Connected, Limited Device Configuration

Specification (First Release)

Java 2 Platform Micro Edition

Please
Recycle

Adobe PostScript™

Contents

Figures

Preface

This document, *Connected, Limited Device Configuration Specification*, defines the *Connected, Limited Device Configuration* (CLDC) of Java™ 2 Micro Edition (J2ME™).

A *configuration* of J2ME specifies the subset of Java programming language features supported, the subset of functionality of the configuration's Java virtual machine, the networking, security, installation and possibly other core platform APIs supported, all to support a certain group of embedded consumer products.

The Connected, Limited Device Configuration is the basis for one or more *profiles*. A *profile* of J2ME 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 related publication, *Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)*, Sun Microsystems, Inc.

Who Should Use This Specification

The audience for this document is the Java Community Process (JCP) expert group (JSR-30) defining this configuration, device manufacturers who want to build small Java-enabled devices, and content developers who want to write Java applications for small, resource-constrained, connected devices.

How This Specification Is Organized

The topics in this specification are organized as follows:

Chapter 1, "Introduction and Background," provides a context for the *CLDC Specification*, and lists the names of the companies that have been involved in the specification work.

Chapter 2, "Goals, Requirements and Scope," defines the goals, special 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," defines the variances from the standard Java programming language required by the CLDC configuration.

Chapter 5, "Adherence to Java Virtual Machine Specification," defines the variances from the standard Java virtual machine required by the CLDC configuration.

Chapter 6, "CLDC Libraries," defines the specific Java APIs required by the CLDC configuration.

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

___

Related Literature

The Java™ Language Specification by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3

Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME), Sun Microsystems, Inc.

Java™ 2 Platform: Micro Edition, *A White Paper*, Sun Microsystems, Inc.

The K Virtual Machine (KVM), *A White Paper*, Sun Microsystems, Inc.

1

Introduction and Background

This document specifies the Connected, Limited Device Configuration (CLDC) of Java™ 2 Platform, Micro Edition (J2ME™). The goal of this work is to define a standard Java platform for small, resource-constrained, connected devices characterized as follows:

- 160 kB to 512 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 (often 9600 bps or less) bandwidth

Cell phones, two-way pagers, personal digital assistants (PDAs), organizers, home appliances, 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, input/output, networking and security are the primary topics addressed by this specification.

This specification is the result of a Java Community Process (JCP) expert group JSR-30 consisting of a number of industrial partners. The following companies (in alphabetical order) have been involved in the definition of CLDC:

- America Online
- Bull
- Ericsson
- Fujitsu
- Matsushita
- Mitsubishi
- Motorola
- Nokia
- NTT DoCoMo
- Oracle

- Palm Computing
- RIM
- Samsung
- Sharp
- Siemens
- Sony
- Sun Microsystems
- Symbian

CLDC is core technology that will be used as the basis for one or more *profiles*. A J2ME *profile* defines a more comprehensive and focused Java platform for a particular vertical market, device category or industry.

For definitive information about J2ME configurations and profiles, refer to *Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)*, Sun Microsystems, Inc. Some information about Java 2 Micro Edition, configurations and profiles is provided in Appendix 1. That appendix also provides information on KVM, a Java virtual machine that can be used to run the Connected, Limited Device Configuration.

2

Goals, Requirements and Scope

2.1 Goals

The high-level goal for the Connected, Limited Device Configuration (CLDC) is to define a standard Java platform for small, resource-constrained, connected devices. The devices targeted by the CLDC have the following general characteristics:

- 160 kB to 512 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 (often 9600 bps or less) bandwidth

Typically, these target devices are manufactured in very large quantities (hundreds of thousands or even millions of units), meaning that manufacturers are usually extremely cost-conscious and interested in keeping the per-unit costs as low as possible.

2.1.1 Dynamic delivery of Java applications and content

Based on the feedback from the CLDC expert group, one of the greatest benefits of Java technology in the small device space is the dynamic, secure delivery of interactive content and applications over different kinds of networks to small client devices. Unlike in the past, when small devices such as cell phones and pagers usually came with a hard-coded feature set, device manufacturers are increasingly looking for solutions that allow them to build *extensible devices* that support rich, dynamic, interactive content from 3rd party content providers and developers. With

the recent introduction of Internet-enabled cell phones, communicators and pagers, this transition is already underway. One of the main goals of this *CLDC Specification* is to take this transition several steps further by allowing the use of the Java programming language as the standard platform for secure delivery of dynamic content for these extensible next-generation devices.

2.1.2　Targeting 3rd party application developers

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 *3rd 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 3rd party developers rather than device manufacturers themselves.

This trend has certain implications for the Java platform features and APIs to be included in this specification. First, the specification shall include only high-level libraries that provide sufficient programming power for the 3rd party application developer. For instance, the networking APIs to be included in CLDC should provide the programmer with meaningful high-level abstractions, such as the ability to transfer whole files, applications or web pages at once, rather than require the programmer to know about the details of specific network transmission protocols. Second, we emphasize the importance of generality and portability. The *CLDC Specification* shall not focus on any specific device category or vertical market.

2.2　Requirements

2.2.1　Hardware requirements

CLDC is intended to run on a wide variety of small devices ranging from wireless communication devices such as cellular telephones and two-way pagers to personal organizers, point-of-sale terminals and even home appliances. 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.

