Connected Limited Device Configuration

Specification

Version 1.1

Java™ 2 Platform, Micro Edition (J2ME™)

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A. 650-960-1300

1.1
March 2003

Connected Limited Device Configuration (CLDC) Specification ("Specification")
Version: 1.1
Status: FCS
Release: March 4, 2003

Please
Recycle

Adobe PostScript

Contents

Figures

Preface

In the past four years, Sun and major consumer device manufacturers have collaborated to create a highly portable, small-footprint Java™ application development environment for resource-constrained consumer devices such as cellular telephones, two-way pagers and personal organizers.

This work started with the development of a new, small-footprint Java virtual machine called the K Virtual Machine (KVM). Two Java Community Process (JCP) standardization efforts, *Connected Limited Device Configuration* (CLDC) and *Mobile Information Device Profile* (MIDP), were then carried out to standardize the Java libraries and the associated Java language and virtual machine features across a wide variety of consumer devices. The CLDC and MIDP standards are a key part of the *Java™ 2 Platform, Micro Edition* (J2ME™).

This document, *Connected Limited Device Configuration Specification*, defines the *1.1* version of the J2ME *Connected Limited Device Configuration* (CLDC). This specification is the result of the Java Community Process expert group JSR-139, consisting of *24* companies from all over the world.

A *configuration* of the J2ME platform specifies the subset of the Java programming language, the subset of functionality of the configuration's Java virtual machine, the security and networking features, as well as the core platform libraries, all to support a wide range of consumer products.

The Connected Limited Device Configuration is the basis for one or more *profiles*. A *profile* of the J2ME platform defines additional sets of APIs and features for a particular vertical market, device category or industry. Configurations and profiles are more exactly defined in the *J2ME™ Platform Specification* document.

Who Should Use This Specification

The audience for this document includes:

- the Java Community Process (JCP) expert group JSR-139 defining this configuration,

- application developers and content providers who want to write Java applications for small, resource-constrained, connected devices,
- device manufacturers who want to build small Java-enabled devices conforming to the *CLDC Specification*,
- Java platform vendors who want to build implementations that conform to the *CLDC Specification*.

How This Specification Is Organized

The topics in this specification are organized as follows:

Chapter 1, "Introduction and Background," provides some background information and context for the *CLDC Specification*, and lists the names of the companies that have been involved in the JSR-139 and JSR-30 specification work. The chapter also summarizes the main differences between *CLDC Specification* versions 1.1 and 1.0.

Chapter 2, "Goals, Requirements and Scope," defines the goals, requirements and scope of this specification.

Chapter 3, "High-level Architecture and Security," defines the high-level architecture of the CLDC, and discusses its security features.

Chapter 4, "Adherence to the Java Language Specification," details the variances from the standard Java programming language defined by the *CLDC Specification*.

Chapter 5, "Adherence to Java Virtual Machine Specification," details the variances from the standard Java virtual machine defined by the *CLDC Specification*.

Chapter 6, "CLDC Libraries," defines the Java APIs supported by the *CLDC Specification*.

"Appendices" is a pointer to a number of appendices that accompany this specification.

Related Literature

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704, USA, +1 (800) 854-7179

The Java™ Language Specification (Java Series), Second Edition by James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. Addison-Wesley, 2000, ISBN 0-201-31008-2

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999)

Programming Wireless Devices with the Java™ 2 Platform, Micro Edition (Java Series) by Roger Riggs, Antero Taivalsaari, and Mark VandenBrink. Addison-Wesley, 2001, ISBN 0-201-74627-1

J2ME™ Platform Specification, Java Community Process, Sun Microsystems, Inc.

`http://jcp.org/jsr/detail/68.jsp`

Connected, Limited Device Configuration Specification, version 1.0, Java Community Process, Sun Microsystems, Inc.

`http://jcp.org/aboutJava/communityprocess/final/jsr030/`

Mobile Information Device Profile Specification, version 1.0, Java Community Process, Sun Microsystems, Inc.

`http://jcp.org/aboutJava/communityprocess/final/jsr037/`

Related Web Pages

Java 2 Micro Edition Product Web Page

`http://java.sun.com/j2me/`

K Virtual Machine (KVM) Product Web Page

`http://java.sun.com/products/kvm/`

Connected Limited Device Configuration Specification, Java Community Process, Sun Microsystems, Inc. *(CLDC) Product Web Page*

`http://java.sun.com/products/cldc/`

Mobile Information Device Profile (MIDP) Product Web Page

`http://java.sun.com/products/midp/`

J2ME Wireless Toolkit Product Web Page

`http://java.sun.com/products/j2mewtoolkit/`

Modification History

TABLE 1

Date	Version	Description
October 10, 2001		Started working on the CLDC NG (Next Generation) Specification.
October 22, 2001	1.1 Working Draft 1	The first “complete” working draft with annotations.
November 21, 2001	1.1 Working Draft 2	Second working draft based on feedback from expert group members and other reviewers. Turned on change bars so that changes are easier to track.
December 19, 2001	1.1 Working Draft 3	Integrated feedback from the third expert group meeting.
January 17, 2002	1.1 Community Review	This version was submitted to JCP for Community Review. Removed change bars.
March 11, 2002	1.1 Working Draft 4	Includes feedback from Community Review and from the fourth expert group meeting.
March 20, 2002	1.1 Public Review	This version was submitted to JCP for Public Review. Removed change bars. Updated the CLDC 1.0 vs. 1.1 differences.
May 10, 2002	1.1 Public Review - Internal Revision 1	Revised some library classes and the javadocs after receiving external and internal feedback during the Public Review.
May 31, 2002	1.1 Public Review - Internal Revision 2	Added `Thread.interrupt()`, as decided by the expert group. Added some text related to uncaught exceptions and errors. Some other minor clarifications.
June 7, 2002	1.1 Public Review - Internal Revision 3	Removed the ISO8601 compliant version of `Date.toString()`, because it is incompatible with J2SE. Minor bug fixes to Appendix 1.
July 8, 2002	1.1 Public Review - Internal Revision 4	Minor library documentation updates based on internal feedback. No changes to the main specification document.
July 12, 2002	1.1 Public Review - Public Revision 2	Integrated the most recent feedback from Motorola and internal reviewers. Most of the changes are related to javadocs only. Added method `String.equalsIgnoreCase()`.

TABLE 1

Date	Version	Description
November 15, 2002	1.1 Proposed Final Draft	Minor updates from internal reviewers. Added back `Date.toString()`(J2SE-compliant version).
December 12, 2002	1.1 Proposed Final Draft	Integrated feedback from Nokia and other reviewers. Removed redundant/overlapping definition of the `StackMap` attribute.

Annotations

This document may contain *annotations* that are used for summarizing the discussions and decisions of the JSR-139 expert group. The format of the annotations is illustrated below.

Annotation – This is what annotations look like. These annotations are not part of the actual specification. They are used for summarizing the discussions and decisions of the JSR-139 expert group.

CHAPTER 1

Introduction and Background

This document specifies the Connected, Limited Device Configuration (CLDC) of Java™ 2 Platform, Micro Edition (J2ME™).

The main goal of the *CLDC Specification* is to standardize a *highly portable, minimum-footprint Java™ application development platform for resource-constrained, connected devices.*

Cell phones, two-way pagers, personal digital assistants (PDAs), organizers, home appliances, low-end TV set-top boxes, and point of sale terminals are some, but not all, of the devices that might be supported by this specification.

This J2ME configuration specification defines the minimum required complement of Java technology components and libraries for small connected devices. Java language and virtual machine features, core libraries, security, input/output, and networking are the primary topics addressed by this specification.

CLDC is core technology that will be used as the basis for one or more *profiles*. A J2ME *profile* defines additional libraries are features for a particular vertical market, device category or industry.

1.1 CLDC Expert Groups

JSR-139 expert group. This specification is the result of the work of a Java Community Process expert group JSR-139 consisting of a number of industrial partners. The following companies (in alphabetical order) are full members in the JSR-139 expert group work:

- aJile Systems
- Aplix Corporation
- France Telecom
- Fujitsu
- Insignia Solutions
- Liberate Technologies
- Mitsubishi
- Motorola

- NEC
- Nokia
- NTT DoCoMo
- OpenTV
- Openwave Systems
- Oracle
- Panasonic
- Research In Motion (RIM)
- Samsung
- Siemens
- Sony
- Sony Ericsson Mobile Communications
- Sun Microsystems
- Symbian
- Vulcan Machines
- Zucotto Wireless

In addition, 11 companies and individuals have been following the JSR-139 expert group work as *observers*. The observers have received all the draft specification versions, intermediate documents and meeting minutes, but they did not participate in the actual expert group meetings.

