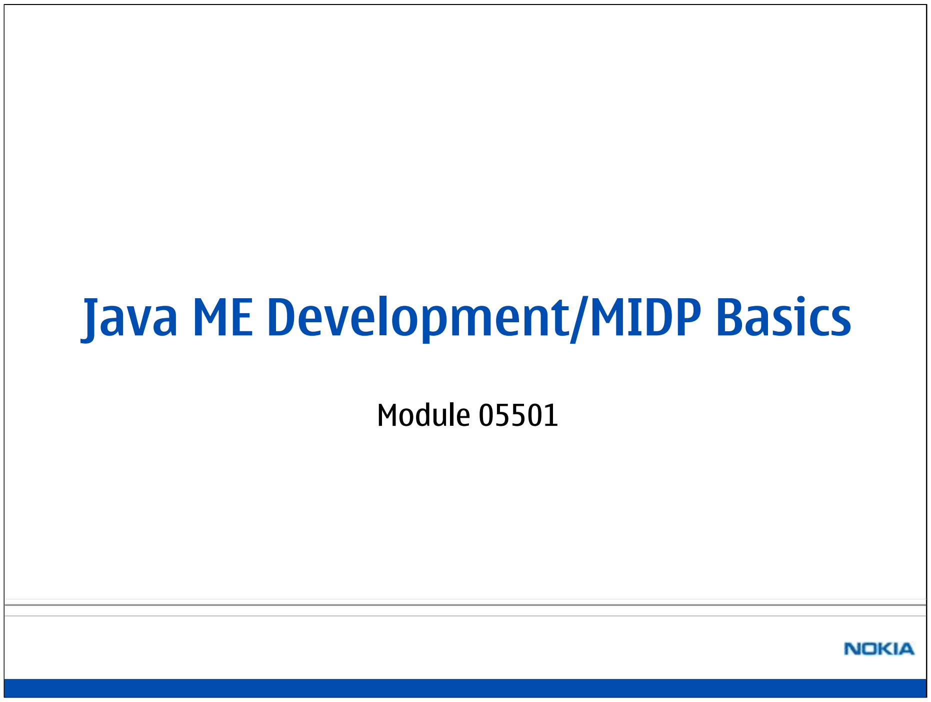


Java ME Development/MIDP Basics

Module 05501

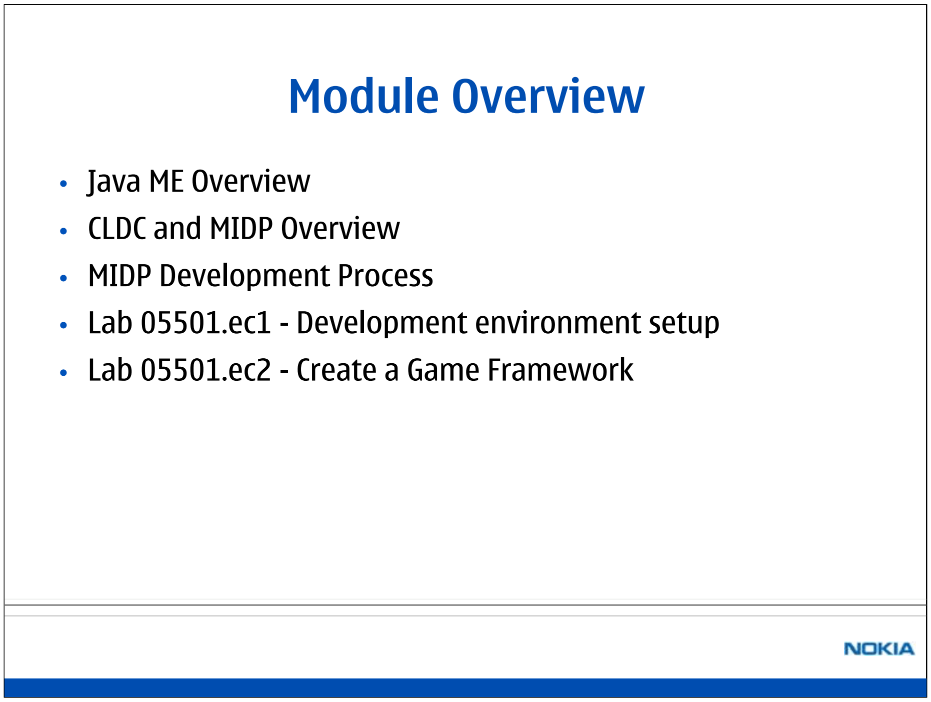
Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

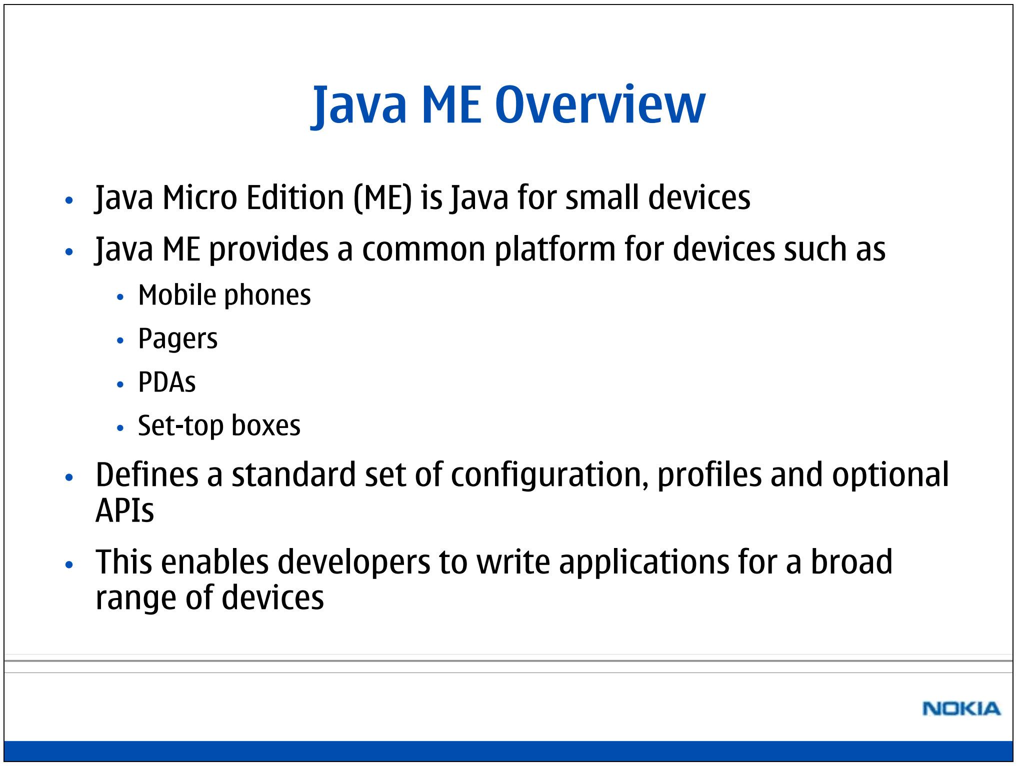
Nokia Corporation retains the right to make changes to this document at any time, without notice.

License



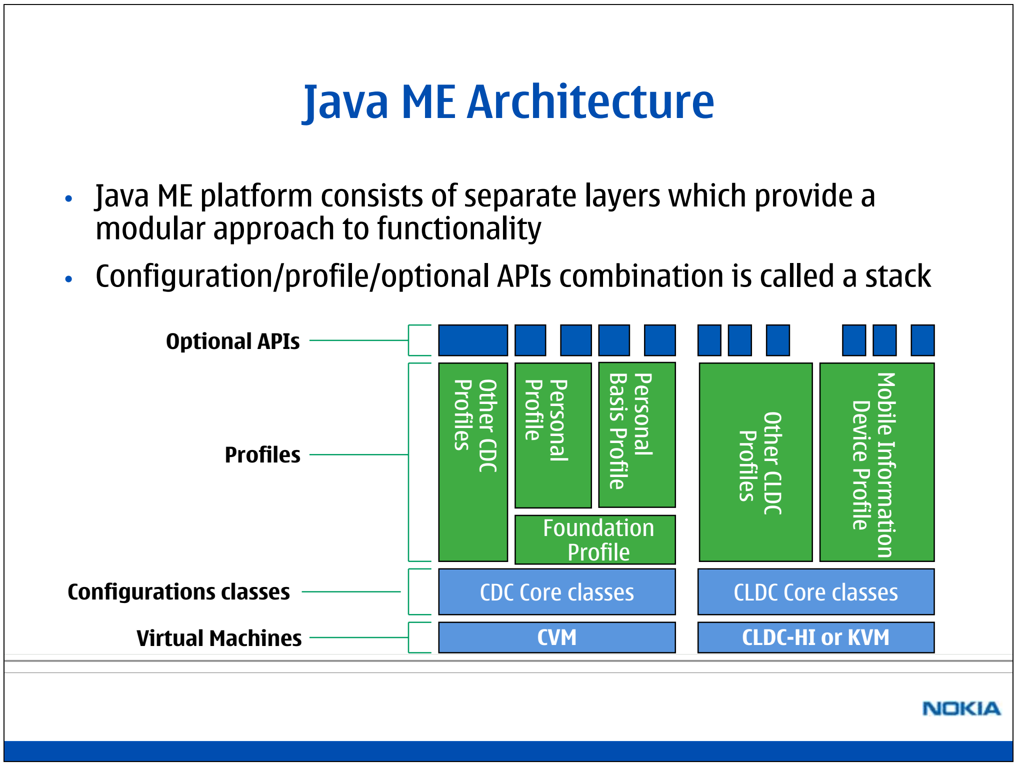
Module Overview

- Java ME Overview
- CLDC and MIDP Overview
- MIDP Development Process
- Lab 05501.ec1 - Development environment setup
- Lab 05501.ec2 - Create a Game Framework



Java ME Overview

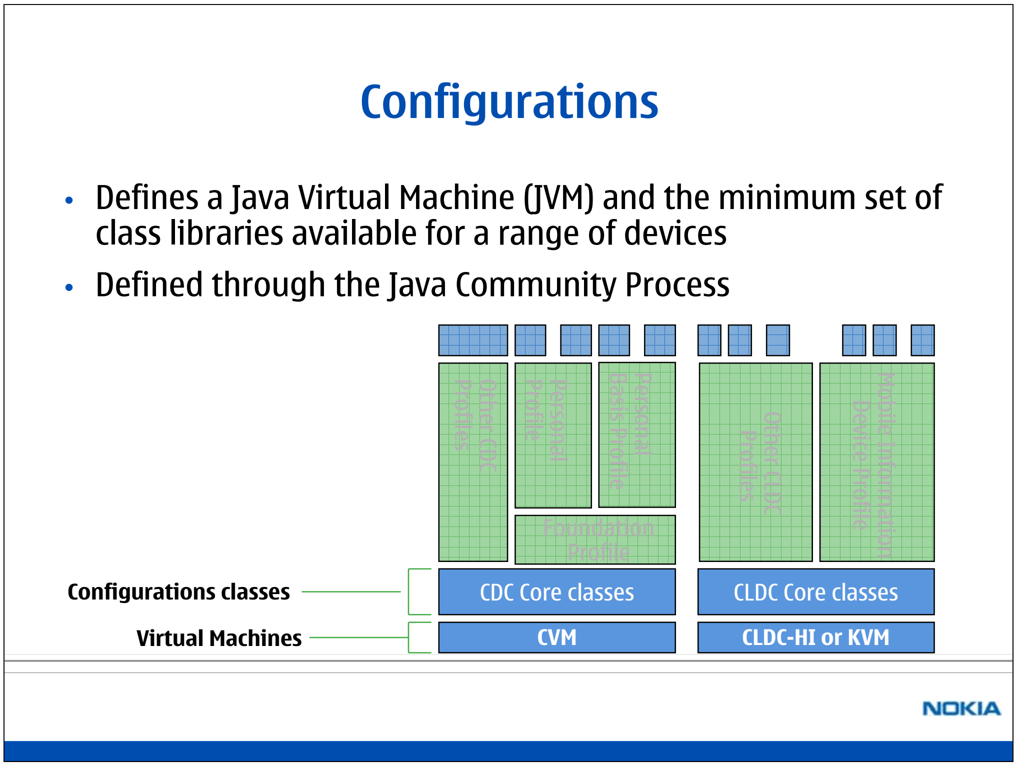
- Java Micro Edition (ME) is Java for small devices
- Java ME provides a common platform for devices such as
  - Mobile phones
  - Pagers
  - PDAs
  - Set-top boxes
- Defines a standard set of configuration, profiles and optional APIs
- This enables developers to write applications for a broad range of devices



Java ME Architecture

- Java ME platform consists of separate layers which provide a modular approach to functionality
- Configuration/profile/optional APIs combination is called a stack

The Java ME architecture defines configurations, profiles and optional packages as elements for building complete Java runtime environments that meet the requirements for a broad range of devices and target markets. Each combination is optimized for the memory, processing power, and I/O capabilities of a related category of devices.



Configurations

- Defines a Java Virtual Machine (JVM) and the minimum set of class libraries available for a range of devices
- Defined through the Java Community Process

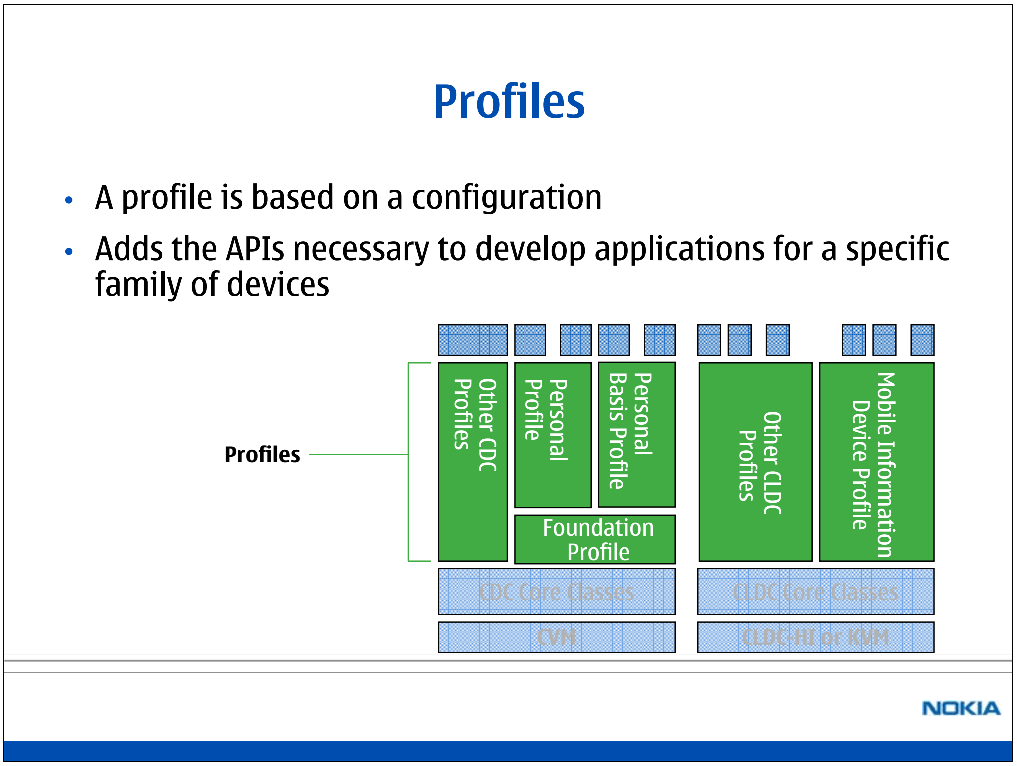
Configurations define the minimum Java libraries and virtual machine capabilities expected to be available on devices that share similar characteristics such as memory size and processing resources. They provide the base functionality that a device manufacturer or a content provider can safely assume to be present on a range of devices. Configurations are defined through the Java Community Process, which is Sun's attempt to involve the international Java community in developing Java specifications.

More specifically, a configuration specifies:

- *The Java programming language features supported,*
- *The Java virtual machine features supported,*
- *The Java libraries and APIs supported.*

Application developers and content providers must design their code to stay within the bounds of the Java virtual machine features and APIs specified by that configuration.

Java ME currently defines two configurations, the Connected Device Configuration (CDC), and the more restrained Connected Limited Device Configuration (CDLC).



Profiles

- A profile is based on a configuration
- Adds the APIs necessary to develop applications for a specific family of devices

A profile is a collection of Java based APIs that supplement a Configuration to provide capabilities for a specific vertical market or device type (for example, wireless: pagers, mobile devices, etc.).

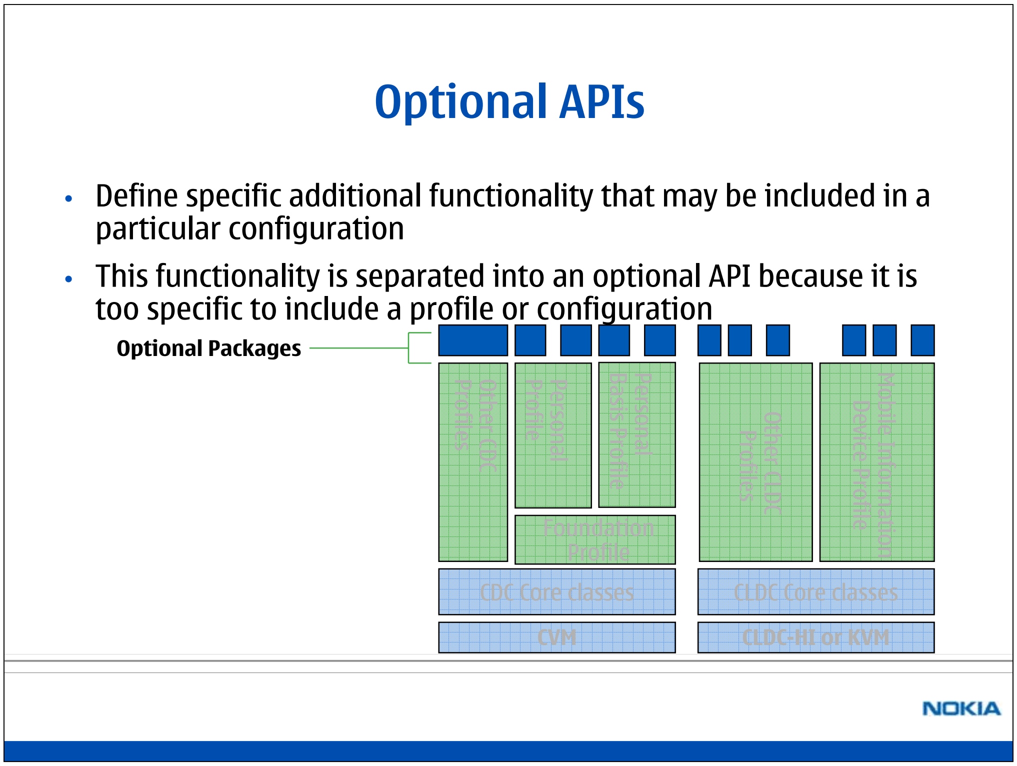
The main goal for a profile is to provide flexibility to the Java Community while still maintaining portability across device types.

Profiles are defined by open industry working groups utilizing the Java Community Process Program. In this way industries can decide for themselves what elements are necessary to provide a complete solution targeted at their industry.

Profiles are implemented on top of a configuration to further define the application life cycle model, the user interface, and access to device specific properties.

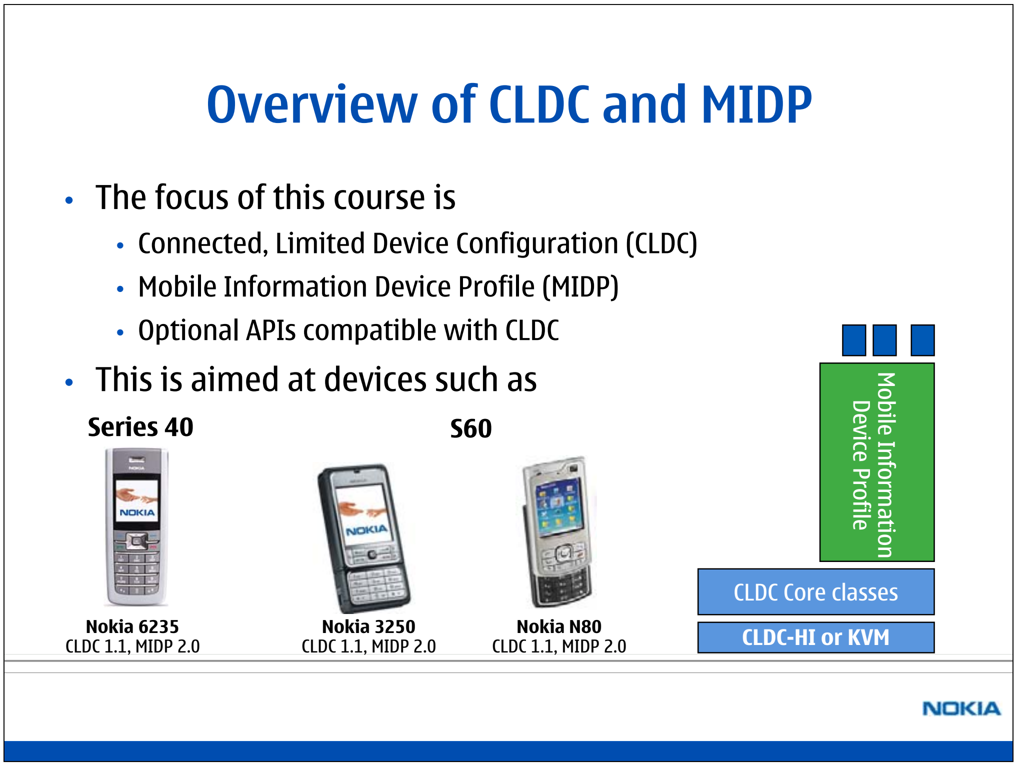
Examples of profiles are the Mobile Information Device Profile (MIDP), Foundation Profile (FP), Personal Profile (PP) and Personal Basic Profile (PBP).

The practical definition for developers is that a configuration specifies everything that is necessary to provide basic Java Technology support in a compliant device, while profiles provide industry-specific or specialized APIs on top of the underlying configuration. The CLDC and MIDP described below together specify both the core VM and Java language features plus the wireless- and mobile-specific APIs needed for mobile devices and pagers with small memory, power constraints, small screens, and limited user interfaces.



Optional APIs

- Define specific additional functionality that may be included in a particular configuration
- This functionality is separated into an optional API because it is too specific to include a profile or configuration



Overview of CLDC and MIDP

- The focus of this course is
  - Connected, Limited Device Configuration (CLDC)
  - Mobile Information Device Profile (MIDP)
  - Optional APIs compatible with CLDC
- This is aimed at devices such as

Series 40

Nokia 6235
CLDC 1.1, MIDP 2.0

S60

Nokia 3250
CLDC 1.1, MIDP 2.0

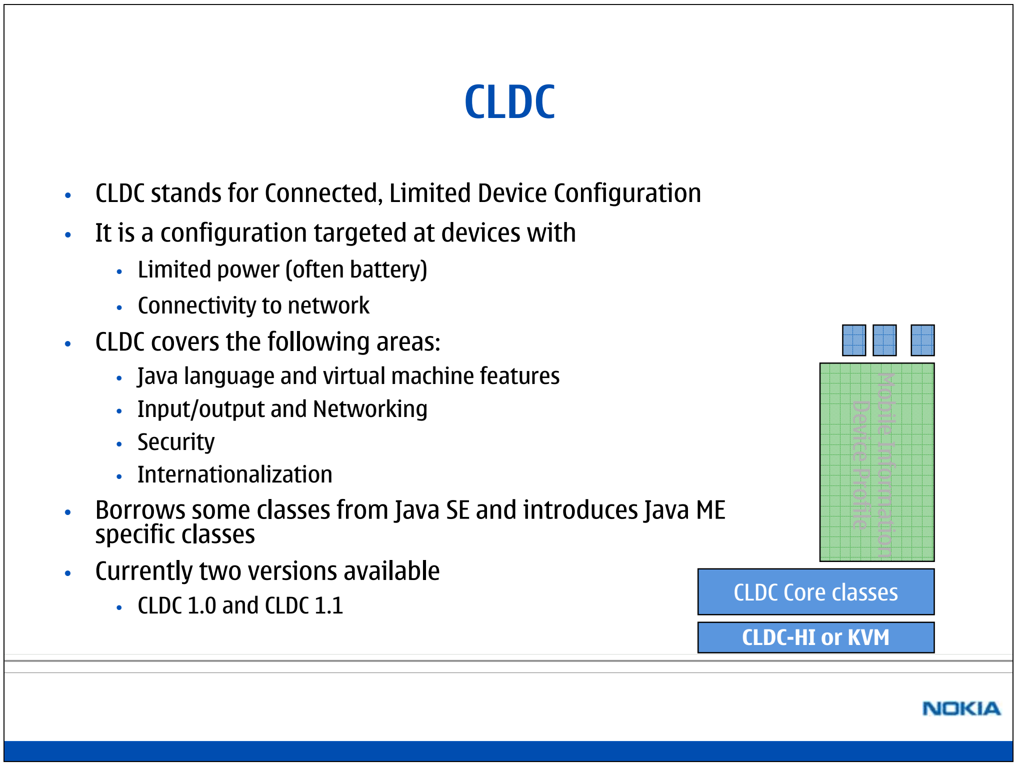
Nokia N80
CLDC 1.1, MIDP 2.0

Mobile Information Device Profile

CLDC Core classes

CLDC-HI or KVM

NOKIA



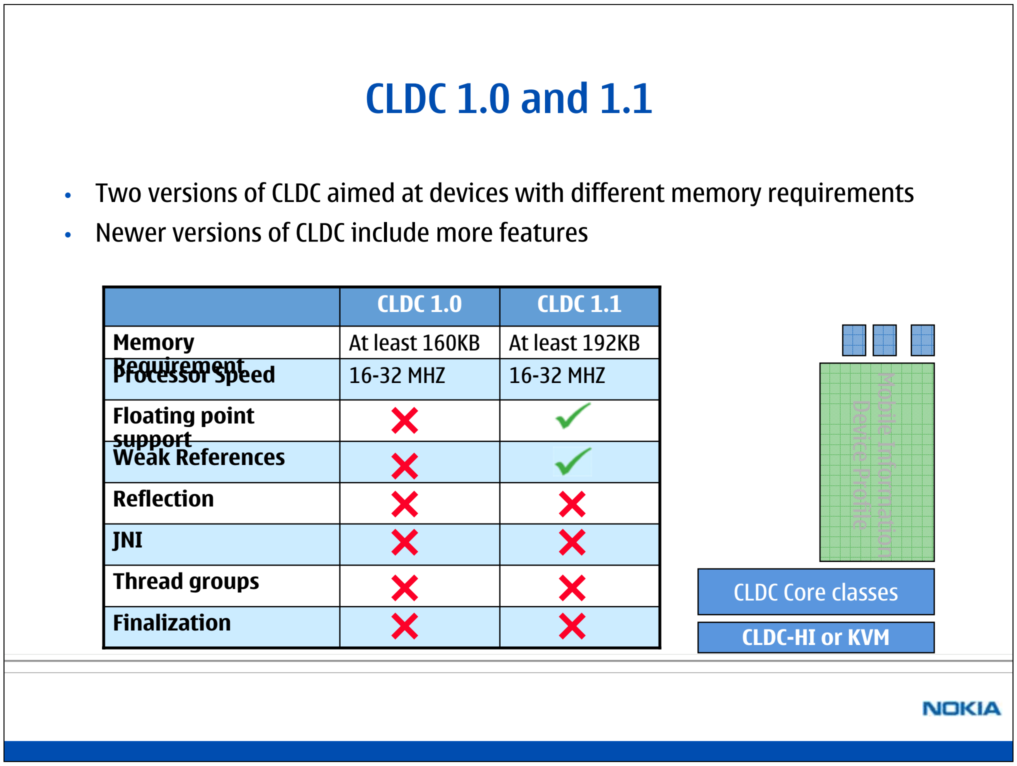
CLDC

- CLDC stands for Connected, Limited Device Configuration
- It is a configuration targeted at devices with
  - Limited power (often battery)
  - Connectivity to network
- CLDC covers the following areas:
  - Java language and virtual machine features
  - Input/output and Networking
  - Security
  - Internationalization
- Borrows some classes from Java SE and introduces Java ME specific classes
- Currently two versions available
  - CLDC 1.0 and CLDC 1.1

Connected Limited Device Configuration (CLDC) specifies the core libraries and virtual machine features for Java ME implementation on small, resource-constrained devices, such as mobile devices, mainstream personal digital assistants, and small retail payment terminals.

CLDC enables dynamic, secure delivery of interactive Java based content to small devices. It also incorporates 3rd-party application development for these devices.

This configuration includes some new classes, not drawn from the Java Standard Edition (SE) APIs, designed specifically to fit the needs of small-footprint devices.



CLDC 1.0 and 1.1

- Two versions of CLDC aimed at devices with different memory requirements
- Newer versions of CLDC include more features

	CLDC 1.0	CLDC 1.1
Memory Requirement	At least 160KB	At least 192KB
Processor Speed	16-32 MHZ	16-32 MHZ
Floating point support	✗	✓
Weak References	✗	✓
Reflection	✗	✗
JNI	✗	✗
Thread groups	✗	✗
Finalization	✗	✗

Mobile Information Device Profile

CLDC Core classes

CLDC-HI or KVM

NOKIA

The general goal for a Java VM supporting CLDC is to be as compliant with the Java. Language Specification as is feasible within the strict memory limits of the target devices. A number of features have been eliminated from a JVM supporting CLDC because of lack of hardware support, 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 limitations include:

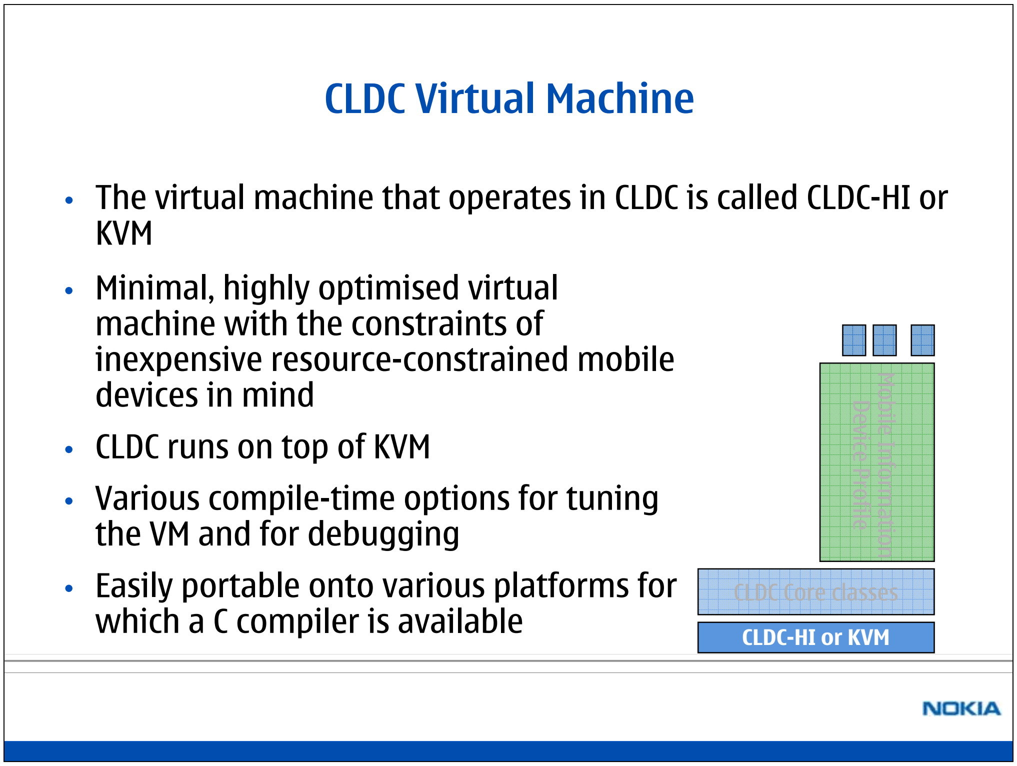
- ***No support for floating point data types (float and double). This was removed because the majority of CLDC target devices do not have hardware floating point support, and since the cost of supporting floating point in software was considered too high.***
- ***No support for finalization of class instances. The method Object.finalize() does not exist.***
- ***No support for the Java Native Interface (JNI). The limited security model provided by CLDC assumes that the set of native functions must be closed. Also the full implementation of JNI was considered too expensive given the strict memory constraints of CLDC target devices.***
- ***No user-defined, Java-level class loaders. The elimination of user-defined class loaders is part of the security restrictions.***
- ***No 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. Consequently, 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.***
- ***No supports for thread groups or daemon threads. Implements multithreading, but does not have support for thread groups or daemon threads. Thread operations such as starting and stopping of threads can be applied only to individual thread objects.***
- ***No weak references, i.e. a reference that does not prevent the referenced object from being garbage collected.***

Error Handling Limitations

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 programmers cannot be expected to worry about device-specific error handling mechanisms and conventions.***

2) ***The 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***



CLDC Virtual Machine

- The virtual machine that operates in CLDC is called CLDC-HI or KVM
- Minimal, highly optimised virtual machine with the constraints of inexpensive resource-constrained mobile devices in mind
- CLDC runs on top of KVM
- Various compile-time options for tuning the VM and for debugging
- Easily portable onto various platforms for which a C compiler is available

The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time.

Sun offers two virtual machines to support the CLDC: The K virtual machine (KVM) and CLDC-HI (HotSpot) virtual machine.

KVM is the minimal VM for the Java ME platform and is the virtual machine of choice for the CLDC configuration. It is a highly optimized virtual machine designed from the ground up with the limitations of inexpensive resource-constrained mobile devices in mind. It is named to reflect that its size is measured in the tens of kilobytes.

More specifically, KVM was designed to be:

- *Small, with a static memory footprint of the core of the virtual machine in the range 40 kilobytes to 80 kilobytes (depending on compilation options and the target platform,)*
- *Clean and highly portable,*
- *Modular and customisable,*
- *As "complete" and "fast" as possible without sacrificing the other design goals.*

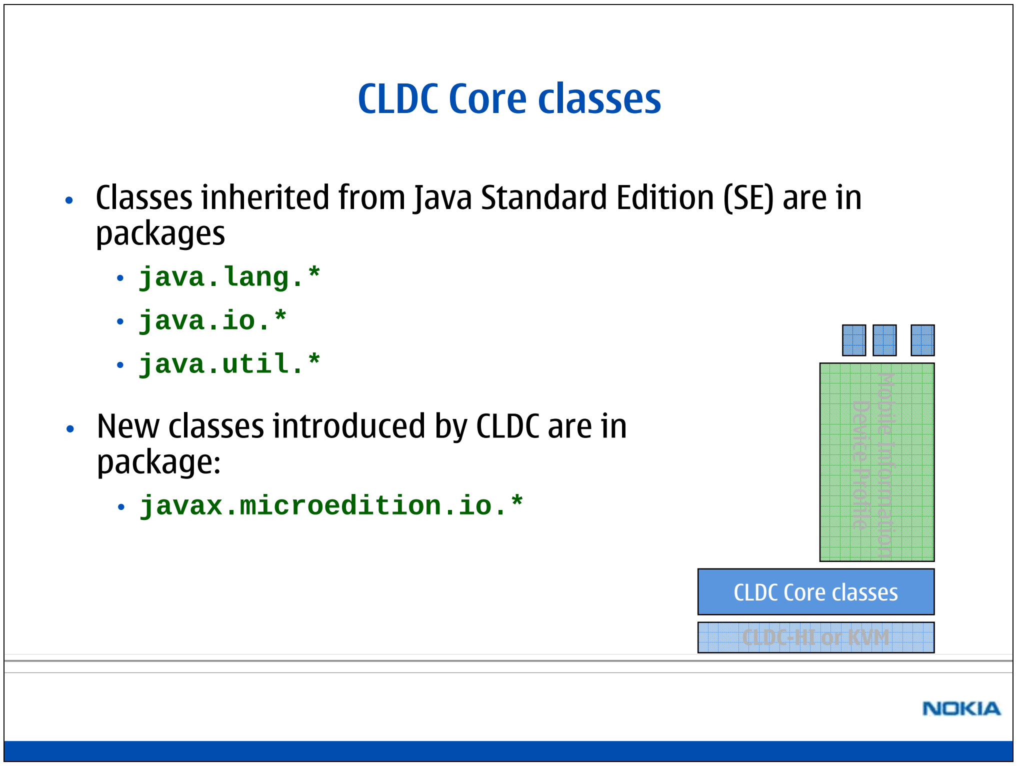
The K virtual machine is extremely small -- starting from approximately 50K in size -- and efficient. CLDC with KVM is suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers, and with as little as 160 KB of total memory available for the Java technology stack. 128 KB of this is for the storage of the actual virtual machine and libraries, and the remainder is for Java applications.

The KVM is implemented in the C programming language, so it can easily be ported onto various platforms for which a C compiler is available. Since the KVM was written from scratch in C, it permitted the engineers to incorporate optimizations that would not have been possible if it was a modification of an existing VM. The virtual machine has been built around a straightforward bytecode interpreter with various compile-time flags and options to aid porting efforts and improve space optimization. This was to address the fact that KVM-based devices would have cramped memory spaces and limited processor power.

Sun provides reference implementations of KVM for Win32 and Solaris. Additional ports for Linux and PalmOS are also provided. Third party vendors have released Java ME CLDC implementations for various hardware and OS platforms.

Like all Java virtual machines, the KVM provides the foundation for the downloading and execution of dynamic content and services. However, it does differ from other virtual machines in that, for example, the virtual machine is highly customizable. It is modular and is built so device manufacturers can easily remove features not needed for a particular target implementation. Such optional features include: large datatypes (long, float, and double), multidimensional arrays, classfile verification, and others. It has various compile- time options for tuning the VM (size vs. speed) and for debugging.

The actual role of KVM in the target devices can vary significantly. In some implementations, KVM is used on top of an existing software stack to give the device the ability to download and run dynamic, interactive, secure Java

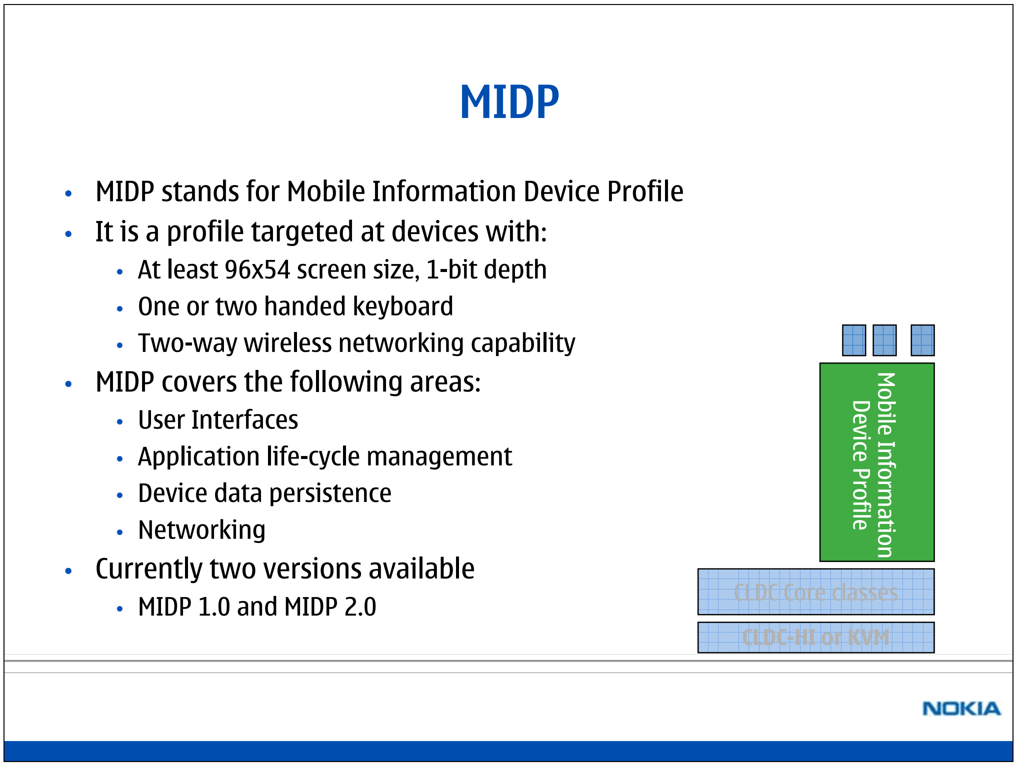


CLDC Core classes

- Classes inherited from Java Standard Edition (SE) are in packages
  - `java.lang.*`
  - `java.io.*`
  - `java.util.*`
- New classes introduced by CLDC are in package:
  - `javax.microedition.io.*`

Mobile Information Device Profile
CLDC Core classes
CLDC-HI or KVM
NOKIA

In order to ensure upward compatibility and portability of applications, the majority of the class libraries included in CLDC are a subset of those specified for the larger Java editions (SE and EE). Only those classes that are appropriate for mobile devices are specified by CLDC.



MIDP

- MIDP stands for Mobile Information Device Profile
- It is a profile targeted at devices with:
  - At least 96x54 screen size, 1-bit depth
  - One or two handed keyboard
  - Two-way wireless networking capability
- MIDP covers the following areas:
  - User Interfaces
  - Application life-cycle management
  - Device data persistence
  - Networking
- Currently two versions available
  - MIDP 1.0 and MIDP 2.0

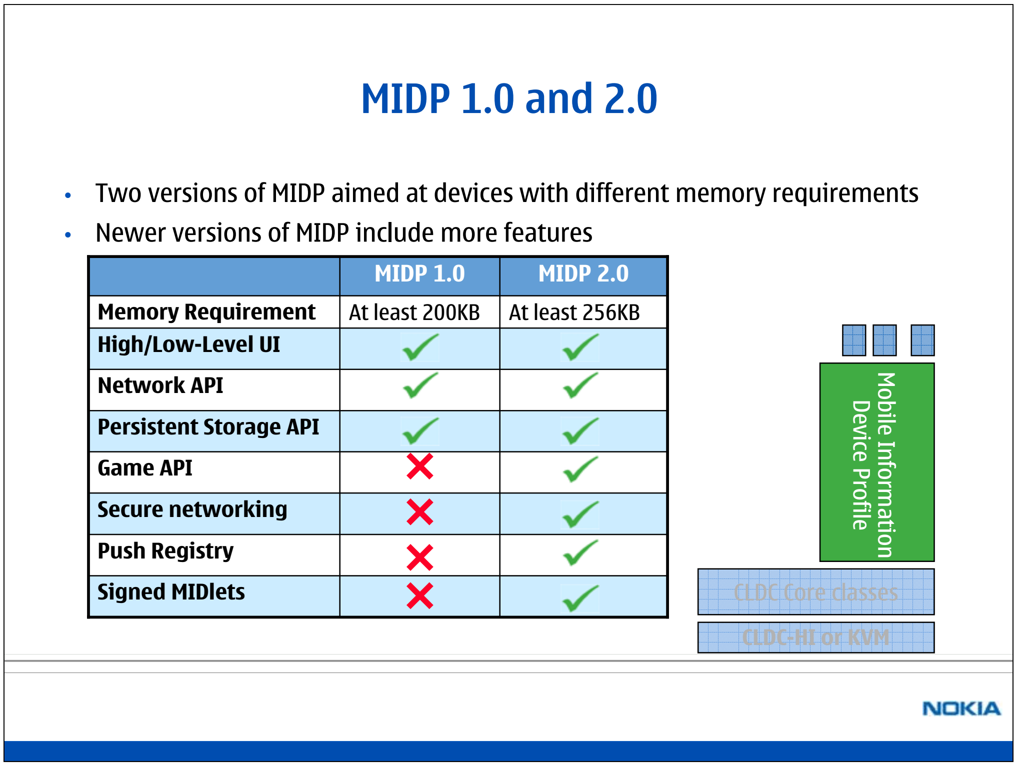
Mobile Information Device Profile

CLDC Core classes

CLDC-HI or KVM

NOKIA

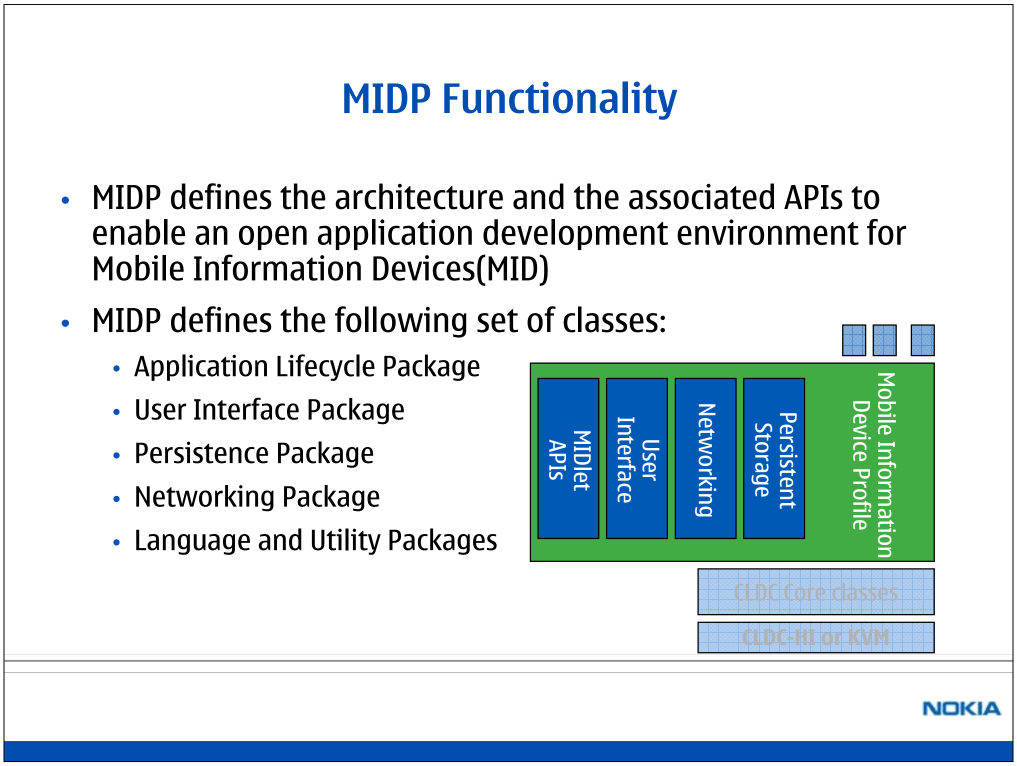
The Mobile Information Device Profile (MIDP) is a profile designed for mobile devices, two-way pages and entry-level PDAs. It offers the core application functionality required by mobile applications, including the user interface, network connectivity, local data storage, and application management. Combined with CLDC, MIDP provides a complete Java runtime environment that leverages the capabilities of handheld devices and minimizes both memory and power consumption.