This specification assumes that the virtual machine, the configuration libraries, the profile libraries and the applications must all fit within a total memory budget of 160-512 kilobytes. More specifically, we assume that:

- 128 kilobytes of non-volatile[1] memory is available for the Java virtual machine and CLDC libraries.
- at least 32 kilobytes of volatile[2] memory is available for the Java runtime and object memory.

The ratio of volatile to non-volatile memory in the total memory budget varies 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 have been built in a device, then applications can be prelinked and a very limited amount of volatile memory will be 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 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 system software available in CLDC devices.

Generally, this specification assumes that a minimal *host operating system* (see Section 3.1, "Virtual machine environment") 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 must it 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*. Features

---

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 this 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. Actual memory technology is outside the scope of this specification.

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 this 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.

specific to a certain vertical market, device category or industry should be defined in a *profile* rather than in CLDC. This means that the scope of CLDC is limited and must generally be complemented by profiles.

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

For further information on the rules and guidelines for defining J2ME configurations and profiles, refer to *Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)*, Sun Microsystems, Inc.

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 execution techniques (e.g., JIT 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 expert group, this *CLDC Specification* will address the following areas:

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

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

■ Application life-cycle management (application installation, launching, deletion)
■ 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 is intentionally trying to keep small the number of areas addressed by CLDC. It seems better to restrict the scope of CLDC in order not to exceed the strict memory limitations or to exclude any particular device category. Future versions of this *CLDC Specification* might address additional areas.

The rest of this specification is organized as follows. It starts with a discussion of the high-level architecture of a typical CLDC environment. Then, it compares the Java language and virtual machine features of a Java virtual machine (JVM) supporting CLDC to a conventional Java environment. Finally, it details the Java libraries included in CLDC.

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.
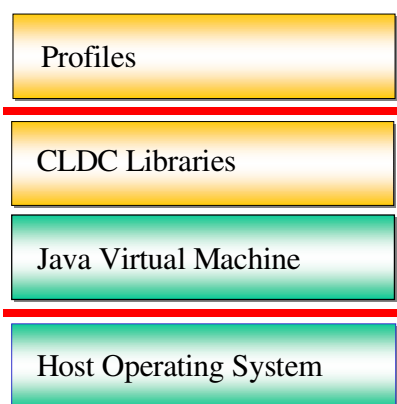
FIGURE 3-1   High-level architecture

3.1   Virtual machine environment

The high-level architecture of a typical CLDC device is illustrated in FIGURE 3-1. At the heart of a CLDC implementation is the *Java Virtual Machine*, 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*. These libraries are divided into two categories:

1. those defined by the Connected, Limited, Device Configuration, and

2. those defined by profiles

3.2 The concept of a Java Application

Throughout this specification, the term *Java application* is used to refer to a collection of Java classfiles containing a single, unique method `main` that identifies the launchpoint of the application. As specified in *Java™ Virtual Machine Specification* (JVMS), §5.2, §2.17.1, the method `main` must be declared `public`, `static` and `void`, as shown below:

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

A JVM supporting CLDC starts the execution of a Java application by calling the method `main`.

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 classes downloaded from an external source are stored persistently on the device. Rather, the virtual machine might just load the classfiles and discard them after loading.

However, in many potential CLDC devices it is beneficial to be able to execute the same Java applications several times without having to download the applications over and over again. This is particularly important if applications are being downloaded over a wireless network, where the user could incur high downloading expenses.

We assume that a device implementing CLDC has capabilities for managing the Java applications that have been stored in the device. At the high level, application management refers to the ability to:

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

Due to significant variations and feature differences among potential CLDC devices, the details of application management are highly device-specific and implementation-dependent. Consequently, application management capabilities are often written in the C programming language or some other low-level programming language specific to the host operating system. The actual details regarding application management are outside the scope of *CLDC Specification*.

3.4 Security

One of the greatest promises of the Java platform is the ability to dynamically deliver interactive content and applications in a secure way to client devices over different kinds of networks.

Unfortunately, the total amount of code devoted to security in Java 2 Standard Edition far exceeds the memory budget available for a Java virtual machine supporting CLDC. Therefore, some compromises are necessary when defining the security model for the CLDC. Our general guideline for defining the security model for CLDC has been to keep things simple, and allow for more comprehensive security solutions in later versions of this *CLDC Specification*.

One of the difficulties in defining the security model for CLDC is that security means different things to different people. For instance, wireless device manufacturers are usually very concerned about network-level security, such as ensuring that nobody can decode or modify the information that is being transmitted over the wireless network. Such solutions are beyond the scope of this *CLDC Specification*.

The focus in this specification is on two areas:

1. low-level virtual machine security, and

2. application-level security.

3.4.1 Low-level virtual machine security

In this specification, low-level virtual machine security means that a Java application executed by the virtual machine must not be able to harm the device in which it is running. In a standard Java virtual machine implementation, this constraint is guaranteed by the Java classfile verifier, which ensures that the Java bytecodes and other items stored in Java classfiles cannot contain references to invalid memory locations or memory areas that are outside the Java object memory (the Java heap).

The role of the classfile verifier is to ensure that classfiles loaded into the virtual machine shall not execute in any way that is not allowed by the *Java™ Virtual Machine Specification*.

