. It is an extensible architecture, and optional packages that deal with sensitive APIs will likely define additional permission types.

This part describes some of the flashier new features in MIDP 2.0; there are many more, including:

- Standardized connection strings for serial port access.
- Standardized connection strings for datagram, socket, and server socket connections. The specification doesn't require support for these types of connections but it recommends their implementation and provides supporting APIs.
- A push registry that allows MIDlets to be launched in response to incoming network connections.
- The Over-the-Air (OTA) recommended practice, an addendum to the MIDP 1.0 specification, is now incorporated in the MIDP 2.0 specification.
- The MIDlet class has a new platformRequest() method that asks the underlying platform to display a URL.
- Record stores may be shared between MIDlets.
MIDP 1.0 established a standard Java environment for small devices with wireless network connectivity. MIDP 2.0 expands considerably on the original specification with far-reaching support for advanced user interfaces, multimedia, secure HTTP networking, and many other useful features.
MIDP 3.0 is the enhanced version of MIDP 2.0

The main enhancements are:

- Enable and specify proper behavior for MIDlets on each of CLDC, CDC, and OSGi, including:
- Enable multiple concurrent MIDlets
- Specify proper firewalling, runtime behaviors, and lifecycle management issues for MIDlets
- Enable background MIDlets (faceless MIDlets with no UI)
- Enable auto-launched MIDlets (started at device boot time)
- Enable inter-MIDlet communications

1. Enable shared libraries for MIDlets
2. Tighten spec in all areas to improve cross-device interoperability
3. Increase functionality in all areas, including:

- Improve UI expressability and extensibility
- Better support for devices with larger displays
- Enable MIDlets to draw to secondary display(s)
- Enable richer and higher performance games
- Secure RMS stores
- Removable/remote RMS stores
- IPv6
- Multiple network interfaces per device

1. Specify standard ways for doing MIDlet provisioning through other means (e.g. OMA (SyncML) DM/DS, Bluetooth, removable media, MMS, JSR 232, etc.)
2. Extensive device capabilities query
3. Localization & Internationalization (if appropriate, integrating/augmenting JSR 238 as needed)
The main thing is that it will be an open source & also backwardly compatible to MIDP 2.0 applications.

A key design goal of MIDP3 will be backward compatibility with MIDP2 content.
Now it's the time that I can answer to some important questions regarding to JAVA ME specially about JSR and MidLet that can complete the feasibility analysis on this project that finally we can find a needed area on Java ME technology to develop certain applications:

1. What is the target Java platform? (i.e., desktop, server, personal, embedded, card, etc.)

MIDP on Mobile Handsets

2. The Executive Committees would like to ensure JSR submitters think about how their proposed technology relates to all of the Java platform editions. Please provide details here for which platform editions are being targeted by this JSR, and how this JSR has considered the relationship with the other platform editions.

The target platform is the MID Profile in J2ME.

3. Should this JSR be voted on by both Executive Committees?

No

4. What need of the Java community will be addressed by the proposed specification?

This JSR will specify the 3rd generation Mobile Information Device Profile, expanding upon the functionality in all areas as well as improving interoperability across devices.

5. Why isn't this need met by existing specifications?

N/A

6. Please give a short description of the underlying technology or technologies:

CLDC, CDC, OSGi, and MIDP 2.0

7. Is there a proposed package name for the API Specification? (i.e., javapi.something, org.something, etc.)

javax.microedition.*

8. Does the proposed specification have any dependencies on specific operating systems, CPUs, or I/O devices that you know of?

No
9. Are there any security issues that cannot be addressed by the current security model?

No

10. Are there any internationalization or localization issues?

No

11. Are there any existing specifications that might be rendered obsolete, deprecated, or in need of revision as a result of this work?

Yes. JSR-118 MIDP 2.0

12. Please describe the anticipated working model for the Expert Group working on developing this specification.

The Expert Group will conduct its work in a similar fashion to the MIDP 2 Expert Group. That is, the primary means of communication will be via email list(s) and web site(s). In addition, periodic face-to-face meetings will be held approximately every 6-8 weeks, and phone conferences as needed for special issues that need to be resolved between meetings. Decisions will be made by technical consensus.

13. It is important to the success of the community and each JSR that the work of the Expert Group be handled in a manner which provides the community and the public with insight into the work the Expert Group is doing, and the decisions that the Expert Group has made. The Executive Committees would like to ensure Spec Leads understand the value of this transparency and ask that each JSR have an operating plan in place for how their JSR will address the involvement of the community and the public. Please provide your plan here, and refer to the Spec Lead Guide for a more detailed description and a set of example questions you may wish to answer in your plan.

Transparency plan:

a) All members of the expert group will be allowed to subscribe to and participate in discussions on the email list(s) as they choose. The face-to-face meetings will be open to all active participants with a limit of one representative per company in order to keep the meeting space to a reasonable size (i.e. approx 30-35 people). Any free seats will be made available to Observers on some kind of simple lottery-type system, similar to how MIDP2 was organized.

b) The specification lead will, on a quarterly basis, provide a brief JSR status to the JCP PMO, for publication to the Java community. This will include the current schedule for the JSR and notes on any major events that have occurred in the previous quarter.
14. Please describe how the RI and TCK will be delivered, i.e. as part of a profile or platform edition, or stand-alone, or both. Include version information for the profile or platform in your answer.