MIDP 1.0 and 2.0

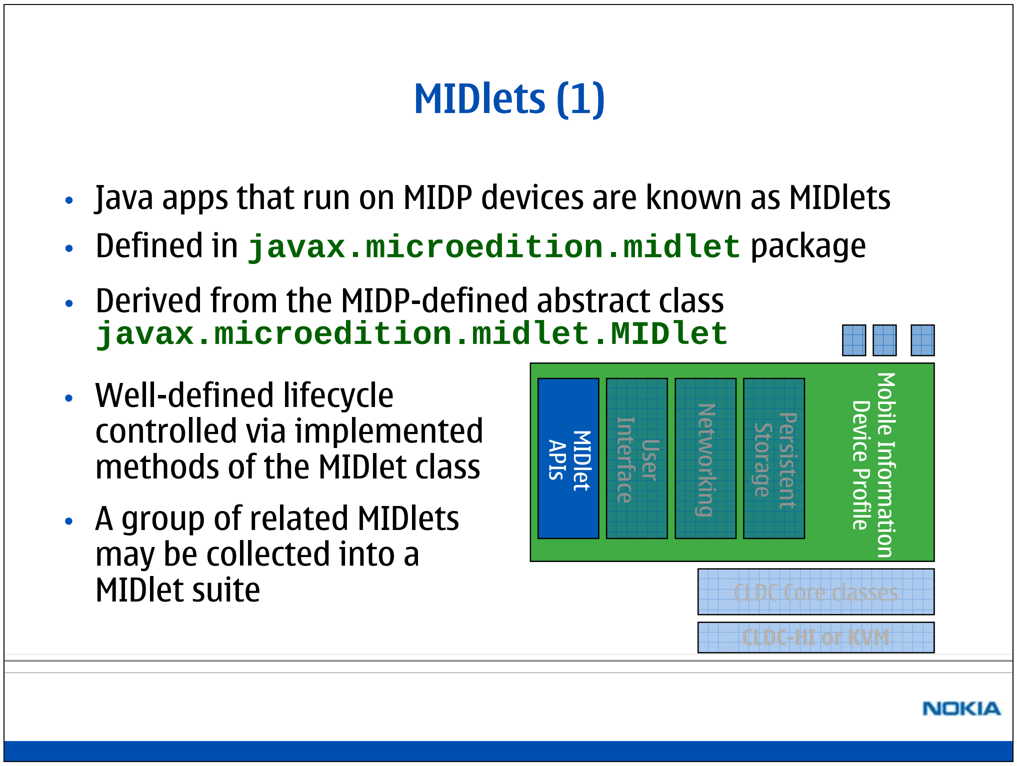
- Two versions of MIDP aimed at devices with different memory requirements
- Newer versions of MIDP include more features

	MIDP 1.0	MIDP 2.0
Memory Requirement	At least 200KB	At least 256KB
High/Low-Level UI	✓	✓
Network API	✓	✓
Persistent Storage API	✓	✓
Game API	✗	✓
Secure networking	✗	✓
Push Registry	✗	✓
Signed MIDlets	✗	✓



MIDP Functionality

- MIDP defines the architecture and the associated APIs to enable an open application development environment for Mobile Information Devices(MID)
- MIDP defines the following set of classes:
  - Application Lifecycle Package
  - User Interface Package
  - Persistence Package
  - Networking Package
  - Language and Utility Packages



MIDlets (1)

- Java apps that run on MIDP devices are known as MIDlets
- Defined in `javax.microedition.midlet` package
- Derived from the MIDP-defined abstract class `javax.microedition.midlet.MIDlet`
- Well-defined lifecycle controlled via implemented methods of the MIDlet class
- A group of related MIDlets may be collected into a MIDlet suite

MIDlet APIs

User Interface

Networking

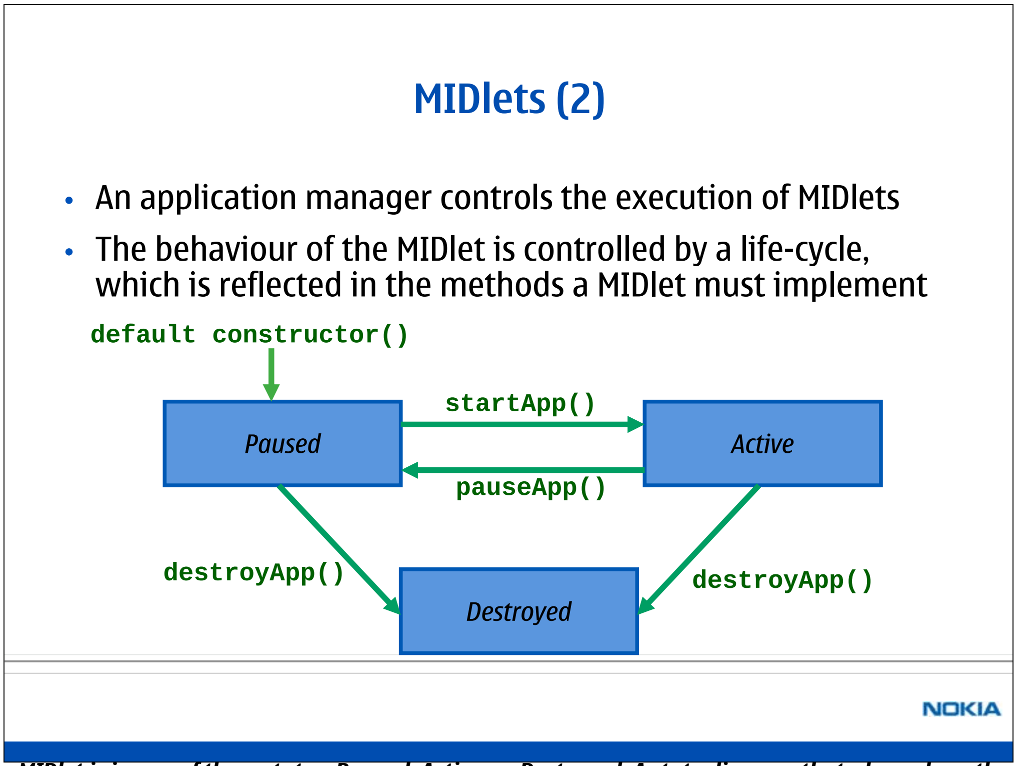
Persistent Storage

Mobile Information Device Profile

CLDC Core classes

CLDC-HI or KVM

Java applications that run on MIDP devices are known as MIDlets. A MIDlet consists of at least one Java class that must be derived from the MIDP-defined abstract class javax.microedition.midlet.MIDlet. MIDlets run in an execution environment within the Java VM that provides a well-defined lifecycle controlled via methods of the MIDlet class that each MIDlet must implement. A MIDlet can also use methods in the MIDlet class to obtain services from its environment, and it must use only the APIs defined in the MIDP specification if it is to be device-portable.



MIDlets (2)

- An application manager controls the execution of MIDlets
- The behaviour of the MIDlet is controlled by a life-cycle, which is reflected in the methods a MIDlet must implement

At any given time, a MIDlet is in one of three states: Paused, Active, or Destroyed. A state diagram that shows how these states are related and the legal state transitions is shown in this slide.

When a MIDlet is loaded, it is initially in the Paused state. The usual class and instance initialization is then performed - that is, static initializers are called the first time the MIDlet class is loaded, all instance initializers are invoked when the MIDlet instance is created, and its public, no-argument constructor is then invoked. If the MIDlet throws an exception during the execution of its constructor, the MIDlet is destroyed. If the MIDlet does not throw an exception, it is scheduled for execution at some later time. Its state is changed from Paused to Active, and its startApp( ) method is called. The MIDlet class declares this method as follows:

```
protected void startApp(  ) throws MIDletStateChangeException;
```

That this method is abstract means that you must implement it in your MIDlet, and that it is protected implies that it will be called either from the MIDlet class itself or from another class in the javax.microedition.midlet package.

The startApp( ) method may complete normally, in which case the MIDlet is allowed to run, or it may inform the MIDP platform that the MIDlet does not want to run at this point. There are several ways to achieve the latter:

- *If the startApp( ) method detects an error condition that stops it from completing, but which might not exist later (i.e., a transient error condition), it should throw a MIDletStateChangeException. This moves the MIDlet back to the Paused state, so that another attempt to start it can be made later.*
- *If the startApp( ) method detects an error condition from which recovery is likely never to be possible (a nontransient error condition), it should call its notifyDestroyed( ) method, which is described a little later.*
- *Finally, the MIDlet may throw an exception other than MIDletStateChangeException, either deliberately or because a method that it invokes throws the exception, and the startApp( ) method does not catch it. In this case, it is assumed that a fatal error has occurred, and the MIDlet is destroyed by calling its destroyApp( ) method (described later).*

If the MIDlet does none of these things, it is in the Active state and will be allowed to run until it is either paused or destroyed.

At any time, the MIDP platform can put a MIDlet into the Paused state. On a cell phone, for example, this might happen when the host software detects an incoming call and needs to release the device's display so the user can answer the call. When a MIDlet is paused, its pauseApp( ) method is called:

```
protected abstract void pauseApp(  );
```

As with startApp( ), a MIDlet is required to provide an implementation for this method. The appropriate response to this state change depends on the MIDlet itself, but, in general, it should release any resources it is holding and save the current state so it can restore itself when it is reactivated later.

If the host platform decides to resume a paused MIDlet, because the incoming call has terminated, for example, the MIDlet's startApp( ) method is invoked again to notify the MIDlet that it has access to the screen. As a consequence, a MIDlet's startApp( ) method should be written carefully to distinguish, if necessary, between the first time that it is called, which signifies that the MIDlet is being started for the first time, and subsequent calls notifying resumption from the Paused state, to prevent resources from being allocated multiple times.

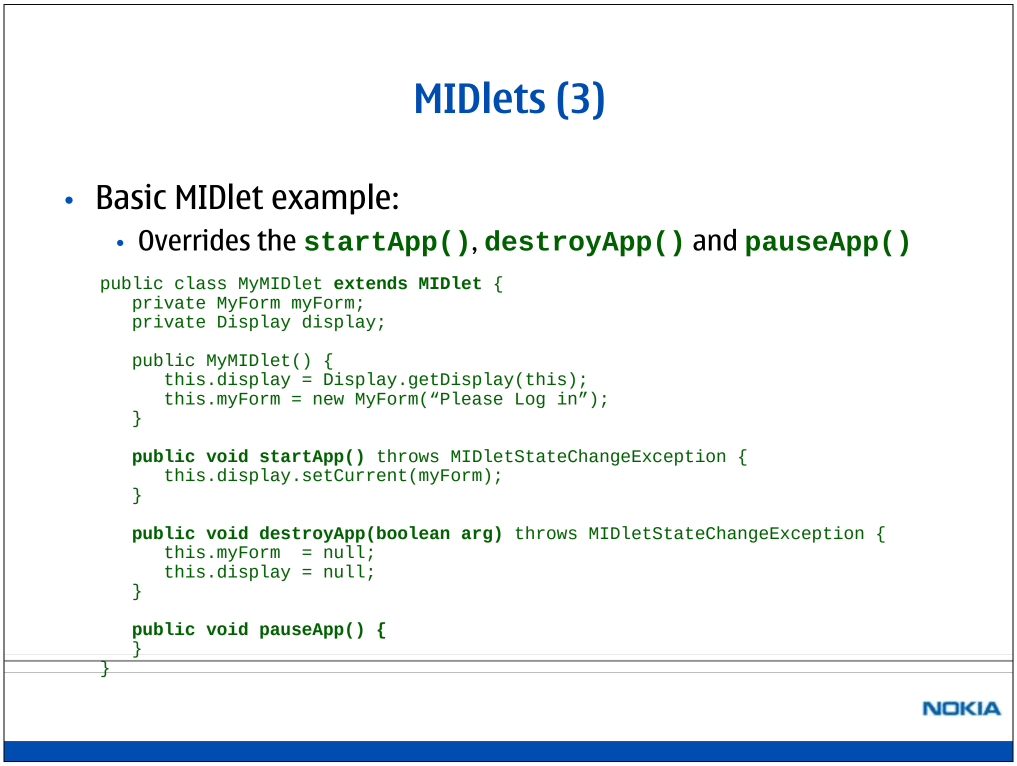
When the host platform needs to terminate a MIDlet, it calls the MIDlet's destroyApp( ) method:

```
public abstract void destroyApp(boolean unconditional) throws MIDletStateChangeException;
```

In the destroyApp( ) method, the MIDlet should release all the resources that it has allocated, terminate any background threads, and stop any active timers. There are two other methods that a MIDlet may invoke to influence its own lifecycle:

```
public final void notifyPaused(  );
public final void resumeRequest(  );
```

The notifyPaused( ) method informs the platform that the MIDlet wishes to be moved to the Paused state; this has the same effect as if the platform had invoked the MIDlet's pauseApp( ) method. When the MIDlet calls notifyPaused( ), the platform does not invoke its pauseApp( ) method, in the same way that it does not call destroyApp( ) in response to notifyDestroyed( ), because it assumes that the MIDlet has

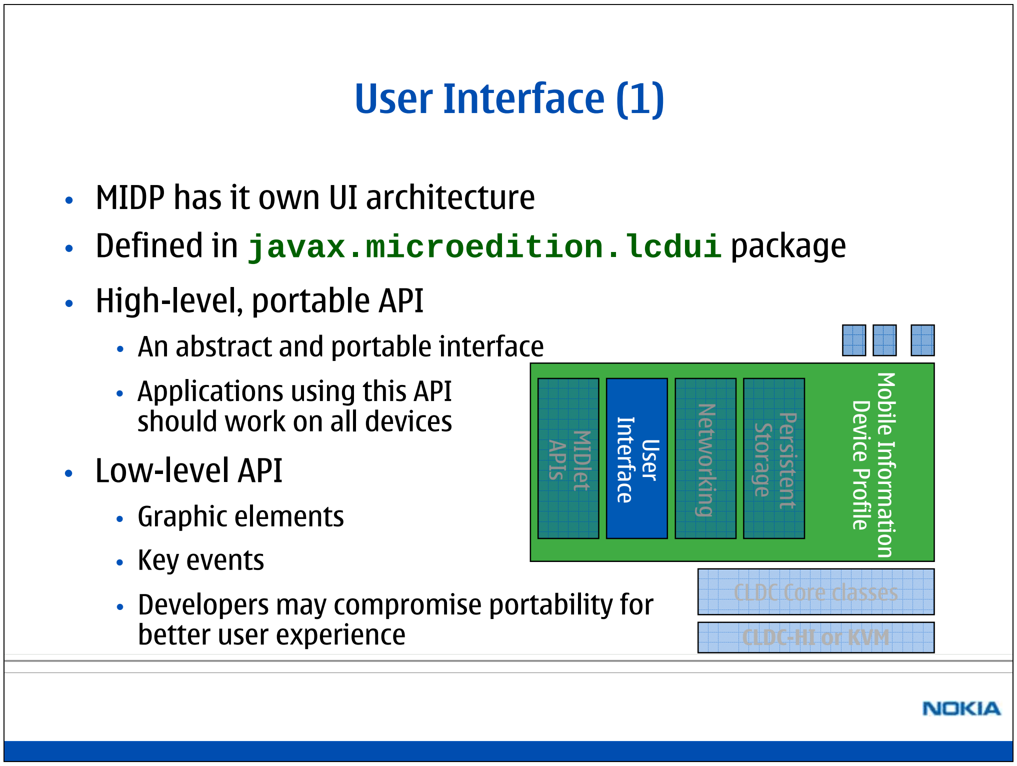


MIDlets (3)

- Basic MIDlet example:
  - Overrides the **`startApp()`**, **`destroyApp()`** and **`pauseApp()`**

```
public class MyMIDlet extends MIDlet {
   private MyForm myForm;
   private Display display;

   public MyMIDlet() {
      this.display = Display.getDisplay(this);
      this.myForm = new MyForm(“Please Log in”);
   }

   public void startApp() throws MIDletStateChangeException {
      this.display.setCurrent(myForm);
   }

   public void destroyApp(boolean arg) throws MIDletStateChangeException {
      this.myForm  = null;
      this.display = null;
   }

   public void pauseApp() {
   }
}
```

All MIDlets are derived from the abstract base class javax.microedition.midlet.MIDlet, which contains methods that the MIDP platform calls to control the MIDlet's lifecycle, as well as methods that the MIDlet itself can use to request a change in its state. A MIDlet must have a public default constructor (that is, a constructor that requires no arguments), which may be one supplied by the developer if there is any initialization to perform or, when there are no explicit constructors, the empty default constructor inserted by the Java compiler.



User Interface (1)

- MIDP has it own UI architecture
- Defined in `javax.microedition.lcdui` package
- High-level, portable API
  - An abstract and portable interface
  - Applications using this API should work on all devices
- Low-level API
  - Graphic elements
  - Key events
  - Developers may compromise portability for better user experience

MIDlet APIs
User Interface
Networking
Persistent Storage
Mobile Information Device Profile
CLDC Core classes
CLDC-HI or KVM
NOKIA

The user interface requirements for handheld devices are different from those for desktop computers. For example, the display size of handheld devices is smaller, and the input devices do not always include pointing devices, for example, a mouse or pen input. For these reasons, you cannot follow the same user interface programming guidelines for applications running on desktop computers and hand-held devices.

The CLDC provides foundation classes for all resource-constrained Java-enabled handheld devices and needs to be complemented by profiles. For example, the CLDC itself does not define any Graphical User Interface (GUI) functionality. Official GUI classes are included in profiles such as the MIDP. The GUI classes included in the MIDP are defined in the javax.microedition.lcdui package. These are not based on the Abstract Window Toolkit (AWT).

The MIDP UI classes were designed with the consumer devices in mind. They consist of high-level and low-level APIs.

High-Level API

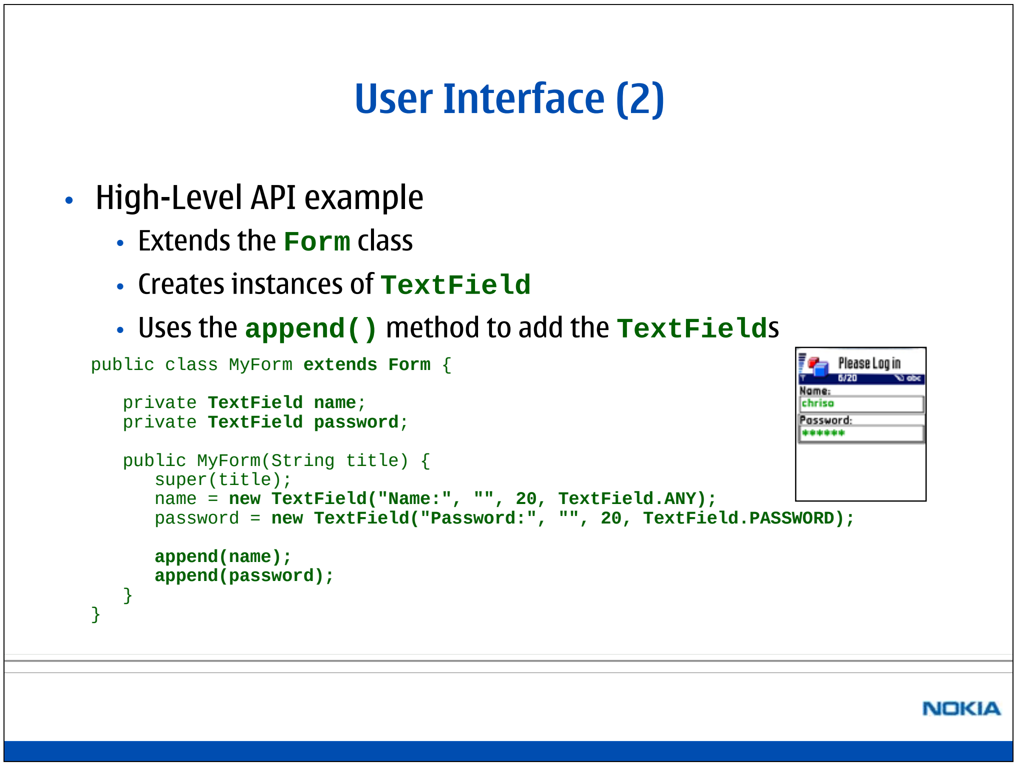
The high-level API is designed for applications whose client parts run on mobile information devices where portability is important. Applications that use this API should work on all devices. To achieve portability, the API employs a high-level abstraction and gives little control over its look and feel. For example, there is no direct access to device features (colours, sizes, input). Interaction with components is encapsulated by the implementation and the application is not aware of such interactions.

When the high-level API is used, the underlying implementation does the necessary adaptation to the device's hardware and native user interface style. The high-level API is implemented by classes that inherit from the Screen class. It is much simpler and less powerful than AWT.

Low-Level API

The low-level API provides little abstraction. It is designed for applications that need precise placement and control of graphic elements and access to low-level key events. This API gives the application full control over what is being drawn on the display. The Canvas and Graphics classes implement the low-level API.

It is important to note that MIDlets that access the low-level API are not guaranteed to be portable because this API provides mechanisms to access details that are specific to a particular device. This compromise on portability however may result in a better user experience.



User Interface (2)

- High-Level API example
  - Extends the `Form` class
  - Creates instances of `TextField`
  - Uses the `append()` method to add the `TextField`s

```
public class MyForm extends Form {

   private TextField name;
   private TextField password;

   public MyForm(String title) {
      super(title);
      name = new TextField("Name:", "", 20, TextField.ANY);
      password = new TextField("Password:", "", 20, TextField.PASSWORD);

      append(name);
      append(password);
   }
}
```

Please Log in

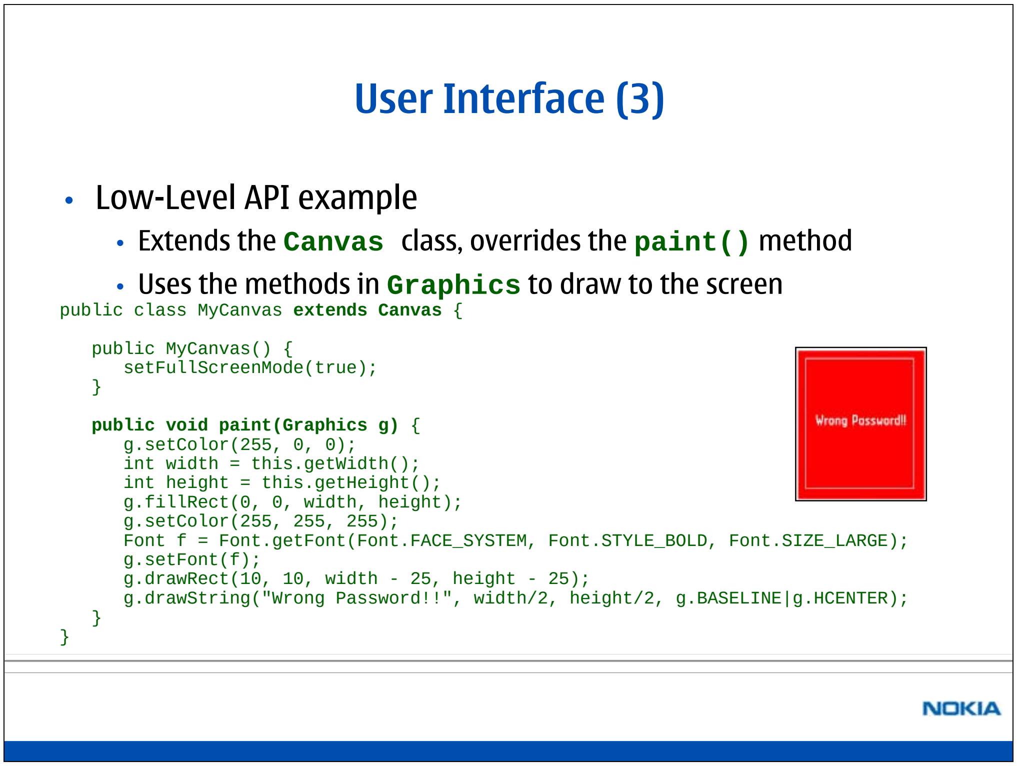
6/20 abc

Name:

chrisa

Password:

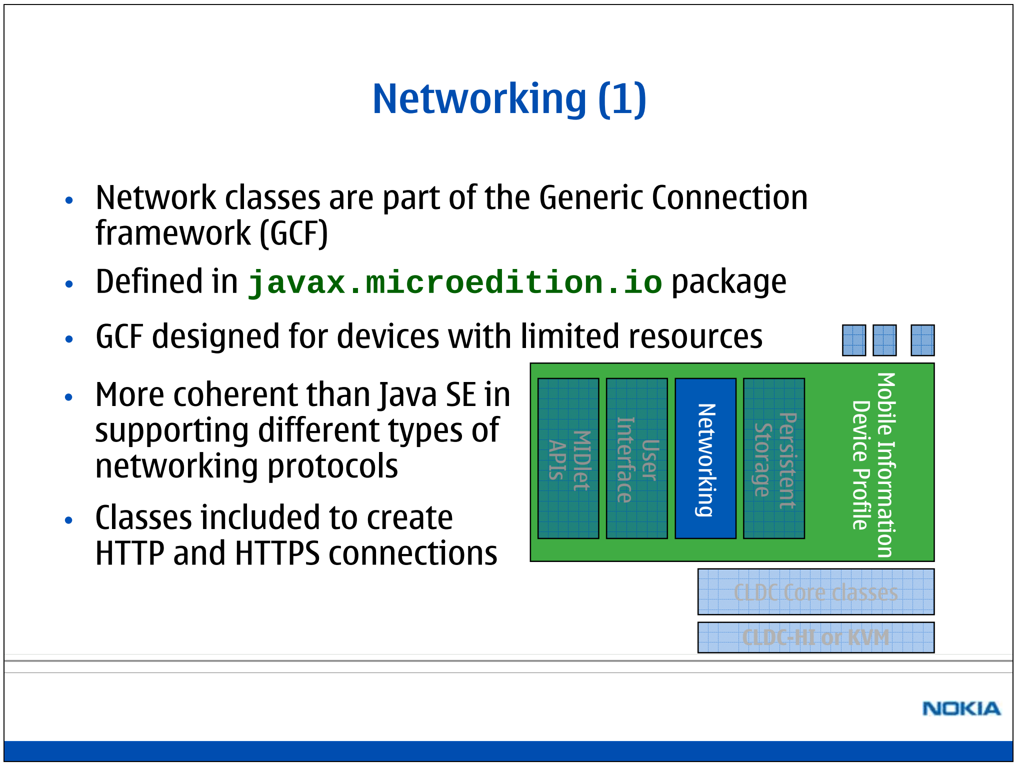
******



User Interface (3)

- Low-Level API example
  - Extends the **`Canvas`** class, overrides the **`paint()`** method
  - Uses the methods in **`Graphics`** to draw to the screen

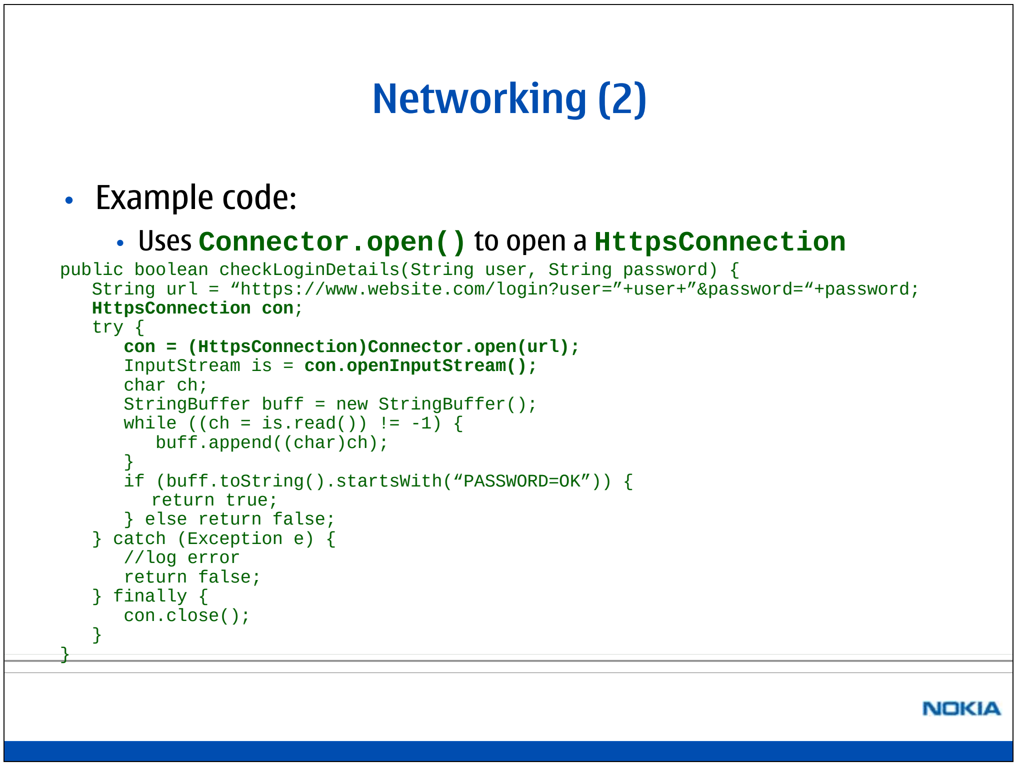
```
public class MyCanvas extends Canvas {

   public MyCanvas() {
      setFullScreenMode(true);
   }

   public void paint(Graphics g) {
      g.setColor(255, 0, 0);
      int width = this.getWidth();
      int height = this.getHeight();
      g.fillRect(0, 0, width, height);
      g.setColor(255, 255, 255);
      Font f = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD, Font.SIZE_LARGE);
      g.setFont(f);
      g.drawRect(10, 10, width - 25, height - 25);
      g.drawString("Wrong Password!!", width/2, height/2, g.BASELINE|g.HCENTER);
   }
}
```



Networking (1)

- Network classes are part of the Generic Connection framework (GCF)
- Defined in `javax.microedition.io` package
- GCF designed for devices with limited resources
- More coherent than Java SE in supporting different types of networking protocols
- Classes included to create HTTP and HTTPS connections

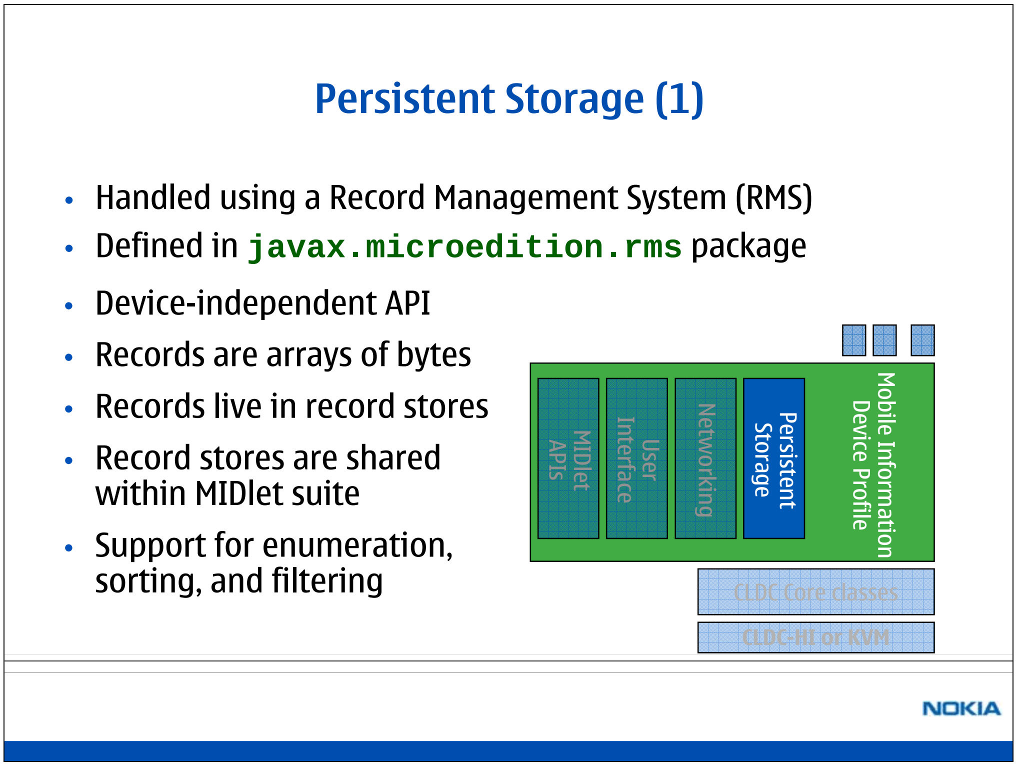
MIDlet APIs
User Interface
Networking
Persistent Storage
Mobile Information Device Profile
CLDC Core classes
CLDC-HI or KVM
NOKIA



Networking (2)

- Example code:
  - Uses `Connector.open()` to open a `HttpsConnection`

```
public boolean checkLoginDetails(String user, String password) {
   String url = “https://www.website.com/login?user=”+user+”&password=“+password;
   HttpsConnection con;
   try {
      con = (HttpsConnection)Connector.open(url);
      InputStream is = con.openInputStream();
      char ch;
      StringBuffer buff = new StringBuffer();
      while ((ch = is.read()) != -1) {
         buff.append((char)ch);
      }
      if (buff.toString().startsWith(“PASSWORD=OK”)) {
         return true;
      } else return false;
   } catch (Exception e) {
      //log error
      return false;
   } finally {
      con.close();
   }
}
```



Persistent Storage (1)

- Handled using a Record Management System (RMS)
- Defined in `javax.microedition.rms` package
- Device-independent API
- Records are arrays of bytes
- Records live in record stores
- Record stores are shared within MIDlet suite
- Support for enumeration, sorting, and filtering

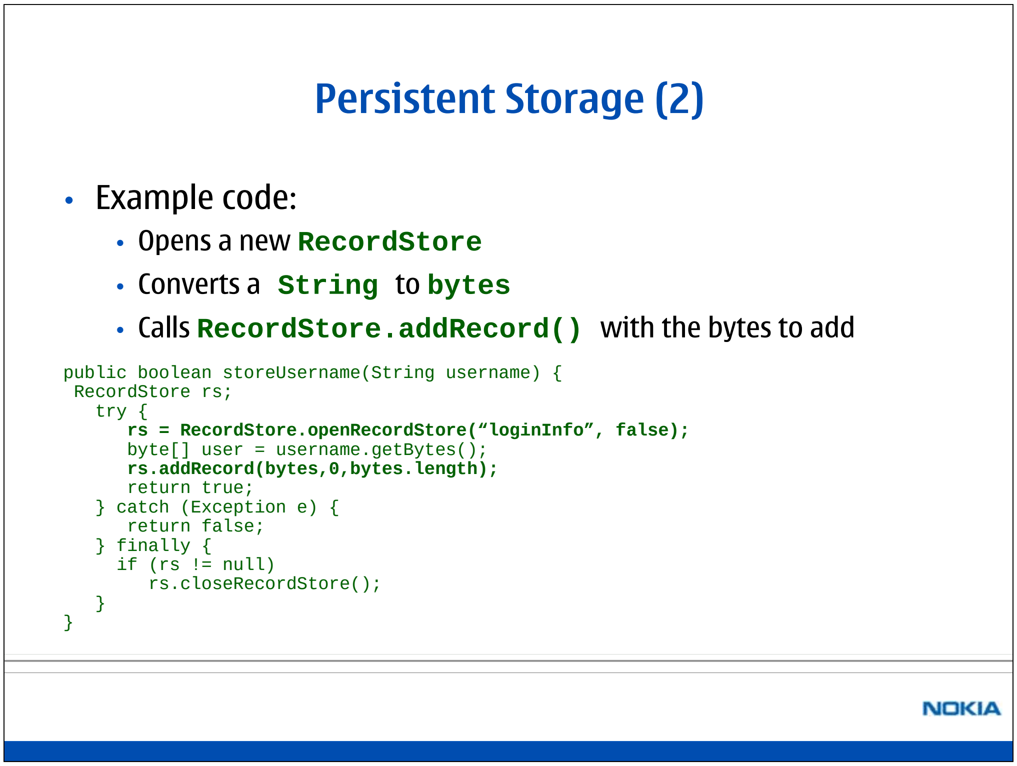
MIDlet APIs
User Interface
Networking
Persistent Storage
Mobile Information Device Profile
CLDC Core classes
CLDC-HI or KVM
NOKIA

Persistent storage is a non-volatile place for storing the state of objects. The MIDP provides a Device- independent API for MIDlets to persistently store data and retrieve it later. This mechanism is a simple record- oriented database called the Record Management System (RMS). A MIDP database (or a record store) consists of a collection of records that remain persistent after the MIDlet exits. When you invoke the MIDlet again, it can retrieve data from the persistent record store.

To use the RMS, import the javax.microedition.rms package.

A record store is a collection of records, and a record is basically a byte array of arbitrary data. Record stores (binary files) are platform-dependent because they are created in platform-dependent locations. MIDlets within a single application (a MIDlet suite) can create multiple record stores (database files) with different names. The RMS APIs provide the following functionality:

- ***Allow MIDlets to manipulate (add and remove) records within a record store.;***
- ***Allow MIDlets in the same application to share records (access one another's record store directly).***
- ***Does not provide a mechanism for sharing records between MIDlets in different applications.***
- ***Different MIDlets in the same MIDlet suite can also use the RMS to share data. (A MIDlet suite is a set of MIDlets packaged together into a single JAR file.)***
- ***There is no way to change just part of a record: the entire record must be read, the change made to the data in memory, and then the entire record is written back to the record store.***

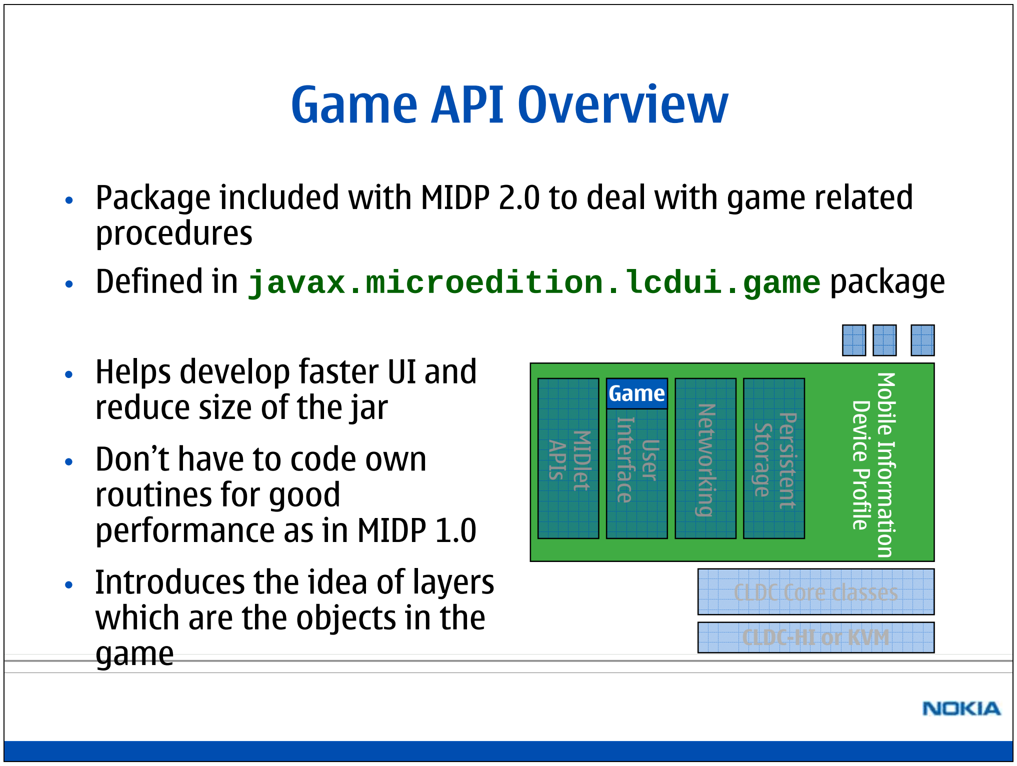


Persistent Storage (2)

- Example code:
  - Opens a new **RecordStore**
  - Converts a **String** to **bytes**
  - Calls **RecordStore.addRecord()** with the bytes to add

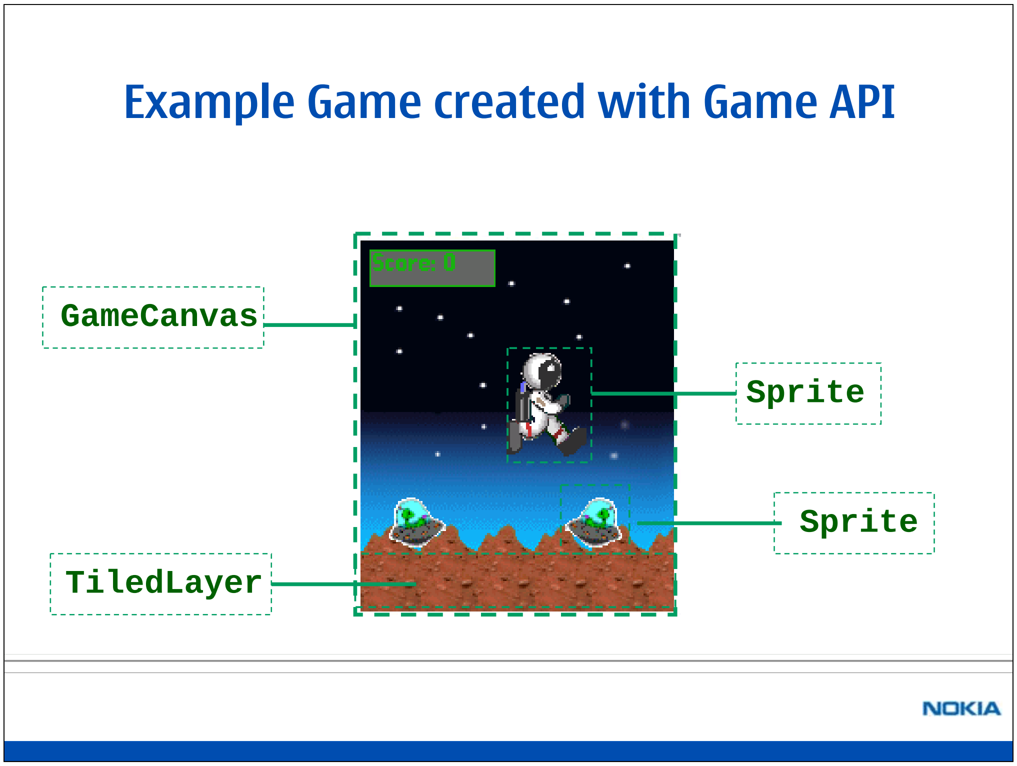
```
public boolean storeUsername(String username) {
 RecordStore rs;
   try {
      rs = RecordStore.openRecordStore(“loginInfo”, false);
      byte[] user = username.getBytes();
      rs.addRecord(bytes,0,bytes.length);
      return true;
   } catch (Exception e) {
      return false;
   } finally {
     if (rs != null)
        rs.closeRecordStore();
   }
}
```

NOKIA



Game API Overview

- Package included with MIDP 2.0 to deal with game related procedures
- Defined in `javax.microedition.lcdui.game` package
- Helps develop faster UI and reduce size of the jar
- Don't have to code own routines for good performance as in MIDP 1.0
- Introduces the idea of layers which are the objects in the game