As will be explained in more detail in Section 5.3, "Classfile verification," this *CLDC Specification* requires that a JVM supporting CLDC must be able to reject invalid classfiles. This can be guaranteed either by using a standard Java classfile verifier, or by utilizing the alternative verification technology defined in Section 5.2.

3.4.2 Application-level security

Even in a standard Java environment, the security provided by the classfile verifier is limited. The verifier can only guarantee that the given program is a valid Java program, 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 or the network is beyond the scope of the verifier.

To provide controlled access to external resources from Java applications, a J2SE or J2EE environment provides the concept of a *security manager*. A security manager is called whenever the various parts of a Java application or the Java runtime system need access to a protected resource. J2SE provides a very comprehensive security model consisting of formal notions of *permissions*, *access controllers*, and *security policies*.

Unfortunately, the J2SE security model is too memory-consuming to be included in a CLDC device that allows only a few hundred kilobytes of total memory budget. Therefore, a simpler solution is needed.

3.4.2.1 Sandbox model

A JVM supporting CLDC provides a simple "sandbox" security model. By a *sandbox* we mean that a Java application must run in a closed environment in which the application can only access those APIs that have been defined by the configuration, profiles and licensee open classes supported by the device.

More specifically, the sandbox model means that:

- Java classfiles have been properly verified and are guaranteed to be valid Java applications (see Section 5.3, "Classfile verification.")
- Only a limited, predefined set of "safe" Java APIs is available to the application programmer (defined by the CLDC, profiles and licensee open classes.)

- The downloading and management of Java applications on the device takes place at the native code level inside the virtual machine, and no user-definable class loaders are provided, in order to prevent the programmer from overriding the standard class loading mechanisms of the virtual machine.

- 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 the CLDC, profiles or licensee open 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. An obvious security hole in the Java virtual machine would be exposed if the downloaded applications could override the system classes provided in packages `java.*`, `javax.microedition.*`, or other profile- or system-specific packages. A CLDC implementation shall ensure that the programmer cannot override the classes in these protected system packages in any way. At the implementation level this can be guaranteed in different ways, depending on whether or not the implementation supports preloading/prelinking of system classes (see Section 5.4, "Classfile format and class loading.") One solution is to require system classes always to be searched first when performing a classfile lookup. For security reasons, it is also required that the application programmer is not allowed to manipulate the classfile lookup order in any way. Classfile lookup order is discussed in more detail in Section 5.4.3, "Classfile lookup order."

3.4.2.3   Supporting multiple, simultaneously running Java applications

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.

3.4.3 Other security areas

A device supporting the CLDC is typically a part of an end-to-end solution such as a wireless network or a payment terminal network. These networks typically require a number of advanced security solutions to ensure secure delivery of data and code between server machines and client devices. All these end-to-end security solutions are assumed to be implementation-dependent and outside the scope of this *CLDC Specification*.

4

Adherence to the Java Language Specification

The general goal for a JVM supporting CLDC is to be as compliant with the *Java™ Language Specification* as is feasible within the strict memory limits. This chapter summarizes the differences between a JVM supporting CLDC, and the J2SE Java virtual machine. Except for the differences indicated herein, a JVM supporting CLDC shall be compatible with Chapters 1 through 17 of *The Java™ Language Specification* by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1.

4.1   No floating point support

The main language-level difference between the full *Java™ Language Specification* and this *CLDC Specification* is that a JVM supporting CLDC does not have floating point support. Floating point support was removed because the majority of CLDC target devices do not have hardware floating point support, and since the cost of supporting floating point in software was considered too high.

> **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).

This means that a JVM supporting CLDC shall not allow the use of floating point literals (JLS §3.10.2), floating point types and values (JLS §4.2.3) and floating point operations (JLS §4.2.4). For further information, refer to Section 5.1, "No floating point support" in this *CLDC Specification*.

4.2 No finalization

CLDC libraries do not include the method `Object.finalize()`, and therefore a JVM supporting CLDC shall not support finalization of class instances (JLS §12.6). No application built on top of a JVM supporting CLDC shall require that finalization is available.

4.3 Error handling limitations

A JVM supporting CLDC shall generally support *exception* handling as defined in JLS Chapter 11. However, the set of *error* classes included in CLDC libraries is limited, and consequently the error handling capabilities of CLDC are restricted. This is because of two reasons:

1) In embedded systems, recovery from error conditions is usually highly device-specific. While some embedded devices may try to recover from serious error conditions, many embedded devices simply soft-reset themselves upon encountering an error. Application programmer cannot be expected to worry about device-specific error handling mechanisms and conventions.

2) As specified in JLS §11.5.2, class `java.lang.Error` and its subclasses are exceptions from which ordinary 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 unnecessary overhead on the implementation given the strict memory constraints in CLDC target devices.

A JVM supporting CLDC shall support a limited set of error classes defined in Section 6.2, "Classes inherited from J2SE." When encountering any other error, the implementation shall handle the error in a manner that is appropriate for the device.

5

Adherence to Java Virtual Machine Specification

The general goal for a JVM supporting CLDC is to be as compliant with the *Java™ Virtual Machine Specification* as is possible within strict memory constraints. This chapter summarizes the differences between a JVM supporting CLDC, and the J2SE Java virtual machine. Except for the differences indicated herein, a JVM supporting 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* will be referred to as JVMS. Sections within the *Java™ Virtual Machine Specification* will be referred to using the § symbol. For example, (JVMS §2.4.3).