JSR-30 expert group. This specification is derived from the *CLDC Specification* version 1.0 that was finished in May 2000. The JSR-30 expert group consisted of representatives from the following companies (in alphabetical order):

- America Online
- Bull
- Ericsson
- Fujitsu
- Matsushita
- Mitsubishi
- Motorola
- Nokia
- NTT DoCoMo
- Oracle
- Palm Computing
- Research In Motion (RIM)
- Samsung
- Sharp
- Siemens
- Sony
- Sun Microsystems
- Symbian

1.2 Main Differences Between CLDC Specification Versions 1.1 and 1.0

The CLDC 1.1 (JSR-139) expert group members were generally satisfied with the *CLDC Specification* version 1.0, and did not see any need for radical changes in the new specification. Therefore, *CLDC Specification* version 1.1 is primarily an incremental release that is intended to be fully backwards compatible with *CLDC Specification* version 1.0. Some important new functionality, such as floating point support, has been added.

The list below summarizes the main differences between *CLDC Specification* versions 1.1 (JSR-139) and 1.0 (JSR-30):

- Floating point support has been added.
  - All floating point byte codes are supported by CLDC 1.1.
  - Classes `Float` and `Double` have been added.
  - Various methods have been added to the other library classes to handle floating point values.
- Weak reference support (small subset of the J2SE weak reference classes) has been added.
- Classes `Calendar`, `Date` and `TimeZone` have been redesigned to be more J2SE-compliant.
- Error handling requirements have been clarified, and one new error class, `NoClassDefFoundError`, has been added.
- In CLDC 1.1, `Thread` objects have names like threads in J2SE do. The method `Thread.getName()` has been introduced, and the `Thread` class has a few new constructors that have been inherited from J2SE.
- Various minor library changes and bug fixes have been included, such as the addition of the following fields and methods:
  - `Boolean.TRUE and Boolean.FALSE`
  - `Date.toString()`
  - `Random.nextInt(int n)`
  - `String.intern()`
  - `String.equalsIgnoreCase()`
  - `Thread.interrupt()`
- Minimum memory budget has been raised from 160 to 192 kilobytes, mainly because of the added floating point functionality.
- Specification text has been tightened and obsolete subsections removed.
- Much more detailed verifier specification ("CLDC Byte Code Typechecker Specification") is provided as an appendix.

CHAPTER **2**

Goals, Requirements and Scope

2.1 Goals

Summary of goals. The goal of the *CLDC Specification* is to standardize a *highly portable, minimum-footprint Java™ application development platform for resource-constrained, connected devices.*

The devices targeted by the *CLDC Specification* have the following general characteristics:

- at least 192 kB of total memory budget available for the Java platform (see Section 2.2.1 "Hardware requirements"),
- a 16-bit or 32-bit processor,
- low power consumption, often operating with battery power,
- connectivity to some kind of network, often with a wireless, intermittent connection and with limited bandwidth.

Cell phones, two-way pagers, personal digital assistants (PDAs), organizers, home appliances, low-end TV set-top boxes, and point of sale terminals are some, but not all, of the devices that might be supported by this specification.

More specifically, the *CLDC Specification* defines a Java application development platform with the following characteristics and goals:

- **Keep footprint small**. Consumer devices such as cellular phones are manufactured in very large quantities (hundreds of thousands, millions, or even tens of millions of units per year). They are sold to price-conscious consumers at very low, often subsidized prices. To maintain profit margins, device manufacturers want to keep the per-unit costs of the devices as low as possible. Additional processing power or precious dynamic memory will not be added unless the consumers are willing to pay for the added capabilities. To meet the needs of the device manufacturers, the *CLDC Specification* defines a "lowest common denominator" standard that includes only the minimal Java platform features and APIs for a wide range of consumer devices.

- **Focus on application programming rather than systems programming**. CLDC is intended to be primarily an *application development platform*, rather than a systems programming environment. This has certain implications for the Java platform features and APIs to be included in this specification. First, the *CLDC Specification* shall include only high-level libraries that provide sufficient programming power for the application developer. Second, we emphasize the importance of generality and portability. The *CLDC Specification* shall not provide any APIs that are specific to a certain device category, vertical market or system functionality.
- **Enable dynamic downloading of applications and encourage third-party application development**. Unlike in the past, when small devices such as cell phones and pagers usually came with a feature set that was hard-coded at the factory, device manufacturers are increasingly looking for solutions that allow them to build *extensible* devices that support the dynamic downloading of interactive content from content providers and third party developers. With the recent introduction of Internet-enabled cell phones, communicators and pagers, this transition is already underway. One of the key goals of the *CLDC Specification* is to define an environment that is well-suited for the dynamic, secure downloading of Java applications over different kinds of networks to small client devices.

The focus on dynamically delivered Java applications means that this specification is intended not just for hardware manufacturers and their system programmers, but also for *third party application developers*. In fact, we assume that once small Java-enabled devices become commonplace, the vast majority of application developers for these devices will be third party developers rather than device manufacturers themselves.

2.2 Requirements

2.2.1 Hardware requirements

CLDC is intended to run on a wide variety of small devices. The underlying hardware capabilities of these devices vary considerably, and therefore the *CLDC Specification* does not impose any specific hardware requirements other than memory requirements. Even for memory limits, the *CLDC Specification* defines only *minimum* limits. The actual CLDC target devices may have significantly more memory than the minimum.

The *CLDC Specification* assumes that:

- At least 160 kilobytes of non-volatile[1] memory is available for the virtual machine and CLDC libraries.
- At least 32 kilobytes of volatile memory[2] is available for the virtual machine runtime (for example, the object heap.)

The ratio of volatile to non-volatile memory in the total memory budget can vary considerably depending on the target device and the role of the Java platform in the device. If the Java platform is used strictly for running system applications that are built in a device, then applications can be prelinked and preloaded, and a very limited amount of volatile memory is needed. If the Java platform is used for running dynamically downloaded content, then devices will need a higher ratio of volatile memory.

2.2.2 Software requirements

Like the hardware capabilities, the system software in CLDC target devices varies considerably. For instance, some of the devices may have a full-featured operating system that supports multiple, concurrent operating system processes and a hierarchical file system. Many other devices may have extremely limited system software with no notion of a file system. Faced with such variety, CLDC makes minimal assumptions about the underlying system software.

Generally, the *CLDC Specification* assumes that a minimal *host operating system* or kernel is available to manage the underlying hardware. This host operating system must provide at least one schedulable entity to run the Java virtual machine. The host operating system does not need to support separate address spaces or processes, nor does it have to make any guarantees about the real-time scheduling or latency behavior.

2.2.3 J2ME requirements

CLDC is defined as a Java 2 Micro Edition (J2ME) *configuration*. This has certain important implications for the *CLDC Specification*:

- A J2ME configuration shall only define a *minimum complement* or the "*lowest common denominator*" of Java technology. All the features included in a configuration must be generally *applicable to a wide variety of devices*. This means that the scope of the *Connected Limited Device Configuration* is limited and possibly incomplete for real target devices. Additional features specific to a certain vertical market, device category or industry should be defined in a J2ME *profile* specification.
- Since the goal of the configuration is to guarantee portability and interoperability between various kinds of resource-constrained devices, the configuration *shall not define any optional features*. This limitation has a significant

1. The term *non-volatile* is used to indicate that the memory is expected to retain its contents between the user turning the device "on" or "off". For the purposes of the *CLDC Specification*, it is assumed that non-volatile memory is usually accessed in read mode, and that special setup may be required to write to it. Examples of non-volatile memory include ROM, Flash and battery-packed SDRAM. The *CLDC Specification* does not define which memory technology a device must have, nor does it define the behavior of such memory in a power-loss scenario.

2. The term *volatile* is used to indicate that the memory is not expected to retain its contents between the user turning the device "on" or "off". For the purposes of the *CLDC Specification*, it is assumed that volatile memory can be read and written to directly without any special setup. The most common type of volatile memory is DRAM.

impact on what can be included in the configuration and what should not be included. The more domain-specific functionality must be defined in J2ME *profiles* or optional packages rather than in CLDC.

- A J2ME configuration specification shall generally define a *subset* of the Java technology features and libraries provided by the Java 2 Platform, Standard Edition (J2SE). Consequently, rather than providing a complete description of all the supported features, the *CLDC Specification* shall only define the variances and differences compared to the full Java Language Specification (JLS) and Java Virtual Machine Specification (JVMS). If something is not explicitly specified in the *CLDC Specification*, then it is assumed that a virtual machine conforming to the *CLDC Specification* shall comply with the JLS and JVMS.

For further information on the rules and guidelines for defining J2ME configurations and profiles, refer to the *J2ME™ Platform Specification*.

Note that the absence of optional features in CLDC does not preclude the use of various *implementation-level optimizations*. For instance, at the implementation level alternative bytecode execution techniques (such as Just-In-Time compilation) or class representation techniques can be used, as long as the observable user-level semantics of the implementation remain the same as defined by the *CLDC Specification*.

2.3 Scope

Based on the decisions of the JSR-30 and JSR-139 expert groups, the *CLDC Specification* shall address the following areas:

- Java language and virtual machine features
- Core Java libraries (`java.lang.*`, `java.util.*`)
- Input/output (`java.io.*`)
- Security
- Networking
- Internationalization

This *CLDC Specification* shall *not* address the following areas:

- Application installation and life-cycle management
- User interface functionality
- Event handling
- High-level application model (the interaction between the user and the application)

These features can be addressed by profiles implemented on top of the CLDC.