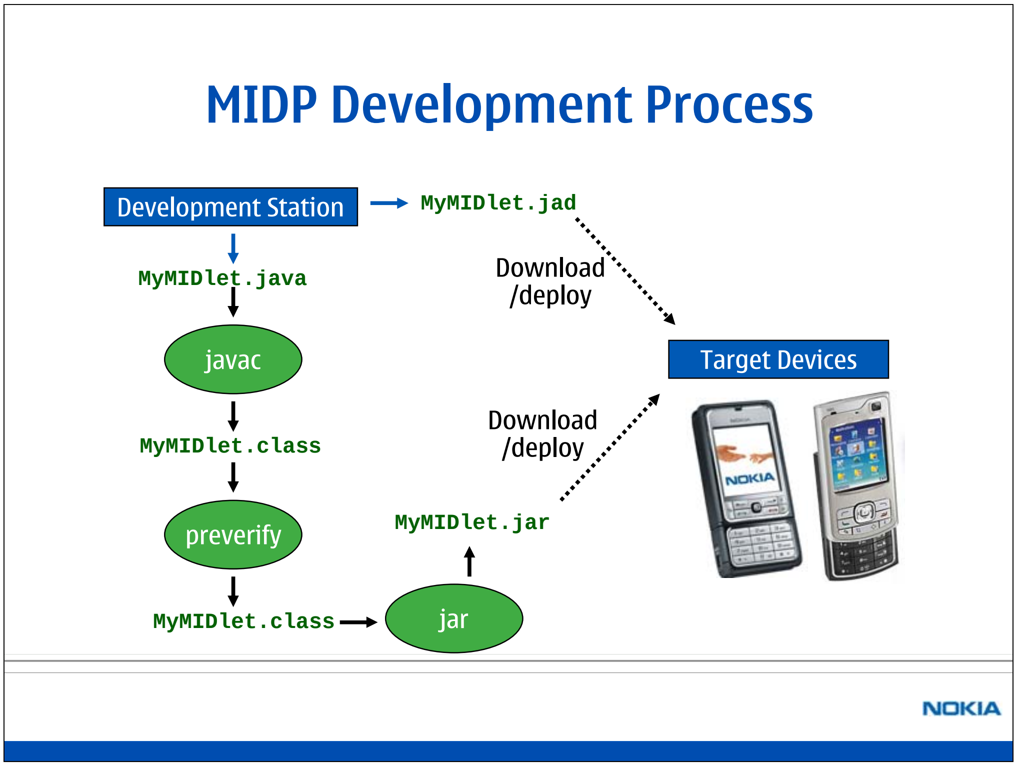
Example Game created with Game API

GameCanvas

Sprite

Sprite

TiledLayer



MIDP Development Process

Development Station

MyMIDlet.jad

MyMIDlet.java

javac

MyMIDlet.class

preverify

MyMIDlet.class

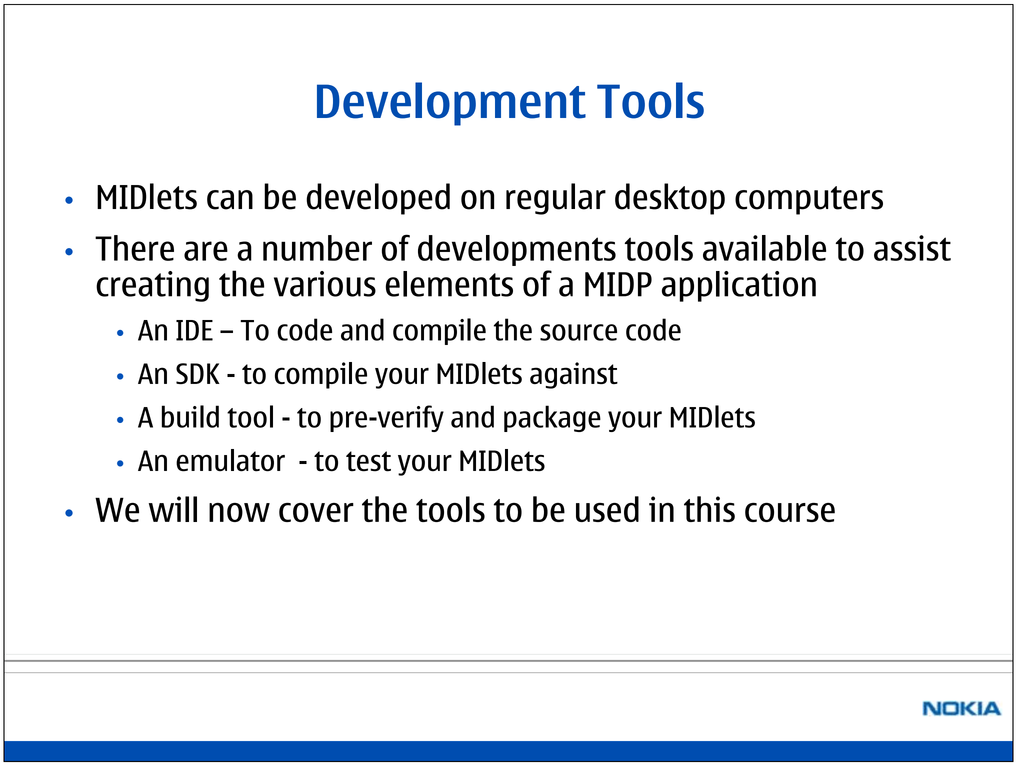
jar

MyMIDlet.jar

Download
/deploy

Download
/deploy

Target Devices

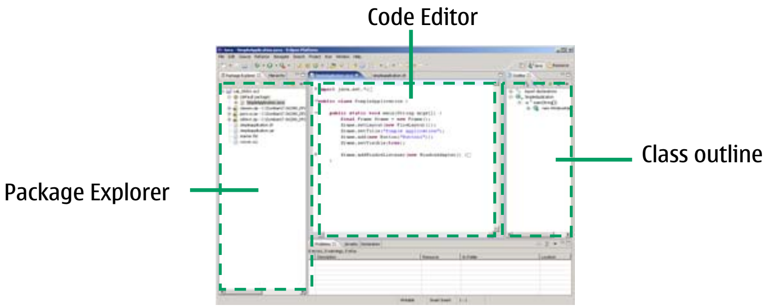


Development Tools

- MIDlets can be developed on regular desktop computers
- There are a number of developments tools available to assist creating the various elements of a MIDP application
  - An IDE – To code and compile the source code
  - An SDK - to compile your MIDlets against
  - A build tool - to pre-verify and package your MIDlets
  - An emulator - to test your MIDlets
- We will now cover the tools to be used in this course

IDE - Eclipse

- General purpose Java IDE
- Can be used to edit, compile and debug MIDlets
- Can be integrated with MIDlet build tools



NOKIA

Build Tool – Carbide.j

- Used to pre-verify, package, and deploy MIDlets

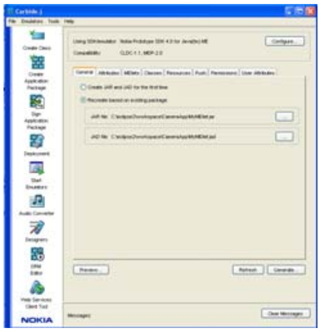
Creating New Classes

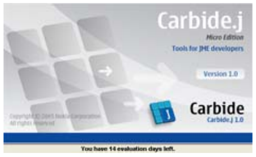
Generating JAR and JAD files

Signing applications

Deploying applications

Running emulators



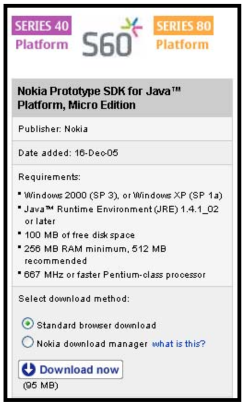


SDK - Nokia Prototype SDK for Java ME

- Provides development kit and emulators
- Download from http://www.forum.nokia.com
- Contains several Developer Platform emulators

Series 40

Series 60



Nokia Prototype SDK for Java™ Platform, Micro Edition

Publisher: Nokia

Date added: 16-Dec-05

Requirements:

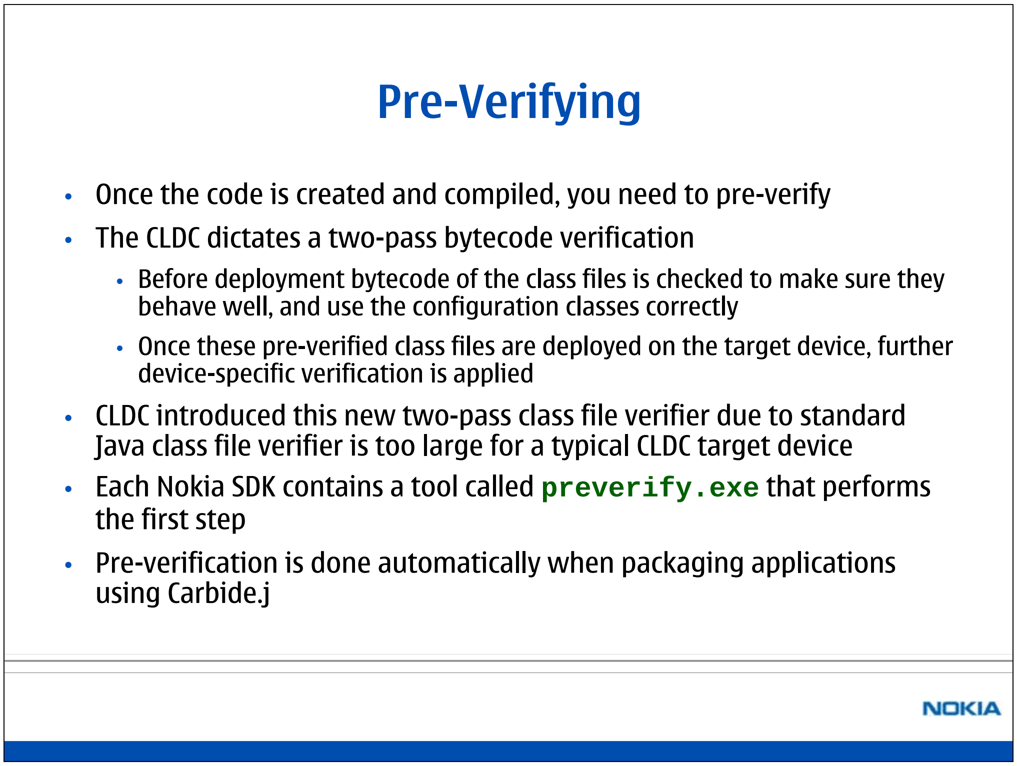
- Windows 2000 (SP 3), or Windows XP (SP 1a)
- Java™ Runtime Environment (JRE) 1.4.1_02 or later
- 100 MB of free disk space
- 256 MB RAM minimum, 512 MB recommended
- 667 MHz or faster Pentium-class processor

Select download method:

● Standard browser download

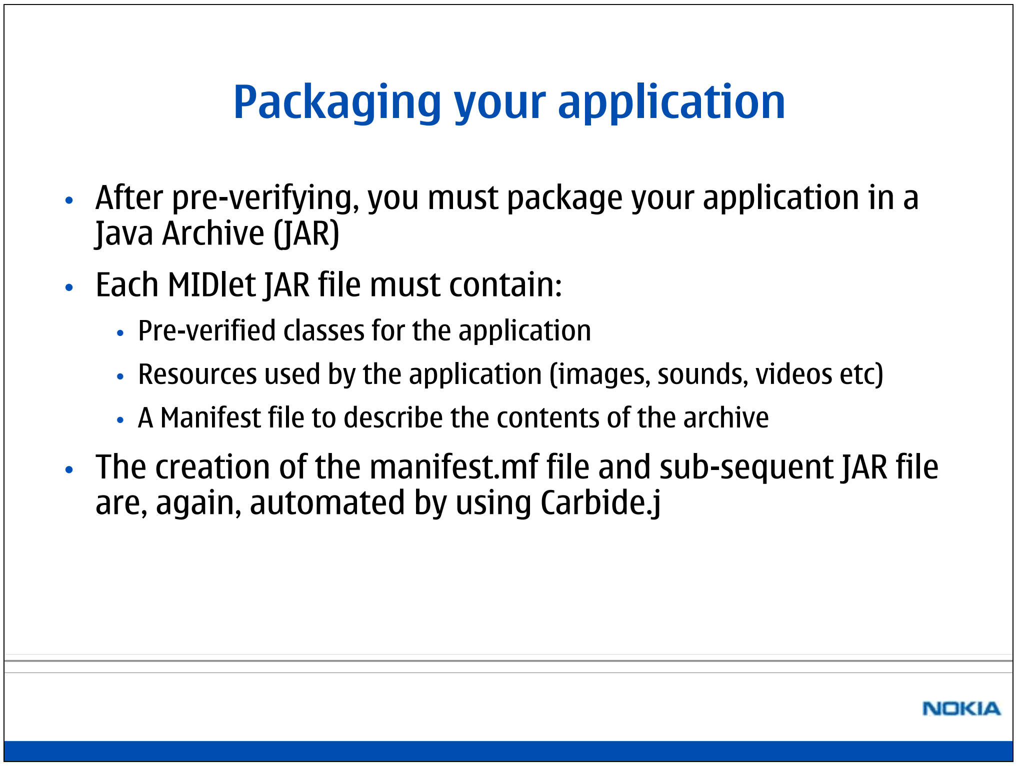
○ Nokia download manager what is this?

(95 MB)



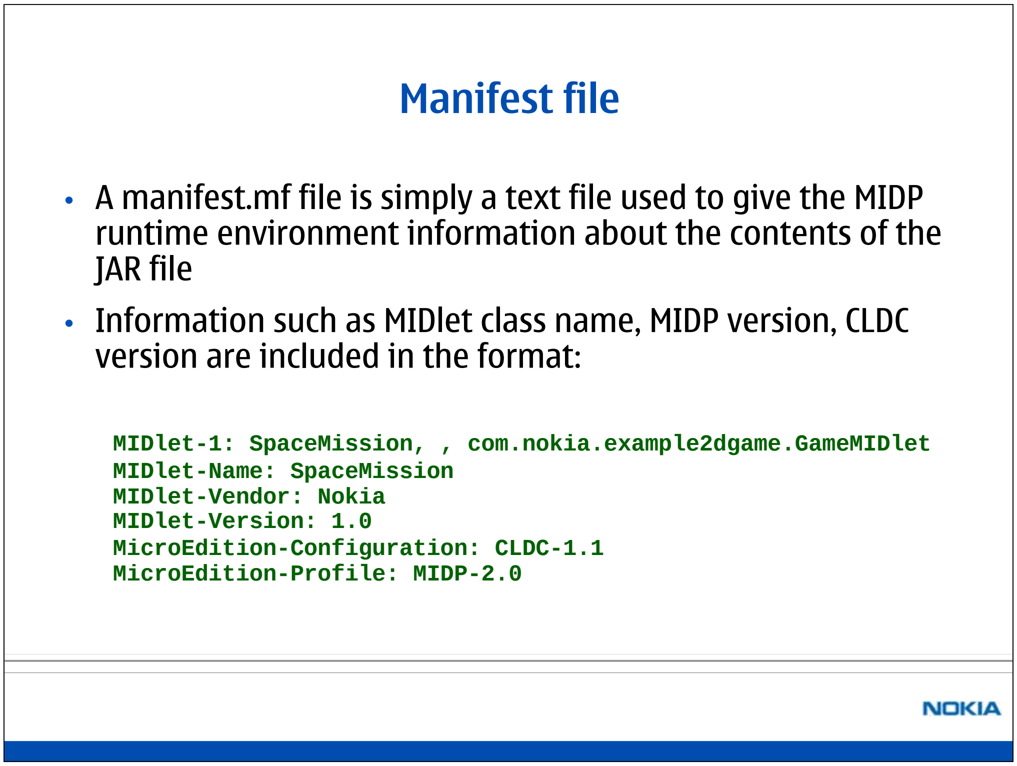
Pre-Verifying

- Once the code is created and compiled, you need to pre-verify
- The CLDC dictates a two-pass bytecode verification
  - Before deployment bytecode of the class files is checked to make sure they behave well, and use the configuration classes correctly
  - Once these pre-verified class files are deployed on the target device, further device-specific verification is applied
- CLDC introduced this new two-pass class file verifier due to standard Java class file verifier is too large for a typical CLDC target device
- Each Nokia SDK contains a tool called `preverify.exe` that performs the first step
- Pre-verification is done automatically when packaging applications using Carbide.j



Packaging your application

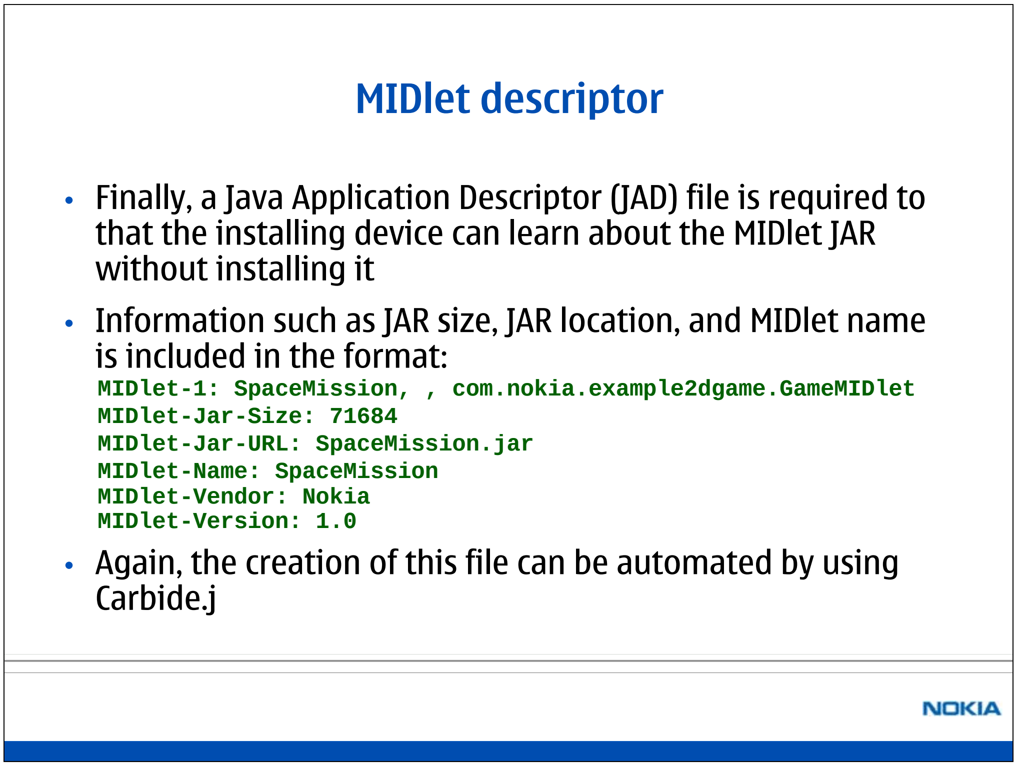
- After pre-verifying, you must package your application in a Java Archive (JAR)
- Each MIDlet JAR file must contain:
  - Pre-verified classes for the application
  - Resources used by the application (images, sounds, videos etc)
  - A Manifest file to describe the contents of the archive
- The creation of the manifest.mf file and sub-sequent JAR file are, again, automated by using Carbide.j



Manifest file

- A manifest.mf file is simply a text file used to give the MIDP runtime environment information about the contents of the JAR file
- Information such as MIDlet class name, MIDP version, CLDC version are included in the format:

```
MIDlet-1: SpaceMission, , com.nokia.example2dgame.GameMIDlet
MIDlet-Name: SpaceMission
MIDlet-Vendor: Nokia
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
```

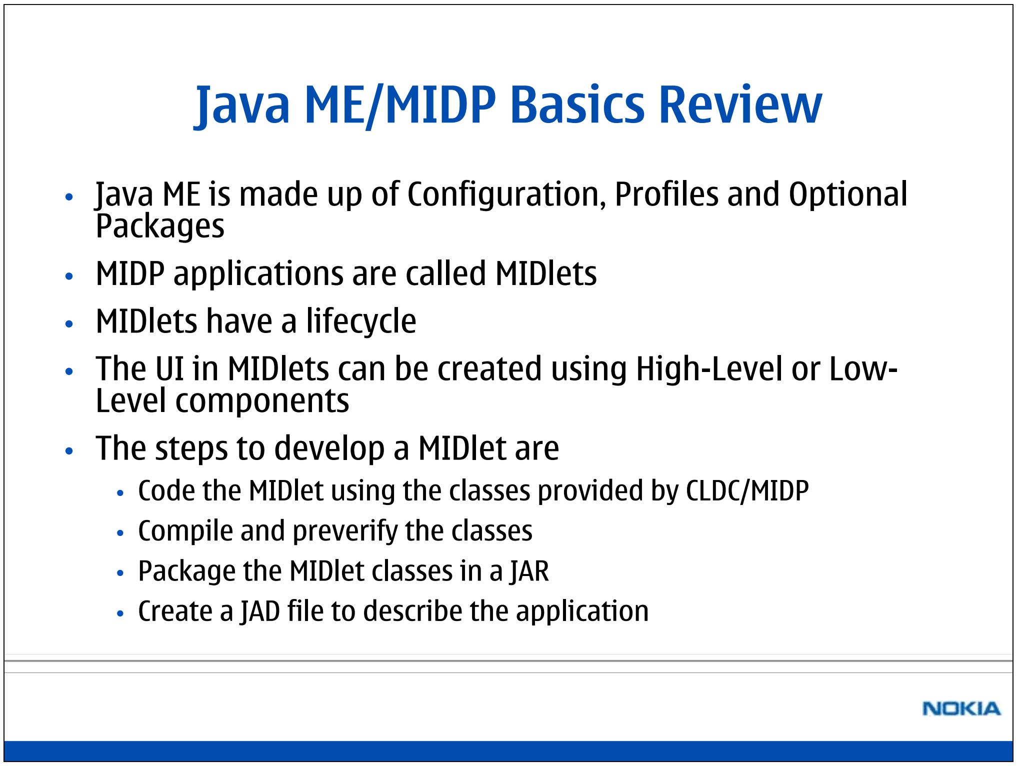


MIDlet descriptor

- Finally, a Java Application Descriptor (JAD) file is required to that the installing device can learn about the MIDlet JAR without installing it
- Information such as JAR size, JAR location, and MIDlet name is included in the format:

```
MIDlet-1: SpaceMission, , com.nokia.example2dgame.GameMIDlet
MIDlet-Jar-Size: 71684
MIDlet-Jar-URL: SpaceMission.jar
MIDlet-Name: SpaceMission
MIDlet-Vendor: Nokia
MIDlet-Version: 1.0
```

- Again, the creation of this file can be automated by using Carbide.j

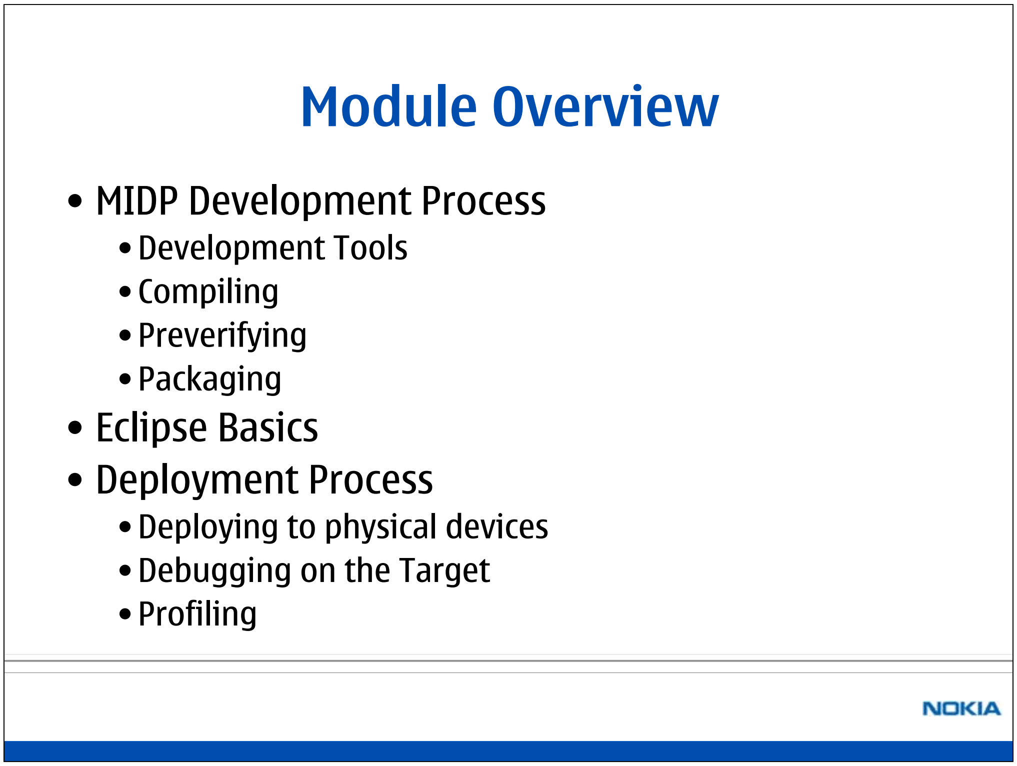


Java ME/MIDP Basics Review

- Java ME is made up of Configuration, Profiles and Optional Packages
- MIDP applications are called MIDlets
- MIDlets have a lifecycle
- The UI in MIDlets can be created using High-Level or Low-Level components
- The steps to develop a MIDlet are
  - Code the MIDlet using the classes provided by CLDC/MIDP
  - Compile and preverify the classes
  - Package the MIDlet classes in a JAR
  - Create a JAD file to describe the application

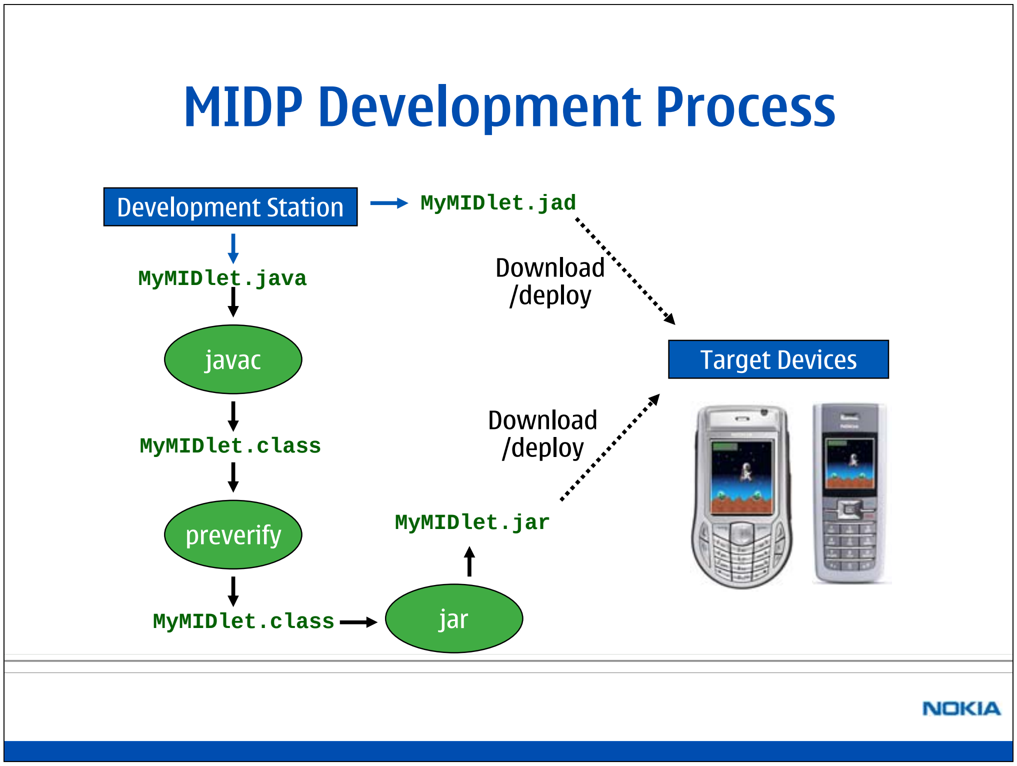
Developing and Deploying MIDP Applications

Module 05502



Module Overview

- MIDP Development Process
  - Development Tools
  - Compiling
  - Preverifying
  - Packaging
- Eclipse Basics
- Deployment Process
  - Deploying to physical devices
  - Debugging on the Target
  - Profiling



MIDP Development Process

Development Station

MyMIDlet.jad

MyMIDlet.java

javac

MyMIDlet.class

preverify

MyMIDlet.class

jar

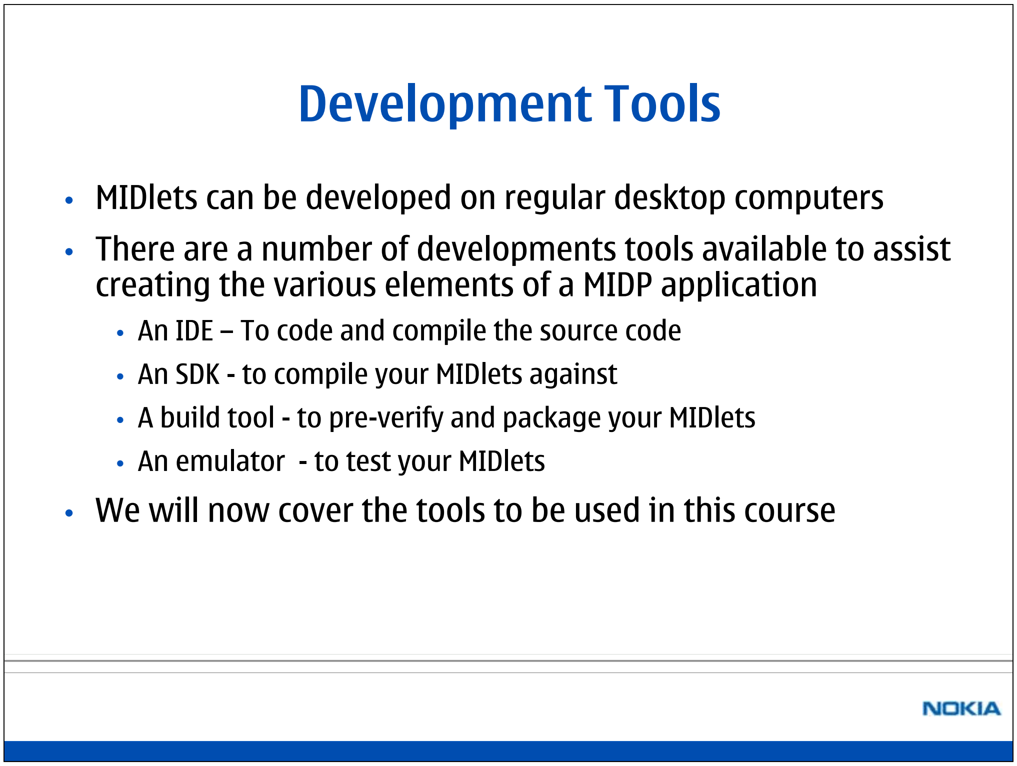
MyMIDlet.jar

Download /deploy

Download /deploy

Target Devices

NOKIA

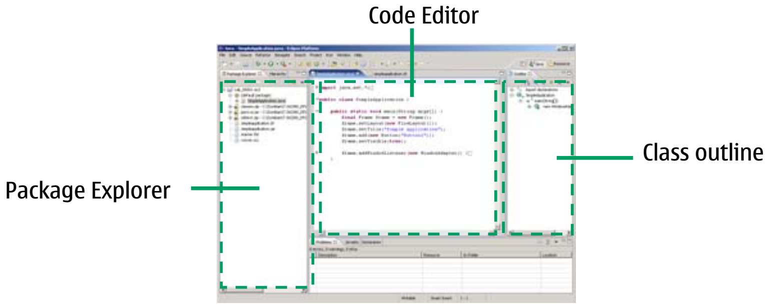


Development Tools

- MIDlets can be developed on regular desktop computers
- There are a number of developments tools available to assist creating the various elements of a MIDP application
  - An IDE – To code and compile the source code
  - An SDK - to compile your MIDlets against
  - A build tool - to pre-verify and package your MIDlets
  - An emulator - to test your MIDlets
- We will now cover the tools to be used in this course

IDE - Eclipse

- General purpose Java IDE
- Can be used to edit, compile and debug MIDlets
- Can be integrated with MIDlet build tools



Build Tool – Carbide.j

- Used to pre-verify, package, and deploy MIDlets

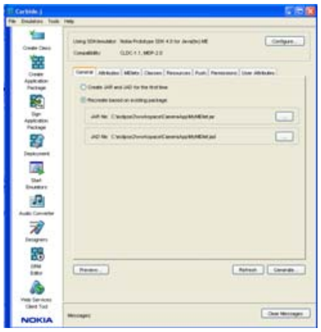
Creating New Classes

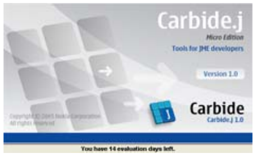
Generating JAR and JAD files

Signing applications

Deploying applications

Running emulators



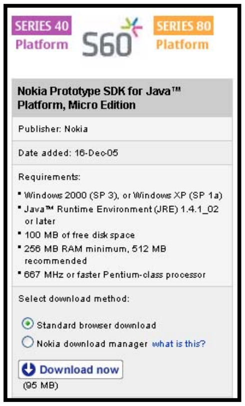


SDK - Nokia Prototype SDK for Java ME

- Provides development kit and emulators
- Download from http://www.forum.nokia.com
- Contains several Developer Platform emulators

Series 40

Series 60



Nokia Prototype SDK for Java™ Platform, Micro Edition

Publisher: Nokia

Date added: 16-Dec-05

Requirements:

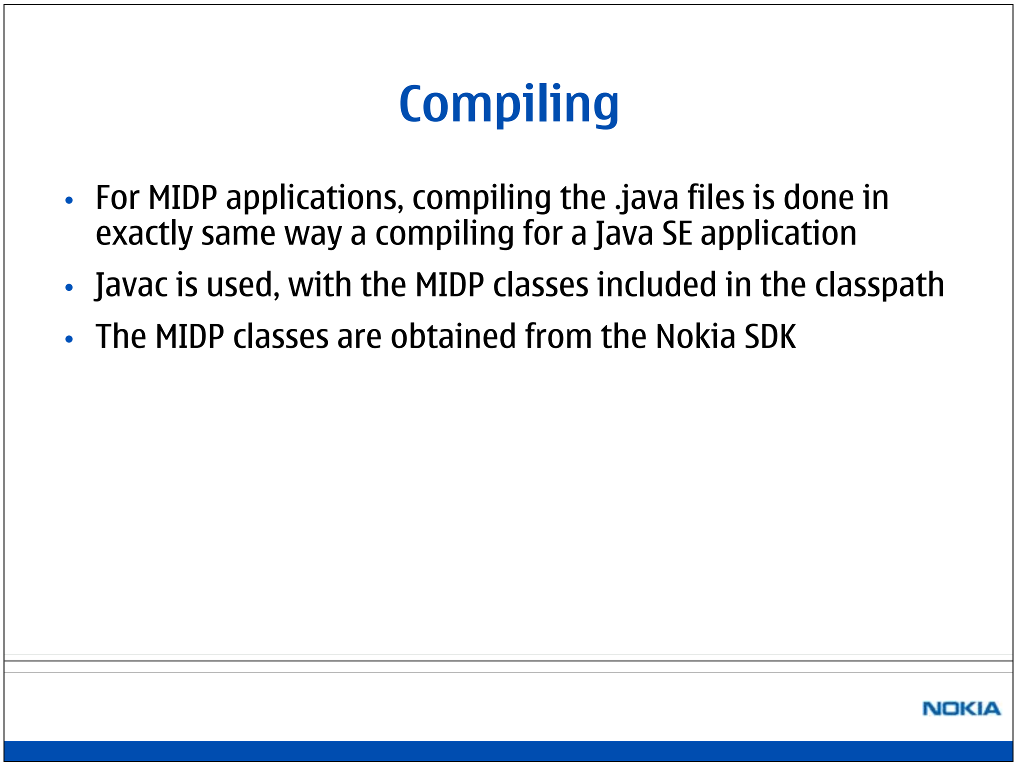
- Windows 2000 (SP 3), or Windows XP (SP 1a)
- Java™ Runtime Environment (JRE) 1.4.1_02 or later
- 100 MB of free disk space
- 256 MB RAM minimum, 512 MB recommended
- 667 MHz or faster Pentium-class processor

Select download method:

● Standard browser download
○ Nokia download manager what is this?

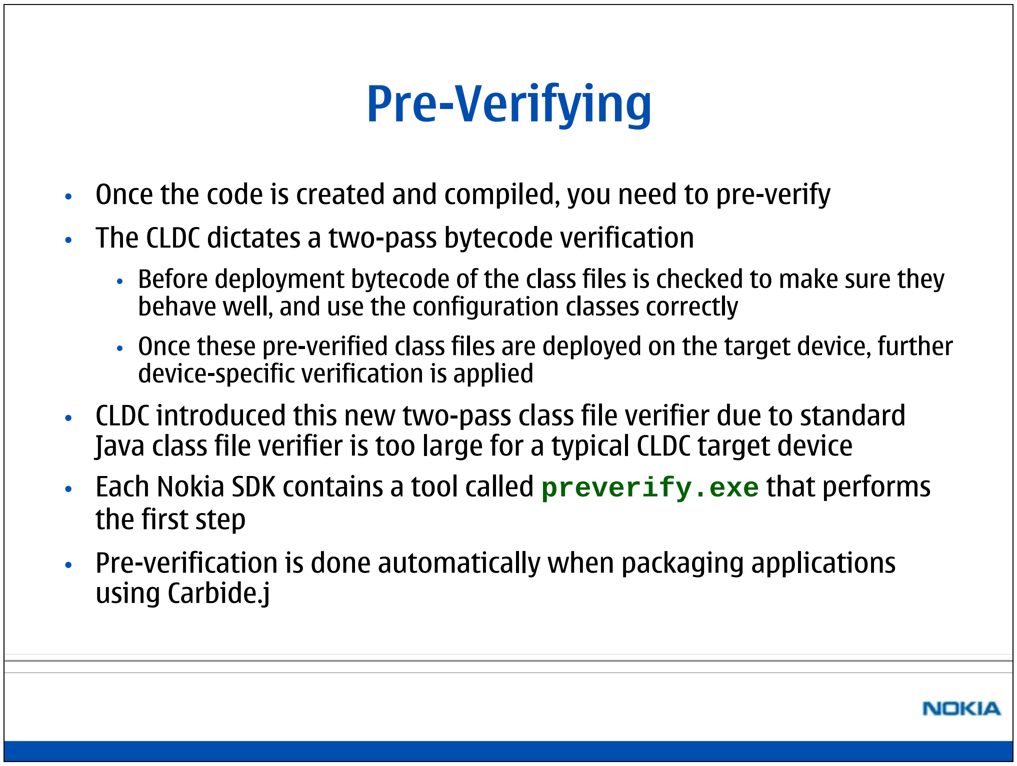
(95 MB)

NOKIA



Compiling

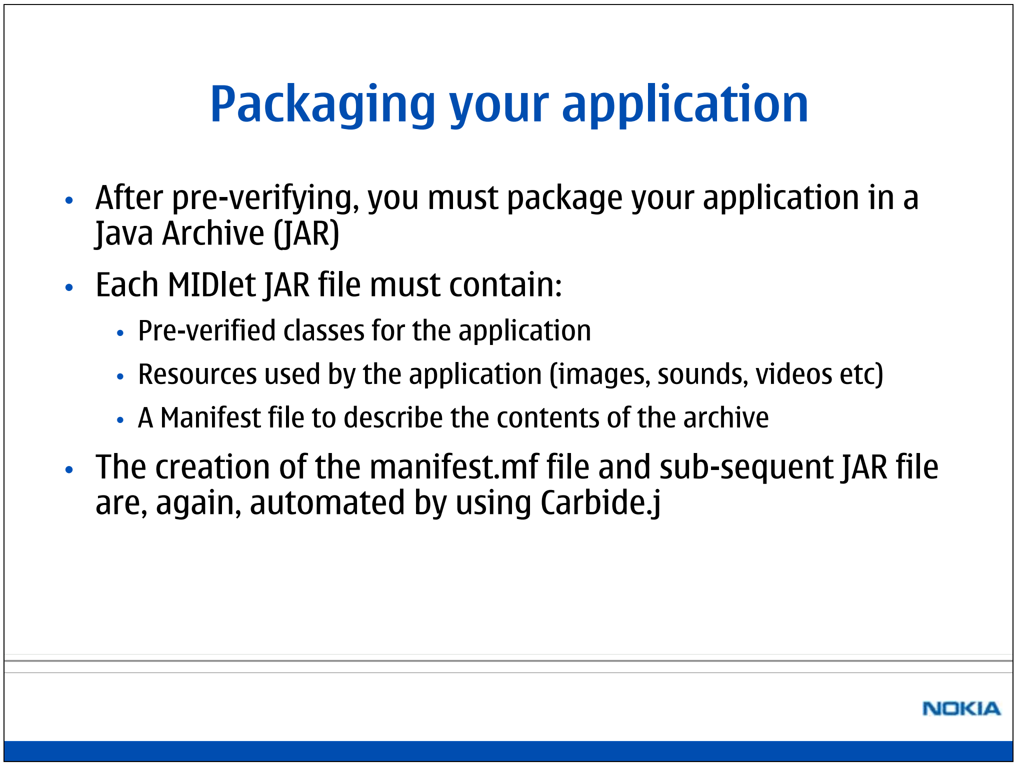
- For MIDP applications, compiling the .java files is done in exactly same way a compiling for a Java SE application
- Javac is used, with the MIDP classes included in the classpath
- The MIDP classes are obtained from the Nokia SDK



Pre-Verifying

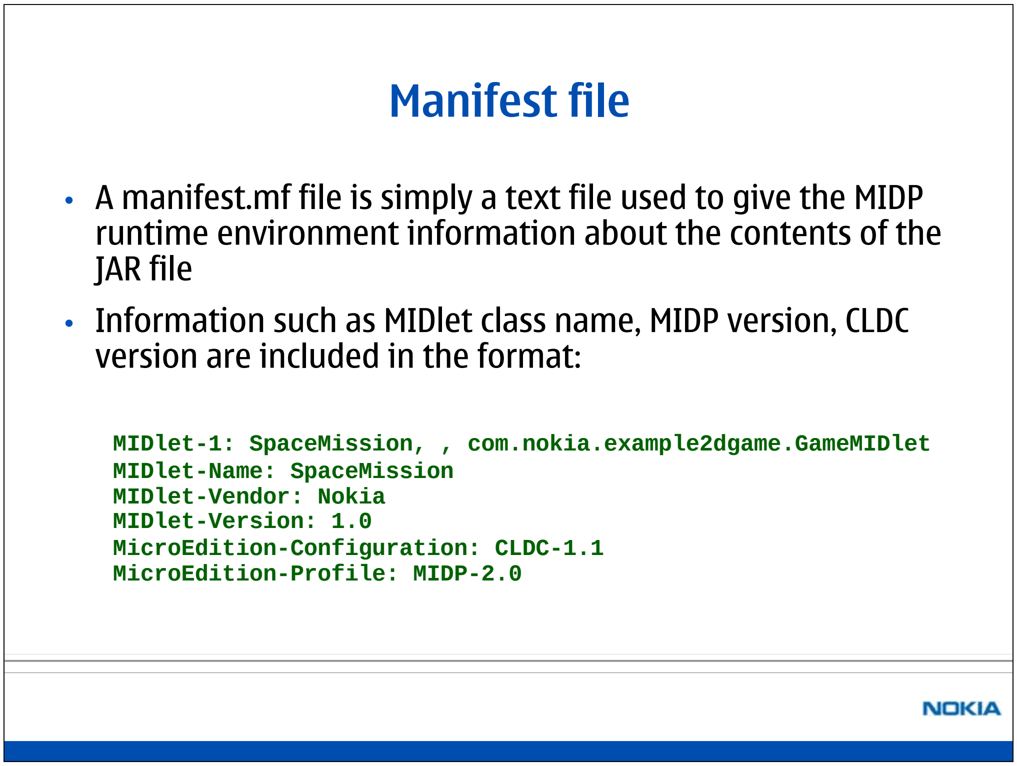
- Once the code is created and compiled, you need to pre-verify
- The CLDC dictates a two-pass bytecode verification
  - Before deployment bytecode of the class files is checked to make sure they behave well, and use the configuration classes correctly
  - Once these pre-verified class files are deployed on the target device, further device-specific verification is applied
- CLDC introduced this new two-pass class file verifier due to standard Java class file verifier is too large for a typical CLDC target device
- Each Nokia SDK contains a tool called `preverify.exe` that performs the first step
- Pre-verification is done automatically when packaging applications using Carbide.j

NOKIA



Packaging your application

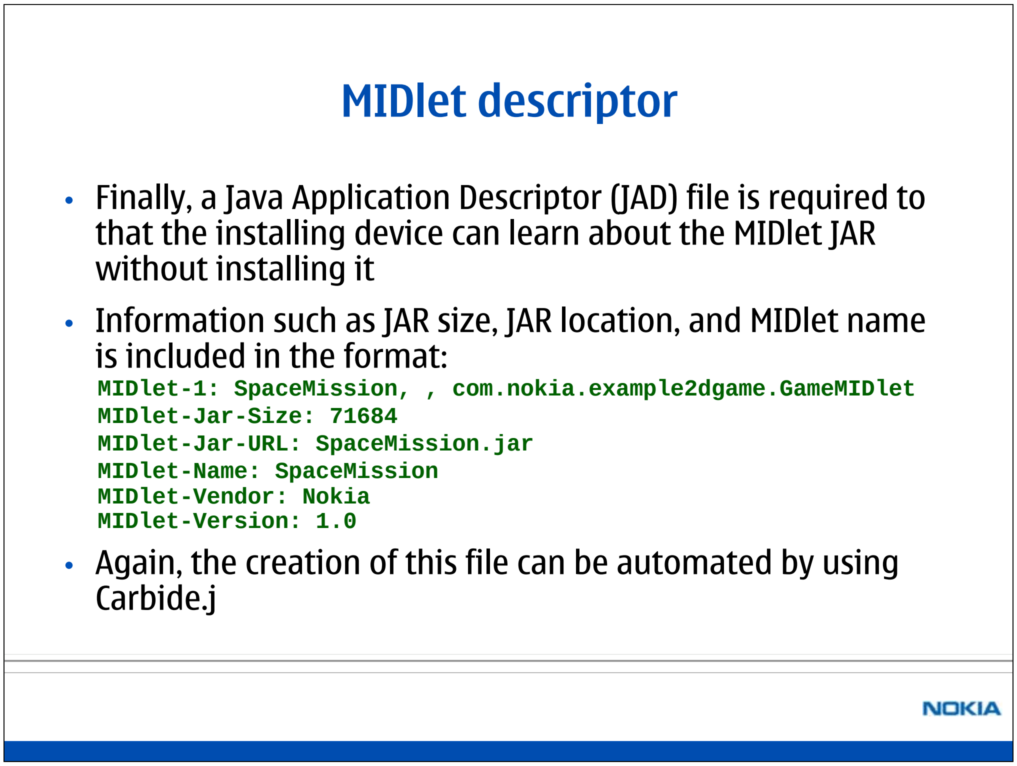
- After pre-verifying, you must package your application in a Java Archive (JAR)
- Each MIDlet JAR file must contain:
    - Pre-verified classes for the application
    - Resources used by the application (images, sounds, videos etc)
    - A Manifest file to describe the contents of the archive
- The creation of the manifest.mf file and sub-sequent JAR file are, again, automated by using Carbide.j



Manifest file

- A manifest.mf file is simply a text file used to give the MIDP runtime environment information about the contents of the JAR file
- Information such as MIDlet class name, MIDP version, CLDC version are included in the format:

```
MIDlet-1: SpaceMission, , com.nokia.example2dgame.GameMIDlet
MIDlet-Name: SpaceMission
MIDlet-Vendor: Nokia
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
```