5.1   No floating point support

A JVM supporting CLDC does not have floating point support. Floating point support was removed because the majority of CLDC target devices do not have hardware floating point support, and since the cost of supporting floating point in software was considered too high. Consequently, a JVM supporting CLDC shall not support the following bytecodes:

Constants

fconst_0, fconst_1, fconst_2, dconst_0, dconst_1

Loads

fload, fload_x, dload, dload_x

Stores

```
fstore, fstore_x, dstore, dstore_x
```

Arrays

```
faload, daload, fastore, dastore, newarray T_DOUBLE, newarray T_FLOAT
```

Arithmetic

```
fadd, dadd, fsub, dsub, fmul, dmul, fdiv, ddiv, frem, drem, fneg,
dneg, fcmpl, fcmpg, dcmpl, dcmpg
```

Conversion

```
i2f, f2i, i2d, d2i, l2f, l2d, f2l, d2l, f2d, d2f
```

Returns

```
freturn, dreturn
```

All user-supplied classes and methods running on top of a JVM supporting CLDC must satisfy the following constraints:

- No method shall use any of the above forbidden bytecodes.
- No field can have as its type `float` or `double`, or an array of one of those types.
- No method can have an argument or return type of type `float` or `double` or an array whose component type is `float` or `double`.
- No constant pool entry can be of type `CONSTANT_Float` or `CONSTANT_Double`.
- No constant pool entry can be of type `CONSTANT_Class` in which the class is an array of type `float` or `double`.

Due to the lack of floating point support, the following sections and subsections of the *Java™ Virtual Machine Specification* (JVMS) are not applicable to a JVM supporting CLDC: §2.4.3, §2.4.4, §2.18, §3.3.2 and §3.8. In addition, all the other parts of the JVMS that refer to floating point data types (`float` or `double`) or operations are beyond the scope of this *CLDC Specification*.

5.2 Features eliminated because of library changes or security concerns

A number of features have been eliminated from a JVM supporting CLDC because the Java libraries included in CLDC are substantially more limited than regular J2SE libraries, and/or the presence of the feature would have posed security problems in the absence of the full Java security model. The eliminated features include:

- Java Native Interface (JNI)
- User-defined class loaders
- Reflection
- Thread groups and daemon threads
- Finalization
- Weak references

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.

5.2.1 Java Native Interface (JNI)

A JVM supporting CLDC does not implement the Java Native Interface (JNI). The way in which the virtual machine invokes native functionality is implementation-dependent. Support for JNI was eliminated mainly because of two reasons:

1) the limited security model provided by CLDC assumes that the set of native functions must be closed (see Section 3.4.2.1, "Sandbox model".)

2) the full implementation of JNI was considered too expensive given the strict memory constraints of CLDC target devices.

5.2.2 User-defined class loaders

A JVM supporting CLDC does not support user-defined, Java-level class loaders (see JVMS §5.3, §2.17.2.) A JVM supporting CLDC shall have a built-in class loader that cannot be overridden, replaced, or reconfigured by the user. The actual class loading implementation as well as any error conditions occurring during class loading are implementation-dependent.

5.2.3 Reflection

A JVM supporting CLDC does not have reflection features, i.e., features that allow a Java program to inspect the number and the contents of classes, objects, methods, fields, threads, execution stacks and other runtime structures inside the virtual machine.

A Java application built on top of a JVM supporting CLDC shall not rely on features that require reflection capabilities. Consequently, a JVM supporting CLDC does not support RMI, object serialization, JVMDI (Debugging Interface), JVMPI (Profiler Interface) or any other advanced features of J2SE that depend on the presence of reflective capabilities.

5.2.4 Thread groups and daemon threads

A JVM supporting CLDC implements multithreading, but shall not have support for thread groups or daemon threads. (See JVMS §2.19, §8.12-14). Thread operations such as starting and stopping 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.

5.2.5 Finalization

CLDC libraries do not include the method `Object.finalize()`, and therefore a JVM supporting CLDC does not support finalization of class instances. (See JVMS §2.17.7). No application built on top of a JVM supporting CLDC shall require that finalization is available.

5.2.6 Weak references

A JVM supporting CLDC does not support weak references. No application built on top of a JVM supporting CLDC shall require that weak references are available.

5.2.7 Errors

As discussed earlier in Section 4.3, "Error handling limitations," the error handling capabilities of a JVM supporting CLDC are limited. Apart from supporting the error classes defined in Section 6.2, "Classes inherited from J2SE", the error handling capabilities of a JVM supporting CLDC are assumed to defined in a manner that is appropriate for the target device.

5.3 Classfile verification

Like a standard J2SE Java virtual machine, a JVM supporting CLDC must be able to reject invalid classfiles. However, since the static and dynamic memory footprint of the standard Java classfile verifier is excessive for a typical device supporting CLDC, a more compact and efficient verification solution has been specified. This solution is described below.