The CLDC expert group intentionally decided to keep small the number of areas addressed by CLDC. It seemed better to restrict the scope of CLDC in order not to exceed the strict memory limitations or to exclude any particular device category. Additional functionality areas are better addressed in J2ME profile specifications.

The rest of this specification is organized as follows. The specification starts with a discussion of the high-level architecture of a typical CLDC environment. Then, the specification compares the Java language and virtual machine features of a virtual machine conforming to CLDC to a conventional Java environment meeting the full *Java™ Language Specification* and *Java™ Virtual Machine Specification*. Finally, the specification describes the Java libraries included in CLDC.

CHAPTER **3**

High-level Architecture and Security

This chapter discusses the high-level architecture of a typical CLDC environment. This discussion serves as a starting point for more detailed specification in later chapters.

3.1 CLDC High-Level Architecture

The high-level architecture of a typical J2ME device is illustrated in FIGURE 1. At the heart of a CLDC implementation ("Configuration") is the *Java Virtual Machine (JVM)*, which, apart from specific differences defined later in this specification, is compliant with the *Java™ Virtual Machine Specification* and *Java™ Language Specification*. The virtual machine typically runs on top of a *Host Operating System* that is outside the scope of CLDC.

On top of the virtual machine are the Java *libraries*. Some of these libraries are defined by the Connected Limited Device Configuration itself, as represented in FIGURE 1. In addition, J2ME profiles may define additional libraries and features that sit on top of the configuration layer.

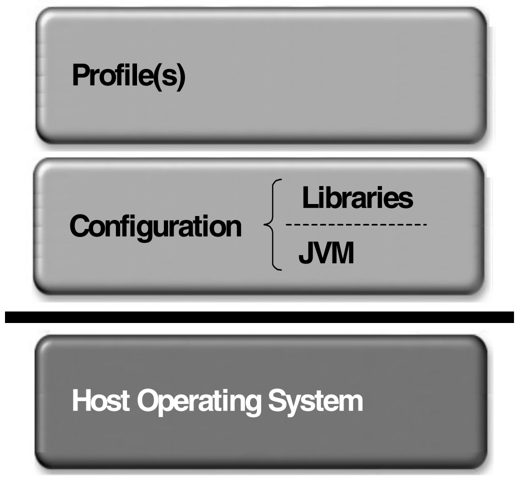

FIGURE 1 High-level architecture

3.2 The Concept of a Java Application

The Connected Limited Device Configuration is not targeted to any specific device category. Many of the target devices may have an advanced graphical user interface, some target devices may be operated from a textual, character-based user interface, while some other target devices may not have any visual user interface or display at all. To cater to such a broad variety of devices, the application model defined in the *CLDC Specification* is intentionally very simple.

In the *CLDC Specification*, the term *Java application* is used to refer to a collection of Java class files containing a single, unique method `main` that identifies the launch point of the application. As specified in *Java™ Virtual Machine Specification*, §5.2 and §2.17.1, the method `main` must be declared `public`, `static` and `void`, as shown below:

```
public static void main(String[] args)
```

A virtual machine conforming to the *CLDC Specification* starts the execution of a Java application by calling the method `main`.

J2ME profiles such as MIDP may define alternative application models that extend or override the basic application model defined by the *CLDC Specification*.

3.3 Application Management

Many small, resource-constrained devices do not have a file system or any other standard mechanism for storing dynamically downloaded information on the device. Therefore, a CLDC implementation shall not require that Java applications downloaded from an external source are stored persistently on the device. Rather, the implementation might just load the application and discard it immediately after execution.

However, in many potential CLDC devices, it is beneficial to be able to execute the same Java applications multiple times without having to download the applications over and over again. This is particularly important if applications are being downloaded over a wireless network, and the user could incur high downloading expenses. If a device implementing CLDC is capable of storing applications persistently, we assume that the device implementing has capabilities for managing the applications that have been stored in the device. At the high level, *application management* refers to the ability to:

- download and install Java applications,
- inspect existing Java applications stored on the device,
- select and launch Java applications,
- delete existing Java applications (if applicable).

Depending on the resources available on the device, a CLDC system can allow multiple Java applications to execute concurrently, or can restrict the system to permit only the execution of one Java application at a time. It is up to the particular CLDC implementation to decide if the execution of multiple Java applications is supported by utilizing the multitasking capabilities (if they exist) of the underlying host operating system, or by instantiating multiple logical virtual machines to run the concurrent Java applications.

Due to significant variations and feature differences among potential CLDC devices, the details of application management are highly device-specific and implementation-dependent. The actual details regarding application management are outside the scope of the *CLDC Specification*.

3.4 Security

With corporations and individuals depending increasingly on critical information stored in computer systems and networks, security issues are becoming ever more important, and even more so in the context of mobile computing and wireless networks. Due to its inherent security architecture, the Java development platform is particularly well-suited to security-critical environments. The security model provided by the Java 2 Platform, Standard Edition (J2SE) provides developers with a powerful and flexible security framework that is built into the Java platform.

Developers can create fine-grained security policies and articulate independent permissions for individual applications, all while appearing transparent to the end user.

Unfortunately, the total amount of code devoted to security in the Java 2 Platform, Standard Edition far exceeds the total memory budget available for CLDC. Therefore, some simplifications are necessary when defining the security model for CLDC. The security model of CLDC is defined at three different levels:

1. *Low-level security*. Low-level security, also known as *virtual machine security*, ensures that the applications running in the virtual machine follow the semantics of the Java programming language, and that an ill-formed or maliciously-encoded class file does not crash or in any other way harm the target device.

2. *Application-level security*. Application-level security means that a Java application running on a device can access only those libraries, system resources and other components that the device and the Java application environment allows it to access.

3. *End-to-end security*. End-to-end security refers to a model that guarantees that any transaction initiated on a device is protected along the entire path from the device to/from the entity providing the services for that transaction (e.g, a server located on the Internet). Encryption or other means may be necessary to achieve this. End-to-end security is outside the scope of the *CLDC Specification*, and is assumed to be defined by those J2ME profiles that provide facilities for end-to-end software development.

Below we take a more detailed look at each of these levels.

3.4.1 Low-level (virtual machine) security

A key requirement for a Java virtual machine in a mobile information device is *low-level virtual machine security*. An application running in the virtual machine must not be able to harm the device in which the virtual machine is running, or crash the device. In a standard Java virtual machine implementation, this constraint is guaranteed by the *class file verifier*, which ensures that the bytecodes and other items stored in class files cannot contain illegal instructions, cannot be executed in an illegal order, and cannot contain references to invalid memory locations or memory areas that are outside the Java object memory (the object heap). In general, the role of the class file verifier is to ensure that class files loaded into the virtual machine do not execute in any way that is not allowed by the *Java™ Virtual Machine Specification*.

As will be explained in more detail in Section 5.2 "Class File Verification," the *CLDC Specification* requires that a Java virtual machine conforming to the CLDC standard must be able to reject invalid class files. This is guaranteed by the class file verification technology presented in Section 5.2 "Class File Verification."

3.4.2 Application-level security

Even though class file verification plays a critical role in ensuring the security of the Java platform, the security provided by the class file verifier is insufficient by itself. The class file verifier can only guarantee that the given program is a valid Java application and nothing more. There are still several other potential security threats that will go unnoticed by the verifier. For instance, access to external resources such as the file system, printers, infrared devices, native libraries or the network is beyond the scope of the class file verifier. By *application-level security,* we mean that a *Java application can access only those libraries, system resources and other components that the device and the Java application environment allows it to access*. The details of application-level security are discussed below.

3.4.2.1 Sandbox model

In CLDC, application-level security is accomplished by using a metaphor of a closed "*sandbox*." An application must run in a closed environment in which the application can access only those libraries that have been defined by the configuration, profiles, and other classes supported by the device. Java applications cannot escape from this sandbox or access any libraries or resources that are not part of the predefined functionality. The sandbox ensures that a malicious or possibly erroneous application cannot gain access to system resources.

More specifically, the CLDC sandbox model requires that:

- Class files must be properly verified and guaranteed to be valid Java applications. (Class file verification is discussed in more detail in Section 5.2 "Class File Verification")
- The downloading, installation, and management of Java applications on the device takes place in such a way that the application programmer cannot modify or bypass the standard class loading mechanisms of the virtual machine.
- A closed, predefined set of Java APIs is available to the application programmer, as defined by CLDC, profiles (such as MIDP) and manufacturer-specific classes.
- The set of native functions accessible to the virtual machine is closed, meaning that the application programmer cannot download any new libraries containing native functionality or access any native functions that are not part of the Java libraries provided by CLDC, profiles or manufacturer-specific classes.

J2ME profiles may provide additional security solutions. Profiles also define which additional APIs are available to the application programmer.

3.4.2.2 Protecting system classes

A central requirement for CLDC is the ability to support dynamic downloading of Java applications to the virtual machine. A possible application-level security hole in the Java virtual machine would be exposed if the downloaded applications could override or extend the set of the system classes provided in packages `java.*`, `javax.microedition.*` or other profile-specific or manufacturer-specific packages. A CLDC implementation shall ensure that the application programmer cannot override, modify, or add any classes to these protected system packages.