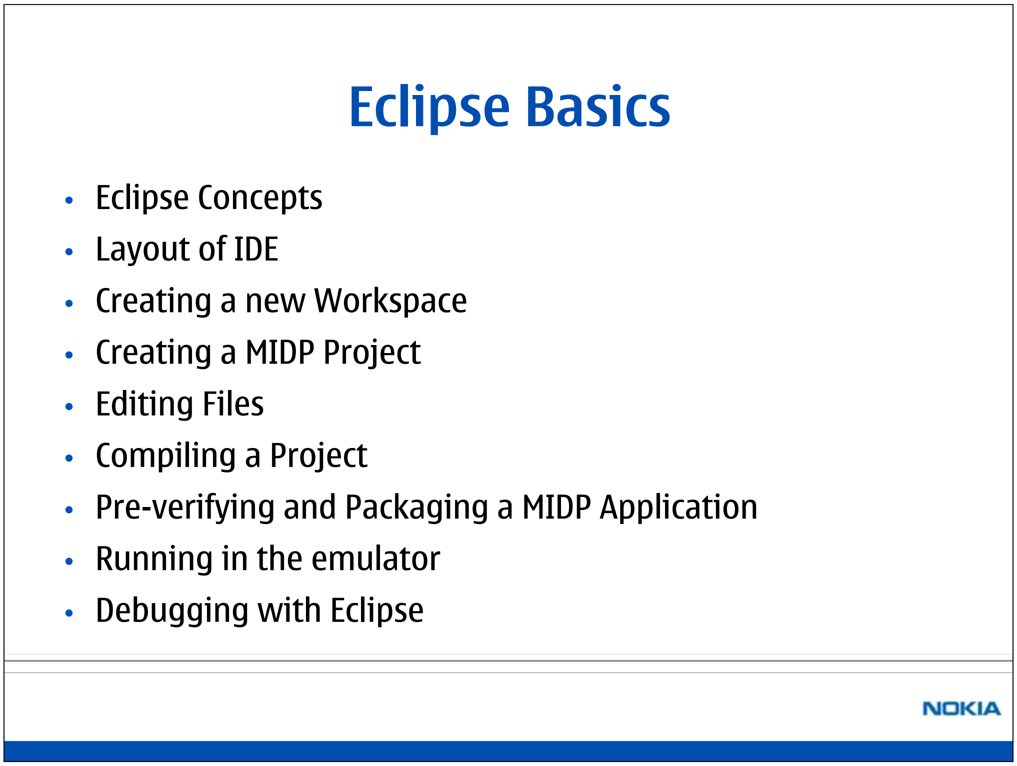


MIDlet descriptor

- Finally, a Java Application Descriptor (JAD) file is required to that the installing device can learn about the MIDlet JAR without installing it
- Information such as JAR size, JAR location, and MIDlet name is included in the format:

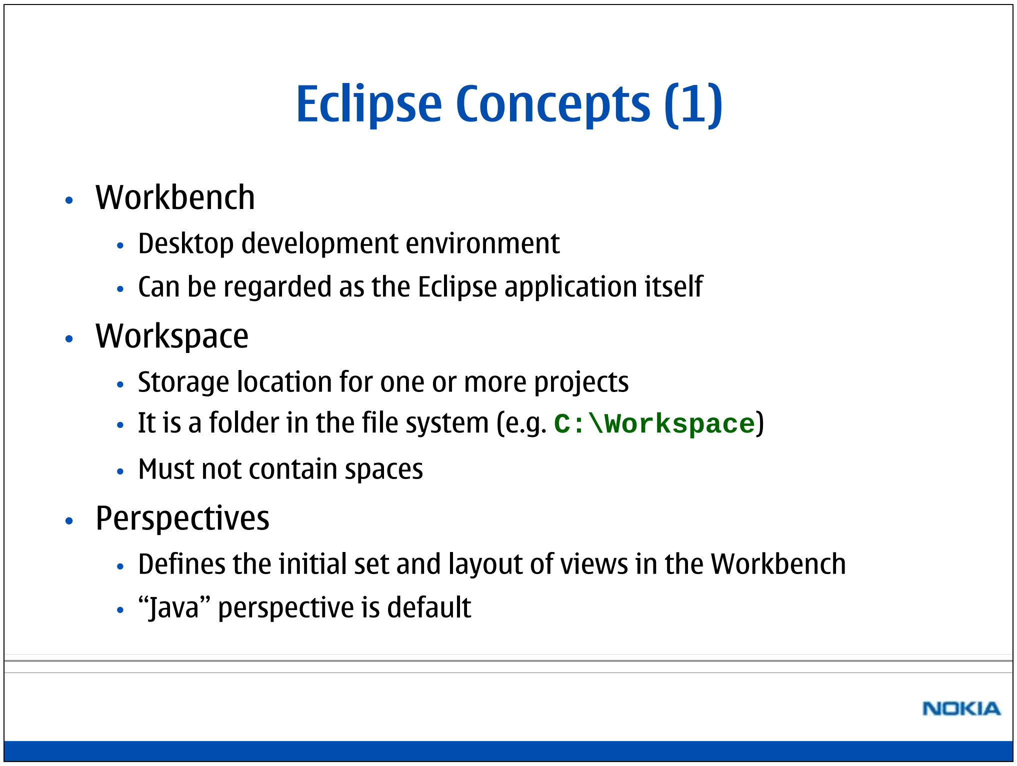
```
MIDlet-1: SpaceMission, , com.nokia.example2dgame.GameMIDlet
MIDlet-Jar-Size: 71684
MIDlet-Jar-URL: SpaceMission.jar
MIDlet-Name: SpaceMission
MIDlet-Vendor: Nokia
MIDlet-Version: 1.0
```

- Again, the creation of this file can be automated by using Carbide.j



Eclipse Basics

- Eclipse Concepts
- Layout of IDE
- Creating a new Workspace
- Creating a MIDP Project
- Editing Files
- Compiling a Project
- Pre-verifying and Packaging a MIDP Application
- Running in the emulator
- Debugging with Eclipse



Eclipse Concepts (1)

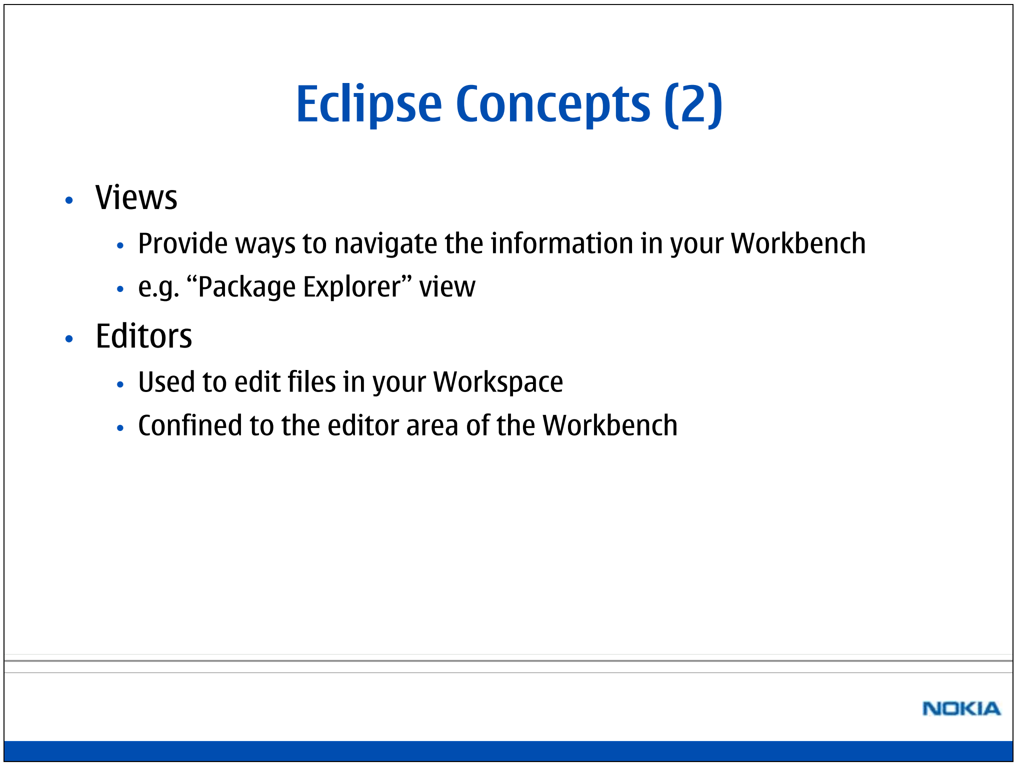
- Workbench
  - Desktop development environment
  - Can be regarded as the Eclipse application itself
- Workspace
  - Storage location for one or more projects
  - It is a folder in the file system (e.g. `C:\Workspace`)
  - Must not contain spaces
- Perspectives
  - Defines the initial set and layout of views in the Workbench
  - “Java” perspective is default

The common concepts associated with Eclipse are discussed below.

In the Eclipse Help documentation, the Workbench is referred to as the desktop development environment. For the purposes of this course it can be regarded as the Eclipse application itself.

The projects, folders and files that you create with the Workbench are all stored under a single directory that represents your Workspace.

Each Workbench window contains one or more perspectives. Perspectives define the initial set and layout of views in the Workbench and control what appears in certain menus and tool bars. More than one Workbench window can exist on the desktop at any given time. The default perspective is “Java” and should be used for all development work.



Eclipse Concepts (2)

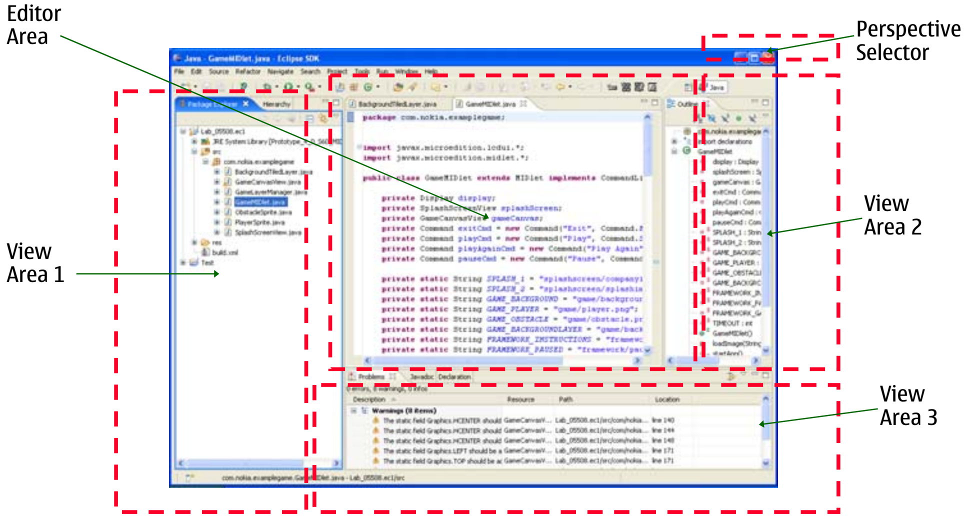
- Views
  - Provide ways to navigate the information in your Workbench
  - e.g. “Package Explorer” view
- Editors
  - Used to edit files in your Workspace
  - Confined to the editor area of the Workbench

Views are UI components that provide ways to navigate the information in your Workbench. For example, the “Package Explorer” view displays the projects located in the current workspace using a tree structure. Top level nodes are ‘project’ nodes that can contain child ‘folder’ nodes. These nodes themselves can contain child ‘file’ nodes.

Editors are used to edit files in your workspace. They are another UI component, but unlike Views are associated with a particular file and are confined to the editor area of the Workbench. Editors identify the file they are modifying by a tab containing the name of the file.

See the Eclipse Help documentation for more information on any of these topics including specific information about each of the views. Select the “Help -> Help Contents” menu item to open the help window. Once this window is open, type a search string in the top left search field and click the Go button.

Layout of IDE



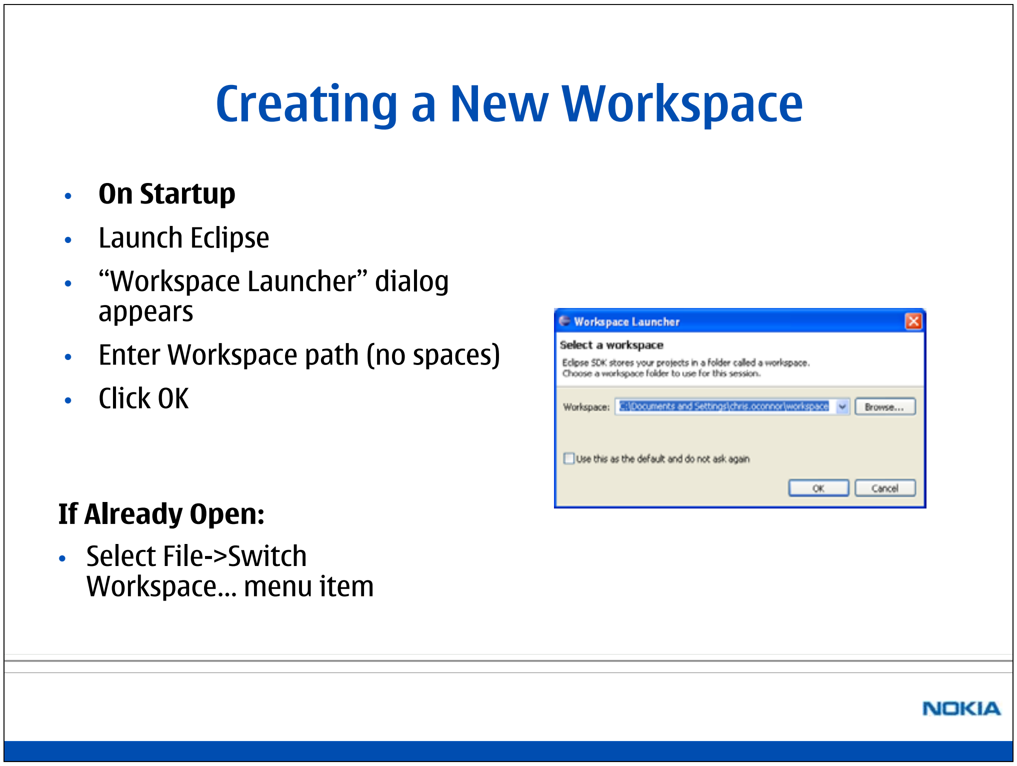
The above slide illustrates the default layout of the Eclipse IDE (for the Java perspective).

In the top right corner of the IDE a control exists to display the current perspective and allow the selection of other perspectives.

The editor area appears in the centre of the IDE and is surrounded by view windows (although in the slide it is only surrounded on three sides).

The default view layout is shown in the above slide with three view areas. However, in practise you can have any number of view areas. View windows may be moved from one view area to another or even into a completely new view area by dragging it's title bar with the mouse into another area of the IDE. Views may also be detached rather than appearing in a specific view area. In such cases the view has its own window which is independent from the main Eclipse IDE window.

To reset the initial layout of your current perspective select the "Window -> Reset Perspective" menu item.



Creating a New Workspace

- **On Startup**
- Launch Eclipse
- “Workspace Launcher” dialog appears
- Enter Workspace path (no spaces)
- Click OK

If Already Open:

- Select File->Switch Workspace... menu item

Select a workspace

Eclipse SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Documents and Settings\chris.oconnor\workspace

□ Use this as the default and do not ask again

The first task in using Eclipse is to select a workspace. As mentioned before, your workspace is a single directory under which all the projects, folders and files that you create with Eclipse are all stored.

When you open Eclipse for the first time you will be presented with the “Workspace Launcher” dialog. You can choose the workspace folder in one of three ways:

- ***Type name of the folder (including its path) into the Workspace field.***
- ***Select the Browse button and browse to the folder of your choosing.***
- ***Select (an existing) workspace from the drop down menu.***

A new workspace is simply a folder that has not previously been used as a Eclipse workspace and can be selected by either of the first two methods.

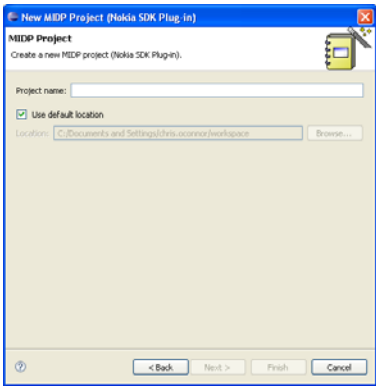
An existing workspace is a folder that has previously been used as a Eclipse workspace and can be selected by any of the above methods.

Note that the workspace folder (including its path) must not contain any spaces. Once you are happy with your choice click the OK button. After a brief initialization period the Eclipse window will be displayed.

You may switch between different workspaces at any time using the “File -> Switch Workspace...“ menu item.

Creating a MIDP Project (1)

- Use built in wizard to create new MIDP Application
- Select “File-> New Project” menu item
- Select “MIDP Project (Nokia SDK Plug-in)”
- Click Next
- “MIDP Project” dialog appears
- Enter Project Name
- Click Next



New MIDP Project (Nokia SDK Plug-in)

MIDP Project

Create a new MIDP project (Nokia SDK Plug-in).

Project name:

☑ Use default location

Location: C:/Documents and Settings/chris.oconnor/workspace Browse...

NOKIA

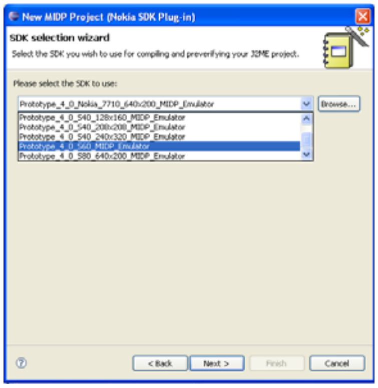
To create a new Eclipse MIDP project, one of the application creation wizards should be used. These wizards generate the code and settings, for a simple application, that can be used as a starting point for writing your own application.

This slide, and the following ones, illustrate how to create a new Eclipse project using the “MIDP Project (Nokia SDK Plug-in)” wizard.

First select the “File -> New Project” menu item. Under the Java folder, select “MIDP Project (Nokia SDK Plug-in)” to display the wizard dialog. The first screen of this wizard allows you to select a name for your project. Type a name in the “Project name” field and click the Next button.

Creating a MIDP Project (2)

- Select a SDK
- Click Next



New MIDP Project (Nokia SDK Plug-in)

SDK selection wizard

Select the SDK you wish to use for compiling and preverifying your J2ME project.

Please select the SDK to use:

Prototype_4_0_Nokia_7710_640x200_MIDP_Emulator

Prototype_4_0_S40_128x160_MIDP_Emulator
Prototype_4_0_S40_208x208_MIDP_Emulator
Prototype_4_0_S40_240x320_MIDP_Emulator
Prototype_4_0_S60_MIDP_Emulator
Prototype_4_0_S80_640x200_MIDP_Emulator

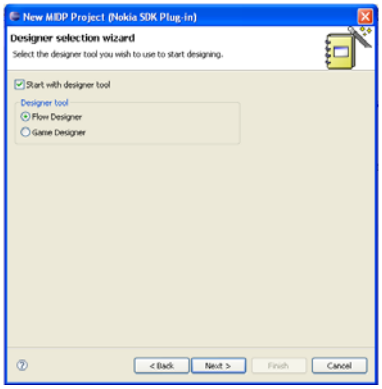
NOKIA

The next screen of the wizard allows you to choose an SDK for your application. A simple explanation is given in the description pane (see slide) for the template currently selected in the list.

To create a project the S60 Platform, select the "Prototype_X_x_S60_MIDP_Emulator" SDK and click the Next button.

Creating a MIDP Project (3)

- To use the “Designer” tool to create your MIDlet, select the “Start with designer tool” tickbox
- Otherwise, deselect this
- Click Next



New MIDP Project (Nokia SDK Plug-in)

Designer selection wizard

Select the designer tool you wish to use to start designing.

☑ Start with designer tool

Designer tool

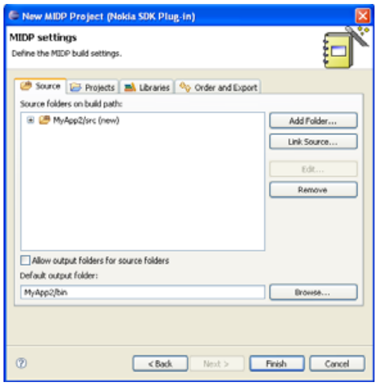
● Flow Designer

○ Game Designer

NOKIA

Creating a MIDP Project (4)

- Click Finish to complete the "New MIDP Project" creation



New MIDP Project (Nokia SDK Plug-in)

MIDP settings

Define the MIDP build settings.

Source folders on build path:

MyApp2/src (new)

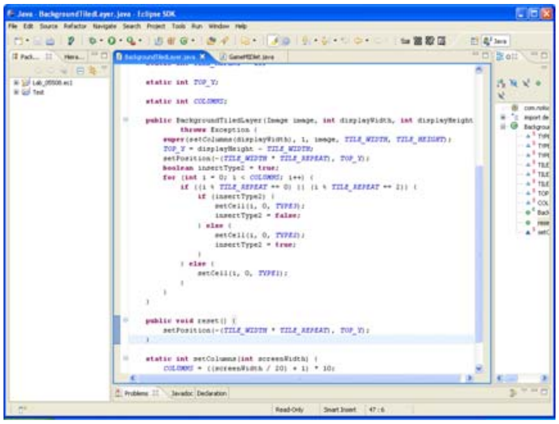
□ Allow output folders for source folders

Default output folder: MyApp2/bin

NOKIA

Editing Files

- Double click source file node in “Project Explorer” view
- A new editor window opens in the editor area containing file contents
- Edit file “as usual”

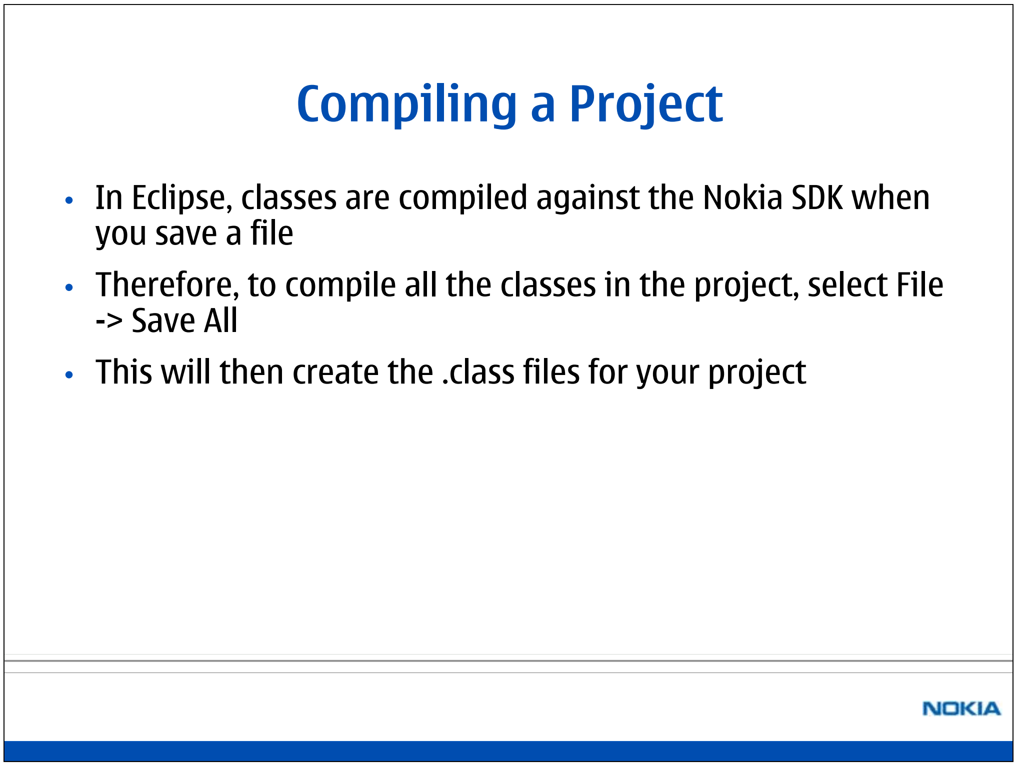


```
    static int TOP_Y;

    static int COLUMNS;

    public BackgroundTiledLayer(Image image, int displayWidth, int displayHeight
            throws Exception {
        super(setColumns(displayWidth), 1, image, TILE_WIDTH, TILE_HEIGHT);
        TOP_Y = displayHeight - TILE_WIDTH;
        setPosition(-(TILE_WIDTH * TILE_REPEAT), TOP_Y);
        boolean insertType2 = true;
        for (int i = 0; i < COLUMNS; i++) {
            if ((i % TILE_REPEAT == 0) || (i % TILE_REPEAT == 2)) {
                if (insertType2) {
                    setCell(i, 0, TYPE3);
                    insertType2 = false;
                } else {
                    setCell(i, 0, TYPE2);
                    insertType2 = true;
                }
            } else {
                setCell(i, 0, TYPE1);
            }
        }
    }

    public void reset() {
        setPosition(-(TILE_WIDTH * TILE_REPEAT), TOP_Y);
    }

    static int setColumns(int screenWidth) {
        COLUMNS = ((screenWidth / 20) + 1) * 10;
```

Java - BackgroundTiledLayer.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Tools Run Window Help
BackgroundTiledLayer.java
GameMIDlet.java
Problems
Javadoc
Declaration
Read-Only
Smart Insert
47 : 6

NOKIA

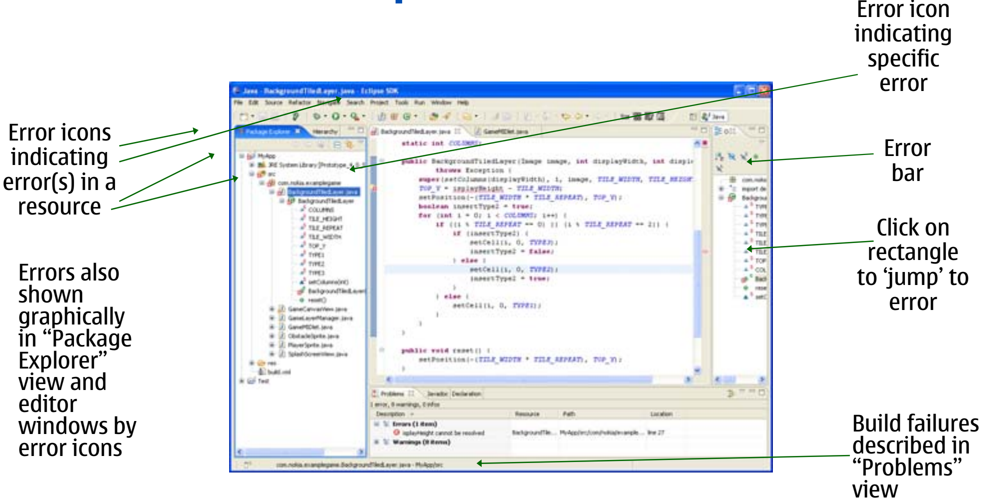
Editing of files in Eclipse is very straightforward. In the “Package Explorer” view, double click on a source file node that you want to edit. A new editor window opens in the editor area containing the file contents. Edit the file as “usual”; i.e. as you would using any other standard text editor.



Compiling a Project

- In Eclipse, classes are compiled against the Nokia SDK when you save a file
- Therefore, to compile all the classes in the project, select File -> Save All
- This will then create the .class files for your project

Compilation Errors



```
static int COLUMNS;

public BackgroundTiledLayer(Image image, int displayWidth, int displ
        throws Exception {
    super(setColumns(displayWidth), 1, image, TILE_WIDTH, TILE_HEIGH
    TOP_Y = isplayHeight - TILE_WIDTH;
    setPosition(-(TILE_WIDTH * TILE_REPEAT), TOP_Y);
    boolean insertType2 = true;
    for (int i = 0; i < COLUMNS; i++) {
        if ((i % TILE_REPEAT == 0) || (i % TILE_REPEAT == 2)) {
            if (insertType2) {
                setCell(i, 0, TYPE3);
                insertType2 = false;
            } else {
                setCell(i, 0, TYPE2);
                insertType2 = true;
            }
        } else {
            setCell(i, 0, TYPE1);
        }
    }
}

public void reset() {
    setPosition(-(TILE_WIDTH * TILE_REPEAT), TOP_Y);
}
```

Description	Resource	Path	Location
Errors (1 item)			
isplayHeight cannot be resolved	BackgroundTile...	MyApp/src/com/nokia/example...	line 27
Warnings (0 items)			

Error icons indicating error(s) in a resource

Errors also shown graphically in "Package Explorer" view and editor windows by error icons

Error icon indicating specific error

Error bar

Click on rectangle to 'jump' to error

Build failures described in "Problems" view

If you have compilation failures in your project then you must fix them. The errors will all be described in the log that was output to the Problems view during the compilation. This will tell you the file and line number the error occurred on along with a reason for the error.

Errors are also indicated, and can be located, using the graphical error icons. These are used to indicate whether an error is present in a particular resource.

To show how to track an error, using this graphical approach, let us look at the example on the slide. First look at the "MyApp" project node in the "Package Explorer" view. This has an error icon associated with it so there are error(s) in the project.

Expand the "MyApp" project node to view its child nodes. At least one of these nodes will have an associated error icon indicating there are error(s) associated with that resource. The "src" folder node is the resource with the associated error icon and so contains the error(s).

Expand the "src" folder node to view its child nodes. Like before at least one of these nodes will have an associated error icon indicating there are error(s) associated with that resource. In this case, it is the BackgroundTiledLayer.java source-file node that contains the error(s).

You can repeat this procedure and expand the BackgroundTiledLayer.java source-file node to find which method(s) the error(s) occurred in. In any event, the next stage is to open the source file BackgroundTiledLayer.java by double clicking on the source-file node or a method node with an associated error icon. This will open an editor window containing the contents of the file.

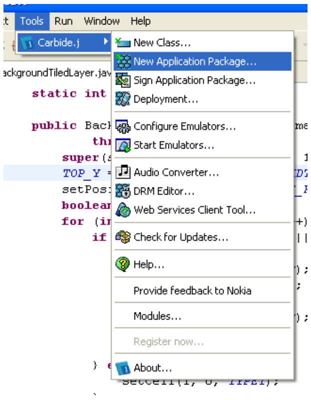
The error bar is the area immediately to the right of an editor window's vertical scroll bar. It works in conjunction with the scroll bar to show the location of all the errors in the file. Each error is represented by a red rectangle. Clicking on a rectangle causes the editor window to jump to the part of the file where the error is located.

The editor window also displays an error icon in the marker bar for each line of the file on which an error occurs. The marker bar is located to the left hand side of the editor window. In addition the line itself on which the error occurs is underlined in red.

If you move the mouse pointer over an error icon in the marker bar or an error rectangle in the error bar, the reason for the error (as described in the build log) will be shown as a tooltip.

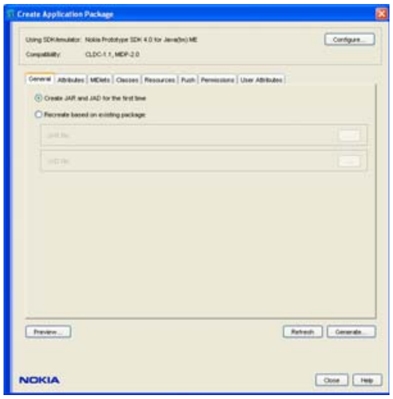
Pre-verifying and Packaging a MIDP Application (1)

- MIDP Projects can be pre-verified and packed into a JAD and JAR file by using Carbide.j
- Do this by selecting the project from the “Package Explorer” and selecting Tools -> Carbide.j -> New Application Package



Pre-verifying and Packaging a MIDP Application (2)

- Select to Create a new JAR and JAD
- Click Generate
- This will then pre-verify and package your MIDP Application
- A JAR and JAD file will be added to your project

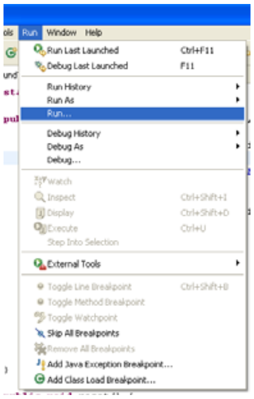


Using SDK/emulator: Nokia Prototype SDK 4.0 for Java(tm) ME
Compatibility: CLDC-1.1, MIDP-2.0

● Create JAR and JAD for the first time
○ Recreate based on existing package

Running in the emulator (1)

- A “Configuration” needs to be created to run the application in the emulator
- Select the project name in the “Package Explorer”
- Select “Run -> Run...”



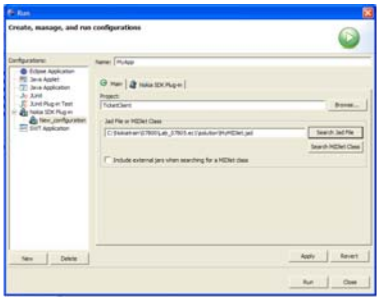
NOKIA

Once you have fixed all your errors and built your project successfully you can now test your application out. The simplest way to test your application is to use the emulator that comes with an SDK. There are two choices available:

1. ***Run the application in the emulator. Select the “Run -> Run...” menu item. This is to create a configuration to start the emulator will launch without the debugger.***
2. ***Debug the application in the emulator. select the “Run -> Debug...” menu item. This is to select the configuration to start the emulator and the debugger.***

Running in the emulator (2)

- Create a new configuration by selecting “Nokia SDK Plug-in”, click the “New” button
- Type a name for the configuration
- Press the “Search Jad file” and select the JAD file
- Ensure the correct emulator is selected by clicking on the “Nokia SDK Plug-in” Tab
- Once this configuration has been created click the “Run” button
- The emulator will launch **without** the debugger



Name: MyApp

Project: TicketClient

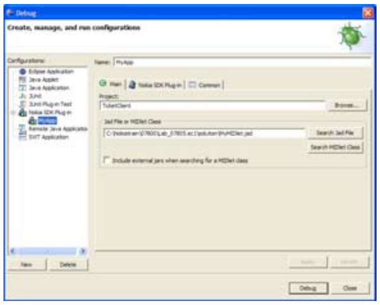
Jad File or MIDlet Class: [illegible]

□ Include external jars when searching for a MIDlet class

NOKIA

Running in the emulator in Debug mode

- Select Run -> Debug..
- Click the configuration name of the application you want to debug
- Click the Debug button
- The emulator will launch and the debugger will start



Name: MyApp

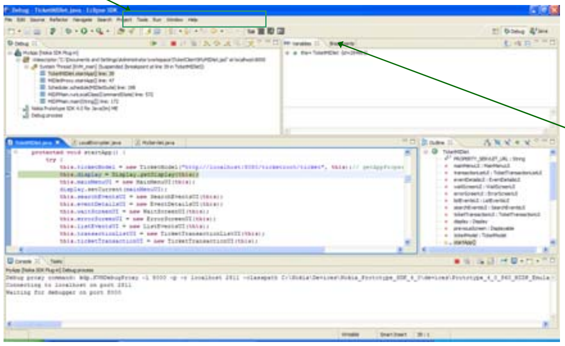
Project: TicketClient

Jad File or MIDlet Class: C:\[illegible]\MyMIDlet.jad

☐ Include external jars when searching for a MIDlet class

NOKIA

Debugging with Eclipse



Use “Debug” view toolbar to step through code

Suspended application thread

Call stack shown by child nodes of suspended thread

Double click in margin to set/remove breakpoints

Editor file highlights location of next line of code to execute

IDE switches to “Debug” perspective whilst debugging

“Variables” view displays descriptor contents

NOKIA

Once the debugger has started, Eclipse automatically switches to the Debug perspective. This perspective displays views that are useful when debugging an application.

In order to debug code you must place a breakpoint somewhere in your code. This is done by double clicking in the marker bar (to the immediate left of the editor window) alongside the line of code where you want to place the breakpoint. A blue circle appears to indicate that a breakpoint has been set. To remove a breakpoint double click on the breakpoint icon in the marker bar.

When the breakpoint you set has been reached by the execution flow, the debugger will halt execution of your application and focus will switch from the emulator to the Eclipse IDE. The line of code that is next in line to be executed is highlighted in green in the editor window.

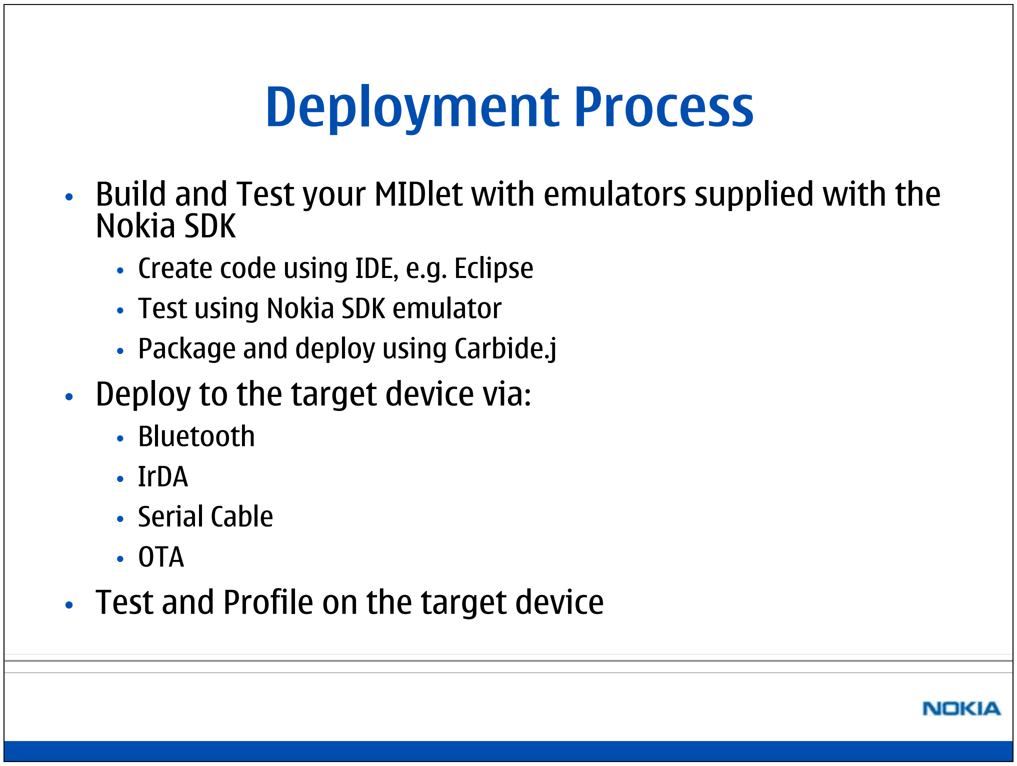
When the debugger halted execution of your application, it suspended your application’s thread of execution. This is shown in the Debug view which contains a list of all the Java threads running under the emulator. One of the thread nodes in the list is marked “Suspended”; this is your application’s thread.

Under your application’s thread node there are a number of child nodes which form the call stack for the thread. Each child node represents a stack frame in the call stack.

The Debug view also has a debugging toolbar that provides a number of useful functions including:

- *Resuming thread execution.*
- *Stepping through the code.*
- *Terminating the debug session.*

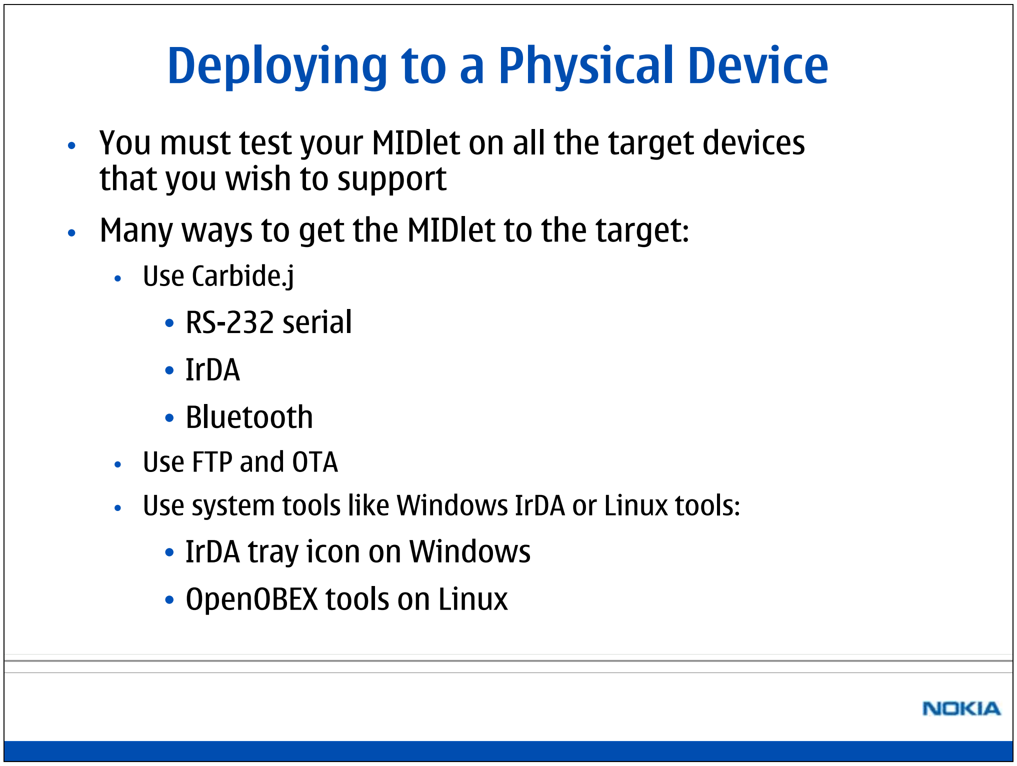
Note that when you finally terminate your debug session, the Eclipse IDE will not automatically switch back to the Java perspective. You have to do this task manually.



Deployment Process

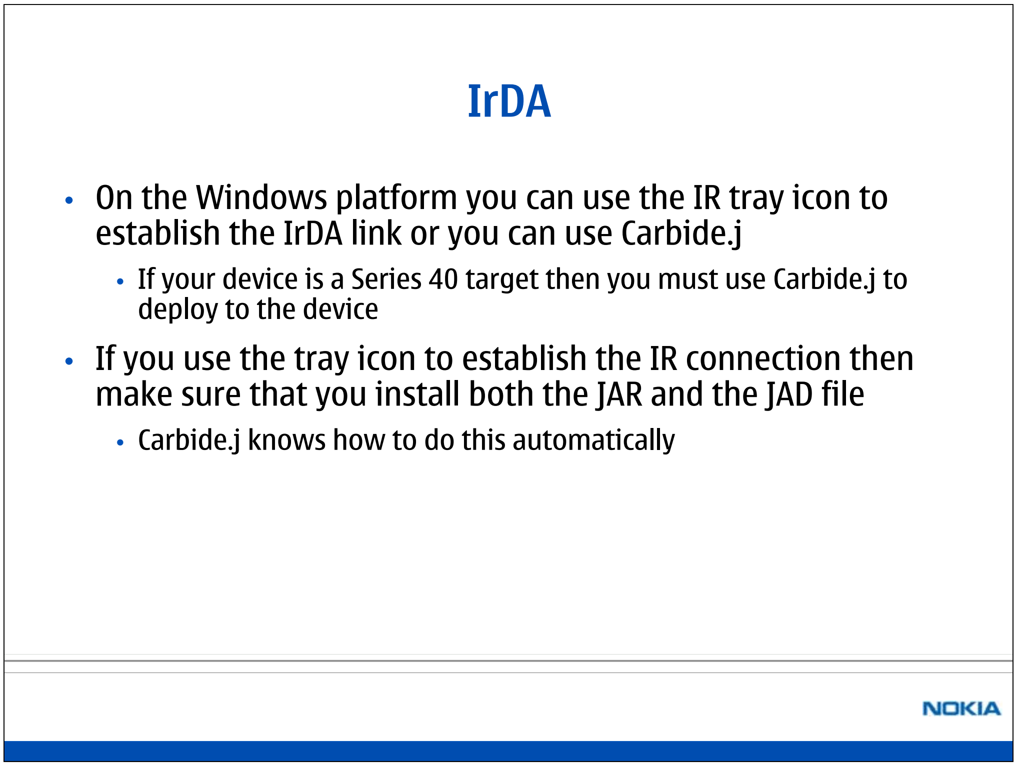
- Build and Test your MIDlet with emulators supplied with the Nokia SDK
  - Create code using IDE, e.g. Eclipse
  - Test using Nokia SDK emulator
  - Package and deploy using Carbide.j
- Deploy to the target device via:
  - Bluetooth
  - IrDA
  - Serial Cable
  - OTA
- Test and Profile on the target device

NOKIA



Deploying to a Physical Device

- You must test your MIDlet on all the target devices that you wish to support
- Many ways to get the MIDlet to the target:
  - Use Carbide.j
    - RS-232 serial
    - IrDA
    - Bluetooth
  - Use FTP and OTA
  - Use system tools like Windows IrDA or Linux tools:
    - IrDA tray icon on Windows
    - OpenOBEX tools on Linux



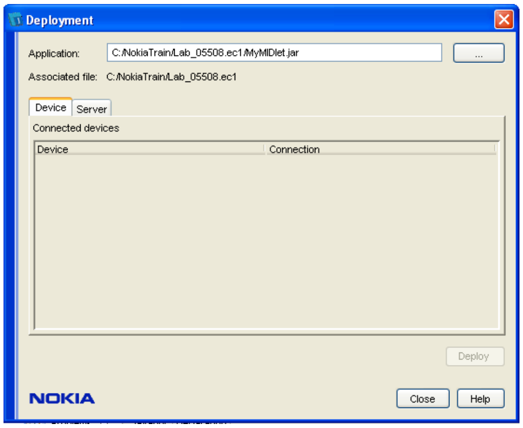
IrDA

- On the Windows platform you can use the IR tray icon to establish the IrDA link or you can use Carbide.j
  - If your device is a Series 40 target then you must use Carbide.j to deploy to the device
- If you use the tray icon to establish the IR connection then make sure that you install both the JAR and the JAD file
  - Carbide.j knows how to do this automatically

If your computer has an IR port, you can point your device to the IR port on your computer and load the MIDlet that way. You can use Carbide.j to do this or if you are programming to a S60 device then it's a simple matter of clicking the tray icon for the IR port and making sure that your target device's IR port is turned on. Similarly, on Linux you can use the OpenOBEX package and the command line tool irobex_plam3 which works fine too.