5.3.1 Off-device pre-verification + runtime verification with stack maps

The existing J2SE classfile verifier defined in *Java™ Virtual Machine Specification* JVMS §4.9 is not ideal for small, resource-constrained devices. The J2SE verifier takes a minimum of 50 kB binary code space, and at least 30-100 kB of dynamic RAM at runtime. In addition, the CPU power needed to perform the 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 verifier requires Java classfiles to contain a special attribute. The new verifier includes a *pre-verification* tool that inserts this attribute into normal class files. A transformed class file is still a valid J2SE class file, with additional attributes (see Section 5.3.1.2, "Stack map attribute definition" and Section 5.4.2, "Public representation of Java applications and resources") that allow verification to be carried out efficiently at run time. These attributes are automatically ignored by the conventional classfile verifier, so the solution is fully upwards compatible with the J2SE virtual machine. Preprocessed class files containing the extra attributes are approximately 5% bigger than the original, unmodified class files.

Runtime byte code verification guarantees type safety. Classes that pass the verifier cannot, for example, violate Java virtual machine's type system and 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 verifier.

5.3.1.1 Verification process

The new classfile verifier operates in two phases: 1) pre-verification and 2) in-device verification. Pre-verification generally takes place off-device, e.g., on a server machine from which Java applications are being downloaded, or on the development workstation where new applications are being developed. In-device verification is carried out inside the device containing the virtual machine. The in-device verifier utilizes the information generated by the pre-verification tool.

The actual verification process is defined below:

Phase 1: Pre-verification (off-device)

The pre-verification 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 classfile. Each method containing these instructions is replaced with semantically equivalent bytecode not containing the jsr, jsr_w, ret and wide ret bytecodes.

- Add special stack map attributes into class files to facilitate runtime verification. The format and the semantics of the stack map attribute is defined in Section 5.3.1.2, "Stack map attribute definition."

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 will be used to store the derived types as the verifier makes a linear pass through the byte code. 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, 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 less general than the derived types, the derived types are set to be the recorded types. If the derived types are less general than the recorded types, the verifier reports a type error.

  - If the current instruction is in the scope of an exception handler, the derived types are matched against the stack map entry that corresponds to the starting byte code offset of the exception handler. The verifier reports an error if there is not a stack map entry that corresponds to the starting byte code 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 byte code 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.

5.3.1.2 Stack map attribute definition

The pre-verification tool described earlier modifies classfiles 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, we call the attribute a "stack map."

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

The stack map 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, ISBN 0-201-43294-3 for a detailed description of the `Code` attribute and how the stack map attribute fits in as part of the `Code` attribute.

The format of the stack map attribute is as follows:

```
StackMap_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_entries;
    {   u2 byte_code_offset;
        u2 number_of_locals;
```

```
        ty types_of_locals[number_of_locals];
        u2 number_of_stack_items;
        ty types_of_stack_items[number_of_stack_items];
    } entries [number_of_entries];
}
```

The items of the `StackMap_attribute` structure are as follows:

`attribute_name_index`

   The value of the `attribute_name_index` item must be a valid index into the
   `constant_pool` table. The `constant_pool` entry at that index must be a
   `CONSTANT_Utf8_info` structure containing the string "`StackMap`".

`attribute_length`

   The value of the `attribute_length` item indicates the length of the attribute,
   excluding the initial 6 bytes.

`number_of_entries`

   The value of the `number_of_entries` item indicates the number of entries in
   the entries array.

`entries[]`

   Every entry records the types of local variables and the types of stack items at a
   given byte code offset. Not all byte code offsets need to have their local variable
   types and stack item types recorded. Local variable and stack item types are
   recorded for a bytecode offset if the offset marks the beginning of a bytecode
   instruction and one or more of the following conditions hold for the instruction:

   1. The instruction is a target of a conditional jump (e.g., `ifeq`), unconditional
   jump (e.g., `goto`), `tableswitch`, or a `lookupswitch` instruction.

   2. The instruction is marked by the `handler_pc` in the exception_table of the
   `Code` attribute.

   3. The instruction immediately follows an unconditional jump (e.g., `goto`),
   `tableswitch`, `lookupswitch`, `athrow`, or `return` instructions. Unless this
   instruction is dead code, it must also fall into cases 1 or 2.

   Each entry contains the following items:

`byte_code_offset`

   The offset of byte code instruction to which the current entry corresponds. The
   instruction at the offset must satisfy one or more of the above three conditions.

`number_of_locals`

   The number of local variables whose types are recorded.

```
types_of_locals
```

The type of local variables. type_of_locals[n] denotes the type of local
variable n (for 0 <= n < number_of_locals). The types (ty) are either 1-byte
or 3-byte entities. They are encoded as follows:

Name	Code	Explanation
ITEM_Bogus	0	an unknown or uninitialized value
ITEM_Integer	1	a 32-bit integer
ITEM_Float	2	(not used by the CLDC implementation of the verifier)
ITEM_Double	3	(not used by the CLDC implementation of the verifier)
ITEM_Long	4	a 64-bit integer
ITEM_Null	5	the type of null
ITEM_InitObject	6	explained in more detail below
ITEM_Object	7	explained in more detail below
ITEM_NewObject	8	explained in more detail below

Note that the first seven types are encoded in 1 byte, and last two types are
encoded in 3 bytes.