For security reasons, it is also required that the application programmer is not allowed to manipulate the class file lookup order in any way. Class file lookup order is discussed in more detail in Section 5.3.3 “Class file lookup order.”

3.4.2.3 Additional restrictions on dynamic class loading

Dynamic loading of Java applications is a key feature of CLDC. However, the *CLDC Specification* defines the class loading mechanism of a virtual machine conforming to CLDC to be implementation-dependent, with one important restriction: by default, a Java application can load application classes only from its own Java Archive (JAR) file. This restriction ensures that Java applications on a device cannot interfere with each other or steal data from each other. Additionally, this ensures that a third-party application cannot gain access to the private or protected components of the Java classes that the device manufacturer or a service provider may have provided as part of the system applications. JAR files and applications representation formats are discussed in more detail in Section 5.3 “Class File Format and Class Loading.”

3.4.3 End-to-end security

A device conforming to the *CLDC Specification* is typically a part of an end-to-end solution such as a wireless network or a payment terminal network. These networks commonly require a number of advanced security solutions (e.g., encryption) to ensure safe delivery of data and code between server machines and client devices. Given the broad diversity of network infrastructure in the world, the CLDC expert group decided not to mandate a single end-to-end security mechanism. Therefore, all the end-to-end security solutions are assumed to be implementation-dependent and outside the scope of *CLDC Specification*.

CHAPTER **4**

Adherence to the Java Language Specification

The general goal for a virtual machine conforming to CLDC is to be as compliant with the *Java™ Language Specification* as is feasible within the strict memory limits of CLDC target devices. This chapter summarizes the differences between a virtual machine conforming to CLDC and the Java virtual machine of Java 2 Standard Edition (J2SE). Except for the differences indicated herein, a virtual machine conforming to CLDC shall be compatible with *The Java™ Language Specification (Java Series), Second Edition* by James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha, Addison-Wesley, 2000, ISBN 0-201-31008-2.

Note – For the remainder of this specification, the *Java™ Language Specification* will be referred to as JLS. Sections within the *Java™ Language Specification* will be referred to using the § symbol. For example, (JLS §4.2.4).

4.1 No Finalization of Class Instances

CLDC libraries do not include the method `Object.finalize()`. Therefore, a virtual machine conforming to CLDC does not support finalization of class instances (JLS §12.6). No application built to conform to the Connected Limited Device Configuration shall require that finalization is available.

4.2 Exception and Error Handling Limitations

A virtual machine conforming to CLDC shall generally support *exception* handling as defined in JLS Chapter 11, with the exception that *asynchronous exceptions* (JLS §11.3.2) are not supported.

In contrast, the set of *error* classes included in CLDC libraries is limited, and consequently the error handling capabilities of CLDC are considerably more limited. This is because of two reasons:

1) In embedded systems, recovery from error conditions is usually highly device-specific. Application programmers cannot be expected to worry about device-specific error handling mechanisms and conventions.

2) As specified in JLS §11.5, class `java.lang.Error` and its subclasses are exceptions from which programs are not ordinarily expected to recover. Implementing the error handling capabilities fully according to the *Java™ Language Specification* is rather expensive, and mandating the presence and handling of all the error classes would impose an overhead on the virtual machine implementation.

A virtual machine conforming to CLDC shall support the set of `Error` classes defined in Section 6.2 "Classes Derived from Java 2 Standard Edition." When encountering any other error, the implementation shall behave as follows:

- either the virtual machine halts in an implementation-specific manner,
- or the virtual machine throws an `Error` that is the nearest CLDC-supported superclass of the `Error` class that must be thrown according to the *Java™ Language Specification*.

If the virtual machine conforming to CLDC implements additional error checks that are part of the *Java™ Language Specification* but that are not required by the *CLDC Specification*, the implementation shall throw the nearest CLDC-supported superclass of the `Error` class that is defined by the *Java™ Language Specification*.

CHAPTER 5

Adherence to Java Virtual Machine Specification

The general goal for a virtual machine conforming to CLDC is to be as compliant with the *Java™ Virtual Machine Specification* as is possible within strict memory constraints of CLDC target devices. This chapter summarizes the differences between a virtual machine conforming to CLDC and the Java virtual machine of Java 2 Standard Edition (J2SE). Except for the differences indicated herein, a virtual machine conforming to CLDC shall be compatible with the Java virtual machine as specified in the *The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999), ISBN 0-201-43294-3.

Note – For the remainder of this specification, the *Java™ Virtual Machine Specification* are referred to as JVMS. Sections within the *Java™ Virtual Machine Specification* are referred to using the § symbol. For example, (JVMS §2.4.3).

5.1 Features Eliminated from the Virtual Machine

A number of features have been eliminated from a virtual machine conforming to CLDC because the Java libraries included in CLDC are substantially more limited than the class libraries of Java 2 Standard Edition, and/or the presence of those features would have posed security problems in the absence of the full J2SE security model. The eliminated features include:

- User-defined class loaders (JVMS §5.3.2)
- Thread groups and daemon threads (JVMS §2.19, §8.12)
- Finalization of class instances (JVMS §2.17.7)
- Asynchronous exceptions (JVMS §2.16.1)

In addition, a virtual machine conforming to CLDC has a significantly more limited set of error classes than a full J2SE virtual machine.

Applications written for CLDC shall not rely on any of the features above. Each of the features in this list is discussed in more detail below.

5.1.1 User-defined class loaders

A virtual machine conforming to CLDC does not support user-defined, Java-level class loaders (JVMS §5.3, §2.17.2). A virtual machine conforming to CLDC shall have a built-in "bootstrap" class loader that cannot be overridden, replaced, or reconfigured. The elimination of user-defined class loaders is part of the security restrictions presented in Section 3.4.2.1 "Sandbox model."

5.1.2 Thread groups and daemon threads

A virtual machine conforming to CLDC implements multithreading, but does not have support for thread groups or daemon threads (JVMS §2.19, §8.12). Thread operations such as starting of threads can be applied only to individual thread objects. If application programmers want to perform thread operations for groups of threads, explicit collection objects must be used at the application level to store the thread objects.

5.1.3 Finalization of class instances

CLDC libraries do not include the method `Object.finalize()`. Therefore, a virtual machine conforming to CLDC does not support finalization of class instances (JVMS §2.17.7). No application running on top of a virtual machine conforming to CLDC shall require that finalization be available.

5.1.4 Errors and asynchronous exceptions

As discussed earlier in Section 4.2 "Exception and Error Handling Limitations," the error handling capabilities of a virtual machine conforming to CLDC are limited.

A virtual machine conforming to CLDC shall support the set of `Error` classes defined in Section 6.2 "Classes Derived from Java 2 Standard Edition." When encountering any other error, the implementation shall behave as follows:

- either the virtual machine halts in an implementation-specific manner,
- or the virtual machine throws an `Error` that is the nearest CLDC-supported superclass of the `Error` class that must be thrown according to the *Java™ Virtual Machine Specification*.

If the virtual machine conforming to CLDC implements additional error checks that are part of the *Java™ Virtual Machine Specification* but that are not required by the *CLDC Specification*, the implementation shall throw the nearest CLDC-supported superclass of the `Error` class that is defined by the *Java™ Virtual Machine Specification*.

A virtual machine conforming to CLDC shall generally support *exception* handling as defined by *Java™ Virtual Machine Specification*. However, a virtual machine conforming to CLDC does not support *asynchronous exceptions* (JVMS §2.16.1).

5.2 Class File Verification

Like the Java virtual machine of Java 2 Standard Edition, a virtual machine conforming to CLDC must be able to reject invalid class files. This means that a CLDC implementation must support class file verification.

Class file verification in CLDC can be implemented in two different ways[1]:

1. Using the standard class file verification approach defined in the *Java™ Virtual Machine Specification* (JVMS §4.9).
2. Using the alternative, more efficient verification approach defined in this specification. This approach is described below and in Appendix 1.

5.2.1 Off-device preverification and runtime verification with stack maps

The conventional J2SE class file verifier defined in *Java™ Virtual Machine Specification* (JVMS §4.9) is not ideal for small, resource-constrained devices. The conventional verifier takes a minimum of 50 kB binary code space, and typically at least 30-100 kB of dynamic RAM at runtime. In addition, the CPU power needed to perform the complex iterative dataflow algorithm in the conventional verifier can be substantial.

The verification approach described in this subsection is significantly smaller and more efficient in resource-constrained devices than the existing J2SE verifier. The implementation of the new verifier in Sun's KVM requires about ten kilobytes of Intel x86 binary code and less than 100 bytes of dynamic RAM at runtime for typical class files. The verifier performs only a linear scan of the bytecode, without the need of a costly iterative dataflow algorithm.

The new class file verifier operates in two phases, as illustrated in FIGURE 2:

1. Important: Regardless of which class file verification solution is used, the class files must be preverified and contain the `StackMap` attributes. See Section 5.3.2 "Public representation of Java applications and resources" for further information.