An independent RI and TCK will be produced as a part of this JSR with each being licensed separately.

15. Please state the rationale if previous versions are available stand-alone and you are now proposing in 2.13 to only deliver RI and TCK as part of a profile or platform edition (See sections 1.1.5 and 1.1.6 of the JCP 2 document).

N/A

16. Please provide a description of the business terms for the Specification, RI and TCK that will apply when this JSR is final.

These terms only represent the initial commercial terms to be used and remain subject to the execution of final legal agreements covering the subject matter hereof to be determined by Motorola at its sole discretion.

License will issue to all interested parties.

Independent implementations will be allowed - TCK and RI will be licensed separately.

For TCK we plan to charge a single one-time fee for access and an annual maintenance fee for a term of four years. The actual size of the license fees will be based on the associated development cost, with a target of cost recovery. Since MIDP is much larger and more complex than most other JSRs, we anticipate the fees of $100,000 USD for one-time fee and $50,000 for annual maintenance fee.

TCK will include both binary environment and source code of the test suite.

Maintenance fee covers limited basic support, first level TCK appeals process, bug fixes when available and updates, which are due to changes in the specification. Major new releases may be subject to additional single one time fee.

For RI in source code form we will charge a per unit license fee. Maintenance releases will cover bug fixes, updates and new releases necessary due to spec changes, and when made generally available by specification lead.

17. Please list any existing documents, specifications, or implementations that describe the technology. Please include links to the documents if they are publicly available.
JSR-118 MIDP 2.0

18. Explanation of how these items might be used as a starting point for the work.

This specification will serve as the basis for beginning the work, which will be extended, enhanced, and clarified. Note that backward compatibility with specified MIDP 2.0 behaviors will be of paramount importance wherever possible.

So, according to the feasibility analysis on the wireless devices and result of the simple graphics and connection testing to get the good feasibility, now we find out the applications which we need to coding and developing, should developed on the CLDC technology which can have the MIDLet and according to the CLDC basis and packages we can develop applications on Java ME basis and by applying the CLDC Library's we will get the certain and particular result as we expect and define on the requirement of the project.

Instead to accessing a large number of packages and libraries we can also use CDC technology with combination of Personal Profile which we will go through it later.
This part examines the performance of Symbian OS phones and S60 devices in the marketplace and the business case for utilizing the S60 platform from the perspective of various potential stakeholders.

Market results
Symbian OS will be the leading platform for smartphones for the foreseeable future. In September 2005, research company Ovum estimated that Symbian OS phones will account for 70 percent of all smartphone shipments by 2009.
The success of Symbian OS comes from offering rich devices from the midrange to the top end, capturing new markets in the process. S60 devices will account for the bulk of Symbian OS device shipments. Nokia alone has shipped more than 70 million S60 devices (as of October 2005). More than 12 S60 3rd Edition models have been launched, and S60 devices are sold by the majority of retail operators worldwide. The S60 platform is licensed to four device manufacturers. All these factors mean that the S60 platform offers a significant and growing market of potential customers.

Stakeholder benefits
The S60 platform is the established leader in the smartphone market, making it an important source of revenue for third-party application and content developers. S60 3rd Edition significantly extends the business opportunities and introduces major enhancements designed to provide long-term stability for stakeholders. There are greater opportunities for market segmentation, with a range of features and applications targeted at the enterprise, game, music, video, personal productivity, and other sectors.

The S60 platform gives the developer community access to industry-standard technologies and a market that can be measured in tens of millions of consumers. The opportunities extend beyond S60 devices, because the platform provides standard technologies that allow developers to build applications and content for Series 40 devices, Series 80 devices, and the Nokia 7710 widescreen smartphone, as well as for devices provided by other manufacturers.
S60 market resources
This section provides information on where to get help with producing and marketing applications.

Nokia Content Discoverer
The Nokia Content Discoverer client is an on-device content portal that makes it easy for mobile consumers to discover, download, and purchase great content and applications. With its ability to integrate with multiple content-delivery systems, the client helps operators maximize mobile application and content sales.

Product Creation Community
The S60 platform has development resources available that are collectively known as the Product Creation Community. The Product Creation Community can provide assistance with application development, device manufacture, training, and consultancy.

Future prospects
The S60 platform has performed well in an emerging market and is set to continue on that path for the foreseeable future. The latest versions of the S60 platform allow for greater diversity of form factors with underlying hardware flexibility. The latest kernel and binary promise greater code optimization and efficiency in addition to the robust multitasking capability of the system. There will be a diversity of hardware solutions, because there is now the option to use either a single-chip or dual-chip architecture.

It is expected that S60 will move beyond the classic smartphone market and into greater differentiation, given the hardware options and range of potential form factors. The APIs are continually being enhanced, providing rich functionality suitable for diverse market segments such as enterprise and games.

Expectation
The S60 platforms will get the around 70 percent of the mobiles handset market, so by implementing our coding and development on this types of mobiles, we can expect that in future the big mobile handset producer will find out the needed of this types of facilitation on
the next handset generation and will search on the market to find, is anybody worked or developed such types of facility on these series platform or not.

Behalf of what mentioned above, I can also offer to this such companies, that, they come to know about these types of works and by increasing this platform series, the leader manufactures in the market wants to keep their stage in the market, so, they will contribute with this types of developers like me, and the benefit will be for the both sides.