The meanings of the above types ITEM_InitObject, ITEM_Object, and
ITEM_NewObject are as follows:

ITEM_InitObject

Before a constructor (the <init> method) for a class other than
java.lang.Object calls a constructor (the <init> method) of one of its
superclasses, the "this" pointer has type ITEM_InitObject. (Comment: The
verifier uses this type to enforce that a constructor must first invoke a
superclass constructor before performing other operations on the "this"
pointer.)

ITEM_Object

A class instance. The 1-byte type code (7) is followed by a 2-byte
type_name_index (a u2). The type_name_index value must be a valid entry
to the constant_pool table. The constant_pool entry at that index must be a
CONSTANT_Class_info structure.

ITEM_NewObject

An uninitialized class instance. The class instance has just been created by the
new instruction, but a constructor (the <init> method) has not yet been
invoked on it. The type code 8 is followed by a 2-byte
new_instruction_index (a u2). The new_instruction_index must be a
valid offset of a byte code instruction. The opcode of the byte code instruction

must be `new`. (Comment: The uninitialized object is created by this `new` instruction. The verifier uses this type to enforce that an instance cannot be used until it is fully constructed.)

`number_of_stack_items`

The number of items on the stack.

`types_of_stack_items`

The type of items on the stack. `types_of_stack_items[0]` denotes the bottom of the operand stack, and `types_of_stack_items[n]` (for 0 < n < `number_of_stack_items`) denotes the stack item above `types_of_stack_items[n-1]`.

5.4   Classfile format and class loading

An essential requirement for the Connected, Limited Device Configuration is the ability to support dynamic downloading of Java applications and 3rd 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 JVM supporting CLDC.

5.4.1   Supported file formats

It is assumed that a CLDC implementation must be able to read standard Java classfiles (defined in Chapter 4 of JVMS) with the changes defined in Section 5.4.2, "Public representation of Java applications and resources".) In addition, a CLDC implementation must support compressed Java Archive (JAR) files. This requirement has been added to maintain upward compatibility with larger Java environments and existing Java tools, but with a smaller footprint than with regular classfiles. Detailed information about JAR format is provided at http://java.sun.com/products/jdk/1.2/docs/guide/jar.

In general, network bandwidth conservation is very important in today's low-bandwidth wireless networks. The compressed JAR format provides 30-50% compression over regular Java classfiles without loss of any symbolic information or compatibility problems with existing Java systems.

5.4.2 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 single, closed network system where the vendor 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 regular Java classfiles with the following restrictions and additional requirements:

- stack map attributes (Section 5.3.1.2, "Stack map attribute definition") must be included in classfiles.
- the classfile must not contain any of the following Java bytecodes: `jsr` (JVMS p. 304, `jsr_w` (JVMS p. 305), `ret` (JVMS p. 352) and `wide ret` (JVMS p. 360).

Sun's reference implementation includes a preverification tool for performing the above modifications to a Java classfile. Note that the stack map attributes are automatically ignored by the conventional classfile verifier described in JVMS §4.9, i.e., the format specified here is fully upwards compatible with larger Java environments such as J2SE or J2EE.

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 library documentation for details.)

5.4.3 Classfile lookup order

The *Java™ Language Specification* and *Java™ Virtual Machine Specification* do not specify the order in which classfiles are searched when new classfiles 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 classfile 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 JVM supporting CLDC is not required to support the notion of `classpath`, but may do so at the implementation level.

Two restrictions apply to classfile lookup order. First, as explained in Section 3.4.2.2, "Protecting system classes," a JVM supporting CLDC must guarantee that the application programmer cannot override the system classes (classes belonging to the CLDC or supported profiles) in any way. Second, it is required that the application programmer must not be able to manipulate the classfile lookup order in any way. Both these restrictions are important for security reasons.

5.4.4 Implementation optimizations

This *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 previous subsection) and internally inside the virtual machine at runtime, alternative formats can be used. For instance, in low-bandwidth wireless networks at the network transport level it is often a good idea to use alternative, more compact transport formats 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 definition of more compact classfile representations is assumed to be implementation-dependent and outside the scope of this *CLDC Specification*.

5.4.5 Preloading/prelinking ("ROMizing")

A JVM supporting CLDC may choose to preload/prelink some classes. This technology is referred to informally as *ROMizing.* Typically, small virtual machine implementations choose to preload all the system classes (classes belonging to a specific configuration or profile), and perform application downloading dynamically. The actual mechanisms for preloading are implementation-dependent and beyond the scope of this *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.

5.4.6 Future classfile formats

Regular Java classfiles are not optimized for network transport in bandwidth-limited environments. This is because each Java classfile is an independent unit that contains its own constant pool (symbol table), method, field and exception tables, bytecodes, and some other information. The self-contained nature of classfiles 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 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 is 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 classfiles are not designed for such execution.

Later releases of CLDC might utilize a variation of "split VM" prelinked classfiles such as "cap" files as specified in the *Java Card™ 2.1 Virtual Machine Specification, Revision 1.1*, Sun Microsystems, Inc.

6

CLDC Libraries

6.1 Overview

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

6.1.1 General goals

A general goal for designing the Java libraries for CLDC is to provide a minimum useful set of libraries for practical application development and profile definition for a variety of small 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 please 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 too high for many others.