1. First, class files have to be run through a *preverifier* tool in order to remove certain bytecodes and augment the class files with additional `StackMap` attributes to speed up runtime verification. The preverification phase is typically performed on the development workstation that the application developer uses for writing and compiling the applications.
2. At runtime, the *runtime verifier* component of the virtual machine uses the additional `StackMap` attributes generated by the preverifier to perform the actual class file verification efficiently.

The execution of bytecodes in a class file can start only when the class file has successfully passed the runtime verifier.

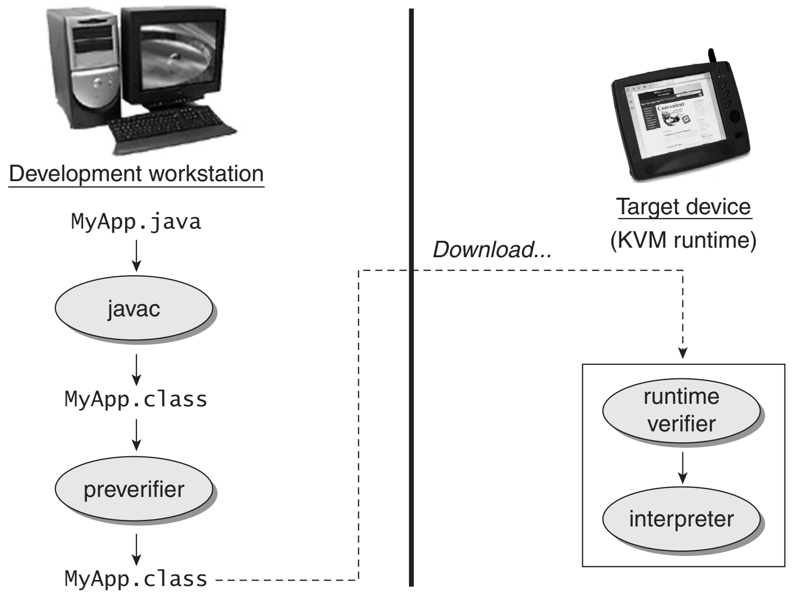

FIGURE 2 Two-phase class file verification in CLDC

Runtime class file verification guarantees type safety. Classes that pass the runtime verifier cannot, for example, violate the type system of the Java virtual machine or corrupt the memory. Unlike approaches based on code signing, such a guarantee does not rely on the verification attribute to be authentic or trusted. A missing, incorrect or corrupted verification attribute causes the class to be rejected by the runtime verifier.

The new verifier requires the methods in class files to contain a special `StackMap` attribute. The *preverifier* tool inserts this attribute into normal class files. A transformed class file is still a valid Java class file, with additional `StackMap` attributes that allow verification to be carried out efficiently at runtime. These attributes are automatically ignored by the conventional class file verifier used in Java 2 Standard Edition, so the solution is fully upward compatible with the Java virtual machine of Java 2 Standard Edition. Preprocessed class files containing the extra attributes are approximately 5 to 15 percent larger than the original, unmodified class files.

Additionally, the new verifier requires that *all the subroutines in the bytecodes of class files are inlined*. In Java class files, subroutines are special bytecode sequences that contain the bytecodes `jsr`, `jsr_w`, `ret` or `wide ret`. The inlining process removes all the `jsr`, `jsr_w`, `ret` and `wide ret` bytecodes from all the methods in class files, replacing these instructions with semantically equivalent bytecode. The inlining process makes runtime verification significantly easier and faster.

5.2.1.1 Verification process

Below is a more detailed description of the two-phase verification process (further details are provided in Appendix 1):

Phase 1: Preverification (off-device)

The preverification tool provided with the new verifier performs the following two operations:

- Inline all subroutines and remove all the `jsr` (JVMS p. 304, `jsr_w` (JVMS p. 305), `ret` (JVMS p. 352) and `wide ret` (JVMS p. 360) bytecodes from the class file. Each method containing these instructions is replaced with semantically equivalent bytecode not containing the `jsr`, `jsr_w`, `ret` and `wide ret` bytecodes.
- Add special `StackMap` attributes into class files to facilitate runtime verification. The format and the semantics of the `StackMap` attribute is defined in Appendix 1.

In addition, the preverification tool will perform other non-semantical code transformations in order to guarantee that, if passed a valid Java class or interface as defined by the JVMS, the resulting code will pass in-device verification, provided that the run-time type hierarchy is identical to the type hierarchy that was present during preverification. If passed an invalid Java class or interface, the preverification tool will not produce a class or interface that can pass in-device verification.

Phase 2: In-device verification

The in-device verification algorithm consists of the following steps:

- First, the verifier allocates enough memory for storing the types of all local variables and operand stack items of a given method. The memory size is determined by the maximum number of local variables and maximum stack depth specified in the `Code` attribute. This memory area is used to store the derived types as the verifier makes a linear pass through the bytecode. This is the only piece of memory the verifier allocates.
- Second, the verifier initializes the derived types to be the type of the `this` pointer for instance methods, argument types, and an empty operand stack.

- Third, the verifier linearly iterates through each instruction. For each instruction, the following happens:
  - If the previous instruction is either unconditional jump (e.g., `goto`), return `(e.g., ireturn), athrow, tableswitch`, or `lookupswitch`, or the current instruction is at the starting byte code offset of an exception handler, there is no direct control flow from the previous instruction. The verifier ignores the current derived types and sets the derived types according to the stack map entry recorded for the current instruction. If the current instruction does not have a stack map entry the verifier reports an error.
  - If the previous instruction is not unconditional jump (e.g., `goto`), return `(e.g., ireturn), athrow, tableswitch`, or `lookupswitch`, there is direct control flow from the previous instruction. The verifier checks if there is a stack map entry recorded for the current instruction. If there is, the verifier attempts to match the derived types with the recorded stack map entry. If the recorded types are of the same or a more general type than the derived types, the derived types are set to be the recorded types. Otherwise, the verifier reports an error.
  - If the current instruction is in the scope of an exception handler (`try` block), the derived types are matched against the stack map entry that corresponds to the starting bytecode offset of the exception handler. The verifier reports an error if there is not a stack map entry that corresponds to the starting bytecode offset of the exception handler.
  - The verifier then attempts to match the derived types with what is expected by the instruction. The `iadd` instruction, for example, expects the top two operand stack items to be integers. The derived types are then modified according to what the instruction does. The `iadd` instruction, for example, pops two integers off the operand stack and pushes an integer result onto the operand stack.
  - Finally the verifier attempts to match the derived types with any stack map entries recorded for any successor instructions that do not directly follow the current instruction.
- Finally, the verifier makes sure that the last instruction in the method is a unconditional jump (e.g., `goto`), return `(e.g., ireturn), athrow`, `tableswitch`, or `lookupswitch`. Otherwise it reports a verification error: the control flow falls through the end of the method.

In addition to the steps discussed above, the in-device verifier must perform an additional check: the verifier must distinguish newly allocated objects from those on which a constructor has been invoked. It must make sure that constructors are invoked exactly once on an object allocated by a `new` instruction at a given bytecode offset, that the only legal operation on newly allocated objects is to invoke its constructor, and that there are no newly allocated objects in local variables or on the operand stack when a backward branch may be taken.

5.2.1.2 Stack map attribute definition

The new runtime verifier requires the Java class files to contain special attributes. Because these attributes describe the types of local variables and operand stack items, all of which reside on the interpreter stack, the attribute is known as a "stack map."

Each `StackMap` attribute consists of multiple entries, with each entry recording the types of local variables and operand stack items at a given bytecode offset.

The `StackMap` attribute is a sub-attribute of the `Code` attribute defined in JVMS §4.7.3. See page 120 of *The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999) for a detailed description of the `Code` attribute and how the stack map attribute fits in as part of the `Code` attribute.

Refer to Appendix 1 for the formal definition of the `StackMap` attribute.

5.3 Class File Format and Class Loading

An essential requirement for the Connected Limited Device Configuration is the ability to support dynamic downloading of Java applications and third party content. The dynamic class loading mechanism of the Java platform plays a central role in enabling this. This section discusses the application representation formats and class loading practices required of a virtual machine conforming to CLDC.

5.3.1 Supported file formats

A CLDC implementation must be able to read standard Java class files (defined in JVMS Chapter 4) with the preverification changes defined in Section 5.3.2 "Public representation of Java applications and resources".) In addition, a CLDC implementation must support compressed Java Archive (JAR) files. Detailed information about JAR format is provided at `http://java.sun.com/products/jdk/1.3/docs/guide/jar`.

Network bandwidth conservation is very important in low-bandwidth wireless networks. The compressed JAR format provides 30 to 50 percent compression over regular class files without loss of any symbolic information or compatibility problems with existing Java systems.

A CLDC implementation must be able to read Java class files in all the formats supported by Java 2 Platform Standard Edition, JDK versions 1.1, 1.2, 1.3 and 1.4.