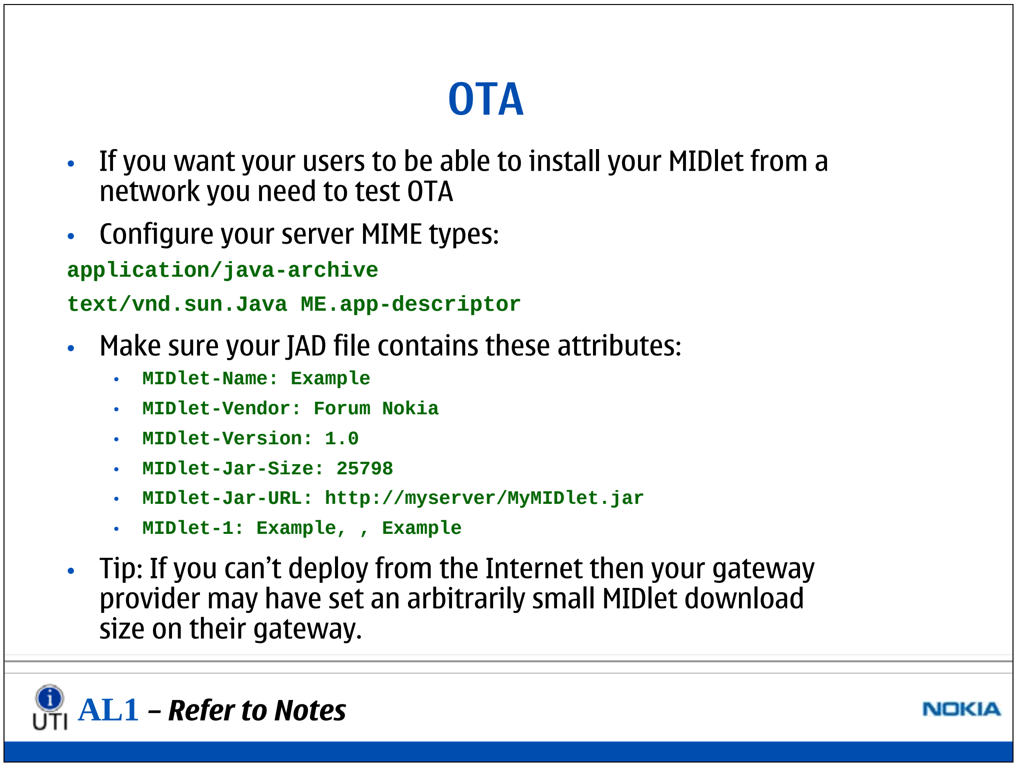
Serial Cable

- The Cabide.j functions as a deployment tool via RS-232



Application: C:/NokiaTrain/Lab_05508.ec1/MyMIDlet.jar

Associated file: C:/NokiaTrain/Lab_05508.ec1



OTA

- If you want your users to be able to install your MIDlet from a network you need to test OTA
- Configure your server MIME types:

```
application/java-archive
text/vnd.sun.Java ME.app-descriptor
```

- Make sure your JAD file contains these attributes:
  - `MIDlet-Name: Example`
  - `MIDlet-Vendor: Forum Nokia`
  - `MIDlet-Version: 1.0`
  - `MIDlet-Jar-Size: 25798`
  - `MIDlet-Jar-URL: http://myserver/MyMIDlet.jar`
  - `MIDlet-1: Example, , Example`
- Tip: If you can't deploy from the Internet then your gateway provider may have set an arbitrarily small MIDlet download size on their gateway.

UTI **AL1** – ***Refer to Notes***

Remember that each Java ME enabled device has a Java Application Manager (JAM) that will handle the installation and the JAM uses the information in the JAD file to control its operations.

To ensure that an application can be installed via OTA, it must include specific attributes in its JAD and manifest file:

- ***MIDlet-Name: The name of the MIDlet suite. The value of this attribute must match the value of the same attribute contained within the manifest file of the downloaded JAR.***
- ***MIDlet-Vendor: The vendor of the MIDlet suite. The value of this attribute must match the value of the same attribute contained within the manifest file of the downloaded JAR.***
- ***MIDlet-Version: This attribute is used to check if there is a newer version of the application available. If an older version of the MIDlet is already installed, notification about updating can then be provided. The value of the attribute must match the value of the same attribute contained within the manifest file of the downloaded JAR.***
- ***MIDlet-Jar-Size: Since the JAM has to check the amount of free space and memory available on a device to ensure an application can be installed, the attribute MIDlet-Jar-Size must be included in the JAD file. However, the actual memory required may be larger or smaller depending on the device, and so this value is only actually used as a hint to the JAM. This value is also used to check that the received JAR file size matches the size specified in the JAD.***
- ***MIDlet-Jar-URL: To be able to locate the JAR file for the MIDlet suite, the JAM uses the MIDlet-Jar-URL attribute.***
- ***MIDlet-1: Name, icon and main class of MIDlet***

Note. A simple way to set up the MIDlet-JAR-URL attribute is to keep the JAR file and the JAD file in the same directory, that way you only need the name of the JAR file for the MIDlet-JAR-URL entry like: MIDlet-Jar-URL: mymidlet.jar

Finally, the default install folder is the Applications folder. If, for example, if you are deploying a game you can control where to install the MIDlet by adding an entry like this to your JAD: Nokia-MIDlet-Category: Game

This will install the MIDlet in the Game folder on the target.

Now, in order to successfully test out OTA provisioning -- which will most likely be the method your end-users use to install MIDlets -- you'll need to set the web server MIME types correctly for your web server. The following MIME types should be set:

.jad text/vnd.sun.Java ME.app-descriptor

.jar application/java-archive

Again, you'll probably need help from your System Administrator for this part since to do this usually requires root privileges.

Now, you are ready to test on your target!

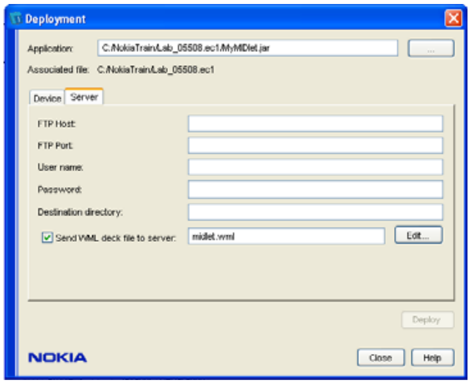
Note: A few words on JAR size. Many WAP gateways have (very) small maximum file size for the JAR -- sometimes as small as 5K -- so you may have to either get your network operator to modify the gateway size or find another network if your JARs are big!

AL1 - The application must install via OTA

This test makes sure that no further configuration is required to allow the application to install correctly OTA. As the slide explains, to ensure the application installs via OTA, your application must have the following attributes in the JAD file: MIDlet-Name, MIDlet-Vendor, MIDlet-

OTA with Carbide.j

- Carbide.j can automate deploying MIDlets to the web server via FTP



Deployment

Application: C:/NokiaTrain/Lab_05508.ec1/MyMIDlet.jar

Associated file: C:/NokiaTrain/Lab_05508.ec1

FTP Host:

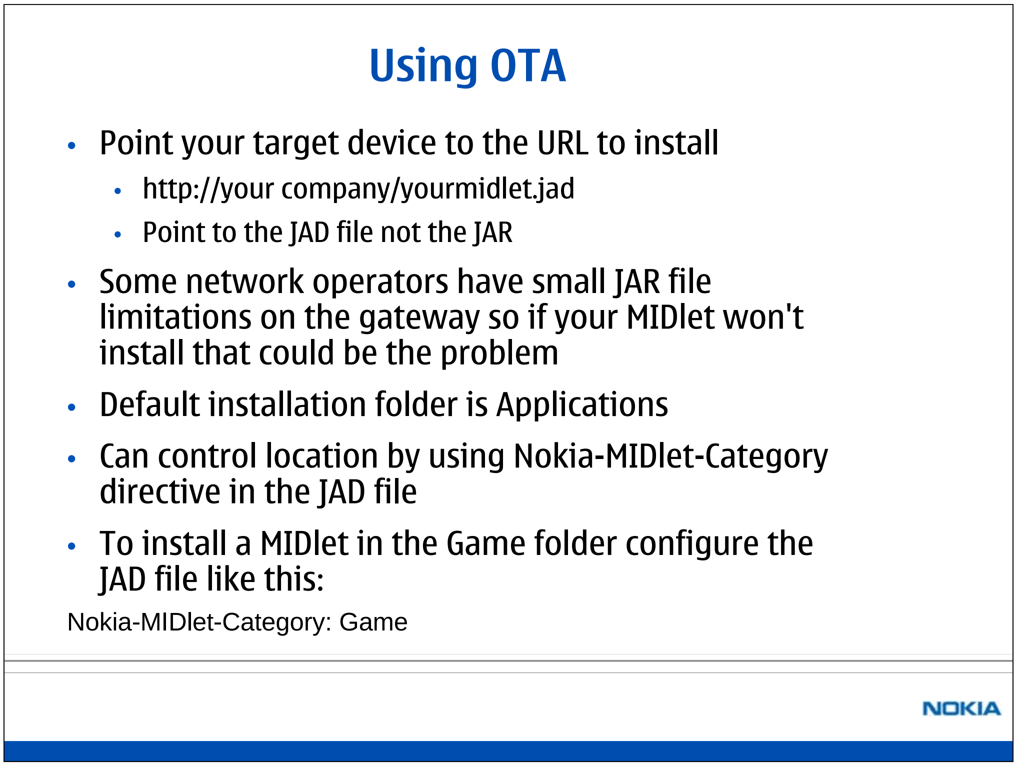
FTP Port:

User name:

Password:

Destination directory:

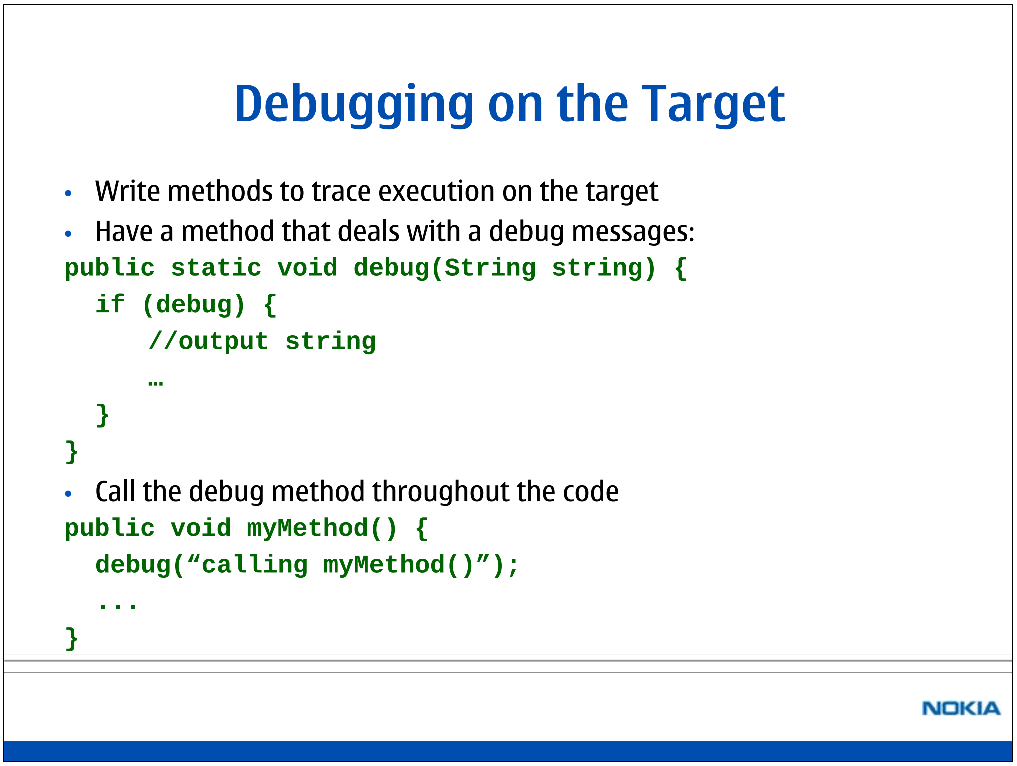
☑ Send WML deck file to server: midlet.wml



Using OTA

- Point your target device to the URL to install
  - http://your company/yourmidlet.jad
  - Point to the JAD file not the JAR
- Some network operators have small JAR file limitations on the gateway so if your MIDlet won't install that could be the problem
- Default installation folder is Applications
- Can control location by using Nokia-MIDlet-Category directive in the JAD file
- To install a MIDlet in the Game folder configure the JAD file like this:

```
Nokia-MIDlet-Category: Game
```



Debugging on the Target

- Write methods to trace execution on the target
- Have a method that deals with a debug messages:

```
public static void debug(String string) {
  if (debug) {
     //output string
     …
  }
}
```

- Call the debug method throughout the code

```
public void myMethod() {
  debug(“calling myMethod()”);
  ...
}
```

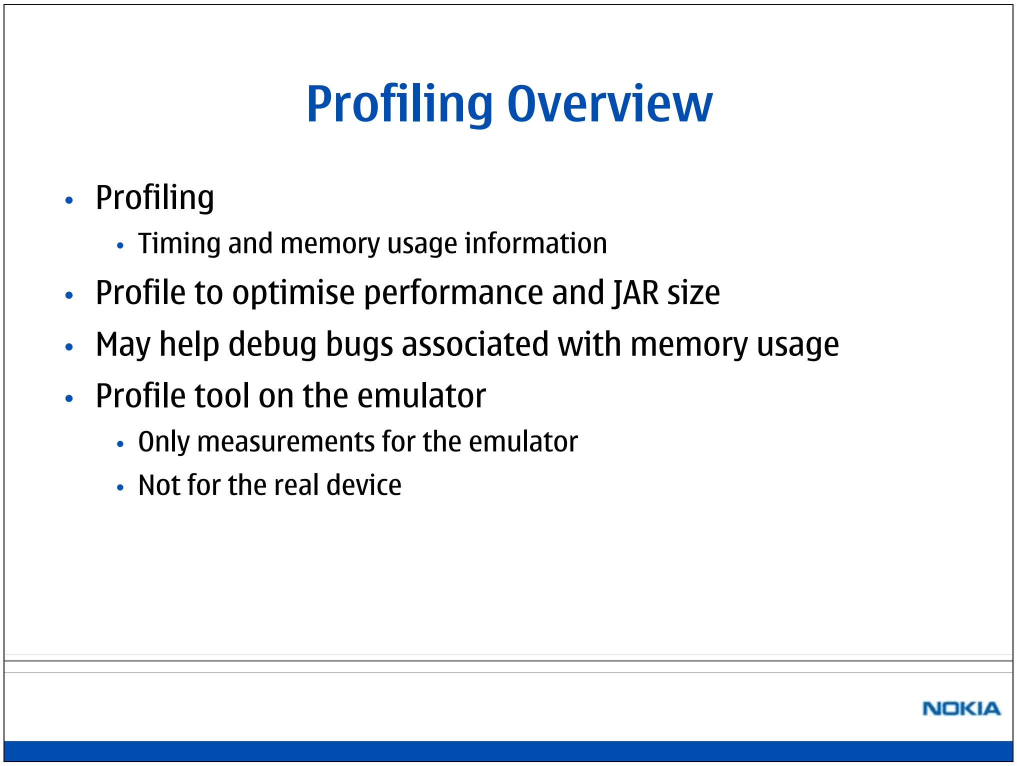
Now, debugging with the emulator can help with logic and design errors, but what happens if you have problems with the MIDlet once it's on the device? One method of doing this is to write test cases and insert code into the MIDlet itself.

Simply add code like:

```
public void myFunc() {
       debug(“calling myFunc”);
       // some useful stuff
}
```

throughout your code and you've got and an easy way to trace the execution on the target to determine where the trouble spots are occurring.

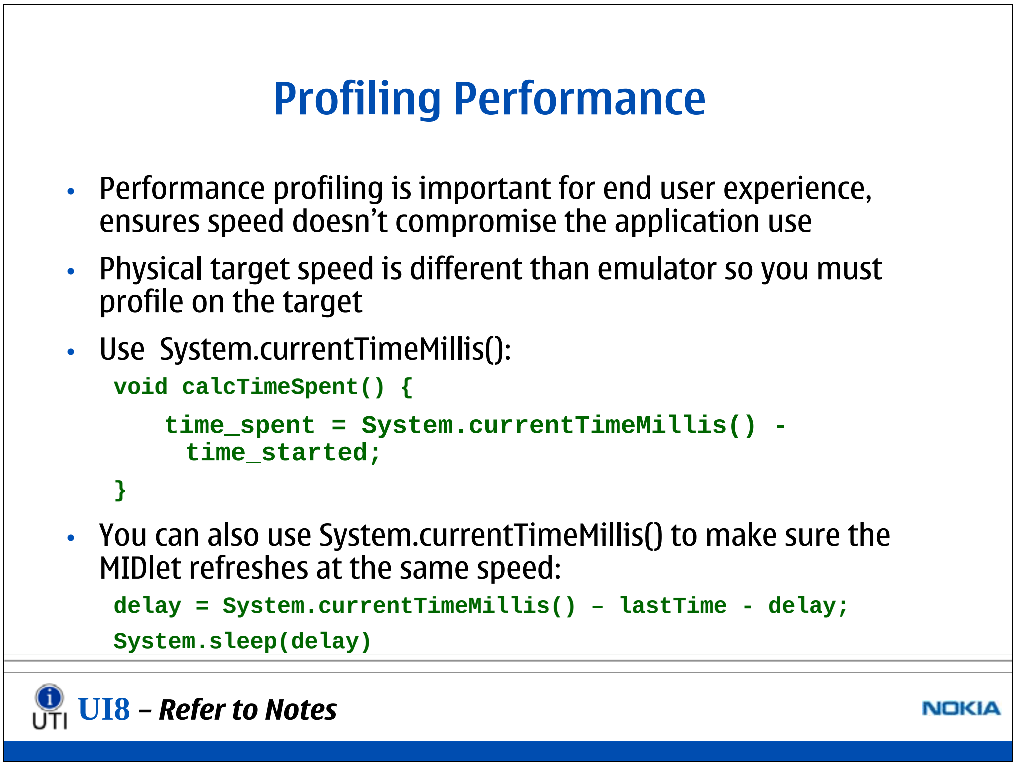
The debug flag can be controlled either as a resource property or even more simply as a static flag that can be turned on or off. Also, you can write a test harness that validates all your assumptions and run these test cases to find out if they pass on the target. Typically called unit testing, you can either role your own test harness or use the Java MEUnit which does the same thing as JUnit which most java programmers are already familiar with.



Profiling Overview

- Profiling
  - Timing and memory usage information
- Profile to optimise performance and JAR size
- May help debug bugs associated with memory usage
- Profile tool on the emulator
  - Only measurements for the emulator
  - Not for the real device

Once your MIDlet is running bug free then you might want to do some profiling to optimize performance and/or JAR size. You may even have to do profiling before you finish debugging, as one of the bugs may in fact have to do with profiling memory usage. One thing to remember is that any profile tool that runs on the emulator can only determine measurements for the emulator and not the device itself. So profiling on the emulator isn't really that useful other than to give you an indication of function call coverage. Real profiling should be done on a real device.



Profiling Performance

- Performance profiling is important for end user experience, ensures speed doesn’t compromise the application use
- Physical target speed is different than emulator so you must profile on the target
- Use System.currentTimeMillis():

```
void calcTimeSpent() {
    time_spent = System.currentTimeMillis() -
     time_started;
}
```

- You can also use System.currentTimeMillis() to make sure the MIDlet refreshes at the same speed:

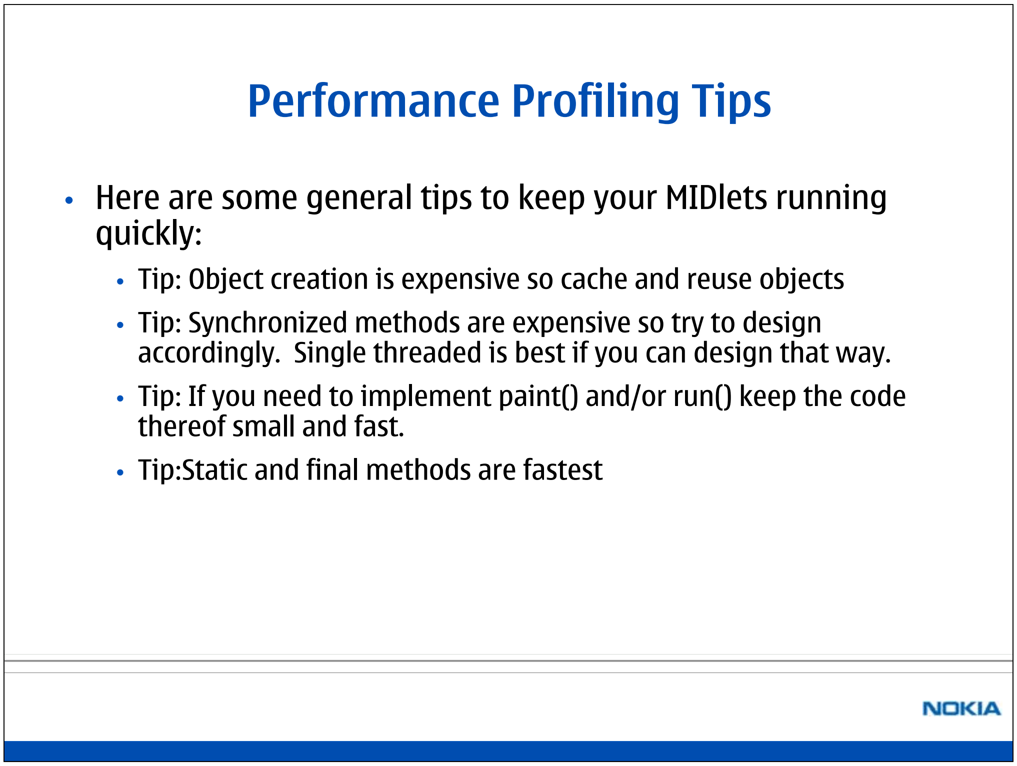
```
delay = System.currentTimeMillis() – lastTime - delay;
System.sleep(delay)
```

UTI **UI8** – ***Refer to Notes***

Performance profiling can be accomplished much the same way as memory profiling. In this case you can use the System.currentTimeMillis() call to find out how fast parts of your program are running.

UI8 - The speed of the application is adequate and it does not compromise the application’s use. The performance of the application is acceptable.

This test makes sure the application is fully useable in real-life situations. As the slide explains, to ensure that the speed of the application is adequate, profile the timing on a real device, rather than on the emulator using System.currentTimeMillis()



Performance Profiling Tips

- Here are some general tips to keep your MIDlets running quickly:
  - Tip: Object creation is expensive so cache and reuse objects
  - Tip: Synchronized methods are expensive so try to design accordingly. Single threaded is best if you can design that way.
  - Tip: If you need to implement paint() and/or run() keep the code thereof small and fast.
  - Tip:Static and final methods are fastest

NOKIA

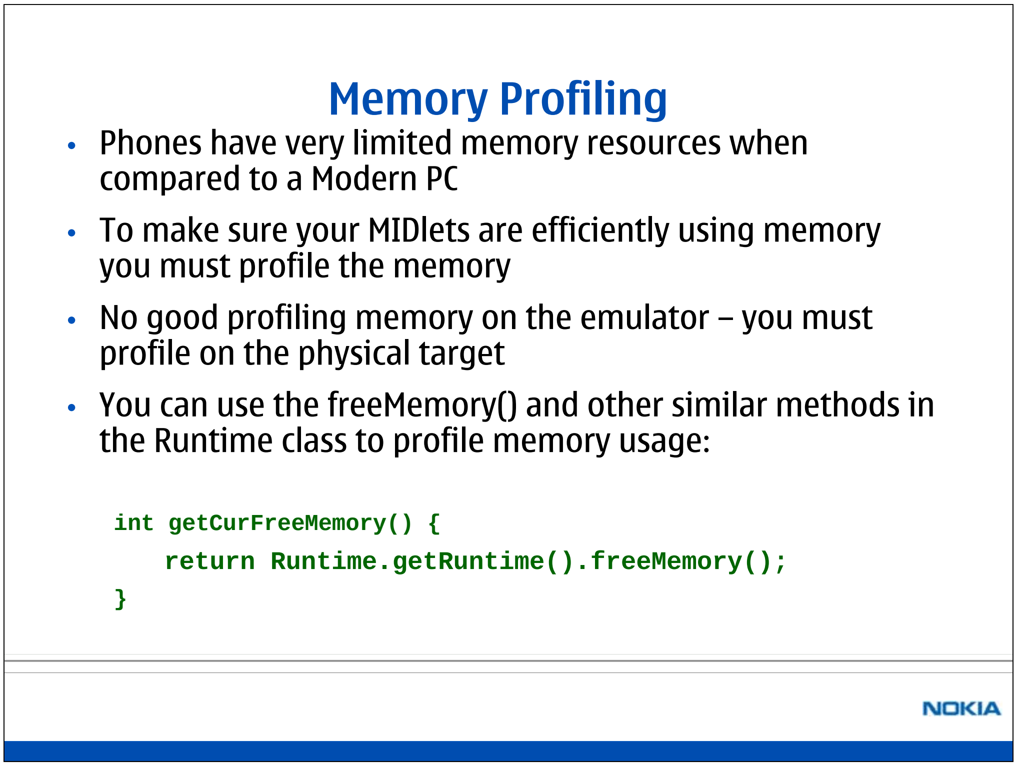
If you remember a few key points about performance your MIDlets will run snappy from the start:

- ***as a rule of thumb about 50% of the execution time is going to be spent in the paint() and run() methods if you implement those methods. So if you can make your MIDlet single threaded and avoid user drawing you'll be better off.***
- ***cache values like crazy when you can.***
- ***the KVM is pretty fast at internal stuff like crunching numbers but slow at making native system calls -- e.g. all the low-level graphics calls dispatch to native calls.***
- ***pre-calculate values***
- ***structure your logic so you don't force repaints too often***
- ***only repaint what's needed***
- ***drawString is unreasonably slow use drawChar with a character array or draw to an offscreen Image***
- ***avoid writing overly elegant OO code***

Some lower-level optimization techniques may also help if you really need the extra performance however be aware that you won't get as much improvements as compared to using better design and smarter algorithms. There are many techniques on low-level optimizations that have been covered countless times and for your curiosity here are a couple of those tips that apply to Java ME:

- ***use the StringBuffer class for concatenation if you're concatenating a lot of Strings.***
- ***declare methods static and final where applicable as they are fastest.***
- ***avoid synchronized methods as they are slowest***
- ***unroll loops (don't worry about code size increasing for unrolled loops as the JAR file compresses these well because it's repetitive code)***
- ***use arithmetic shift for fast math if you can structure the logic using powers of 2***

As you've probably noticed, performance tuning is sometimes at odds with memory usage so you need to decide the right balance -- that's why you're a programmer after all!

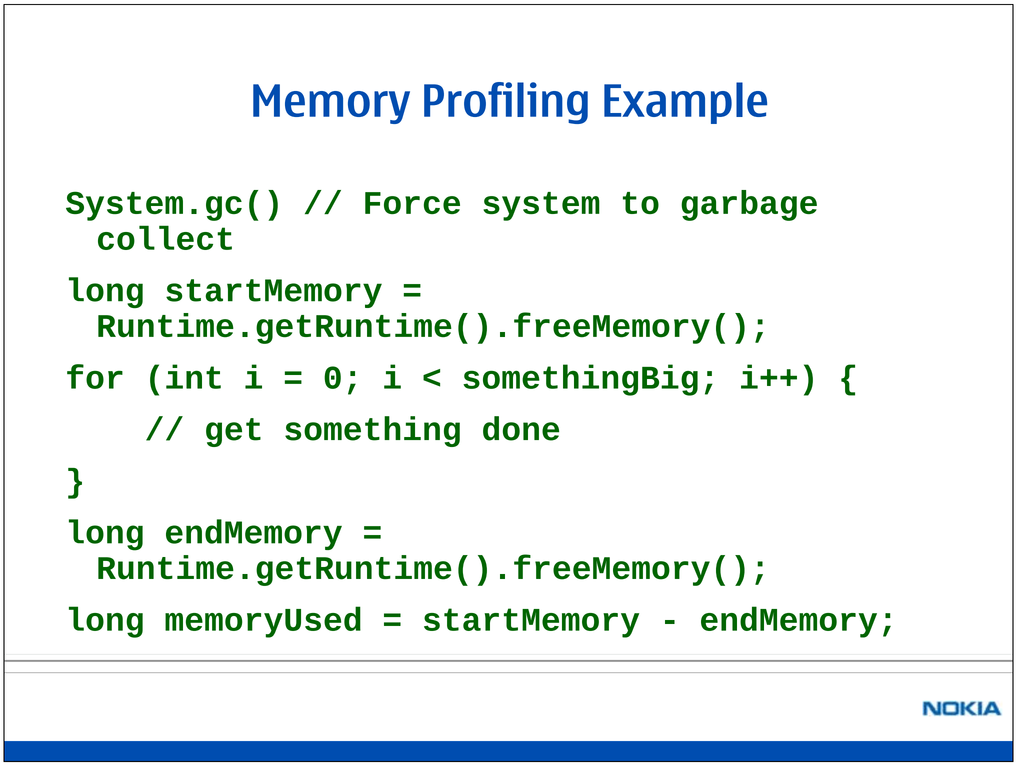


Memory Profiling

- Phones have very limited memory resources when compared to a Modern PC
- To make sure your MIDlets are efficiently using memory you must profile the memory
- No good profiling memory on the emulator – you must profile on the physical target
- You can use the freeMemory() and other similar methods in the Runtime class to profile memory usage:

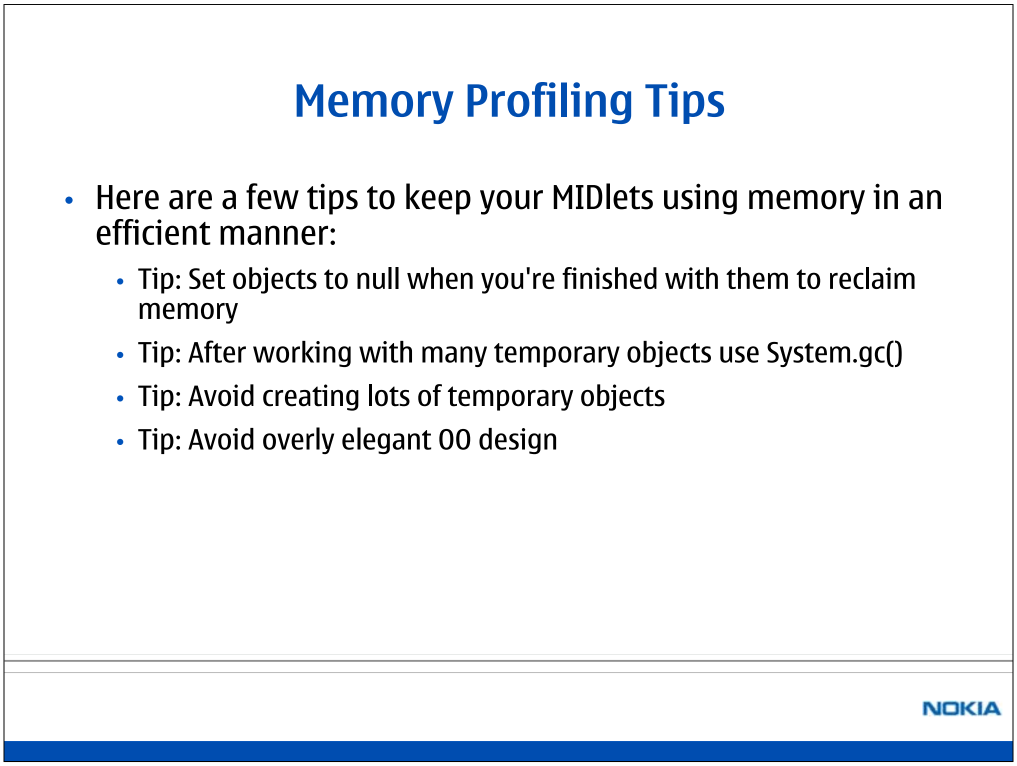
```
int getCurFreeMemory() {
    return Runtime.getRuntime().freeMemory();
}
```

You can do memory profiling quite simply by calling Runtime.freeMemory() (and similar methods that are part of the Runtime class) at junctures in your MIDlet code and then saving and/or outputting those results. We show how to do this method in the accompanying lab.



Memory Profiling Example

```
System.gc() // Force system to garbage
  collect
long startMemory =
  Runtime.getRuntime().freeMemory();
for (int i = 0; i < somethingBig; i++) {
    // get something done
}
long endMemory =
  Runtime.getRuntime().freeMemory();
long memoryUsed = startMemory - endMemory;
```

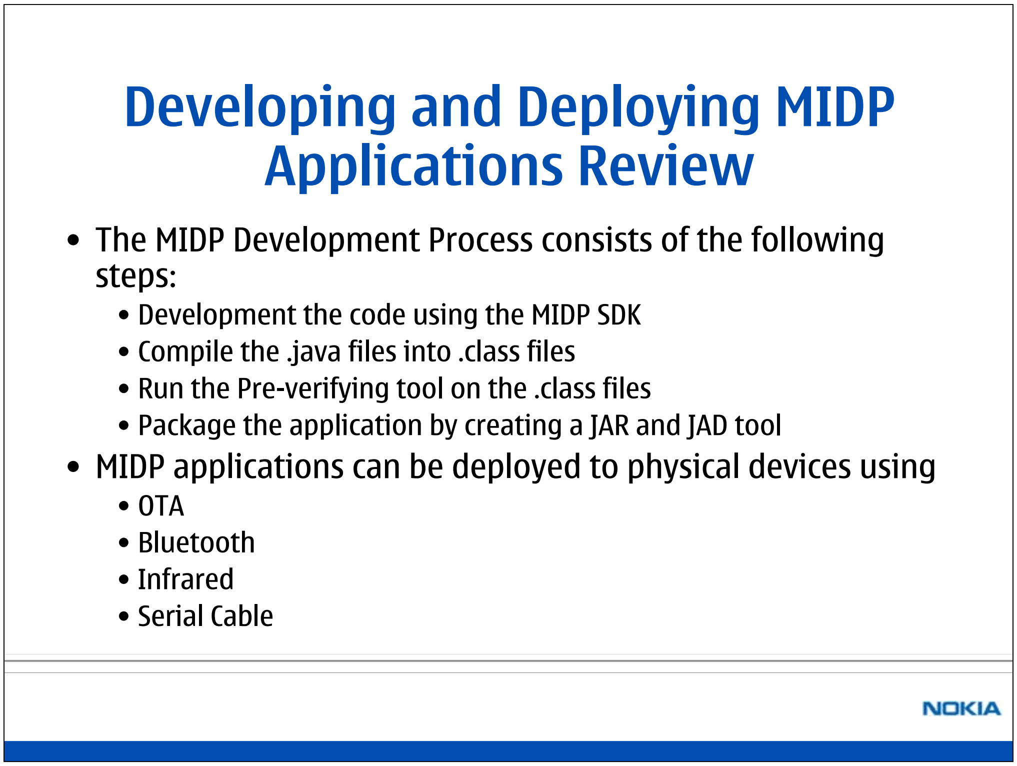


Memory Profiling Tips

- Here are a few tips to keep your MIDlets using memory in an efficient manner:
  - Tip: Set objects to null when you're finished with them to reclaim memory
  - Tip: After working with many temporary objects use System.gc()
  - Tip: Avoid creating lots of temporary objects
  - Tip: Avoid overly elegant OO design

It's important to remember a few key points to keep your memory bugs away at bay:

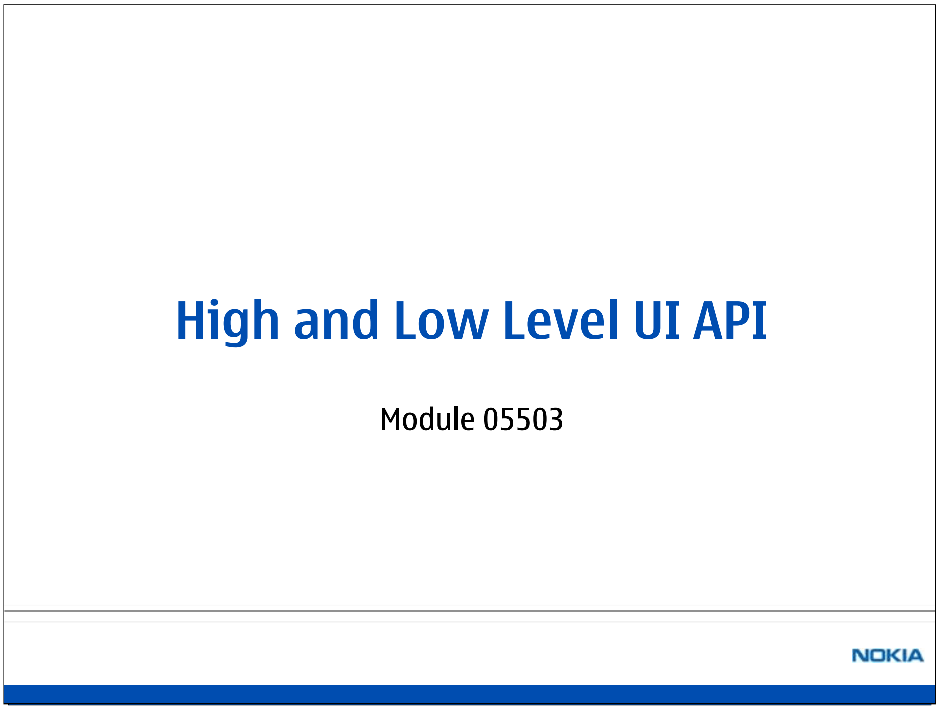
- ***set objects to null when done with an objects -- the KVM garbage collector is quite aggressive***
- ***try to recycle objects so as to limit the number of objects created***
- ***use System.gc() judiciously after you've created and nulled a lot of objects -- again the KVM GC is aggressive and responds fairly well to System.gc() hints***



Developing and Deploying MIDP Applications Review

- The MIDP Development Process consists of the following steps:
  - Development the code using the MIDP SDK
  - Compile the .java files into .class files
  - Run the Pre-verifying tool on the .class files
  - Package the application by creating a JAR and JAD tool
- MIDP applications can be deployed to physical devices using
  - OTA
  - Bluetooth
  - Infrared
  - Serial Cable

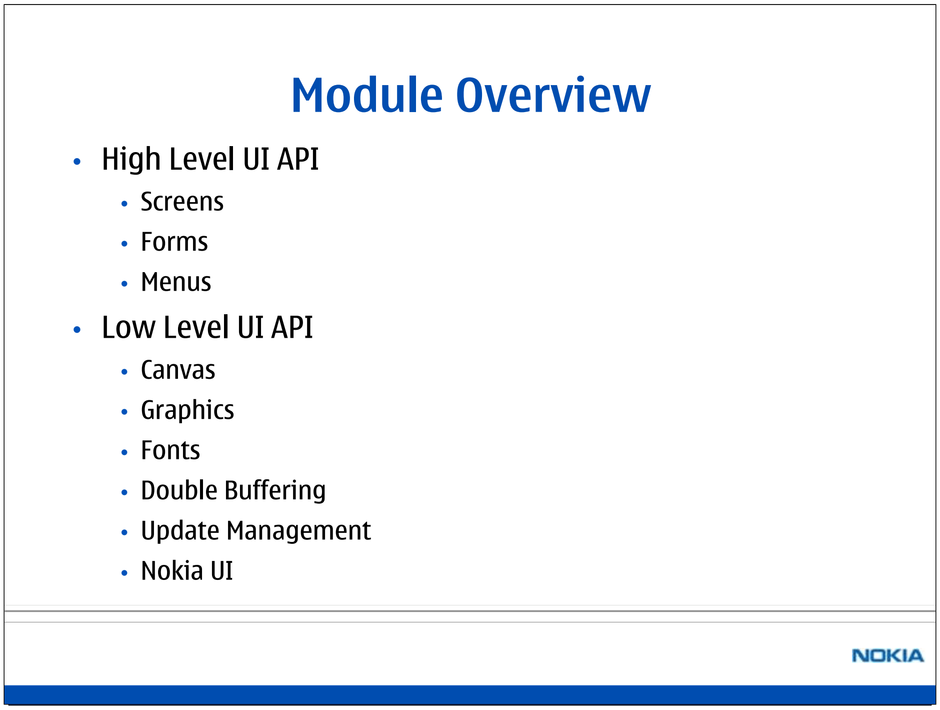
- ***What is the trade-off between memory and speed?***
- ***How can you trace call coverage in your MIDlet?***
- ***How can you use the JDK to debug your MIDlet?***
- ***How can you debug and profile your MIDlet in a target device?***
- ***What are some of the ways you can deploy a MIDlet to a physical device?***
- ***What is OTA and why is it important?***
- ***How can you test your MIDlet so that it supports OTA?***
- ***How can you test the memory usage of your MIDlet?***
- ***How can you test the runtime speed of your MIDlet?***
- ***Why is the emulator useful?***
- ***Why must you always test your MIDlet on a physical device before shipment?***



High and Low Level UI API

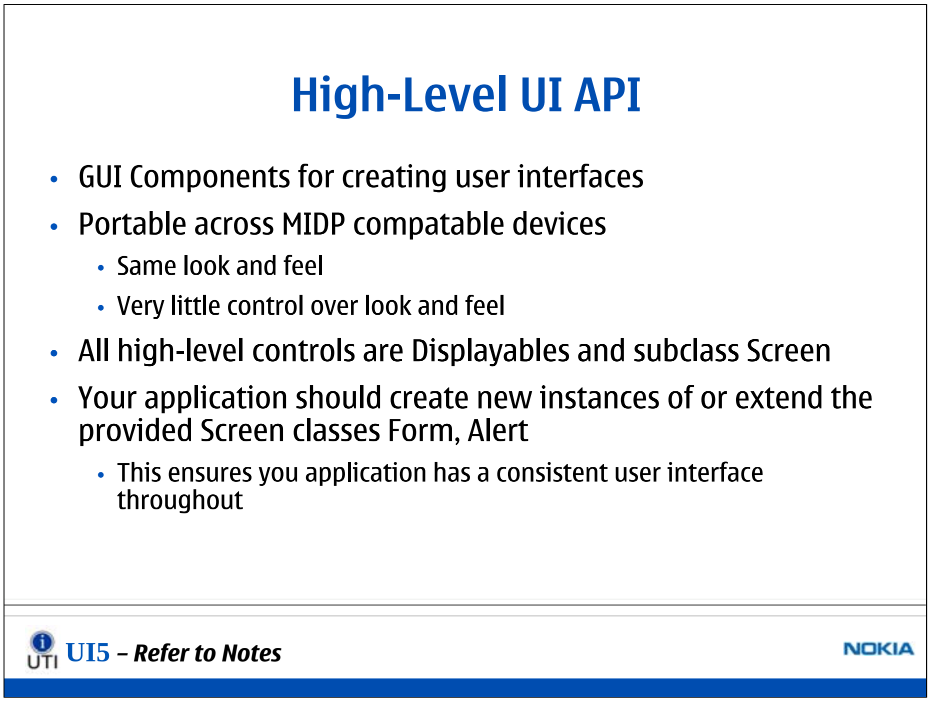
Module 05503

MIDP defines two different ways to interact with the user: the screen and the canvas. Whereas screens implement high-level APIs that define how the user interface components look on the device and thus are portable, the canvas is a low-level API, is non-portable and provides the developer with almost complete control over how the user interface appears. The canvas is particularly geared towards rich applications such as mobile games.



Module Overview

- High Level UI API
  - Screens
  - Forms
  - Menus
- Low Level UI API
  - Canvas
  - Graphics
  - Fonts
  - Double Buffering
  - Update Management
  - Nokia UI



High-Level UI API

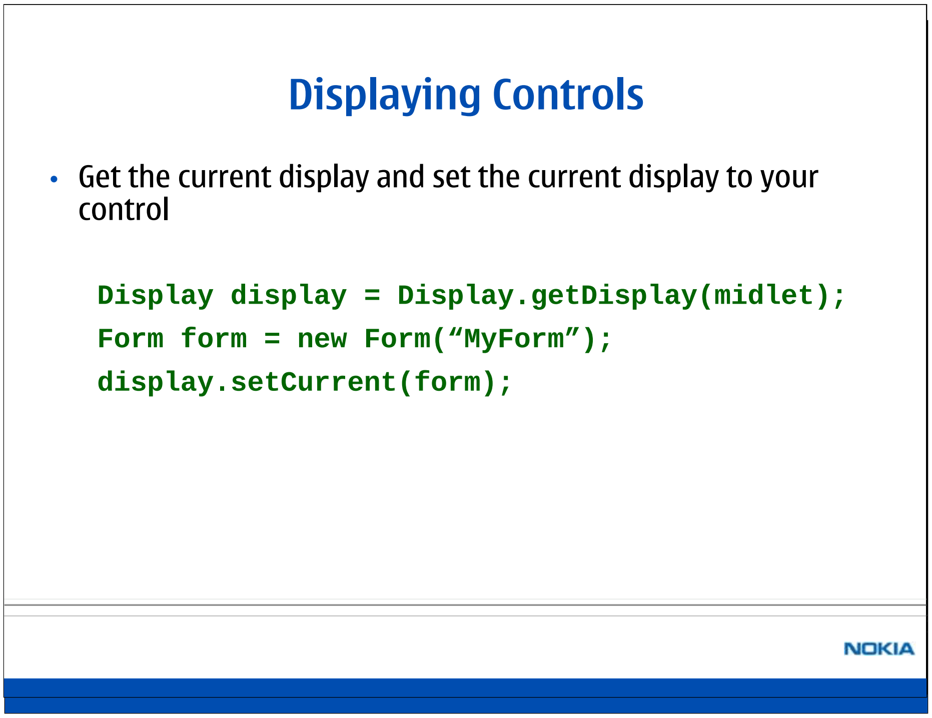
- GUI Components for creating user interfaces
- Portable across MIDP compatable devices
  - Same look and feel
  - Very little control over look and feel
- All high-level controls are Displayables and subclass Screen
- Your application should create new instances of or extend the provided Screen classes Form, Alert
  - This ensures you application has a consistent user interface throughout

UI5 – ***Refer to Notes***

NOKIA

UI5 - The application's user interface should be consistent throughout, e.g. common series of actions, action sequences, terms, layouts, soft button definitions and sounds.

To make sure your application screens, error messages etc have a consistent UI, you should create new class by extending Form/Alerts with the chosen layout for your application. You can then use/extend these new classes to create the elements in your application. Use static final constants for terms, soft button definition and sounds. This way, you can use these throughout your application consistently.



Displaying Controls

- Get the current display and set the current display to your control

```
Display display = Display.getDisplay(midlet);
Form form = new Form(“MyForm”);
display.setCurrent(form);
```

Constructing a component using the high-level API is straightforward. The important thing to remember that there can only be one current component per screen unless you are using a Form and form items to show multiple components. Also, all high-level components are located on the javax.microedition.lcdui package so you need to import that package to use the controls.