In defining the scope of the libraries, we have used the original Java Specification Request JSR-30 as a guideline, and placed a lot of emphasis on *connectivity*. This means that in addition to fundamental system and data type classes, the libraries included in CLDC should provide an extensible set of networking features for both today's and tomorrow's small, connected devices.

6.1.2 Compatibility

The majority of the libraries included in CLDC are a subset of the larger Java editions (J2SE and J2EE) to ensure upward compatibility and portability of applications. While upward compatibility is a very desirable goal, J2SE and J2EE libraries have strong internal dependencies that make subsetting of 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 very 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 areas of networking and I/O.

The CLDC libraries presented in this *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.*`. These classes have been derived from Java 2 Standard Edition version 1.3. A detailed list of these classes is presented in Section 6.2, "Classes inherited from J2SE."

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

6.2 Classes inherited from J2SE

CLDC provides a number of classes that have been inherited from J2SE. 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 official information on these rules, refer to *Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)*, Sun Microsystems, Inc.

For a definitive reference on the classes listed in this section, refer to Appendix 2.

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 J2SE Java compiler (`javac`) generates code that requires that certain functions of classes `String` and `StringBuffer` be available. The following system classes are included.

```
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.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. Classes `Reader`, `Writer`, `InputStreamReader` and `InputStreamWriter` are required in order to support internationalization (see Section 6.2.8, "Internationalization.").

```
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`. By default, only one time zone is supported. Additional time zones may be provided by actual implementations.

```
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` (for data types `int` and `long`) that are frequently used by other Java library classes.

```
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 4.3, "Error handling limitations," the error handling capabilities of CLDC are limited. By default, a JVM supporting 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.ClassNotFoundException
java.lang.IllegalAccessException
java.lang.InstantiationException
java.lang.InterruptedException
java.lang.RuntimeException
java.lang.ArithmeticException
java.lang.ArrayStoreException
java.lang.ClassCastException
java.lang.IllegalArgumentException
java.lang.IllegalThreadStateException
java.lang.NumberFormatException
java.lang.IllegalMonitorStateException
```

```
java.lang.IndexOutOfBoundsException

java.lang.ArrayIndexOutOfBoundsException

java.lang.StringIndexOutOfBoundsException

java.lang.NegativeArraySizeException

java.lang.NullPointerException

java.lang.SecurityException


java.util.EmptyStackException

java.util.NoSuchElementException


java.io.EOFException

java.io.IOException

java.io.InterruptedIOException

java.io.UnsupportedEncodingException

java.io.UTFDataFormatException
```

6.2.7.2   Error classes

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

6.2.8   Internationalization

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 used here using the `InputStreamReader` and `OutputStreamWriter` classes with identical constructors.

```
new InputStreamReader(InputStream is);

new InputStreamReader(InputStream is, String name);

new OutputStreamWriter(OutputStream os);

new OutputStreamWriter(OutputStream os, String name);
```

In the cases where the string 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

`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.

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 assumed to be outside the scope of CLDC.

6.2.9   Property support

A JVM supporting CLDC does not implement class `java.util.Properties`, which is part of J2SE. However, a limited set of properties described in the table below is available. These properties can be accessed by calling the method `System.getProperty(String key)`.

Property `microedition.encoding` describes the default character encoding name.

TABLE 6-1   Standard System Properties

Key	Explanation	Value
`microedition.platform`	Name of the host platform or device	Default `null`
`microedition.encoding`	Default character encoding	Default "ISO8859_1"
`microedition. configuration`	Name and version of the supported configuration	Default "CLDC-1.0"
`microedition.profiles`	Names of the supported profiles	Default `null`

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.

Profiles may define additional properties not included in TABLE 6-1 above.

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 and organize data in a resource-constrained environment.

6.3.1 Background and motivation

The J2SE and J2EE libraries provide a rich set of functionality for handling input and output access to storage and networking systems. The package `java.io.*` of J2SE contains approximately 60 classes and interfaces, and more than 15 exception classes. The package `java.net.*` of J2SE consists of approximately 20 regular classes and 10 exception classes. The total static size of these class files is approximately 200 kilobytes. 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 often need to support specific types of connections such as infrared or Bluetooth, or which do not provide TCP/IP support.

In general, the requirements for the networking and storage libraries vary significantly from one resource-constrained device to another. Those device manufacturers who are dealing with packet-switched networks typically want datagram-based communication mechanisms, while those dealing with circuit-switched networks require 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 requirement for having small footprint J2ME systems has led to the generalization of the J2SE network and I/O classes. The general goal for this new system is to be a precise 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 totally different kinds of abstractions for different forms of communication, a set of related abstractions are used at the application programming level.

Note – These examples are provided for illustration only. It is not expected that a particular implementation would provide support for all these kinds of connections. Actual implementations may also support protocols not shown below.

General form

All connections are created using a single static method in a system class called `Connector`. If successful, this method will return an object that implements one of the generic connection interfaces. There are number of these interfaces that form a hierarchy with the `Connection` interface being the root. The method takes a URL parameter in the general form:

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

HTTP records

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

FTP records