Note – The class file format numbers used by different JDK versions are as follows[1]:
- The 45.* (usually 45.3) version number identifies JDK 1.1 class files.
- The 46.* version number identifies JDK 1.2 class files.
- The 47.* version number identifies JDK 1.3 class files.
- The 48.* version number identifies JDK 1.4 class files.

A virtual machine conforming to CLDC should be able to read Java class files in any of the formats listed above. However, the virtual machine is allowed to ignore certain class file attributes that the CLDC implementation does not need. More specifically, the following attributes can be ignored by a CLDC implementation:

- The `Synthetic` attribute (JVMS §4.7.6)
- The `SourceFile` attribute (JVMS §4.7.7)
- The `LineNumberTable` attribute (JVMS §4.7.8)
- The `LocalVariableTable` attribute (JVMS §4.7.9)
- The `Deprecated` attribute (JVMS §4.7.10)

For historical reasons (JVMS p. 127), a virtual machine conforming to CLDC is not required to check the well-formedness of the `InnerClasses` attribute (JVMS §4.7.5).

5.3.2 Public representation of Java applications and resources

A Java application is considered to be "*represented publicly*" or "*distributed publicly*" when the system it is stored on is open to the public, and the transport layers and protocols it can be accessed with are open standards. In contrast, a device can be part of a *closed network environment* where the vendor (such as the operator of a wireless network) controls all communication. In this case, the application is no longer represented publicly once it enters and is distributed via the closed network system.

Whenever Java applications intended for a CLDC device are represented publicly, the compressed JAR file representation format must be used. The JAR file must contain *preverified* Java class files, as defined in Section 5.2.1 "Off-device preverification and runtime verification with stack maps" of this specification.

Sun's CLDC reference implementation includes a preverification tool for performing these modifications to a class file. As explained earlier, the `StackMap` attributes generated by the preverification tool are automatically ignored by the conventional J2SE class file verifier described in JVMS §4.9. That is to say, the modified class file format is fully upwards compatible with the larger Java environments such as J2SE or J2EE.

1. In reality, the story is a bit more complicated. In pre-JDK1.4 releases, the `javac` compiler produced 45.3 class files by default for compatibility with older VMs. In JDK 1.4, `javac` produces 46.0 (JDK1.2) class files by default. However, the JDK 1.4 `javac` can be instructed to produce other class file formats by using the "`-target`" option. For instance, "`javac -target 1.1`" would produce 45.3 class files.

In general, if a virtual machine conforming to CLDC implements class file verification using the conventional verification approach defined in the *Java™ Virtual Machine Specification* (JVMS §4.9), the virtual machine is allowed to ignore the `StackMap` attributes. Such an implementation may also provide support for the `jsr`, `jsr_w`, `ret` and `wide ret` bytecodes.

Additionally, the JAR file may contain *application-specific resource files* that can be loaded into the virtual machine by calling method `Class.getResourceAsStream(String name)` (see the CLDC library documentation for details.)

5.3.3 Class file lookup order

The *Java™ Language Specification* and *Java™ Virtual Machine Specification* do not specify the order in which class files are searched when new class files are loaded into the virtual machine. At the implementation level, a typical Java virtual machine implementation utilizes a special environment variable `classpath` to define the lookup order.

This *CLDC Specification* assumes class file lookup order to be implementation-dependent, with the restrictions described in the next paragraph. The lookup strategy is typically defined as part of the application management implementation (see Section 3.3 "Application Management.") A virtual machine conforming to CLDC is not required to support the notion of `classpath`, but may do so at the implementation level.

Two restrictions apply to class file lookup order. Both these restrictions are important for security reasons:

1. As explained in Section 3.4.2.2 "Protecting system classes," a virtual machine conforming to CLDC must guarantee that the application programmer cannot override, modify, or add new system classes (classes belonging to the CLDC, supported profiles or manufacturer-specific classes) in any way.
2. The application programmer must not be able to manipulate the class file lookup order in any way.

5.3.4 Implementation optimizations and alternative application representation formats

Preloading/prelinking ("ROMizing"). A virtual machine conforming to CLDC may choose to preload/prelink some classes. This technology is referred to informally as *ROMizing*.[1] Typically, small virtual machine implementations choose to preload all the system classes (for instance, classes belonging to a specific configuration or profile), and perform application loading dynamically.

1. The term *ROMizing* is somewhat misleading, since this technology can be used independently of any specific memory technology. ROMized class files do not necessarily have to be stored in ROM.

The actual mechanisms for preloading are implementation-dependent and beyond the scope of the *CLDC Specification*. In all cases, the runtime effect and semantics of preloading/prelinking must be the same as if the actual class had been loaded in at that point. There must be no visible effects from preloading other than the possible speed-up in application launching. In particular, any class initialization that has a user-visible effect must be performed at the time the class would have first been loaded if it had not been preloaded into the system.

Other implementation-level optimizations. Java class files are not optimized for network transport in bandwidth-limited environments. Each Java class file is an independent unit that contains its own constant pool (symbol table), method, field and exception tables, bytecodes, exception handlers, and some other information. The self-contained nature of class files is one of the virtues of Java technology, allowing applications to be composed of multiple pieces that do not necessarily have to reside in the same location, and making it possible to extend applications dynamically at runtime. However, this flexibility has its price. If Java applications were treated as a sealed unit, a lot of space could be saved by removing the redundancies in multiple constant pools and other structures, especially if full symbolic information was left out. Also, one of the desirable features of an application transport format in a limited-power computing environment is the ability to execute applications "in-place," without any special loading or conversion process between the static representation and runtime representation. Standard Java class files are not designed for such execution.

The *CLDC Specification* mandates the use of compressed JAR files for Java applications that are represented and distributed publicly. However, in closed network environments (see the discussion in Section 5.3.2 "Public representation of Java applications and resources") and internally inside the virtual machine at runtime, alternative formats can be used. For instance, in low-bandwidth wireless networks it is often reasonable to use alternative, more compact transport formats at the network transport level to conserve network bandwidth. Similarly, when storing the downloaded applications in CLDC devices, more compact representations can be used, as long as the observable user-level semantics of the applications remain the same as with the original representation. The alternative formats may not be used for representing or distributing CLDC Java applications publicly, i.e., the public representation format of CLDC Java applications must always be as defined in Section 5.3.2 "Public representation of Java applications and resources."

The definition of alternative application representations is assumed to be implementation-dependent and outside the scope of the *CLDC Specification*.

CHAPTER **6**

CLDC Libraries

6.1 Overview

The Java 2 Platform, Standard Edition (J2SE) and the Java 2 Platform, Enterprise Edition (J2EE) provide a very rich set of libraries for the development of applications for desktop computers and server machines. Unfortunately, these libraries require multiple megabytes of memory to run, and are therefore unsuitable for small devices with limited resources.

A general goal for designing the libraries for the Connected Limited Device Configuration is to provide a minimum useful set of libraries for practical application development and profile definition for a variety of small devices. As explained in Section 2.1 "Goals", CLDC is a "*lowest common denominator*" standard that includes only the minimal Java platform features and APIs for a wide range of consumer devices. Given the strict memory constraints and differing features of today's small devices, it is virtually impossible to come up with a set of libraries that would be ideal for everyone. No matter where the bar for feature inclusion is set, the bar is inevitably going to be too low for some devices and users, and possibly too high for some others.

To ensure upward compatibility with larger editions of the Java 2 Platform, the majority of the libraries included in CLDC are a subset of Java 2 Standard Edition and Java 2 Enterprise Edition. While upward compatibility is a very desirable goal, J2SE and J2EE libraries have strong internal dependencies that make subsetting them difficult in important areas such as security, input/output, user interface definition, networking and storage. These dependencies are a natural consequence of design evolution and reuse that has taken place during the development of Java libraries over time. Unfortunately, these dependencies make it difficult to take just one part of the libraries without including several others. For this reason, we have redesigned some of the libraries, especially in the area of networking.

The CLDC libraries defined by the *CLDC Specification* can be divided into two categories:

- those classes that are a subset of standard J2SE libraries,

- those classes that are specific to CLDC (but which can be mapped onto J2SE).

Classes belonging to the former category are located in packages `java.lang.*`, `java.util`, and `java.io`. A detailed list of these classes is presented in Section 6.2 "Classes Derived from Java 2 Standard Edition."

Classes belonging to the latter category are located in package `javax.microedition.io`. These classes are discussed in Section 6.3 "CLDC-Specific Classes."

6.2 Classes Derived from Java 2 Standard Edition

CLDC supports a number of classes that have been derived from Java 2 Standard Edition, version 1.3.1. The rules for J2ME configurations mandate that each class that has the same name and package name as a J2SE class must be identical to or a subset of the corresponding J2SE class. The semantics of the classes and their methods included in the subset shall not be changed. The classes shall not add any public or protected methods or fields that are not available in the corresponding J2SE classes.

For a definitive reference on the classes listed in this section, refer to Appendix 2 that provides a detailed summary of the CLDC libraries in Javadoc format.