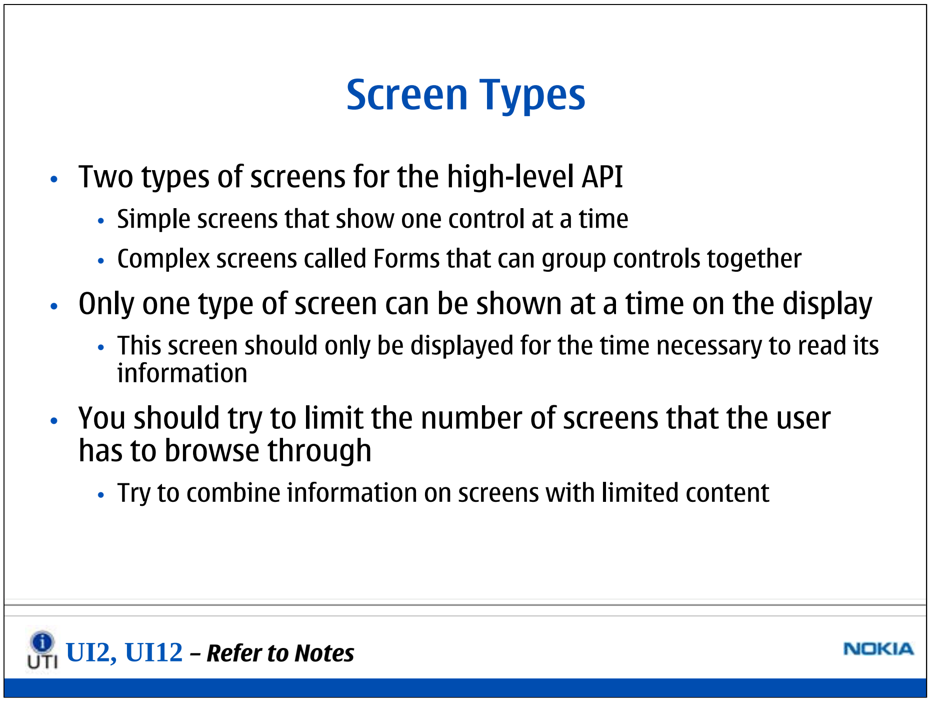
Here's an example of how to grab the current display and set it to your component:

```
class MyMIDlet extends MIDlet {
        Display display;
        void init() {
                display = Display.getDisplay(this);
        }
```

Once you have the display, you can use the object to set the current display to a control that you want to display. For example:

```
void showAlert() {
        Alert alert = new Alert(“Alert”);
        alert.setType(AlertType.ERROR);
        alert.setString(“*****ERROR*****”);
        display.setCurrent(alert); // set the current display
}
```

And that's it! Once you do that your control will be displayed as the current screen.



Screen Types

- Two types of screens for the high-level API
  - Simple screens that show one control at a time
  - Complex screens called Forms that can group controls together
- Only one type of screen can be shown at a time on the display
  - This screen should only be displayed for the time necessary to read its information
- You should try to limit the number of screens that the user has to browse through
  - Try to combine information on screens with limited content

UTI **UI2, UI12** – ***Refer to Notes***

NOKIA

UI2 - Each screen must appear for the time necessary to read all its information

The developer, depending on the nature of the usage, should decide how long a screen should be displayed for.

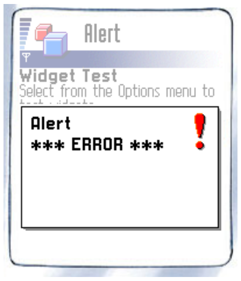
UI12 - The number of screens a user has to browse through must be minimal

Minimizing the number of screens means the user is less likely to get lost within the application. There are a number of ways in which you can keep the number of screen the user has to page through to a minimum. On way is to try to combine the information on multiple screens with limited content. Another idea is to implement shortcuts in your application, which take the user directly to the main areas of the application, without having to trawl through menus and other screens.

Simple Screens

- Use them for simple user input

Alert



Alert

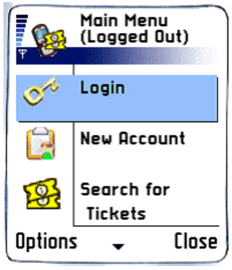
Widget Test

Select from the Options menu to

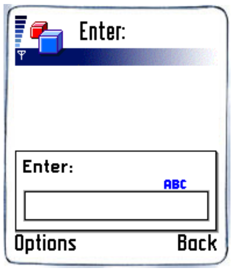
Alert

*** ERROR ***

List

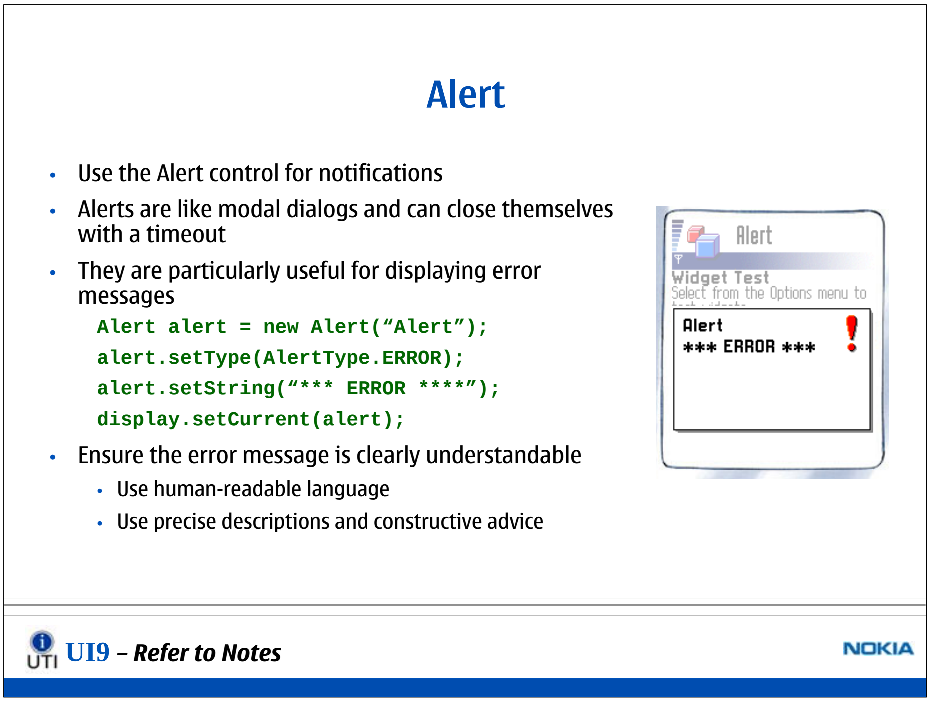


TextBox



Enter:

Enter:



Alert

- Use the Alert control for notifications
- Alerts are like modal dialogs and can close themselves with a timeout
- They are particularly useful for displaying error messages

```
Alert alert = new Alert(“Alert”);
alert.setType(AlertType.ERROR);
alert.setString(“*** ERROR ****”);
display.setCurrent(alert);
```

- Ensure the error message is clearly understandable
  - Use human-readable language
  - Use precise descriptions and constructive advice

Alert
*** ERROR ***

UI9 – *Refer to Notes*

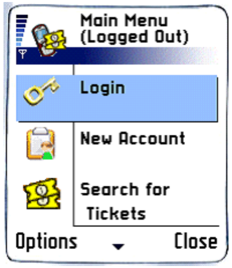
UI9 - Any error messages in the application must be clearly understandable. Error messages must clearly explain to the user the nature of the problem, and indicate what action needs to be taken (where appropriate).

As the slide explains, you can use an Alert to display an error message. To make the error message clearly understandable set the Alert type to be AlertType.ERROR. This way, the implementation will know to display a box that represents an error. Set the title of the alert to be something relating directly to the problem and ensure the text of the error provides constructive advice using a human-readable language (not obscure codes) and precise description of the problem.

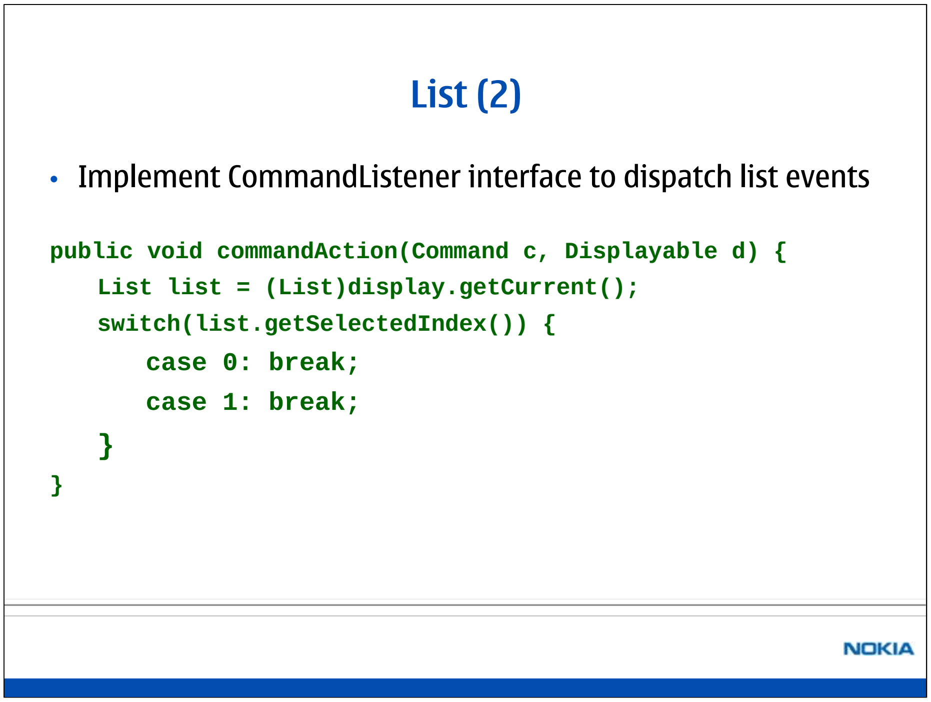
List (1)

- Use List controls to display choices to the user

```
List list = new List(“Main Menu”,
  Choice.IMPLICIT);
list.append(“Login”, image1);
list.append(“New Account”, image2);
list.append(“Search for Tickets”, image3);
display.setCurrent(list);
```



NOKIA



List (2)

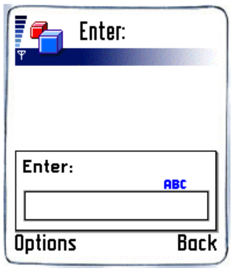
- Implement CommandListener interface to dispatch list events

```
public void commandAction(Command c, Displayable d) {
   List list = (List)display.getCurrent();
   switch(list.getSelectedIndex()) {
      case 0: break;
      case 1: break;
   }
}
```

TextBox

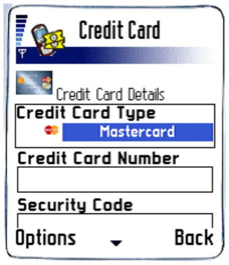
- Use the **TextBox** control to accept user input

```
TextBox textBox
= new TextBox(“Enter:”,””,20,TextField.ANY);

display.setCurrent(textBox);
```



Complex Screens - Forms

- Form class acts as a container for form items.
- Form Items:
  - TextField
  - StringItem
  - DateField
  - ImageItem
  - Gauge
  - CustomItem
  - ChoiceGroup



Credit Card

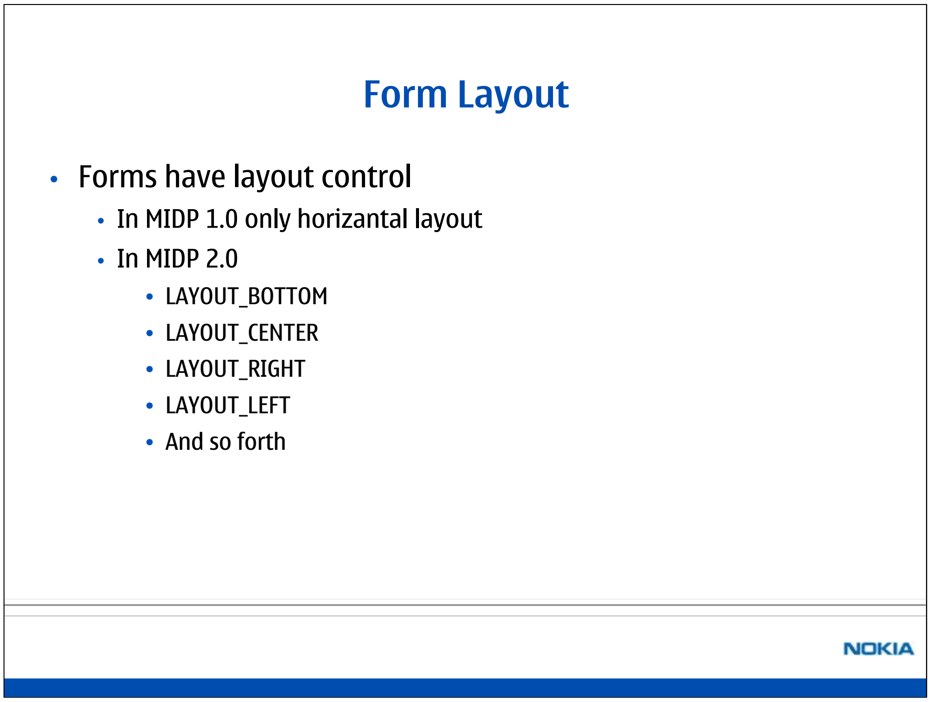
Credit Card Details

Credit Card Type

Mastercard

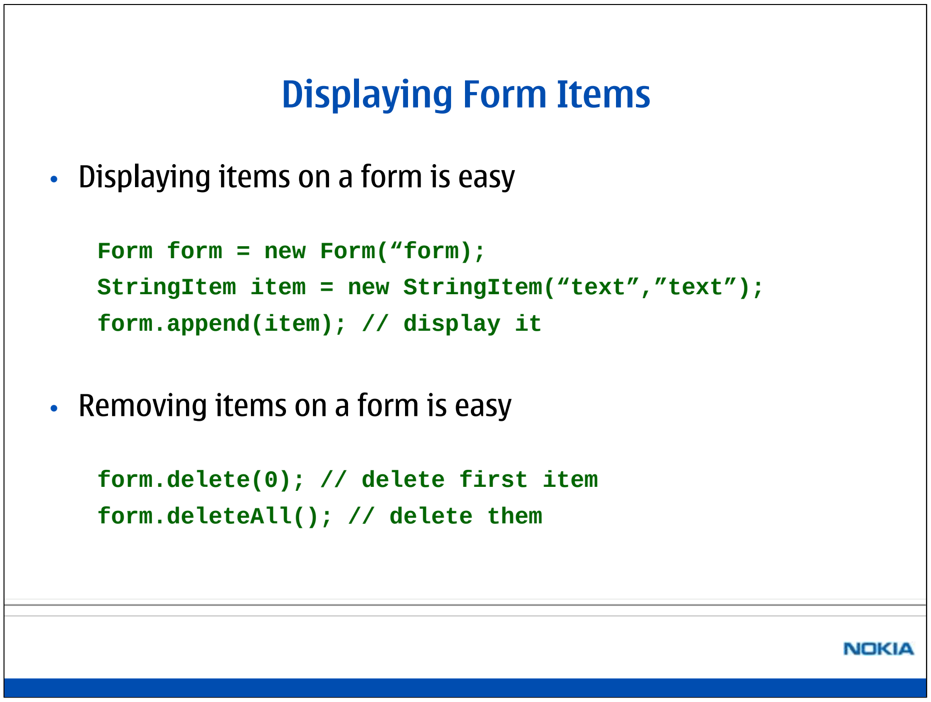
Credit Card Number

Security Code



Form Layout

- Forms have layout control
  - In MIDP 1.0 only horizantal layout
  - In MIDP 2.0
    - LAYOUT_BOTTOM
    - LAYOUT_CENTER
    - LAYOUT_RIGHT
    - LAYOUT_LEFT
    - And so forth



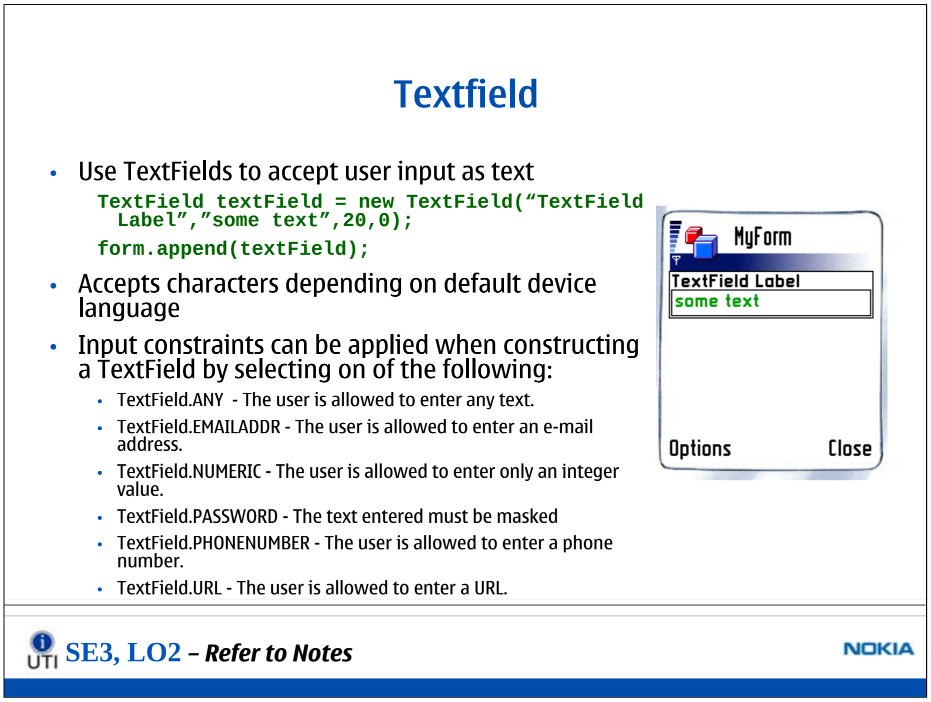
Displaying Form Items

- Displaying items on a form is easy

```
Form form = new Form(“form);
StringItem item = new StringItem(“text”,”text”);
form.append(item); // display it
```

- Removing items on a form is easy

```
form.delete(0); // delete first item
form.deleteAll(); // delete them
```



Textfield

- Use TextFields to accept user input as text

```
TextField textField = new TextField("TextField
  Label","some text",20,0);
form.append(textField);
```

- Accepts characters depending on default device language
- Input constraints can be applied when constructing a TextField by selecting on of the following:
  - TextField.ANY - The user is allowed to enter any text.
  - TextField.EMAILADDR - The user is allowed to enter an e-mail address.
  - TextField.NUMERIC - The user is allowed to enter only an integer value.
  - TextField.PASSWORD - The text entered must be masked
  - TextField.PHONENUMBER - The user is allowed to enter a phone number.
  - TextField.URL - The user is allowed to enter a URL.

UTI SE3, LO2 – *Refer to Notes*

SE3 - The application must not echo the input of sensitive data, e.g., pins and passwords

Your application should use the TextField class for password input box. To ensure the TextField does not echo the input of sensitive data, you should set the value of the input constraints parameter to TextField.PASSWORD

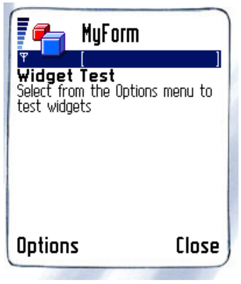
LO2 - Data entry fields must accept and properly display International characters.

TextField supports all input modes available in native text editing. The default input mode is set dependent on the user interface language of the device. In MIDP 2.0, you can use the setInitialInputMode method to define the active input mode, however, this is only a hint to the implementation.

StringItem

- Use StringItem to display static text

```
String str = “Select from the Options menu to test widgets”;
StringItem stringItem
= new StringItem(“Widget Test”, str);
form.append(stringItem);
```

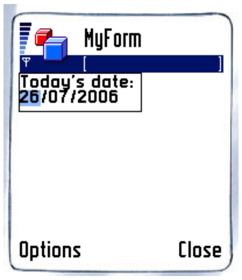


Widget Test
Select from the Options menu to test widgets

DateField

- Use DateField to display dates

```
DateField dateField = new DateField(“Date:”,
  DateField.DATE);
java.util.Date date = new java.util.Date();
dateField.setDate(date);
form.append(dateField);
```



Today's date:
26/07/2006

- Displays the date in the correct format for default device time zone

UTI **LO1** – ***Refer to Notes***

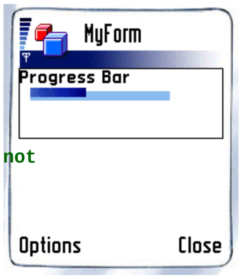
LO1 - Data format must be handled appropriately for the targeted country

Always use the DateField class for displaying/entering dates and times. This is especially useful when thinking about localisation, as when using the DateField class, the date is displayed correctly for the default time zone of the device the application is running on.

Gauge

- Use Gauges to display values in a range as a bar graph if non-interactive or ascending arc if interactive

```
Gauge gauge
= new Gauge(“Progress Bar”, false, 30, 10);
// 2nd parameter determines if interactive or not

form.append(gauge);
```



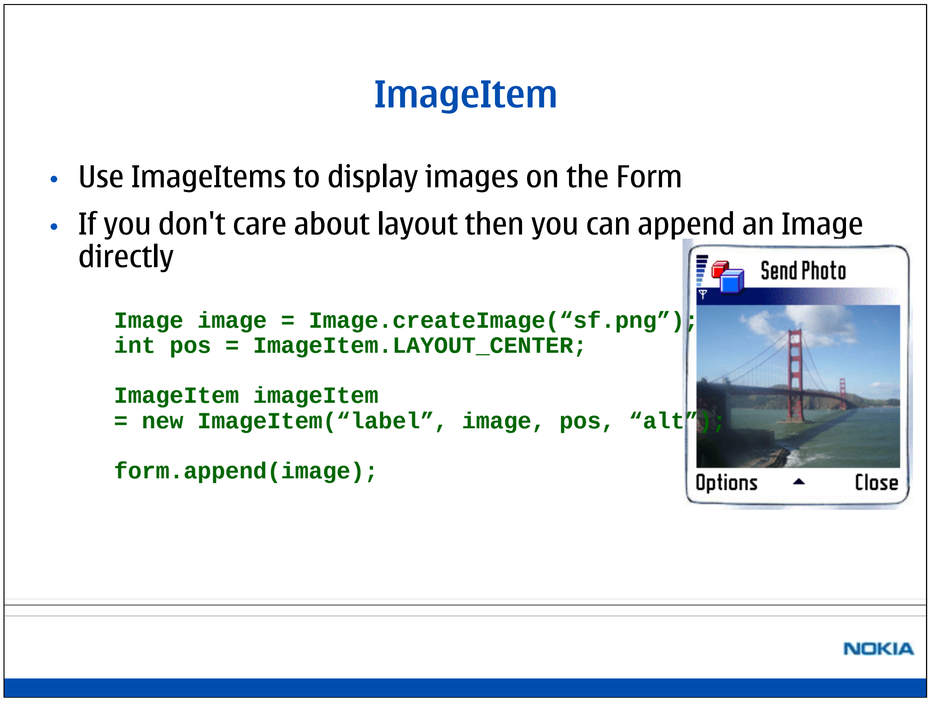
- Use setValue() to change the position of the current level in the bar graph
- Useful for displaying progress screens

UTI **UI13** – ***Refer to Notes***

NOKIA

UI13 - Any selection of a different function in the application should be performed within 5 seconds. Within 1 second, there must be some visual indication that the function will be performed

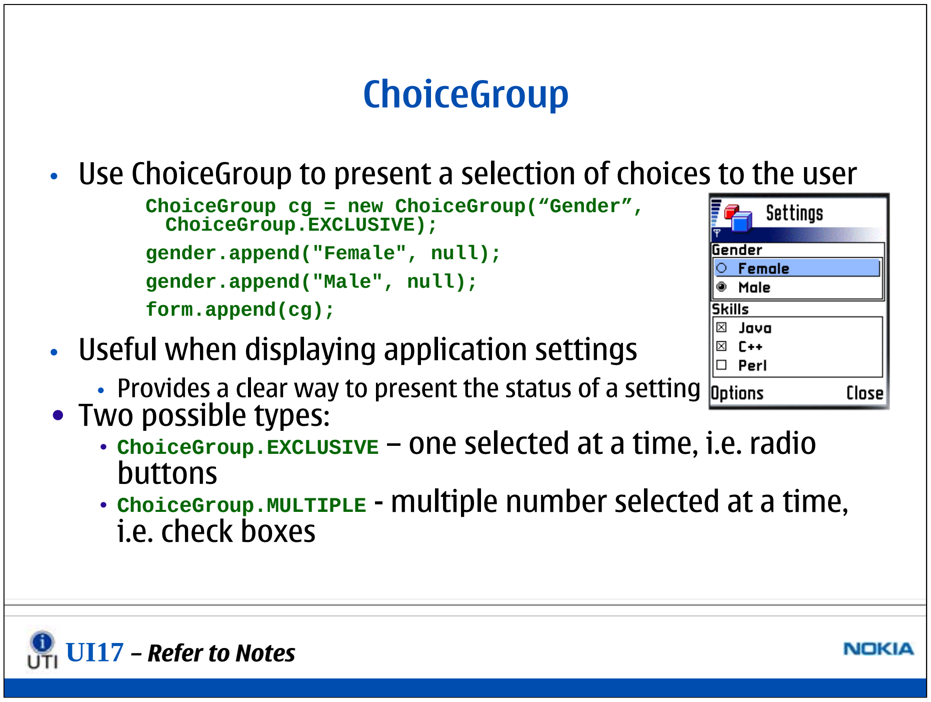
A Gauge can be used to show progress. Your application should update the Gauge as parts of the task are completed, using the setValue method



ImageItem

- Use ImageItems to display images on the Form
- If you don't care about layout then you can append an Image directly

```
Image image = Image.createImage(“sf.png”);
int pos = ImageItem.LAYOUT_CENTER;

ImageItem imageItem
= new ImageItem(“label”, image, pos, “alt”);

form.append(image);
```



ChoiceGroup

- Use ChoiceGroup to present a selection of choices to the user

```
ChoiceGroup cg = new ChoiceGroup(“Gender”,
  ChoiceGroup.EXCLUSIVE);
gender.append("Female", null);
gender.append("Male", null);
form.append(cg);
```

- Useful when displaying application settings
  - Provides a clear way to present the status of a setting
- Two possible types:
  - `ChoiceGroup.EXCLUSIVE` – one selected at a time, i.e. radio buttons
  - `ChoiceGroup.MULTIPLE` - multiple number selected at a time, i.e. check boxes

Settings

Gender

○ Female

● Male

Skills

☒ Java

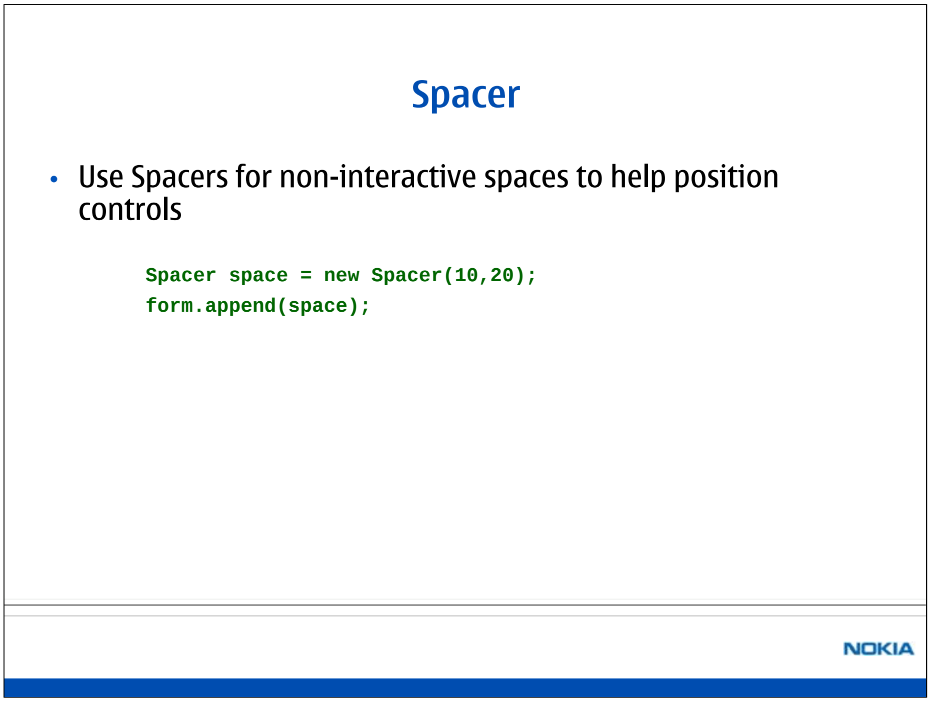
☒ C++

☐ Perl

UI17 – *Refer to Notes*

UI17 - The current status of each setting is clear: the application should make use of check boxes or by changing text.

As the slide shows, a ChoiceGroup can be used to display setting within your application. Check boxes can be implemented by, setting the type to Choice.MULTIPLE



Spacer

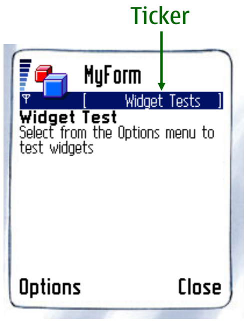
- Use Spacers for non-interactive spaces to help position controls

```
Spacer space = new Spacer(10,20);
form.append(space);
```

Tickers

- Tickers are scrolling text information that appears on the screen
- You can attach a ticker to any Displayable except Canvas

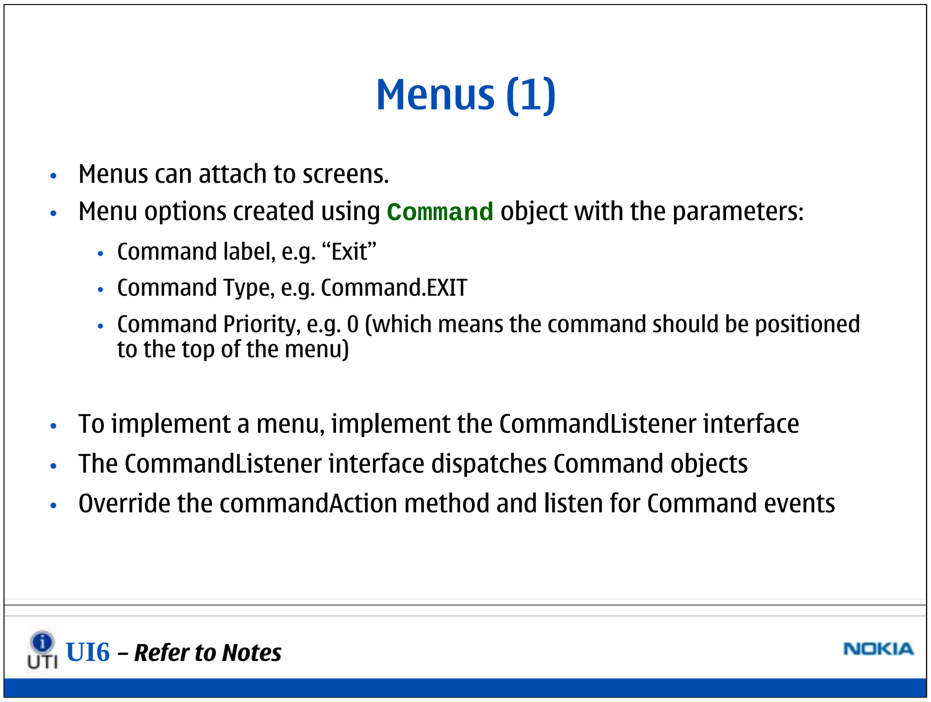
```
Form form = new Form(“MyForm”);
form.setTicker(new Ticker(“Widget Tests”));
```



Widget Test
Select from the Options menu to test widgets

Tickers are scrolling display items that you can attach to a screen item. Typically they are used to display some form of current information such as stock market price information. Tickers can be attached to screen item like this:

```
// using the same list examples
void setListTicker() {
       list.setTicker(new Ticker(“a ticker tape));
}
```



Menus (1)

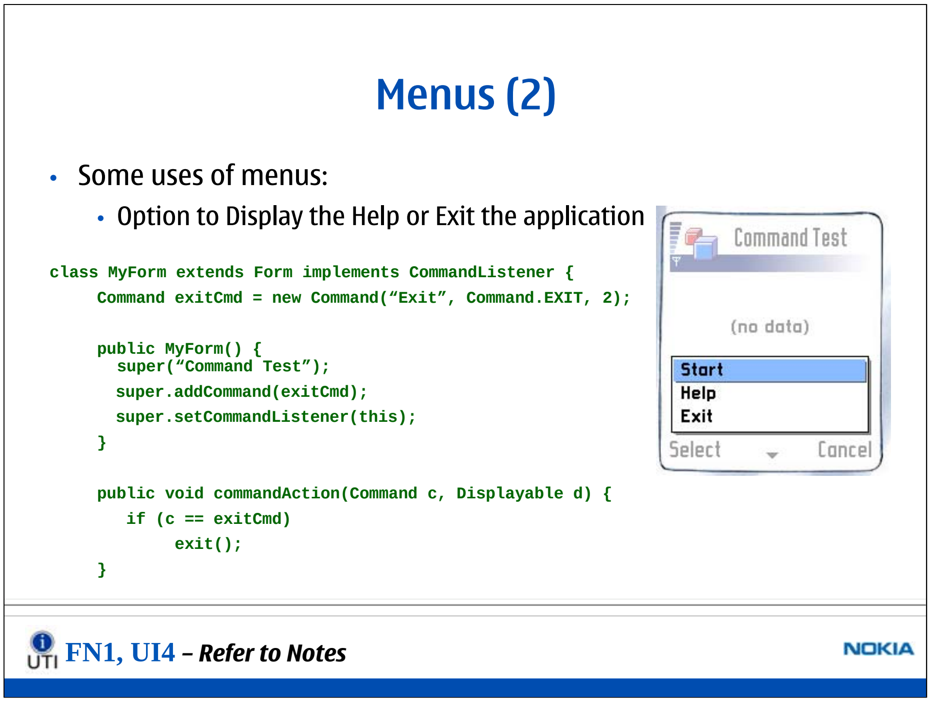
- Menus can attach to screens.
- Menu options created using `Command` object with the parameters:
  - Command label, e.g. “Exit”
  - Command Type, e.g. Command.EXIT
  - Command Priority, e.g. 0 (which means the command should be positioned to the top of the menu)
- To implement a menu, implement the CommandListener interface
- The CommandListener interface dispatches Command objects
- Override the commandAction method and listen for Command events

UTI UI6 – ***Refer to Notes***

NOKIA

UI6 - Where the application uses menu or selection items, the function of the selection and menu items must be clearly understandable to the user. Further, each menu or selection item must perform a valid action (i.e. no menu orphans.)

When adding the command label, you should ensure it appropriate for the target audience. Ideally, the label of the selected command should be reflected in the title of the resultant screen. The order of commands must be considered in your application also. For example, the most relevant command for currenct application state should be at the top, while something such as an ”Exit” command should be at the bottom.



Menus (2)

- Some uses of menus:
  - Option to Display the Help or Exit the application

```
class MyForm extends Form implements CommandListener {
     Command exitCmd = new Command(“Exit”, Command.EXIT, 2);

     public MyForm() {
       super(“Command Test”);
       super.addCommand(exitCmd);
       super.setCommandListener(this);
     }

     public void commandAction(Command c, Displayable d) {
        if (c == exitCmd)
             exit();
     }
```

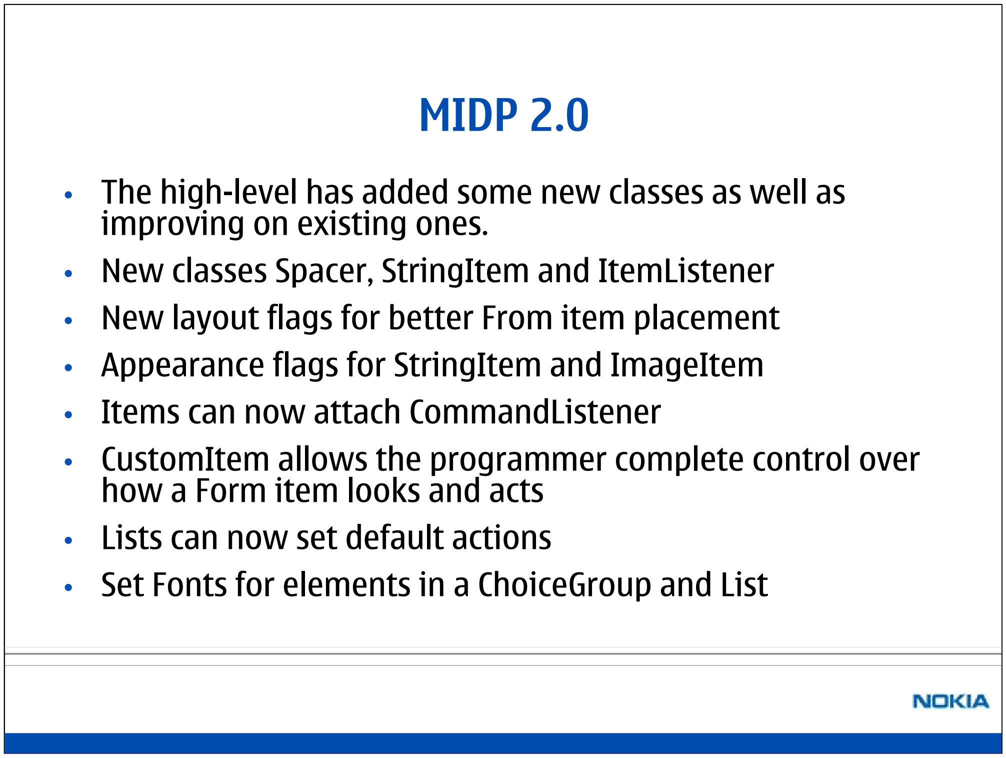
UTI **FN1, UI4** – ***Refer to Notes***

FN1 - An exit functionality is explicitly present in the application (e.g. in a Main Menu)

For every screen in your application, you should have a way to exit the whole application. This can be done by adding a Command with type Command.EXIT to each Form, Alert, Canvas etc. You must ensure that the ”Exit” functionality in your application calls destroyApp then notifyDestroyed. The destroyApp method must release all resources that your application has been using

UI4 - If applicable, the main functionalities of Exit, About and Help must be accessed easily through a Main Menu.

You can create a main menu using Commands



MIDP 2.0

- The high-level has added some new classes as well as improving on existing ones.
- New classes Spacer, StringItem and ItemListener
- New layout flags for better From item placement
- Appearance flags for StringItem and ImageItem
- Items can now attach CommandListener
- CustomItem allows the programmer complete control over how a Form item looks and acts
- Lists can now set default actions
- Set Fonts for elements in a ChoiceGroup and List

Some new classes and improved functionality of the MIDP 1.0 classes have been added that dramatically improve programmer control over the high-level UI. Notably the following improvements have been made:

- ***Ability to control backlight and vibrator***
- ***New methods for managing styles and colors including alpha values for colors***
- ***Methods to query preferred image dimensions for use in lists***
- ***Commands can implement short and long labels which are more aesthetic depending on the context of the where the command is used***
- ***Affinity identifiers that allow for more flexible layout of Form items***
- ***New classes StringItem, Spacer and CustomItem***
- ***New input constraints for TextBox class such as TextBox.PHONENUMBER and TextBox.DECIMAL and input modes for different character sets.***

Alert

Alerts now have the ability to attach custom commands and progress/activity indicators.

List

A List control now has the ability to set the selected command which will be the default when the List is first instantiated. This allows for flexible command handling than using List.SELECT_COMMAND and now MIDlets can define command dispatchers more cleanly. This is valid for implicit lists.

Items

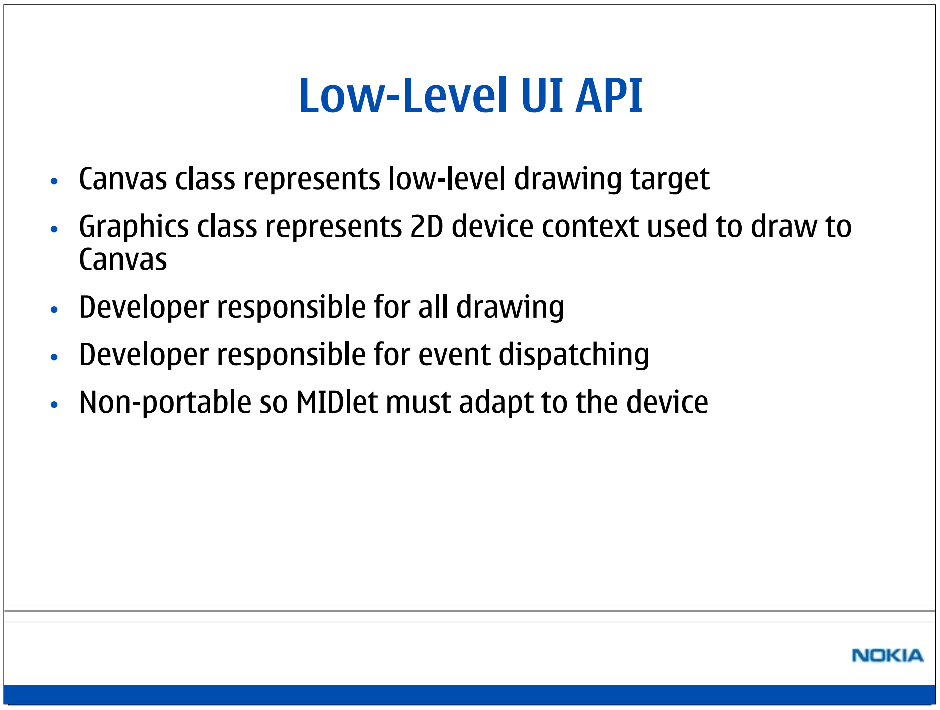
All items now allow individial event listeners to be attached directly to the form item. Items can now also modify the labels in addition to the textual content after the Item is created.

Spacers and Layout

One of the problems with MIDP 1.0 (especially dealing with Forms) was how to place Items on the screen in aesthetic manner. What was sorely needed was a way to space items and to lay items out according to different affinities. Both of these actions are now supported in MIDP 2.0. Spacers are non-interactive items that specify a minimum size in width and height. The new layout affinities such as LAYOUT_BOTTOM, LAYOUT_CENTER, LAYOUT_NEWLINE_AFTER aid in the positioning of form items in the most aesthetic manner.

Custom Items

Finally, if you are completely dissatisfied with the widget elements provided with MIDP 2.0, you are free to implement your own owner-drawn items which sublass CustomItem. The complete visual appearance and user interaction is completely implemented and defined by the programmer so now there is virtually limitless design freedom when it comes to designing MIDP user interfaces.

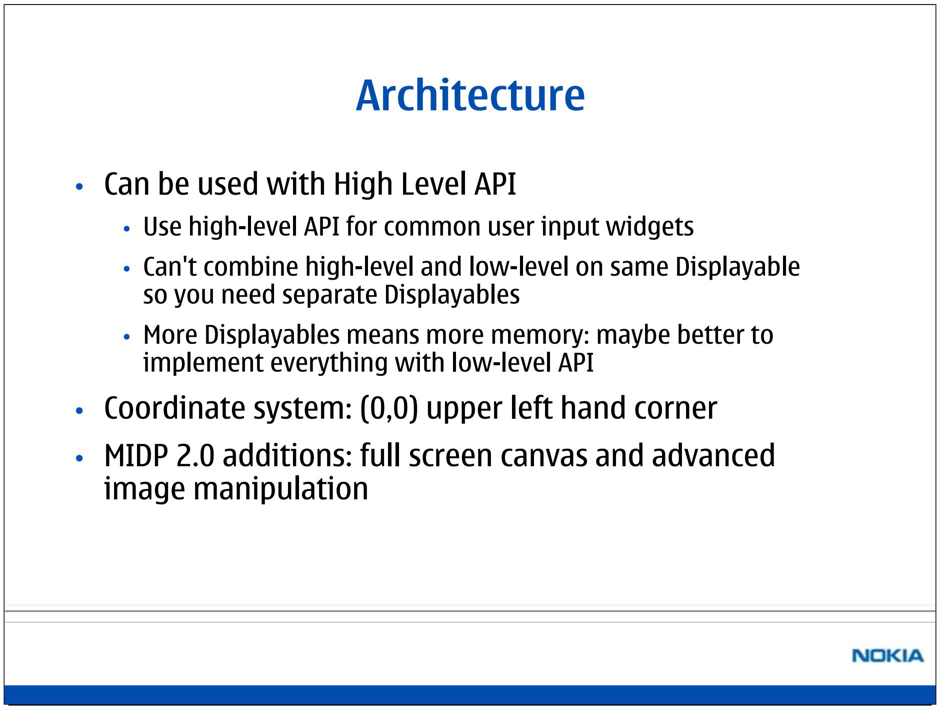


Low-Level UI API

- Canvas class represents low-level drawing target
- Graphics class represents 2D device context used to draw to Canvas
- Developer responsible for all drawing
- Developer responsible for event dispatching
- Non-portable so MIDlet must adapt to the device

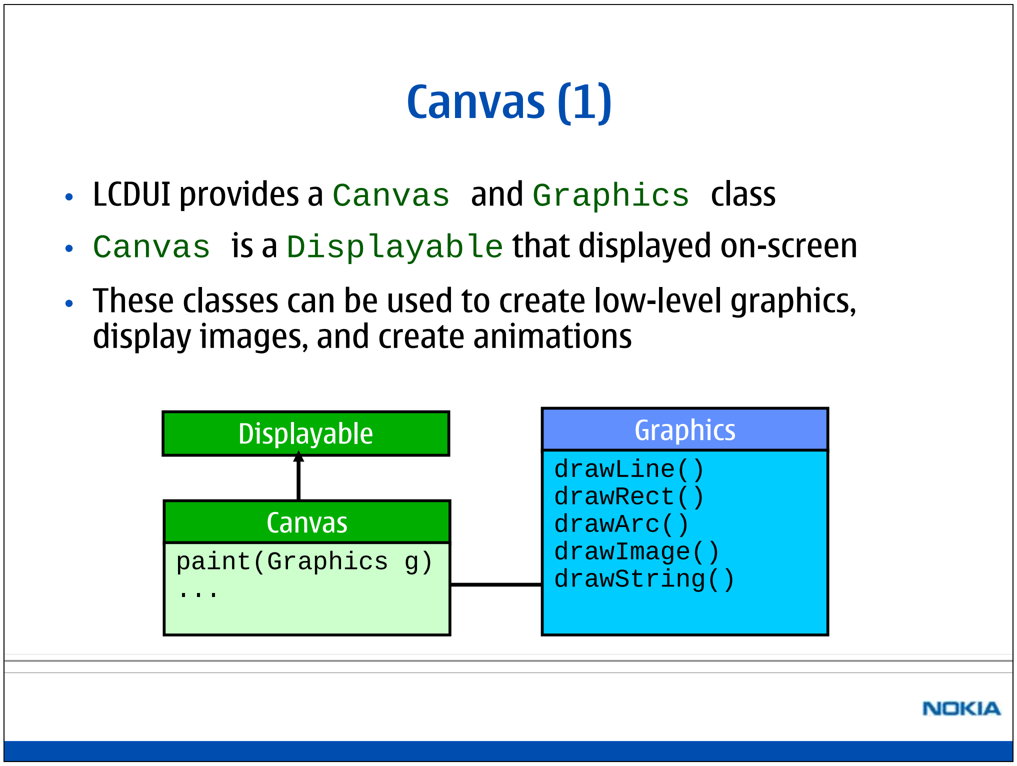
NOKIA

Canvas subclasses javax.microedition.lcdui.Displayable and provides methods for obtaining screen geometry and other informative methods about the canvas. The Graphics class is then used to implement drawing on the Canvas with a number of different primitives. The important part for the developer is to remember that the developer is responsible for all drawing and event handling on the Canvas. The canvas provides the appropriate listener interfaces for handling key and pointer events and you can use the interfaces thereof to construct and control program logic.