```
Connector.open("ftp://www.foo.com/foo.dat");
```

Sockets

```
Connector.open("socket://129.144.111.222:9000");
```

Communication ports

```
Connector.open("comm:9600:18N");
```

Datagrams

```
Connector.open("datagram://129.144.111.333");
```

Files

```
Connector.open("file:/foo.dat");
```

Network file systems

```
Connector.open("nfs:/foo.com/foo.dat");
```

A central objective of this design is to isolate, as much as possible, the differences between the setup of one protocol and another into a string characterizing the type of connection. This string is the parameter to `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 used inside 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 `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 PC or workstation computer.

6.3.3 No implementations provided at the configuration level

The Generic Connection framework included in CLDC does not specify the actual supported protocols or include implementations of specific protocols. 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 (data sent through time is managed quite differently than data sent through space). The Generic Connection framework reflects these different capabilities, ensuring that operations that are logically the same share the same API.

The new framework is implemented using a hierarchy of `Connection` interfaces that group together classes of protocols with the same semantics. This hierarchy consists of seven interfaces. Additionally, there is the `Connector` class, one exception class, one other interface, and a number of data stream classes for reading and writing data (see Appendix 2 for details.) 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 device.
- A basic serial output device.
- A datagram oriented communications device.
- A circuit oriented communications device (TCP, etc.).
- A notification mechanism for a server to be informed 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 application programmers to choose the optimal level of cross-protocol portability for the code they are writing.

The `Connection` interface hierarchy is illustrated in FIGURE 6-1. For a definitive reference on the `Connection` interfaces, refer to Appendix 2.
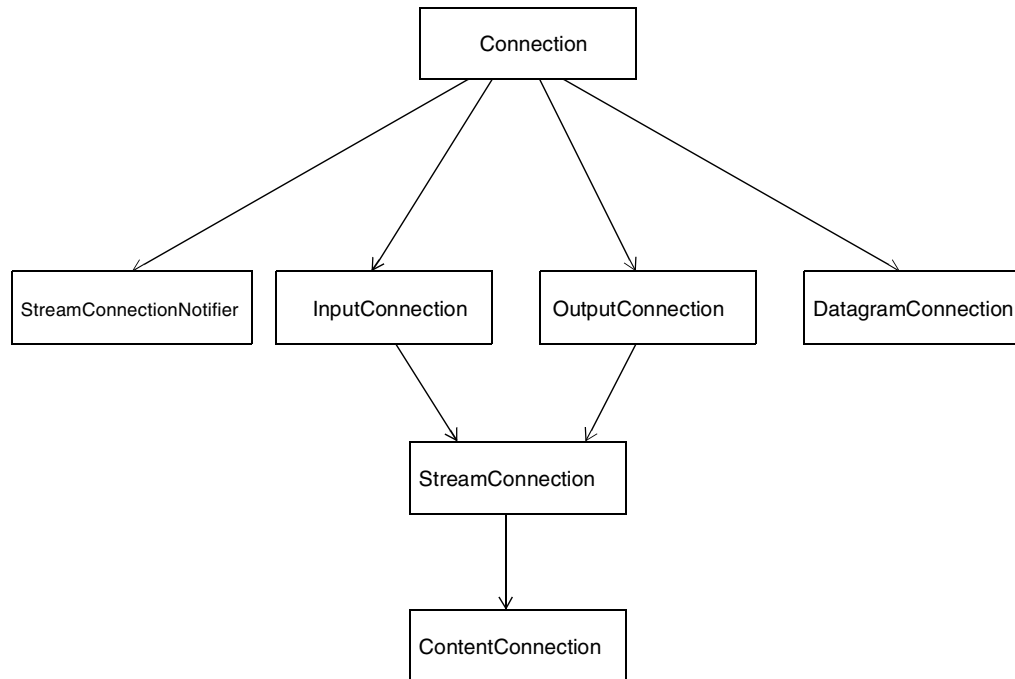
FIGURE 6-1 `Connection` interface hierarchy

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 public because it is always called via the static `open()` method in 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 input stream 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 output stream for the connection.

Methods:

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

6.3.4.4 Interface `StreamConnection`

This is simply an interface that combines the `InputConnection` and `OutputConnection` interfaces. It forms a logical starting point for classes that implement communications interfaces.

6.3.4.5 Interface `ContentConnection`

This is a sub-interface of `StreamConnection`, and provides access to some of the most basic meta data 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 to wait for a connection to be established. The `acceptAndOpen` method of this class will block until a client program makes a connection. It returns a `StreamConnection` on which a communications link has been established. Like all connections, the returned stream connection must be closed when it is no longer required.

Methods:

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

6.3.4.7 Interface `DatagramConnection`

This interface represents a datagram endpoint. In common with the J2SE datagram interface, the address used for opening the connection is the endpoint at which datagrams will be 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 String getAddress();
public int getMaximumLength();
public int getNominalLength();
public void setTimeout(int time);
public void send(Datagram datagram);
public void receive(Datagram datagram);
public Datagram newDatagram(int size);
public Datagram newDatagram(byte[] buf, int size);
public Datagram newDatagram(byte[] buf, int size, String addr);
```

This class requires a data type called `Datagram` that is used to contain the data buffer and 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." Profiles may provide additional stream classes as necessary.

6.3.6 Examples

Examples illustrating the use of the Generic Connection framework are available separately.

Appendices

APPENDIX 1: Introduction to Java 2 Micro Edition and the KVM.

APPENDIX 2: Detailed description of CLDC libraries in javadoc format.