6.2.1 System classes

J2SE class libraries include several classes that are intimately coupled with the Java virtual machine. Similarly, several standard Java tools assume the presence of certain classes in the system. For instance, the standard Java compiler (`javac`) generates code that requires that certain functions of classes `String` and `StringBuffer` be available. The system classes included in the *CLDC Specification* are listed below. Each of these classes is a subset of the corresponding class in J2SE.

```
java.lang.Object
java.lang.Class
java.lang.Runtime
java.lang.System
java.lang.Thread
java.lang.Runnable (interface)
java.lang.String
java.lang.StringBuffer
java.lang.Throwable
```

6.2.2 Data type classes

The following basic data type classes from package `java.lang` are supported. Each of these classes is a subset of the corresponding class in J2SE.

```
java.lang.Boolean
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Float
java.lang.Double
java.lang.Character
```

6.2.3 Collection classes

The following collection classes from package `java.util` are supported.

```
java.util.Vector
java.util.Stack
java.util.Hashtable
java.util.Enumeration (interface)
```

6.2.4 Input/output classes

The following classes from package `java.io` are supported.

```
java.io.InputStream
java.io.OutputStream
java.io.ByteArrayInputStream
java.io.ByteArrayOutputStream
java.io.DataInput (interface)
java.io.DataOutput (interface)
java.io.DataInputStream
java.io.DataOutputStream
```

`java.io.Reader`

`java.io.Writer`

`java.io.InputStreamReader`

`java.io.OutputStreamWriter`

`java.io.PrintStream`

6.2.5 Calendar and time classes

CLDC includes a small subset of the standard J2SE classes `java.util.Calendar`, `java.util.Date`, and `java.util.TimeZone`. To conserve space, the *CLDC Specification* requires only one time zone to be supported. By default, this time zone is GMT. Additional time zones may be provided by manufacturer-specific implementations of CLDC, as long as the time zones are compatible with those provided by Java 2 Standard Edition.

`java.util.Calendar`

`java.util.Date`

`java.util.TimeZone`

6.2.6 Additional utility classes

Two additional utility classes are provided. Class `java.util.Random` provides a simple pseudo-random number generator that is useful for implementing applications such as games. Class `java.lang.Math` provides methods `min`, `max` and `abs` that are frequently used by other Java library classes. Starting from CLDC 1.1, class `java.lang.Math` also includes support for trigonometric functions and square root calculation, as well as some additional utility functions such as `ceil` and `floor`.

`java.util.Random`

`java.lang.Math`

6.2.7 Exception and error classes

Since the libraries included in CLDC are generally intended to be highly compatible with J2SE libraries, the library classes included in CLDC shall throw precisely the same exceptions as regular J2SE classes do. Consequently, a fairly comprehensive set of exception classes has been included.

In contrast, as explained in Section 4.2 “Exception and Error Handling Limitations,” the error handling capabilities of CLDC are limited. By default, a virtual machine conforming to CLDC is required to support only the error classes listed below in Section 6.2.7.2 “Error classes.”

6.2.7.1 Exception classes

```
java.lang.Exception
java.lang.ArithmeticException
java.lang.ArrayIndexOutOfBoundsException
java.lang.ArrayStoreException
java.lang.ClassCastException
java.lang.ClassNotFoundException
java.lang.IllegalAccessException
java.lang.IllegalArgumentException
java.lang.IllegalMonitorStateException
java.lang.IllegalThreadStateException
java.lang.IndexOutOfBoundsException
java.lang.InstantiationException
java.lang.InterruptedException
java.lang.NegativeArraySizeException
java.lang.NullPointerException
java.lang.NumberFormatException
java.lang.RuntimeException
java.lang.SecurityException
java.lang.StringIndexOutOfBoundsException

java.util.EmptyStackException
java.util.NoSuchElementException

java.io.EOFException
java.io.InterruptedIOException
java.io.IOException
java.io.UnsupportedEncodingException
java.io.UTFDataFormatException
```

6.2.7.2 Error classes

```
java.lang.Error
java.lang.NoClassDefFoundError
java.lang.OutOfMemoryError
java.lang.VirtualMachineError
```

6.2.8 Weak References

As of *CLDC Specification* version 1.1, the following weak reference classes are supported:

```
java.lang.ref.Reference
java.lang.ref.WeakReference
```

6.2.9 Internationalization

Character sets and character case conversion support. A CLDC implementation is required to support Unicode characters. Character information is based on the *Unicode Standard, version 3.0*. However, since the full character tables required for Unicode support can be excessively large for devices with tight memory budgets, by default the character property and case conversion facilities in CLDC assume the presence of *ISO Latin-1* range of characters only. More specifically, implementations must provide support for character properties and case conversions for characters in the "*Basic Latin*" and "*Latin-1 Supplement*" blocks of Unicode 3.0. Other Unicode character blocks may be supported as necessary[1].

Character encodings. CLDC includes limited support for the translation of Unicode characters to and from a sequence of bytes. In J2SE this is done using objects called *Readers* and *Writers*, and this same mechanism is utilized in CLDC using the `InputStreamReader` and `OutputStreamWriter` classes with identical constructors.

```
new InputStreamReader(InputStream is);
new InputStreamReader(InputStream is, String enc);
new OutputStreamWriter(OutputStream os);
new OutputStreamWriter(OutputStream os, String enc);
```

If the `enc` parameter is present, it is the name of the encoding to be used. Where it is not, a default encoding (defined by the system property `microedition.encoding`) is used. Additional converters may be provided by particular implementations. If a converter for a certain encoding is not available, an

1. In the CLDC reference implementation, this can be accomplished by overriding an implementation class `com.sun.cldc.i18n.uclc.DefaultCaseConverter`.

UnsupportedEncodingException will be thrown. For official information on character encodings in J2SE, refer to
http://java.sun.com/products/jdk/1.3/docs/guide/intl/encoding.doc.html.

No localization support. Note that CLDC does not provide any *localization* features. This means that all the solutions related to the formatting of dates, times, currencies, and so on are outside the scope of *CLDC Specification*.

6.2.10 Property support

A virtual machine conforming to CLDC does not include the java.util.Properties class familiar from Java 2 Standard Edition. In J2SE, that class is used for storing system properties such as the name of the host operating system, version number of the virtual machine and so on.

In CLDC, a limited set of properties described in TABLE 1 is available. These properties can be accessed by calling the method System.getProperty(*String key*).

TABLE 1 CLDC system properties

Key	**Explanation**	**Value**
microedition.platform	Name of the host platform or device	(implementation-dependent)
microedition.encoding	Default character encoding	Default value: "ISO-8859-1"
microedition.configuration	Name and version of the supported configuration	"CLDC-1.1"
microedition.profiles	Names of the supported profiles	(implementation-dependent)

Property microedition.encoding describes the default character encoding name. This information is used by the system to find the correct class for the default character encoding in supporting internationalization. Property microedition.platform characterizes the host platform or device. Property microedition.configuration describes the current J2ME configuration and version, and property microedition.profiles defines a string containing the names of the supported profiles separated by blanks.

Note that the set of properties defined above can be extended by J2ME profile specifications or device manufacturers. For instance, the MIDP Specification defines additional properties not included in TABLE 1 above.

Manufacturer-specific property definitions should be prefixed with the same package name that the manufacturer-specific classes use (e.g., "com.companyname.propertyname"). The "microedition" namespace, as well as the "java" and "javax" namespaces are reserved, and may only be extended by official J2ME configurations and profiles.

6.3 CLDC-Specific Classes

This section contains a description of the *Generic Connection framework* for supporting input/output and networking in a generalized, extensible fashion. The Generic Connection framework provides a coherent way to access various types of networks in a resource-constrained environment.

6.3.1 Background and motivation

The class libraries included in Java 2 Standard Edition and Java 2 Enterprise Edition provide a rich set of functionality for handling input and output access to storage and networking systems. For instance, in JDK 1.3.1, the package `java.io.*` contained 59 regular classes and interfaces, and 16 exception classes. The package `java.net.*` of JDK 1.3.1 consisted of 31 regular classes and interfaces, and 8 exception classes. It is difficult to make all this functionality fit in a small device with only a few hundred kilobytes of total memory budget. Furthermore, a significant part of the standard I/O and networking functionality is not directly applicable to today's small devices, which do not often provide TCP/IP support, of which often need to support specific types of connections such as IrDA (infrared) or Bluetooth.

In general, the requirements for the networking and storage libraries vary significantly from one resource-constrained device to another. For instance, those device manufacturers who use packet-switched networks typically want datagram-based communication mechanisms, while those using circuit-switched networks commonly prefer stream-based connections. Some of the devices have traditional file systems, while many others have highly device-specific mechanisms. Due to strict memory limitations, manufacturers supporting certain kinds of input/output, networking and storage capabilities generally do not want to support other mechanisms. All this makes the design of these facilities for CLDC very challenging, especially since J2ME configurations are not allowed to define optional features. Also, the presence of multiple networking mechanisms and protocols is potentially very confusing to the application programmer, especially if the programmer has to deal with low-level protocol issues.

6.3.2 The Generic Connection framework

The challenges presented above led to the generalization of the J2SE network and I/O classes. The goal for this generalized design is to be a functional subset of J2SE classes, which can easily map to common low-level hardware or to any J2SE implementation, but with better extensibility, flexibility and coherence in supporting new devices and protocols.