Architecture

- Can be used with High Level API
  - Use high-level API for common user input widgets
  - Can't combine high-level and low-level on same Displayable so you need separate Displayables
  - More Displayables means more memory: maybe better to implement everything with low-level API
- Coordinate system: (0,0) upper left hand corner
- MIDP 2.0 additions: full screen canvas and advanced image manipulation



Canvas (1)

- LCDUI provides a `Canvas` and `Graphics` class
- `Canvas` is a `Displayable` that displayed on-screen
- These classes can be used to create low-level graphics, display images, and create animations

Displayable

Canvas

```
paint(Graphics g)
...
```

Graphics

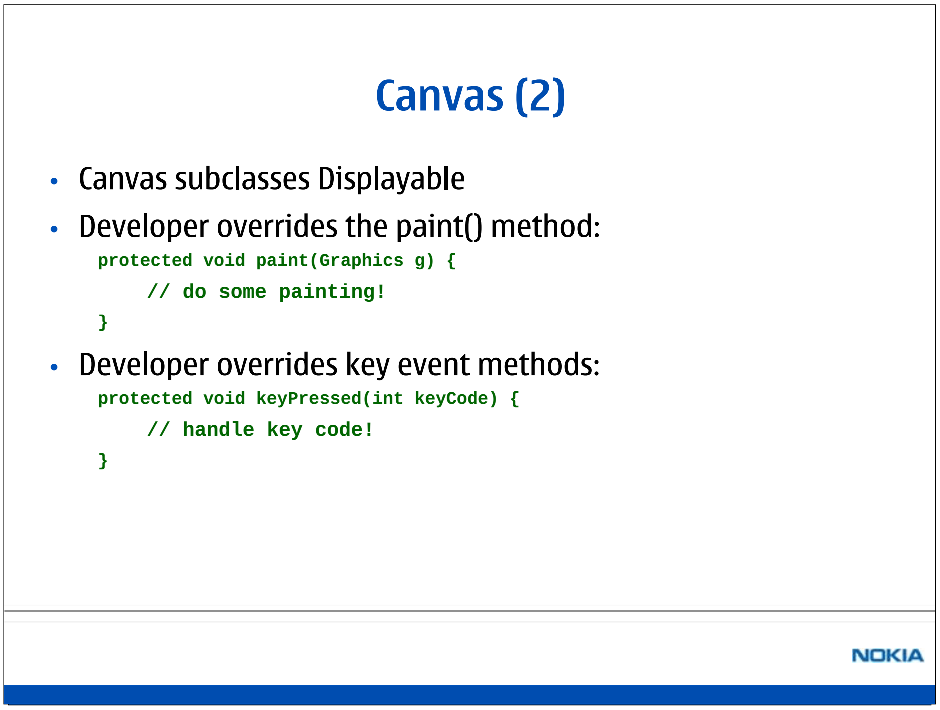
```
drawLine()
drawRect()
drawArc()
drawImage()
drawString()
```

NOKIA

As well as providing classes to add high-level components to your using interfaces, LCDUI also provided class to allow you to draw low-level graphics to the screen. The main two classes are the Canvas class and the Graphics class.

The Canvas class is a displayavle that can be displayed using the Display.setCurrent() method. It represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user.

An application must subclass the Canvas class in order to get useful functionality such as creating a custom component. The paint method must be overridden in order to perform custom graphics on the canvas.



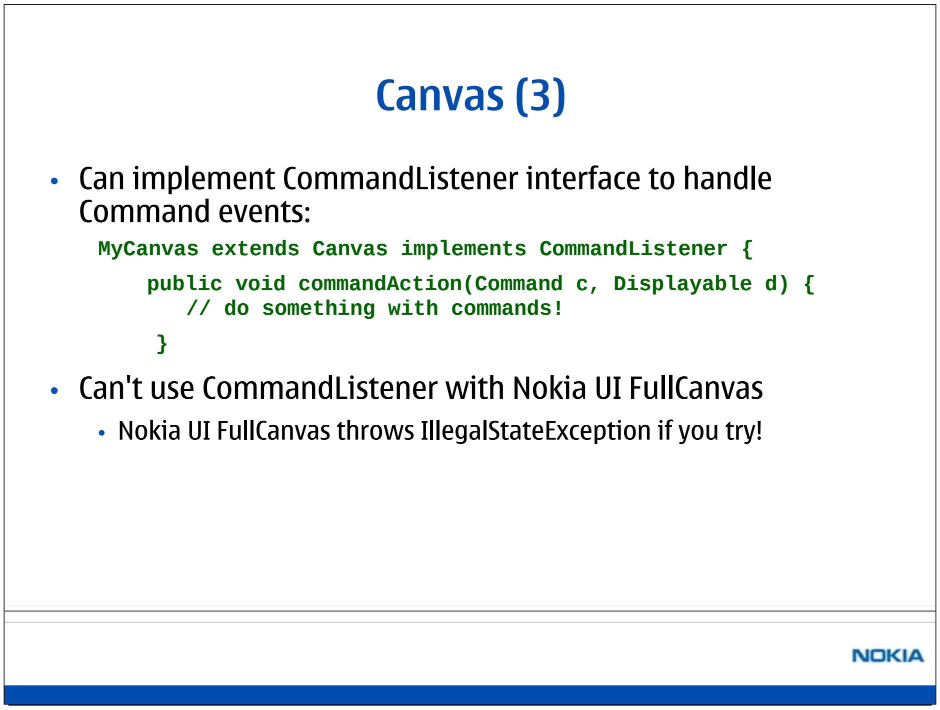
Canvas (2)

- Canvas subclasses Displayable
- Developer overrides the paint() method:

```
protected void paint(Graphics g) {
    // do some painting!
}
```

- Developer overrides key event methods:

```
protected void keyPressed(int keyCode) {
    // handle key code!
}
```

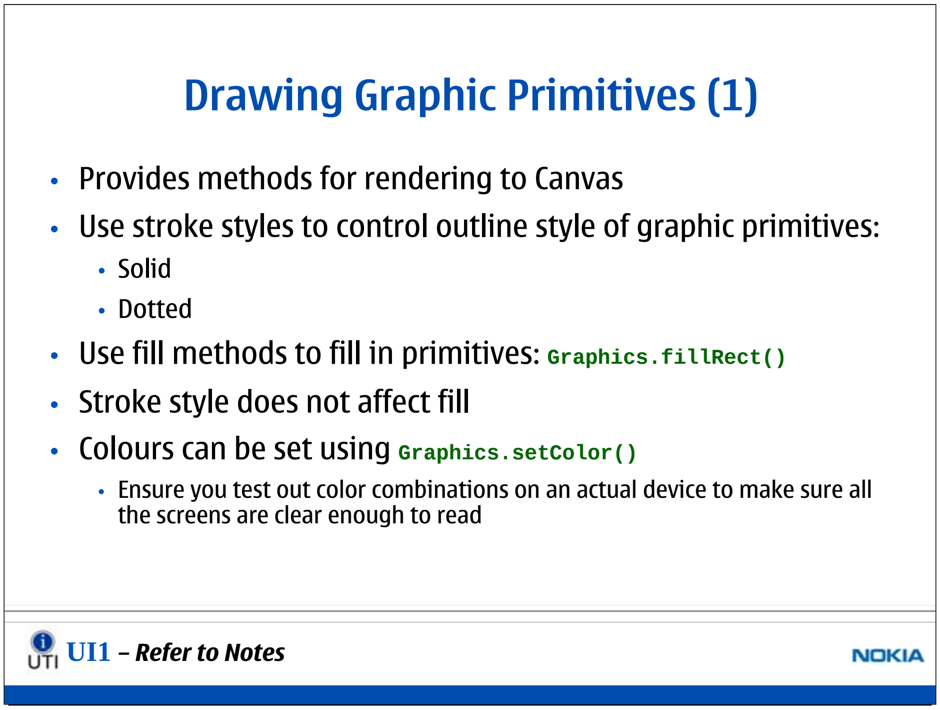


Canvas (3)

- Can implement CommandListener interface to handle Command events:

```
MyCanvas extends Canvas implements CommandListener {
    public void commandAction(Command c, Displayable d) {
       // do something with commands!
     }
```

- Can't use CommandListener with Nokia UI FullCanvas
  - Nokia UI FullCanvas throws IllegalStateException if you try!



Drawing Graphic Primitives (1)

- Provides methods for rendering to Canvas
- Use stroke styles to control outline style of graphic primitives:
  - Solid
  - Dotted
- Use fill methods to fill in primitives: `Graphics.fillRect()`
- Stroke style does not affect fill
- Colours can be set using `Graphics.setColor()`
  - Ensure you test out color combinations on an actual device to make sure all the screens are clear enough to read

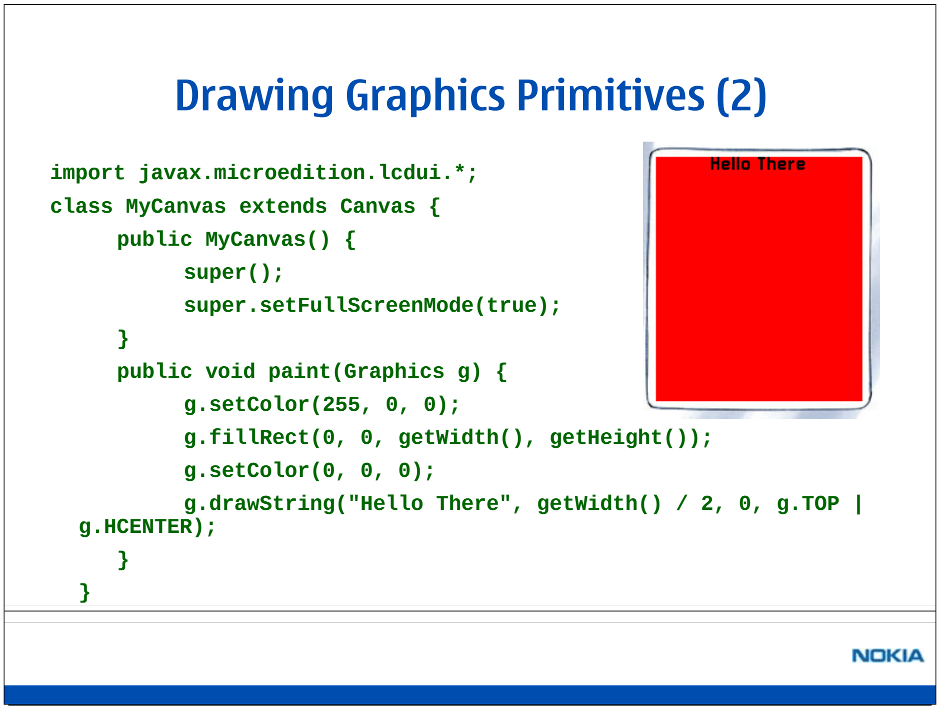
UI1 – ***Refer to Notes***

As indicated above, the Graphics class provides methods for drawing 2D geometric graphic primitives. You can draw primitives such as rectangles, polygons, lines and even load static images. As rich as the API is, there are some limitations when drawing circles and other objects such as 3D objects which rely heavily on sine, cosine and other calculations involving floating points. Currently this is a limitation of the device configuration – namely CLDC 1.0 – which doesn't support floating points and will be addressed in the future. So you need some tricks to draw circles and 3D objects with the low-level API and in the accompanying lab we'll demonstrate how to do these sorts of things. Also note, that color selection is based on a 24 bit, red, green, blue model and you can specify colors like: 0xRRGGBB with 8 bits per color component. However note that not all displays are color so you should use the isColor() and numColors() methods as indicated above for determining the capabilities of your phone. One thing to remember is that backgrounds must also be redrawn each time the paint() method is invoked so be sure to call methods like setFillRect() to assure that you maintain the correct background color for your screen since there is no method to set the background color by default.

```
import javax.microedition.lcdui.*;
public class MyCanvas extends Canvas {
   WIDTH = screen.getWidth();
   m_screenHeight = screen.getHeight();
   m_image = Image.createImage("/m031.png");
   m_imageWidth = m_image.getWidth();
   m_imageHeight = m_image.getHeight();
   m_x = (m_screenWidth - m_imageWidth)/2;
   m_y = (m_screenHeight - m_imageHeight)/2;
   protected void paint(Graphics g)
  {
     g.drawImage(m_image, m_x, m_y, Graphics.TOP|Graphics.LEFT);
  }
}
```

UI1 - All screen content must be clear and readable to the naked eye regardless of information displayed, or choice of font, color scheme etc

Avoid color schemes that are too close in hue, for example a dark blue background with a medium-range blue text. Also try to avoid color schemes that are too close in contrast, for example a dark blue background with dark purple text. When including any text on your application screens, use upper and lower case rather than full uppercase lettering, and try avoid putting too much text on the screen.



Drawing Graphics Primitives (2)

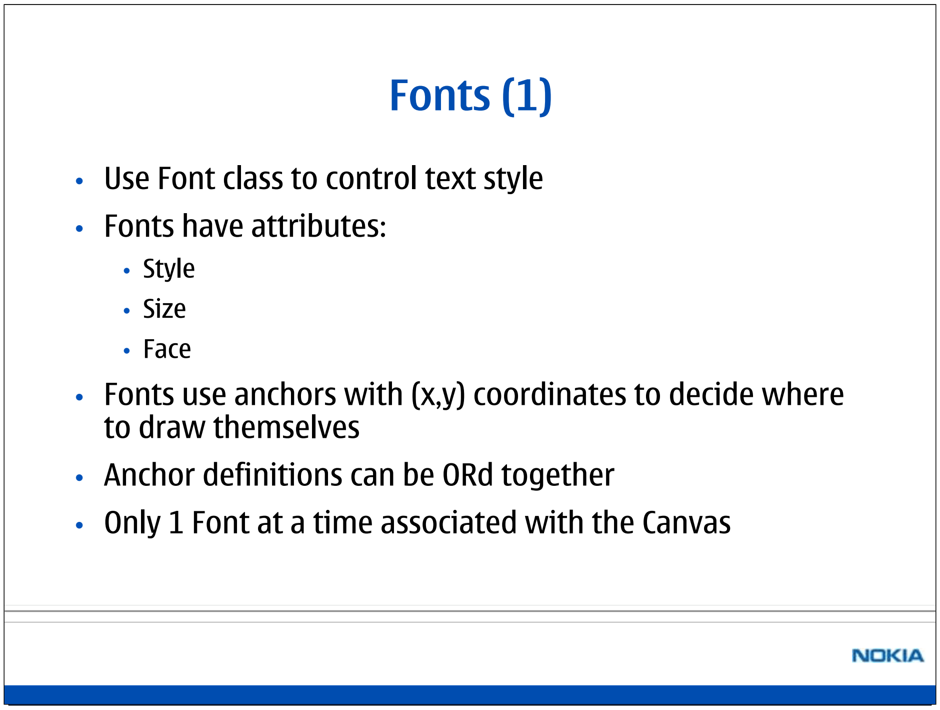
```
import javax.microedition.lcdui.*;
class MyCanvas extends Canvas {
     public MyCanvas() {
          super();
          super.setFullScreenMode(true);
     }
     public void paint(Graphics g) {
          g.setColor(255, 0, 0);
          g.fillRect(0, 0, getWidth(), getHeight());
          g.setColor(0, 0, 0);
          g.drawString("Hello There", getWidth() / 2, 0, g.TOP |
  g.HCENTER);
     }
  }
```

Drawing to the screen is accomplished by first setting the drawable canvas as the current display from the MIDlet entry point.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class MyMidlet extends MIDlet {

   public MyMidlet() { // constructor
        Canvas canvas = new MyCanvas(this)
}
   public void startApp(  ) {
      Display display = Display.getDisplay(this);
            display.setCurrent(canvas);
   }
}
```

After this, we can subclass the Canvas, and override a single method: paint(Graphics g) in order to draw to the screen:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class MyCanvas extends Canvas {
        private MyMidlet midlet;
        private bool isColor;
        private numColors = 0;
        private int HEIGHT;
        private int WIDTH;
        public MyCanvas(MyMidlet midlet) {
                 this.midlet = midlet;
                 isColor = numColors();
                 if (isColor)
                         numColors = numColors();
                 HEIGHT = getHeight();
                 WIDTH = getWidth();
   }
   protected void paint(Graphics g){
        g.setColor(0xffffff, 0xffffff, 0xffffff);
        g.fillRect(0, 0, WIDTH, HEIGHT);
        g.setColor(0, 0, 0);
        g.drawString( "Some Text", WIDTH/2, 0, g.TOP |                     g.HCENTER);
    }
}
```

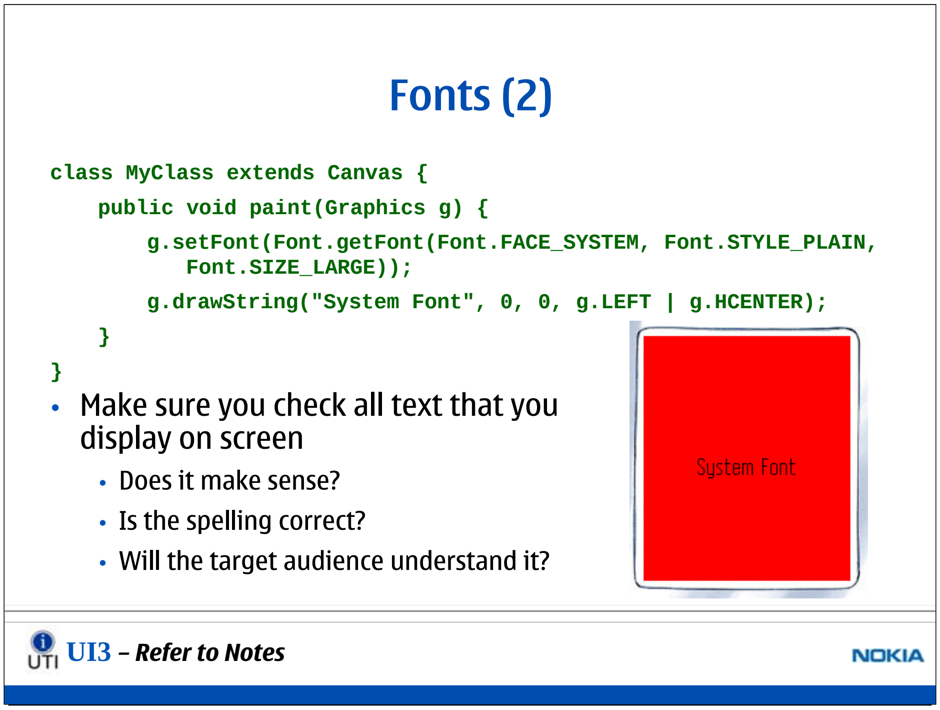


Fonts (1)

- Use Font class to control text style
- Fonts have attributes:
  - Style
  - Size
  - Face
- Fonts use anchors with (x,y) coordinates to decide where to draw themselves
- Anchor definitions can be ORd together
- Only 1 Font at a time associated with the Canvas

Fonts are used to control the way text appears on the canvas. Fonts are styled by three different attributes: Face, Size and Style. The style attribute can be ORd together to produce the desired font however the other attributes – face and size – must be singular. Font's always return a valid font object for a valid Font format which means that if a given font style specification is not available – i.e. the combination of face, size and style requested by the developer is not available – then the instantiation of the font returns the closest matching font format combination thereof. Positioning of the Font string is achieved with (X,Y) coordinates and the "anchor points" using MIDP terminology. Anchor points are hints as to the affinity of how the Font is drawn with respect to the (X,Y) position. Anchor points can be ORd together to produce the desired affinity. For example the anchor point LEFT | TOP would position the string towards the top left corner as being the starting location of the (X,Y) coordinate position of the string. The following example code should clarify:

```
import javax.microedition.lcdui.*;
public class FontCanvas extends Canvas {
   public void paint(Graphics g) {
      g.setColor(0xffffff);
      g.fillRect(0, 0, getWidth(), getHeight(  ));
      g.setColor(0x000000);
      g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN,
          Font.SIZE_LARGE));
      g.drawString("System Font", 0, 0, g.LEFT | g.TOP);
}
```



Fonts (2)

```
class MyClass extends Canvas {
    public void paint(Graphics g) {
        g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN,
           Font.SIZE_LARGE));
        g.drawString("System Font", 0, 0, g.LEFT | g.HCENTER);
    }
}
```

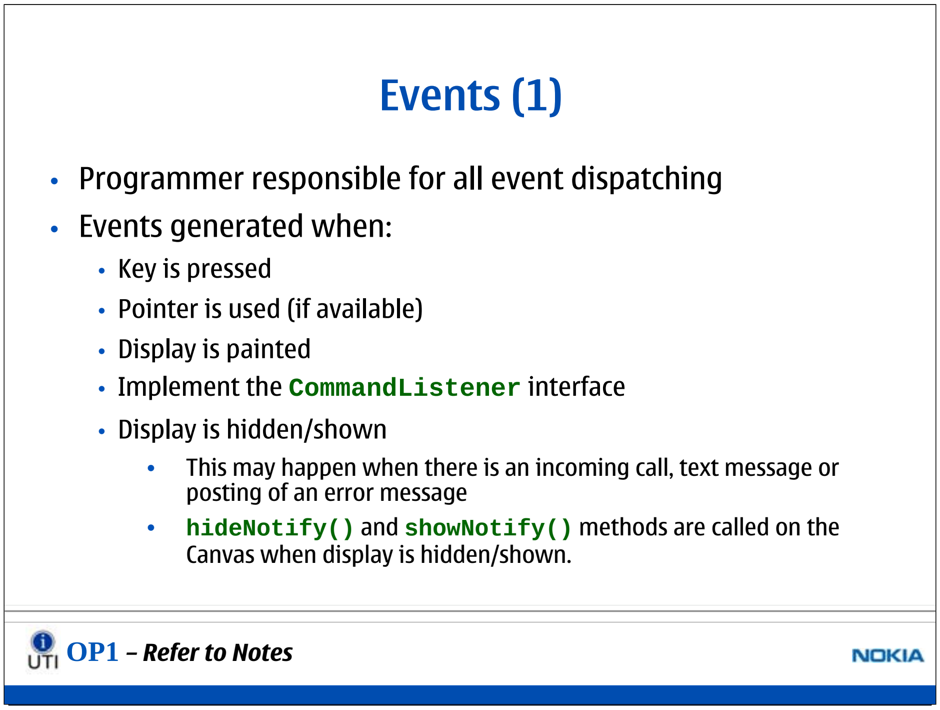
- Make sure you check all text that you display on screen
  - Does it make sense?
  - Is the spelling correct?
  - Will the target audience understand it?

System Font

UI3 – ***Refer to Notes***

UI3 - Text should be understandable to the target user group. The application spelling and grammar should be correct. Text should not be truncated

Ensure that you know the audience of your application to be sure any text is understandable. For example, if your application is for scientific user, ensure that you use symbols and acronyms correctly, or if your application is aimed at young children don't use complicated words or long sentences. Ensure to spell check all text that you use within your application.



Events (1)

- Programmer responsible for all event dispatching
- Events generated when:
  - Key is pressed
  - Pointer is used (if available)
  - Display is painted
  - Implement the `CommandListener` interface
  - Display is hidden/shown
    - This may happen when there is an incoming call, text message or posting of an error message
    - `hideNotify()` and `showNotify()` methods are called on the Canvas when display is hidden/shown.

OP1 – *Refer to Notes*

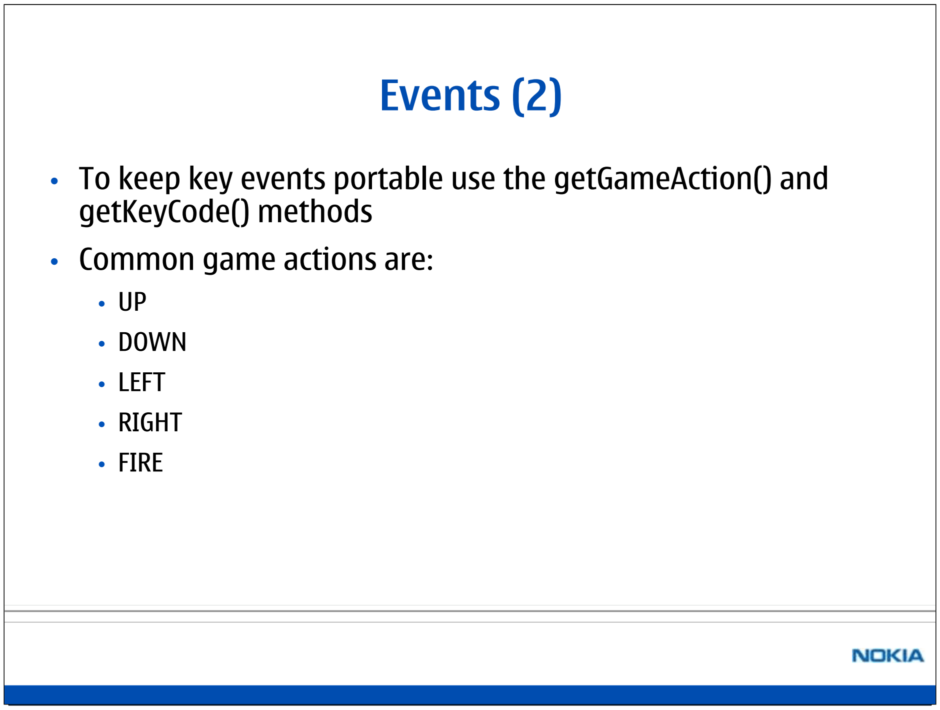
NOKIA

As with drawing, all input events and other canvas notifications is the responsibility of the developer to manage. At the very minimum you need to override the keyPressed() event to exit the MIDlet otherwise there is no way for the MIDlet to exit with end-user control.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class MyCanvas extends Canvas {
    MyMidlet midlet;
   public MyCanvas(MyMidlet midlet) {
       this.midlet = midlet;
       }
    protected void keyPressed(int keyCode){
         midlet.exit();
        }
}
```

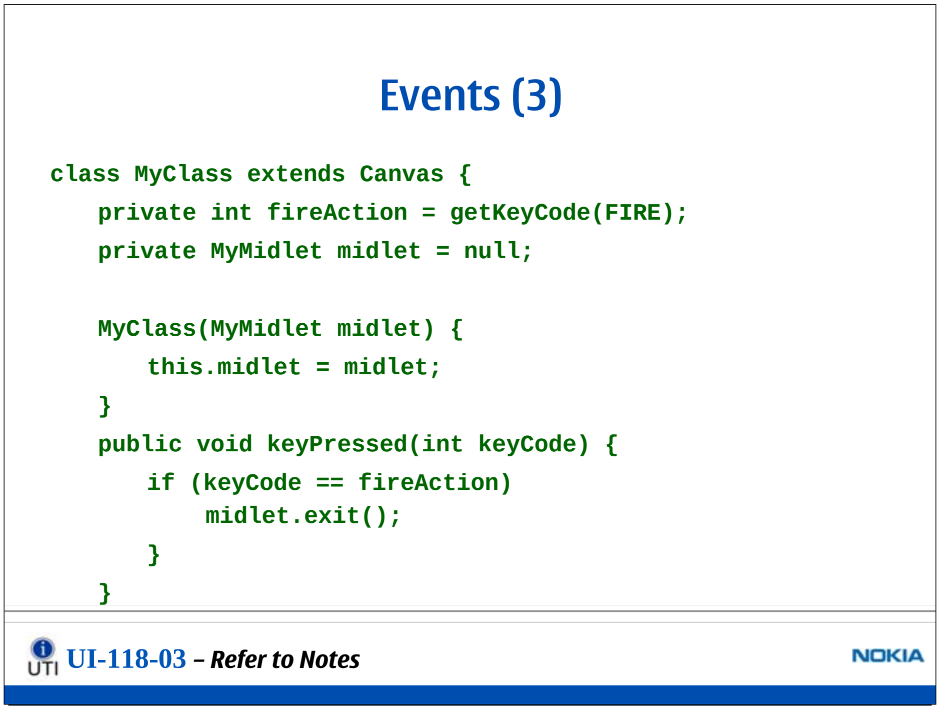
Other interesting events to capture are those that pertain to screen changes (showNotify(), hideNotify()) – each screen is known as a "card" in MIDP terminology and the sum of the screens is known as the deck -- and pointer events for devices that support pointer input.

Keyboard input is non-uniform in the sense that different key events could be mapped to different keys on the input pad in a proprietary way. To alleviate this problem, a portable solution using the getKeycode() method works by defining common key events and returns these keys -- called action keys in MIDP terminology – as mapped for the device.



Events (2)

- To keep key events portable use the getGameAction() and getKeyCode() methods
- Common game actions are:
  - UP
  - DOWN
  - LEFT
  - RIGHT
  - FIRE



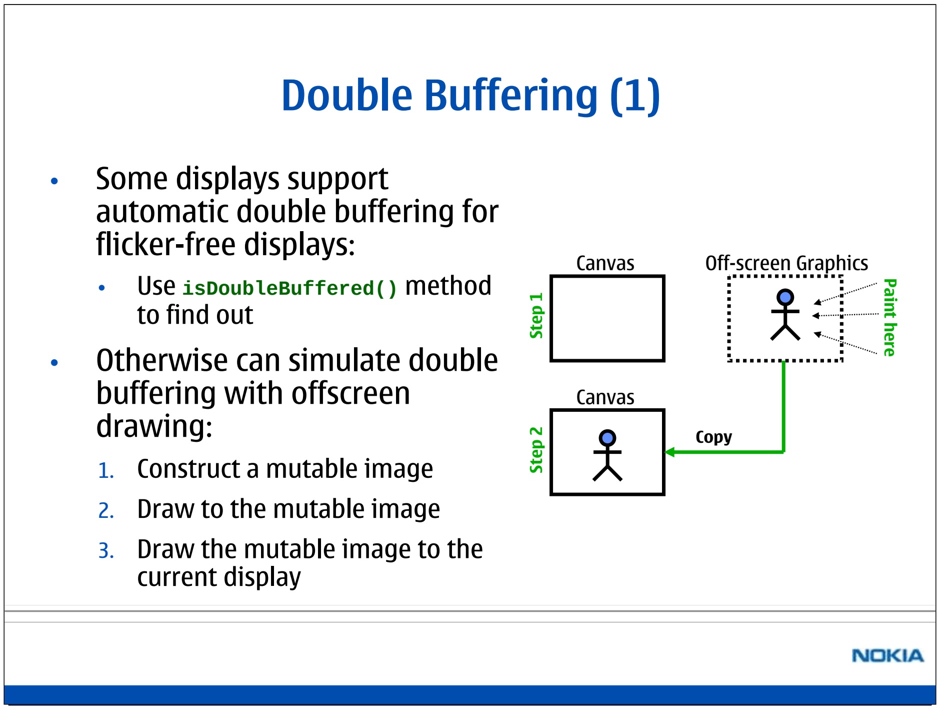
Events (3)

```
class MyClass extends Canvas {
   private int fireAction = getKeyCode(FIRE);
   private MyMidlet midlet = null;

   MyClass(MyMidlet midlet) {
       this.midlet = midlet;
   }
   public void keyPressed(int keyCode) {
       if (keyCode == fireAction)
           midlet.exit();
       }
   }
```

UI-118-03 – *Refer to Notes*

UI-118-03 - A MIDlet must not be able to simulate key-press events to mislead the user.

Your application must not call Canvas.keyPressed manually. This method should only be called when the user actually presses a key.

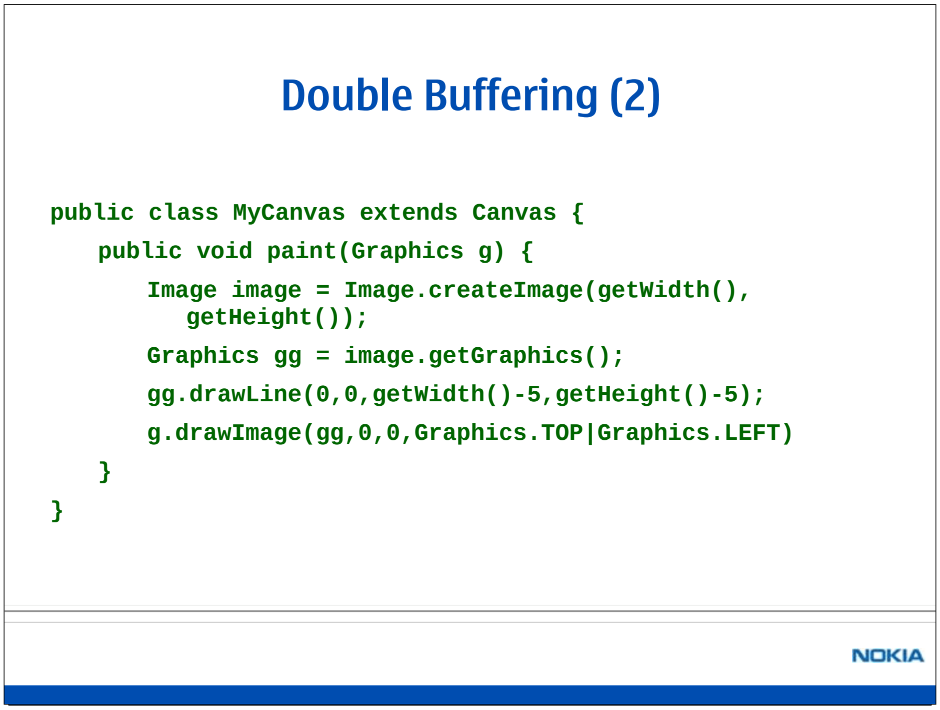


Double Buffering (1)

- Some displays support automatic double buffering for flicker-free displays:
  - Use `isDoubleBuffered()` method to find out
- Otherwise can simulate double buffering with offscreen drawing:
  1. Construct a mutable image
  2. Draw to the mutable image
  3. Draw the mutable image to the current display

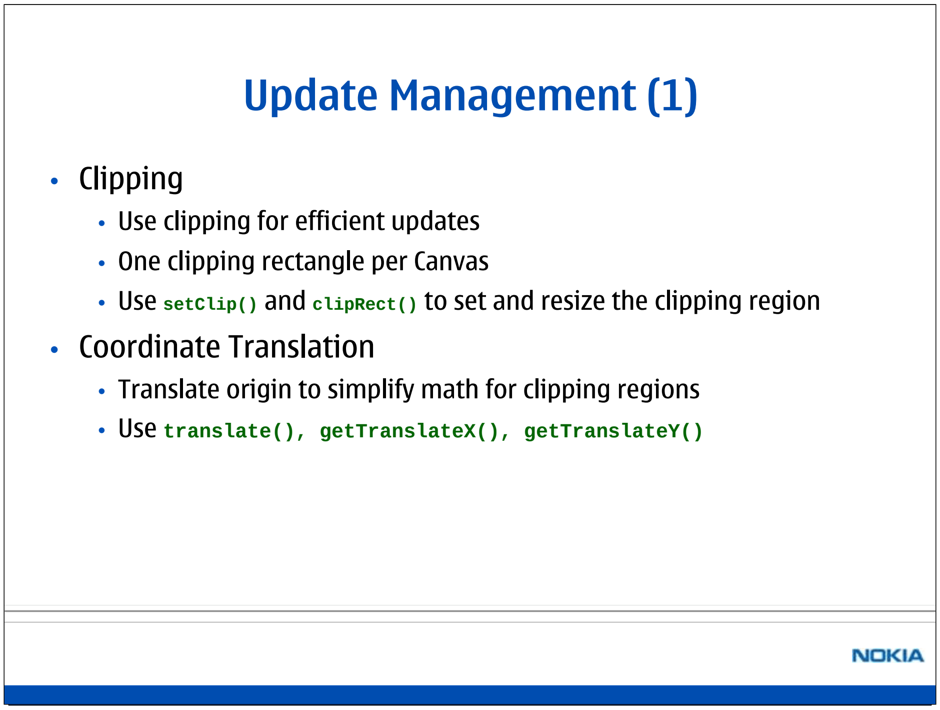
In order to get flicker-free drawing you can simulate well known double buffering techniques if your device doesn't support native double buffering. If your device is not double buffered – use isDoubleBuffered() method to find out -- then you can create an offscreen image, draw to that image and then copy those bits in the offscreen display onto the onscreen display to achieve double buffering and smooth flicker-free display updates. Note that the offscreen image should naturally be constructed mutable for this to work.

```
int width = getWidth();
int height = getHeight();
Image buffer = Image.createImage(width, height);
Graphics gc = buffer.getGraphics();
gc.drawRect(20, 20, 25, 30);
public void paint(Graphics g) {
   g.drawImage(buffer, 0, 0, 0);
}
```



Double Buffering (2)

```
public class MyCanvas extends Canvas {
   public void paint(Graphics g) {
      Image image = Image.createImage(getWidth(),
         getHeight());
      Graphics gg = image.getGraphics();
      gg.drawLine(0,0,getWidth()-5,getHeight()-5);
      g.drawImage(gg,0,0,Graphics.TOP|Graphics.LEFT)
   }
}
```



Update Management (1)

- Clipping
  - Use clipping for efficient updates
  - One clipping rectangle per Canvas
  - Use `setClip()` and `clipRect()` to set and resize the clipping region
- Coordinate Translation
  - Translate origin to simplify math for clipping regions
  - Use `translate(), getTranslateX(), getTranslateY()`

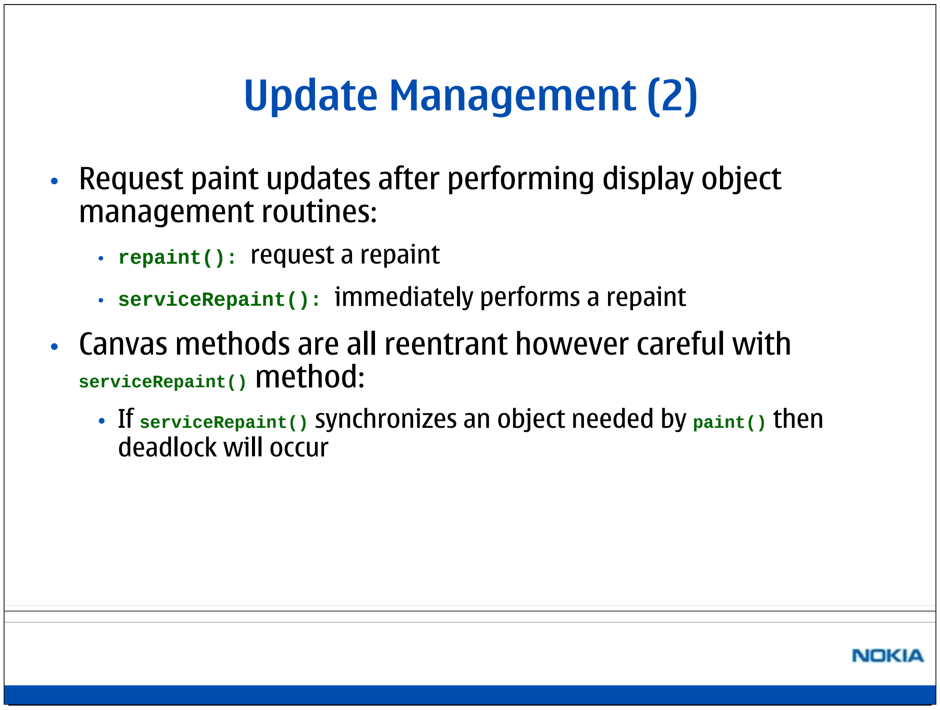
The developer should be aware that while the paint() method is fully threadsafe, care must be taken not to call objects that are synchronized by the serviceRepaint() method and needed by the paint() method. This is because serviceRepaint() does not schedule a repaint(), instead it calls paint() immediately. For example, this call structure would cause the MIDlet to deadlock:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class MyCanvas extends Canvas {
    private MyObject obj;
    public MyCanvas( MyMIDlet midlet ){
        this.midlet = midlet;
        }
         public synchronized myBadUpUpdate() {
        synchronized (this) {
               obj.update();
               serviceRepaints();
        }
         }
         protected void paint(Graphics g) {
        obj.update();
        }
}
```

Similarly, you can call methods that perform update management routines by using the callSerially() method which will invoke your update routines AFTER screen repainting occurs.

```
class MyCanvas extends Canvas implements Runnable {
   private MyObject obj;
   void doSomething(  ) {
      callSerially(this);
   }
   public void run(  ) {
      obj.update();
   }
}
```

If possible, structure your code so that management updates are independent of order so you do not have to serialize the update logic as this introduces a dependency and could impede performance maintainability as your program grows.



Update Management (2)

- Request paint updates after performing display object management routines:
  - `repaint():` request a repaint
  - `serviceRepaint():` immediately performs a repaint
- Canvas methods are all reentrant however careful with `serviceRepaint()` method:
  - If `serviceRepaint()` synchronizes an object needed by `paint()` then deadlock will occur

Nokia UI (1)

- Extends the Canvas and Graphics classes
  - FullCanvas: great for games that need the full screen
  - DirectGraphics: new primitives and greater control over primitives especially control over color
  - DirectUtils: manipulate images and translate Graphics to DirectGraphics
- Adds an alpha channel to color format for transparencies: 0xAARRGGBB
- Has many of the same capabilities of MIDP 2.0

The Nokia UI has many advanced feature with respect to low-level canvas manipulation, sound API, phone vibrations and other user functions. In fact the Nokia UI was developed to overcome the limitations of MIDP 1.0 with respect to implementing games on the currently available profiles and configurations. The purpose of this section is to explain how the Nokia UI Graphics API fits in with low-level canvas manipulation.

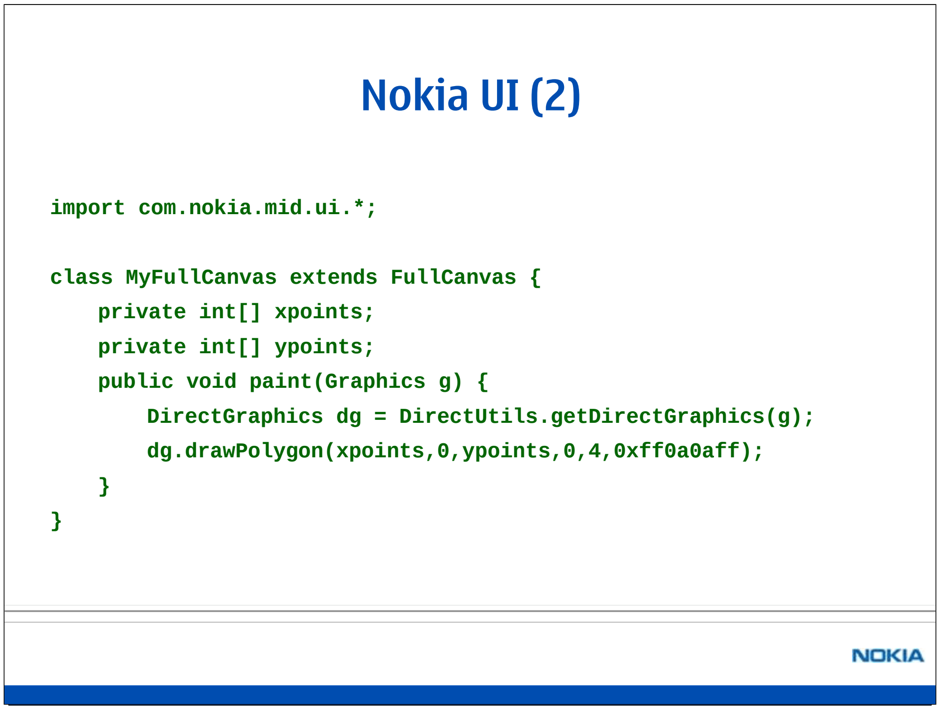
The classes/interfaces of the Nokia UI which concern the developer with low-level canvas manipulation are: DirectGraphics, FullCanvas and DirectUtils.

DirectGraphics extends MIDP 1.0 with new shapes such as polygons and triangles which can be drawn and filled, supports advanced image manipulation such as rotation or inversion, supports alpha channels as well as raw pixel data management used to obtain pixel data directly from the graphics context and draw pixels directly to the graphics context -- hence the name DirectGraphics. With the addition of alpha channels, expressing colors takes the form 0xAARRGGBB and the high order bits signifying the alpha bits. An alpha value of 0 represents a completely transparent pixel.

The DirectUtils class has utility methods for creating images from image offsets, creating mutable images and images with transparencies. The Nokia UI API further has the notion of a full screen canvas which means that the developer has control over the entire display area. This is a great class to use for games.