The general idea is illustrated below. Instead of using a collection of different kinds of abstractions for different forms of communication, a set of related abstractions are used at the application programming level.

General form

`Connector.open("`*`<protocol>`*`:`*`<address>`*`;`*`<parameters>`*`");`

Note – These examples are provided for illustration only. CLDC itself does not define any protocol implementations (see Section 6.3.3 “No network protocol implementations defined in CLDC”). It is not expected that a particular J2ME profile would provide support for all these kinds of connections. J2ME profiles may also support protocols not shown below.

HTTP

```
Connector.open("http://www.sun.com");
```

Sockets

```
Connector.open("socket://129.144.111.222:2800");
```

Communication ports

```
Connector.open("comm:0;baudrate=9600");
```

Datagrams

```
Connector.open("datagram://129.144.111.222:2800");
```

All connections are created by calling the static method `open` of the class `javax.microedition.io.Connector`. If successful, this method will return an object that implements one of the `javax.microedition.io.Connection` interfaces shown in FIGURE 3. The `Connector.open` method takes a `String` parameter in the general form:

`Connector.open("`*`<protocol>`*`:`*`<address>`*`;`*`<parameters>`*`");`

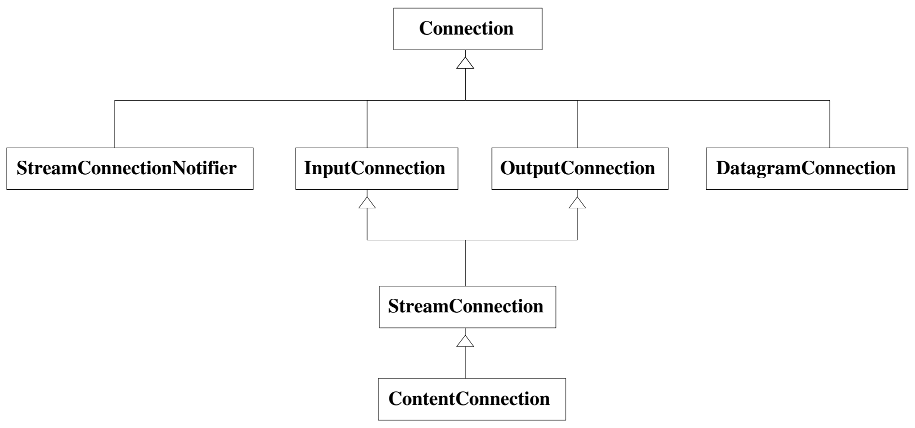

FIGURE 3 `Connection` interface hierarchy

The syntax of the `Connector.open` parameter strings should generally follow the *Uniform Resource Indicator* (URI) syntax as defined in the IETF standard RFC2396 (`http://www.ietf.org/rfc/rfc2396.txt`).

A central objective of this design is to isolate, as much as possible, the differences between the use of one protocol and another into a string characterizing the type of connection. This string is the parameter to the method `Connector.open`. A key benefit of this approach is that the bulk of the application code stays the same regardless of the kind of connection that is used. This is different from traditional implementations, in which the abstractions and data types used in applications often change dramatically when changing from one form of communication to another.

The binding of protocols to a J2ME program is done at runtime. At the implementation level, the string (up to the first occurrence of '`:`') that is provided as the parameter to the method `Connector.open` instructs the system to obtain the desired protocol implementation from a location where all the protocol implementations are stored. It is this *late binding* mechanism which permits a program to dynamically adapt to use different protocols at runtime. Conceptually this is identical to the relationship between application programs and device drivers on a personal computer or workstation.

6.3.3 No network protocol implementations defined in CLDC

The Generic Connection framework defined by the *CLDC Specification* does not specify the actual supported network protocols or mandate implementations of any specific networking protocols. What the *CLDC Specification* provides is an *extensible* *framework* that can be customized by J2ME profiles such as MIDP to support those protocols that specific device categories might need. The actual implementations and decisions regarding supported protocols must be made at the profile level.

6.3.4 Design of the Generic Connection framework

Connections to different types of devices will need to exhibit different forms of behavior. A file, for instance, can be renamed, but no similar operation exists for a TCP/IP socket. The Generic Connection framework reflects these different capabilities, ensuring that operations that are logically the same share the same API.

The Generic Connection framework is implemented using a hierarchy of `Connection` interfaces (located in package `javax.microedition.io`) that group together classes of protocols with the same semantics. This hierarchy consists of seven interfaces shown in Figure 3 above. Additionally, there is the `Connector` class, one exception class (`ConnectionNotFoundException`), one other interface, and a number of data stream classes for reading and writing data. At the implementation level, a minimum of one class is needed for implementing each supported protocol. Often, each protocol implementation class contains simply a number of wrapper functions that call the native functions of the underlying host operating system.

There are six basic interface types that are addressed by the Generic Connection framework:

- A basic serial input connection
- A basic serial output connection
- A datagram oriented connection
- A circuit oriented connection
- A notification mechanism to inform a server of client-server connections
- A basic Web server connection

The collection of `Connection` interfaces forms a hierarchy that becomes progressively more capable as the hierarchy progresses from the root `Connection` interface. This arrangement allows J2ME profile designers or application programmers to choose the optimal level of cross-protocol portability for the libraries and applications they are designing.

The Generic Connection framework is intended to be extensible so that additional interfaces can be added later by J2ME profiles if they require new connection types.

The `Connection` interface hierarchy is illustrated in FIGURE 3. A brief summary of each interface class is provided below. For a definitive reference on the `Connection` interfaces, refer to the documentation that is provided as part of the CLDC reference implementation.

6.3.4.1 Interface `Connection`

This is the most basic connection type that can only be opened and closed. The `open` method is not included in the interface because connections are always opened using the static `open` method of the `Connector` class.

Methods:

```
public void close() throws IOException;
```

6.3.4.2 Interface `InputConnection`

This connection type represents a device from which data can be read. The `openInputStream` method of this interface will return an `InputStream` for the connection. The `openDataInputStream` method of this interface will return a `DataInputStream` for the connection.

Methods:

```
public InputStream openInputStream() throws IOException;
public DataInputStream openDataInputStream() throws IOException;
```

6.3.4.3 Interface `OutputConnection`

This connection type represents a device to which data can be written. The `openOutputStream` method of this interface will return an `OutputStream` for the connection. The `openDataOutputStream` method of this interface will return a `DataOutputStream` for the connection.

Methods:

```
public OutputStream openOutputStream() throws IOException;
public DataOutputStream openDataOutputStream() throws IOException;
```

6.3.4.4 Interface `StreamConnection`

This connection type combines the `InputConnection` and `OutputConnection` interfaces. It forms a logical starting point for classes that implement two-way communication interfaces.

6.3.4.5 Interface `ContentConnection`

This connection type is a sub-interface of `StreamConnection`. It provides access to some of the most basic metadata information provided by HTTP connections.

Methods:

```
public String getType();

public String getEncoding();

public long getLength();
```

6.3.4.6 Interface `StreamConnectionNotifier`

This connection type is used when waiting for a connection to be established. The `acceptAndOpen` method of this interface will block until a client program makes a connection. The method returns a `StreamConnection` on which a communications link has been established. Like all connections, the returned `StreamConnection` must be closed when it is no longer required.

Methods:

```
public StreamConnection acceptAndOpen() throws IOException;
```

6.3.4.7 Interface `DatagramConnection`

This connection type represents a datagram endpoint. In common with the J2SE datagram interface, the address used for opening the connection is the endpoint at which datagrams are received. The destination for datagrams to be sent is placed in the datagram object itself. There is no address object in this API. Instead, a string is used that allows the addressing to be abstracted in a similar way as in the `Connection` interface design.

Methods:

```
public int getMaximumLength() throws IOException;

public int getNominalLength() throws IOException;

public void send(Datagram datagram) throws IOException;

public void receive(Datagram datagram) throws IOException;

public Datagram newDatagram(int size) throws IOException;

public Datagram newDatagram(int size, String addr)
       throws IOException;

public Datagram newDatagram(byte[] buf, int size)
```

```
        throws IOException;
public Datagram newDatagram(byte[] buf, int size, String addr)
        throws IOException;
```

This `DatagramConnection` interface requires a data type called `Datagram` that is used to contain the data buffer and the address associated with it. The `Datagram` interface contains a useful set of access methods with which data can be placed into or extracted from the datagram buffer. These access methods conform to the `DataInput` and `DataOutput` interfaces, meaning that the datagram object also behaves like a stream. A current position is maintained in the datagram. This is automatically reset to the start of the datagram buffer after a read operation is performed.

6.3.5 Additional remarks

In order to read and write data to and from Generic Connections, a number of input and output stream classes are needed. The stream classes supported by CLDC are listed in Section 6.2.4, “Input/output classes.” J2ME profiles may provide additional stream classes as necessary.

APPENDIX **A**

Appendices

APPENDIX 1: CLDC Byte Code Typechecker Specification

APPENDIX 2: CLDC library documentation in *Javadoc* format