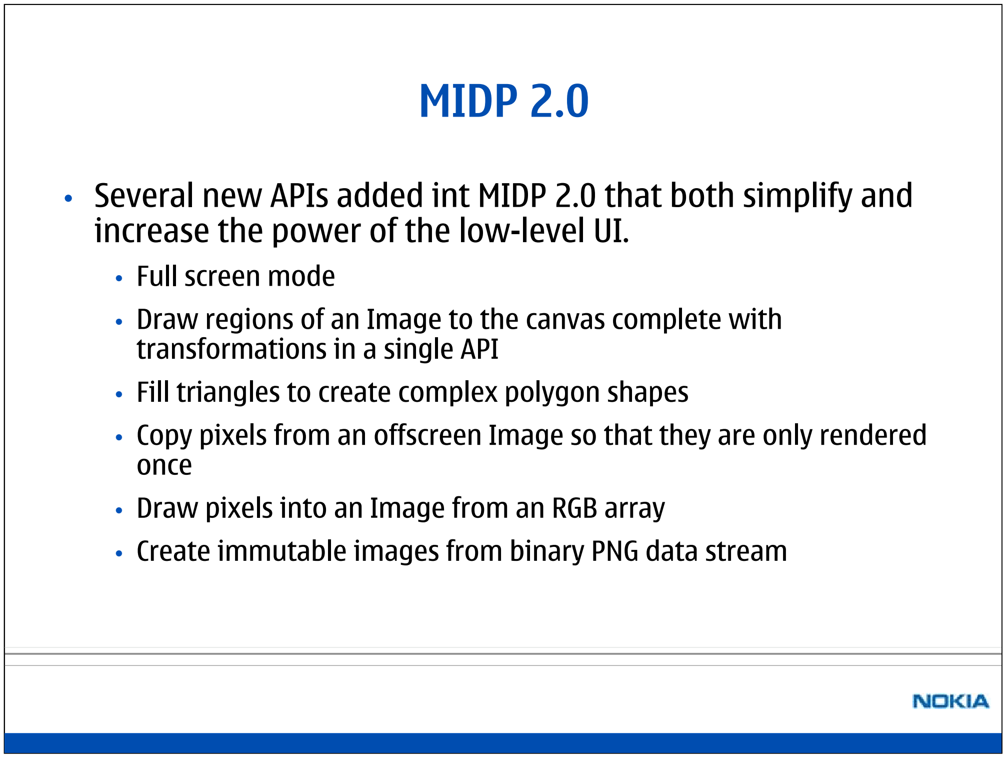
The following code shows how to use the Nokia UI API:

```
import javax.microedition.midlet.*;
import com.nokia.mid.ui.*;

public class TDCanvas extends FullCanvas
{
        private DirectGraphics dg = null;
        int[] xpoints = null;
        int[] ypoints  = null;

        public void paint(Graphics g) {
                dg = DirectUtils.getDirectGraphics(g);
                updatePoints(xpoints,ypoints);
                dg.drawPolygon(xpoints,0,ypoints,0,4,0xff0a550a);
        }
        private void myUpdate(int[] xpoints, int[] ypoints) {
                // some clever code here
        }
}
```



Nokia UI (2)

```
import com.nokia.mid.ui.*;

class MyFullCanvas extends FullCanvas {
    private int[] xpoints;
    private int[] ypoints;
    public void paint(Graphics g) {
        DirectGraphics dg = DirectUtils.getDirectGraphics(g);
        dg.drawPolygon(xpoints,0,ypoints,0,4,0xff0a0aff);
    }
}
```



MIDP 2.0

- Several new APIs added int MIDP 2.0 that both simplify and increase the power of the low-level UI.
  - Full screen mode
  - Draw regions of an Image to the canvas complete with transformations in a single API
  - Fill triangles to create complex polygon shapes
  - Copy pixels from an offscreen Image so that they are only rendered once
  - Draw pixels into an Image from an RGB array
  - Create immutable images from binary PNG data stream

MIDP 2.0 adds several APIS to the low-level UI that makes it much easier and more powerful to use. Certain aspects are specifically geared towards games and others are convenience methods that implement functionality that was previously outside the scope of developer control – notably full canvas mode.

Image

One of the notable improvements in MIDP 2.0 is in the Image class. The ability to create Image objects directly from an InputStream allows the developer to store images as binary data. Also, Image objects can now use the getGraphics() method which returns a Graphics object and all the Graphic object drawing primitives are available so the developer can draw directly on the Image. Perhaps the most interesting improvement is the addition of the ability to create and modify Image objects created as integer arrays. The createRGBImage() and getRGB() methods can be used in conjunction to manipulate Images with very fine grained precision.

Graphics

The new Graphics primitives are expressive and powerful. The goal is to allow the developer more precise control of how primitives are rendered to the screen. While some of the new APIs are complex, an understanding of how they work should definitely simply much low-level UI code.

The simplest of the new APIs is the fillTriangle() method which takes 3 (X,Y) coordinates as input and renders the triangle using the current color. This is especially useful for creating polygon routines that could be used to create a software 3D rendered until JSR-184 (Mobile 3D API) becomes available.

Next, is the drawRegion() API can take a transformation parameter to fix the orientation of an Image drawn on the Canvas. For example:

```
        Graphics g = getGraphics();
        g.drawRegion(image, 2, 2, 8, 6, Sprite.TRANS_MIRROR_ROT90,
          9, 9, Graphics.LEFT | Graphics.TOP);
```

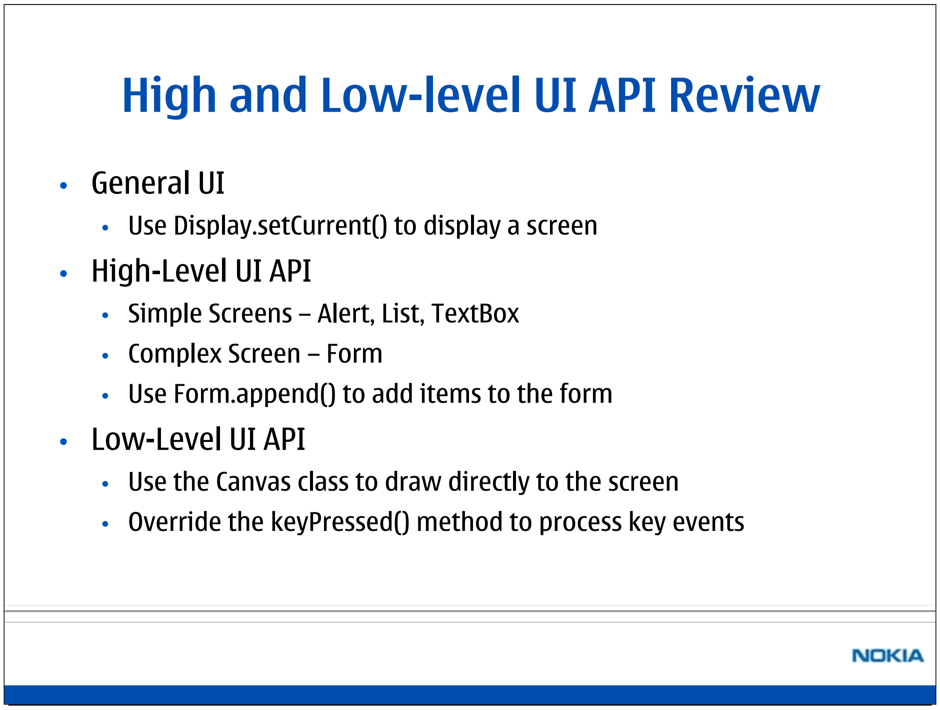
would draw the image, rotated 90 degrees and then mirrored. This is really useful for keeping the size of JAR files down as you will only need one orientation for a bitmap image, and then can use the transformation flags for drawing the image with a great deal of flexibility.

The drawRGB() method allows the developer to draw RGB color values directly to a rectangular region on the Canvas. For example

```
int[] rgbArray = {
    0x123ABC, 0x121212, 0x0ABBAA, 0x000000, 0xFFFFFF, 0xBDBDEE,
    0x123ABC, 0x121212, 0x0ABBAA, 0x000000, 0xFFFFFF, 0xBDBDEE,
};
g.drawRGB(rgb, 1, 4, 4, 2, 3, 2, false);
```

would draw the pixels from offset 1, read in the next 4 pixels and draw them at coordinate (4,2), 3 pixels wide and 2 deep. The offset value may not be long enough (4 in this case) so additional pixel data is read from the array in order to complete drawing of the region.

Finally, the copyRegion() method compliments the Graphics object by allowing rectangular areas of pixels to be copied from one area of an image to another without having to render the same pixels again. This is really useful for implementing scrolling in a canvas



High and Low-level UI API Review

- General UI
  - Use Display.setCurrent() to display a screen
- High-Level UI API
  - Simple Screens – Alert, List, TextBox
  - Complex Screen – Form
  - Use Form.append() to add items to the form
- Low-Level UI API
  - Use the Canvas class to draw directly to the screen
  - Override the keyPressed() method to process key events

True or False?

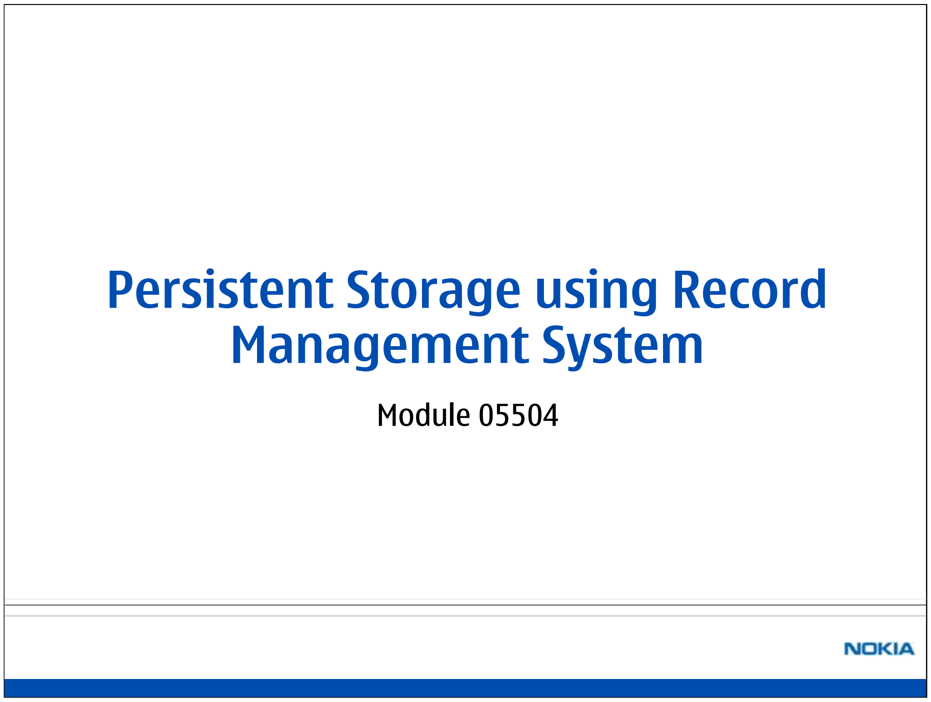
- ***Use the high-level API whenever possible because it is portable***
- ***If you need special screen layouts (e.g. games) then use the low-level API***
- ***Update the Canvas intelligently or your MIDlet will be slow***
- ***The KVM is fast at number crunching but slow at accessing the native OS calls used for drawing***

High-Level UI API Review Questions

- ***What are tickers used for?***
- ***What are abstract commands?***
- ***How to you capture user events?***
- ***How does the look-and-feel of the High Level API vary over platforms?***
- ***How can you create custom widgets?***
- ***What is a Screen?***
- ***What are the different types of Screens?***
- ***How many screens can be attached to a display at the same time?***
- ***How do you override the drawing and customize a Screen or Form Item?***

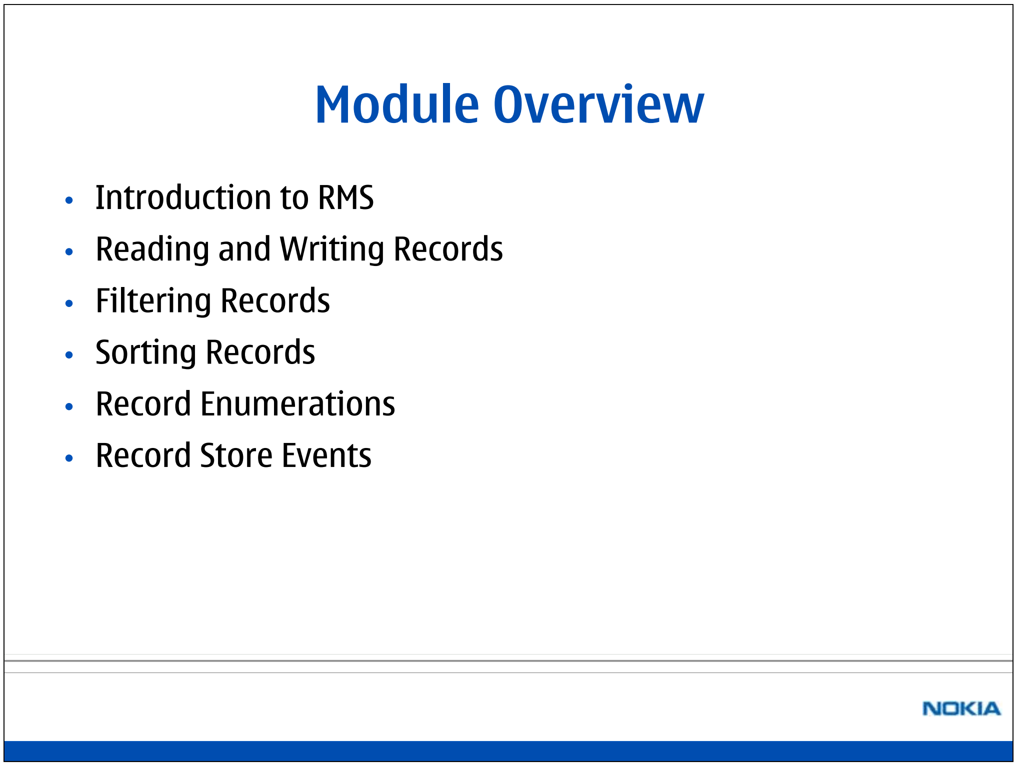
Low-Level UI API Review Questions

- ***In addition to the paint event, why would the developer wish to override the key event even if there was no interaction with the MIDlet via key events?***
- ***Is the low-level graphics API threadsafe? What concurrency issues must the developer always be watchful of?***
- ***How would you implement double buffering if your device doesn't automatically support it?***
- ***Why would it make sense to implement a MIDlet only with the low-level API even though multiple display "cards" or available with each MIDlet deck?***
- ***What is the main reason why the Nokia UI is implemented the way it is?***



Persistent Storage using Record Management System

Module 05504



Module Overview

- Introduction to RMS
- Reading and Writing Records
- Filtering Records
- Sorting Records
- Record Enumerations
- Record Store Events

Introduction. This provides a summary of the Record Management System (RMS) of MIDP. It explains what the RMS is, the architecture of the RMS package, and gives descriptions of the classes, interfaces and exceptions included within this package. A summary of the concept of a record store is also presented.

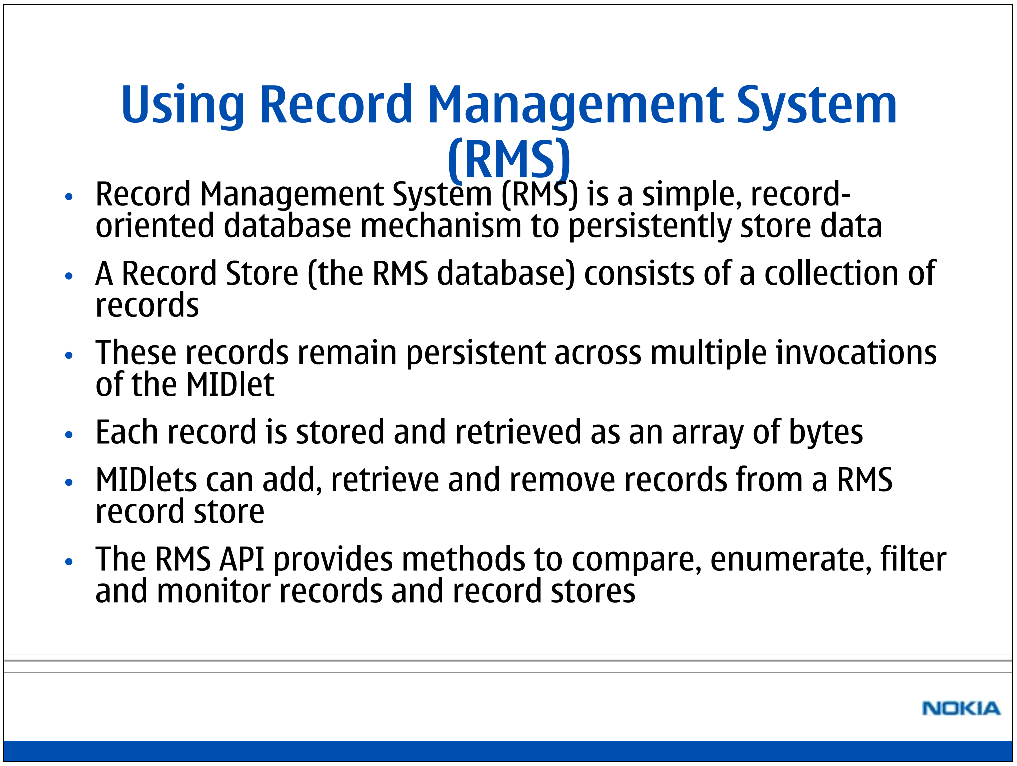
Reading and Writing Records. This section concentrates on how the developer can use the classes provided in the RMS package to manipulate record stores. Overviews of the fundamental record tasks (reading, writing, deleting and updating) are presented with example code, and description of usage.

Filtering Records. Details of how records can be filtered by examining records and seeing if they meet specific criteria is covered here. An overview is presented that explains the usage of the RecordFilter interface. An example then follows this.

Sorting Records. This covers how records can be sorted by comparing two records and determining their relative order. An overview is presented that explains the usage of the RecordComparator interface. An example then follows this.

Record Enumerations. This section first gives an overview of the RecordEnumeration interface that can be used to enumerate over a number of records. Example code is then presented to show how to use the RecordEnumeration with the RecordFilter and RecordComparator.

Record Store Events. This section first gives an overview of record store events. It explains why these can be useful, gives examples of their usage and what classes are used to implement them. This is then followed by some example code.



Using Record Management System (RMS)

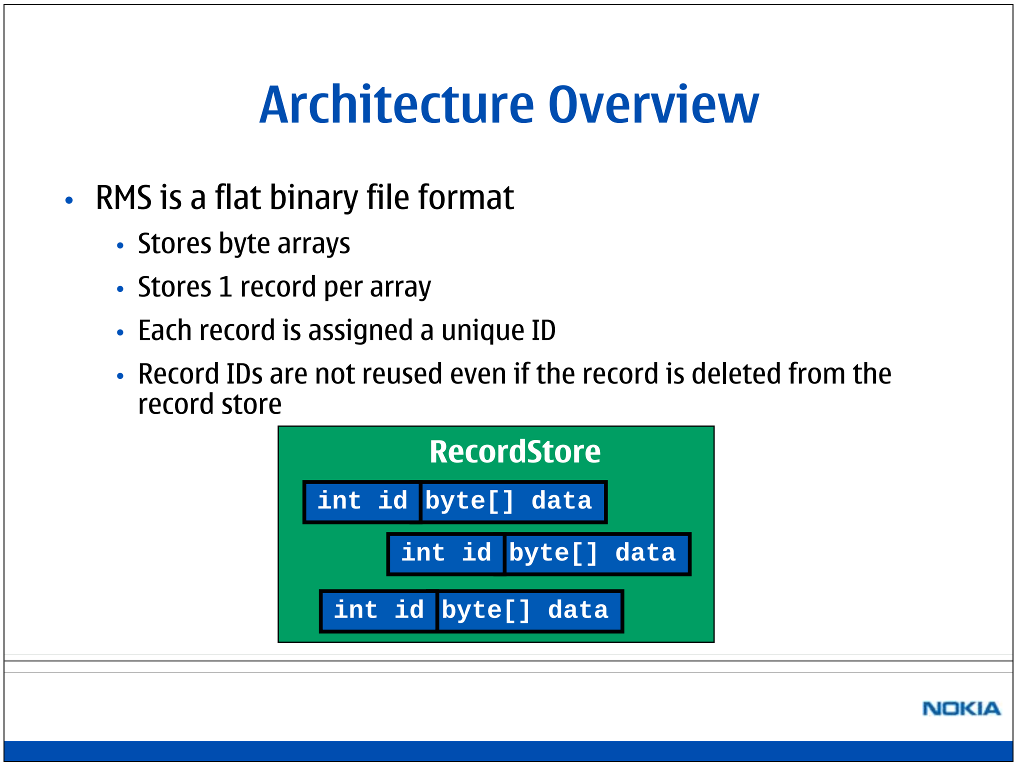
- Record Management System (RMS) is a simple, record-oriented database mechanism to persistently store data
- A Record Store (the RMS database) consists of a collection of records
- These records remain persistent across multiple invocations of the MIDlet
- Each record is stored and retrieved as an array of bytes
- MIDlets can add, retrieve and remove records from a RMS record store
- The RMS API provides methods to compare, enumerate, filter and monitor records and record stores

MIDP defines the Record Management System (RMS), which is a mechanism for MIDlets to persistently store data and retrieve it later. A MIDP database (or a record store) consists of a collection of records that remain persistent after the MIDlet exits. When you invoke the MIDlet again, it can retrieve data from the persistent record store.

RMS has limitations with regard to the amount of data that can be stored and general depends on your target devices data sheet specifications. Make sure you understand this limit and check with the target device's data sheet to know for certain what the hard limits are. MIDlets can share data between them using RMS only if they are part of the same MIDlet suite. In MIDP 2.0 this limitation is relaxed within the confines of the new security model. Otherwise, sharing data with RMS any other way violates the MIDlet security manager and is not possible.

In a record-oriented approach, Java ME RMS comprises multiple record stores. Each record store can be visualized as a collection of records, which will remain persistent across multiple invocations of the MIDlet. The device platform is responsible for making its best effort to maintain the integrity of the MIDlet's record stores throughout the normal use of the platform, including reboots, battery changes, etc.

Each record in a record store is an array of bytes and has a unique integer identifier. The format of RMS is purposely kept simple – each record store is given a unique name and the stores in the records are simply rows, with a unique ID, followed by a series of bytes. Each row can be a different length. It is the responsibility of the programmer to determine what the MIDlet logic is for interpreting the stored bytes. Most programmers are used to 0 indexed arrays – RMS begins the row count at 1 so the programmer has to be careful about accessing the correct row. Also, note that record Ids are unique for the life time of a record store which means that if a record is deleted then it's ID is never recycled. Record ID's are 32 bit values and continued ad infinitum until the maximum value of the storage capacity is reached.



Architecture Overview

- RMS is a flat binary file format
  - Stores byte arrays
  - Stores 1 record per array
  - Each record is assigned a unique ID
  - Record IDs are not reused even if the record is deleted from the record store

RecordStore

int id byte[] data

int id byte[] data

int id byte[] data

To use RMS, the developer needs to import the javax.microedition.rms package. This package has one main class called RecordStore. This class represents a record store, which consists of a collection of records that will remain persistent across multiple invocations of the MIDlet.

Interfaces

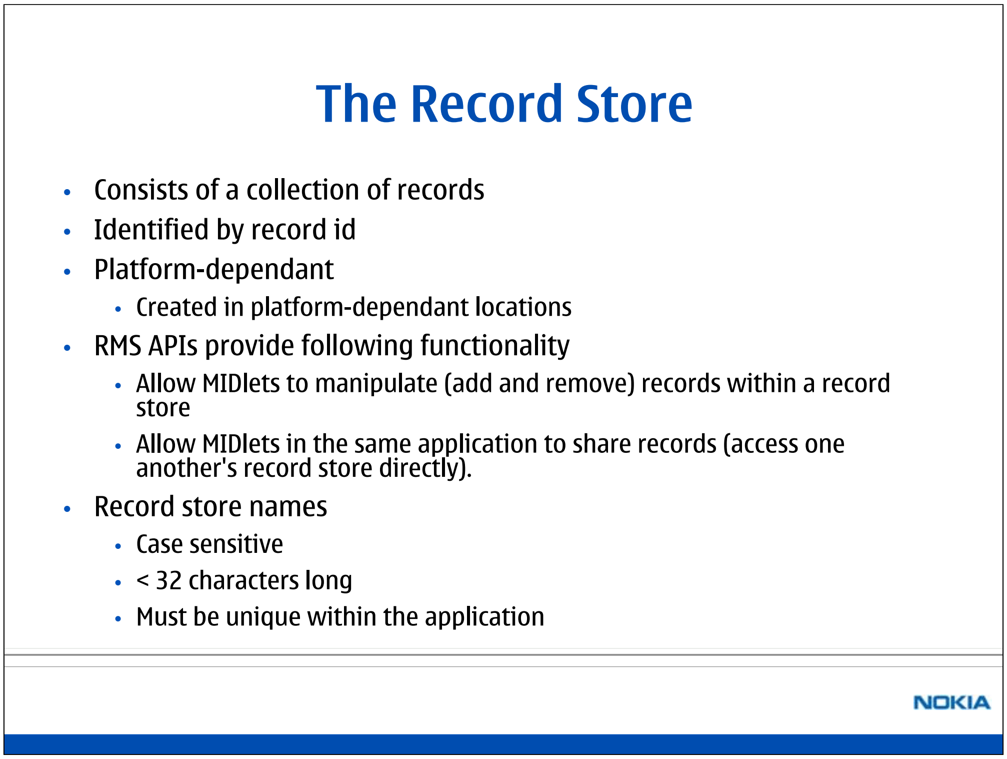
- ***RecordComparator: Defines a comparator to compare two records.***
- ***RecordEnumeration: Represents a bidirectional record enumerator.***
- ***RecordFilter: Defines a filter to examine a record and checks if it matches based on a criteria defined by the application.***
- ***RecordListener: Receives records that were added, changed, or deleted from a record store.***

Classes

- ***RecordStore: Represents a record store.***

Exceptions

- ***InvalidRecordIDException: Thrown to indicate the RecordID is invalid.***
- ***RecordStoreException: Thrown to indicate a general exception was thrown.***
- ***RecordStoreFullException: Thrown to indicate the record store file system is full.***
- ***RecordStoreNotFoundException: Thrown to indicate the record store could not be found.***
- ***RecordStoreNotOpenException: Thrown to indicate an operation on a closed record store.***



The Record Store

- Consists of a collection of records
- Identified by record id
- Platform-dependant
  - Created in platform-dependant locations
- RMS APIs provide following functionality
  - Allow MIDlets to manipulate (add and remove) records within a record store
  - Allow MIDlets in the same application to share records (access one another's record store directly).
- Record store names
  - Case sensitive
  - < 32 characters long
  - Must be unique within the application

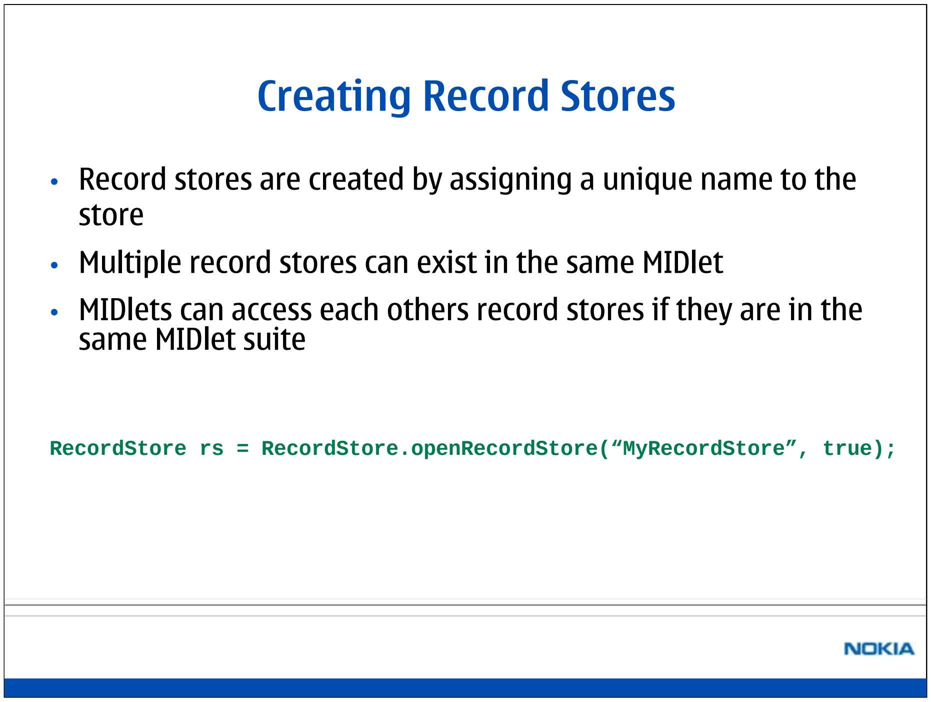
NOKIA

A record store consists of a collection of records that are uniquely identified by their record ID, which is an integer value. The record ID is the primary key for the records. The first record has an ID of 1, and each additional record is assigned an ID that is the previous value plus 1.

Record stores (binary files) are platform-dependent because they are created in platform-dependent locations. MIDlets within a single application (a MIDlet suite) can create multiple record stores (database files) with different names. The RMS APIs provide the following functionality:

- ***Allow MIDlets to manipulate (add and remove) records within a record store***
- ***Allow MIDlets in the same application to share records (access one another's record store directly)***

Record store names are case sensitive, and cannot be more than 32 characters long. Also, a MIDlet cannot create two record stores with the same name in the same application, but it can create a record store with the same name as a MIDlet in another application.



Creating Record Stores

- Record stores are created by assigning a unique name to the store
- Multiple record stores can exist in the same MIDlet
- MIDlets can access each others record stores if they are in the same MIDlet suite

```
RecordStore rs = RecordStore.openRecordStore(“MyRecordStore”, true);
```

The first step towards writing data to the record store is to open the record store with a unique name. If the record store name does not exist in RMS, then a new record store is created; otherwise the existing record store is opened. The code in the slide shows how this can be done.

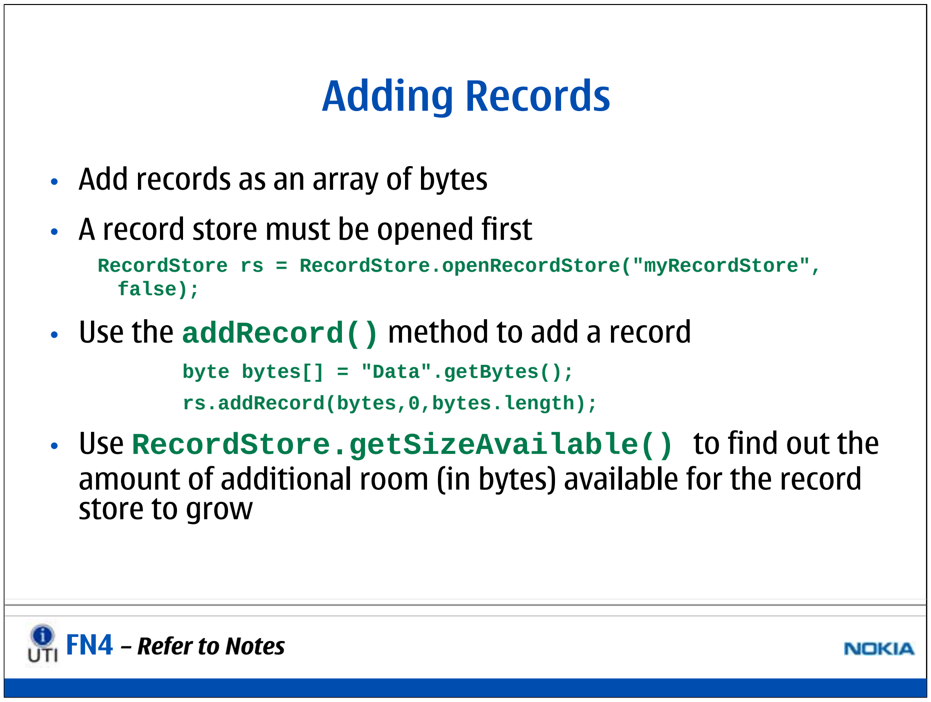
The openRecordStore method takes two arguments, namely a String recordStoreName, and a boolean createIfNecessary. The first argument specifies the name of the store that needs to be opened. The second argument determines whether this method should create the specified store if it cannot be found. Note that record store names must be unique within a MIDlet suite.

It's important to catch exceptions when using RMS especially for writing as you may run out of RMS allotted storage space in the device. The openRecordStore method throws the exceptions RecordStoreException, RecordStoreFullException, RecordStoreNotFoundException.

RecordStoreException - if a record store related exception occurred. This is a general exception that could be thrown, for example, if when opening the record store, there was an interruption in the stream.

RecordStoreNotFoundException - if the record store could not be found. This exception could be thrown if the openRecordStore method is called with its first argument set to a record store name that doesn’t exist its second argument set to false (don’t create if the store cannot be found).

RecordStoreFullException - if the operation cannot be completed because the record store is full.



Adding Records

- Add records as an array of bytes
- A record store must be opened first

```
RecordStore rs = RecordStore.openRecordStore("myRecordStore",
  false);
```

- Use the `addRecord()` method to add a record

```
byte bytes[] = "Data".getBytes();
rs.addRecord(bytes,0,bytes.length);
```

- Use `RecordStore.getSizeAvailable()` to find out the amount of additional room (in bytes) available for the record store to grow

FN4 – ***Refer to Notes***

The MIDlet invokes the addRecord() method of javax.microedition.rms.RecordStore class to insert a new record into the record store. This is a blocking atomic operation and returns the recordId for the new record. The record is written to persistent storage before the method returns.

public int addRecord(byte[] data, int offset, int numBytes) inserts a record represented by an array of bytes data with offset as its starting index and numBytes as its length.

First you have to get the data into a byte array format. For example:

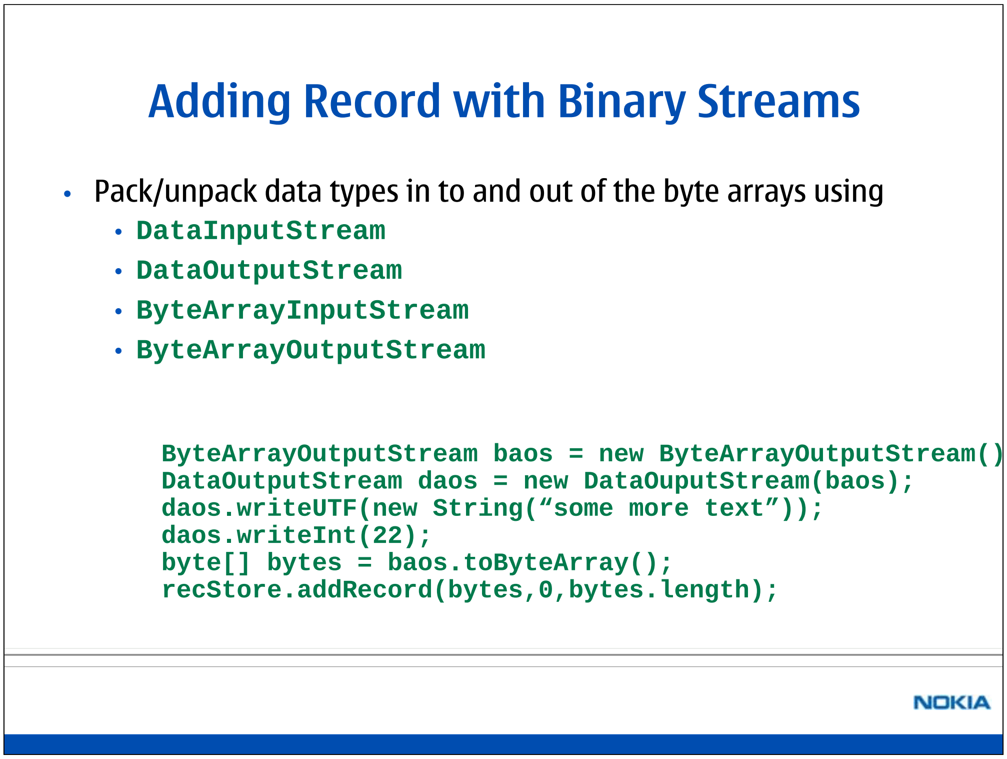
```
String string  = “Some Data”;
bytes[] bytes = string.getBytes();
try {
       recordStore.addRecord(bytes, 0, bytes.length);
} catch (Exception e) {}
```

That's it! Note that the addRecord() method allows you to specify the offsets and how far to read into the array for writing. One thing to keep in mind when writing records is that RMS does not provide locking so that concurrent writes must be managed by the application logic.

The addRecord method throws the following exceptions:

- ***RecordStoreNotOpenException - if the record store is not open.***
- ***RecordStoreException - if a different record store-related exception occurred.***
- ***RecordStoreFullException - if the operation cannot be completed because the record store has no more room.***
- ***NullPointerException - if data is null but numBytes is greater than zero.***

FN4 - The application must not (or attempt to) cause harm to system applications or data stored in the terminal.
If the flash memory becomes full, it may affect the execution of system applications on the device. Your application can prevent the flash memory from being filled up by checking the amount of space that is currenctly available. As the slide explains, this can be done by calling the getSizeAvailable() method on the RecordStore. This returns the amount of additional room (in bytes) available for the record store to grow.



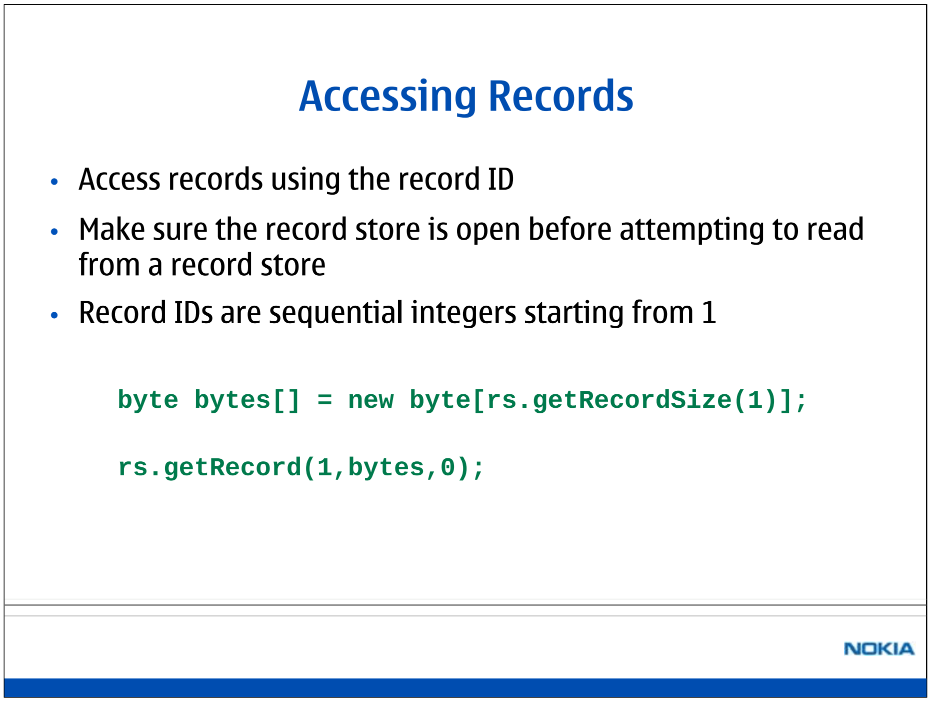
Adding Record with Binary Streams

- Pack/unpack data types in to and out of the byte arrays using
  - DataInputStream
  - DataOutputStream
  - ByteArrayInputStream
  - ByteArrayOutputStream

```
ByteArrayOutputStream baos = new ByteArrayOutputStream()
DataOutputStream daos = new DataOuputStream(baos);
daos.writeUTF(new String(“some more text”));
daos.writeInt(22);
byte[] bytes = baos.toByteArray();
recStore.addRecord(bytes,0,bytes.length);
```

The previous RMS examples demonstrate how to use the RMS API with text only – what about if we want to store mixed data in the same record? We can use the streams classes to do this and gain the advantage of binary formats as well as increased efficiency. The way to do this is to use DataOuputStream and DataInputStream. The example is the slide shows how to write data to RMS using streams.

You construct a DataOutputStream for writing the record to the record store, then you convert the ByteArrayOutputStream to a byte array, and finally you invoke addRecord() to add the record to the record store.



Accessing Records

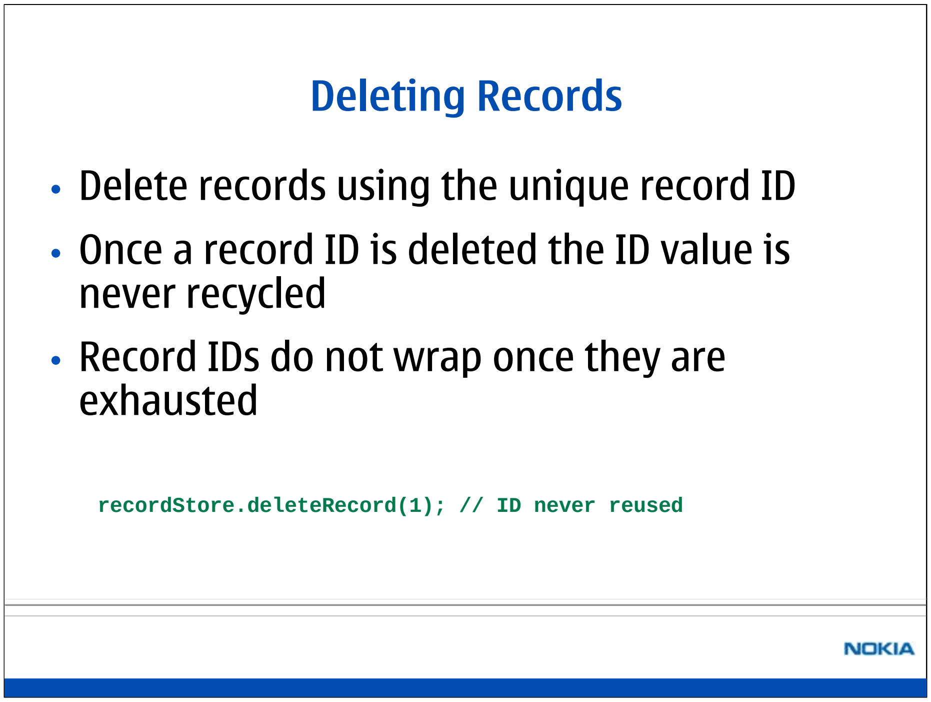
- Access records using the record ID
- Make sure the record store is open before attempting to read from a record store
- Record IDs are sequential integers starting from 1

```
byte bytes[] = new byte[rs.getRecordSize(1)];

rs.getRecord(1,bytes,0);
```

Reading records is accomplished in much the same way as writing records. As with writing records, the record store must first be open and the programmer is responsible for closing the record store after the read is complete. This is an important step as that way the consistency of the record store will be maintained even in the event that there is a power down during the course of the MIDlet session. The programmer is responsible for providing a byte array large enough to hold the requested information and then the array is unpacked according to the program logic. Here's another example:

```
try {
      byte[] bytes;
      for (int i = 1; i < recStore.getNumRecords(); i++) {
            bytes = new byte[recStore.getRecordSize(i)];
            recStore.getRecord(i, bytes, 0);
            saveMe(new String(bytes,0,bytes.length); // save it
      }
} catch (Exception e) {}
```

Note that we start the for loop with index 1 as indicated above because that is where RMS sets the first record index start point.



Deleting Records

- Delete records using the unique record ID
- Once a record ID is deleted the ID value is never recycled
- Record IDs do not wrap once they are exhausted

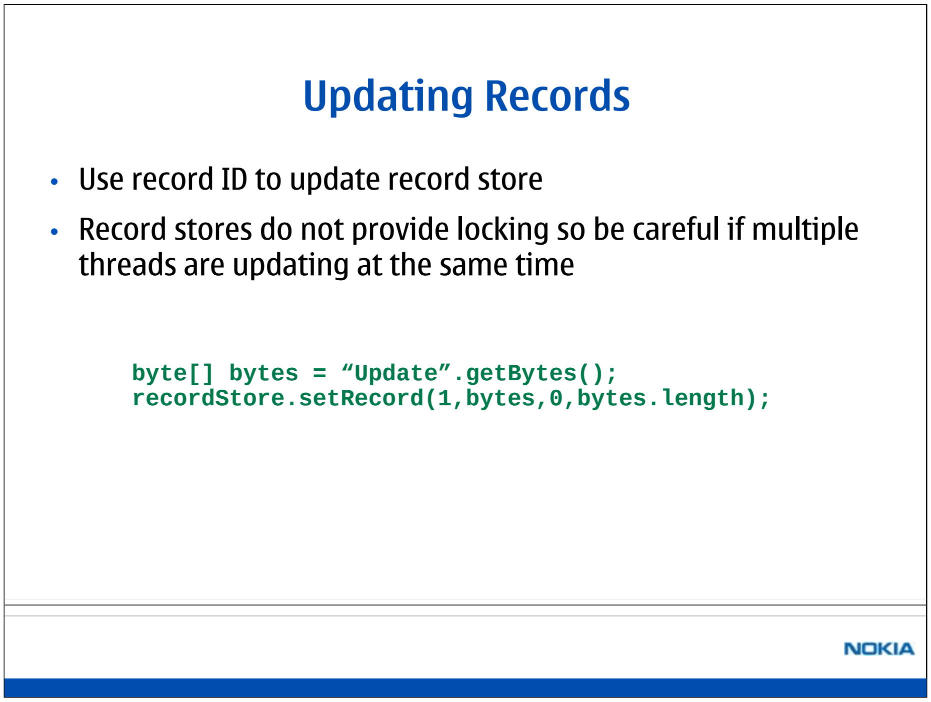
```
recordStore.deleteRecord(1); // ID never reused
```

To delete a record from the record store, you have to know the record ID for the record to be deleted. To delete the record, use the deleteRecord() method. This method takes an integer as a parameter, which is the record ID of the record to be deleted.

Recall that record IDs are unique and never recycled, thus once a record is deleted, the record ID is lost and there is no wrap around of IDs.

There is no method to get the record ID. To work around this, every time you create a new record, add its record ID to a vector like this:

```
       Vector recordIDs = new Vector();
       int lastID = 1;
       //Add a record....parameters are missing here
       db.addRecord();
       // Now add the ID to the vector
       recordIDs.addElement(new Integer(++lastID));
       //Now, to delete a record, find the record ID of the record you
want to delete:
       Enumeration IDs = recordIDs.elements();
       while(IDs.hasMoreElements()) {
              int id = ((Integer) IDs.nextElement()).intValue();
              //Compare to see if this is the record you want
              //Then call
              db.deleteRecord(id);
  }
```



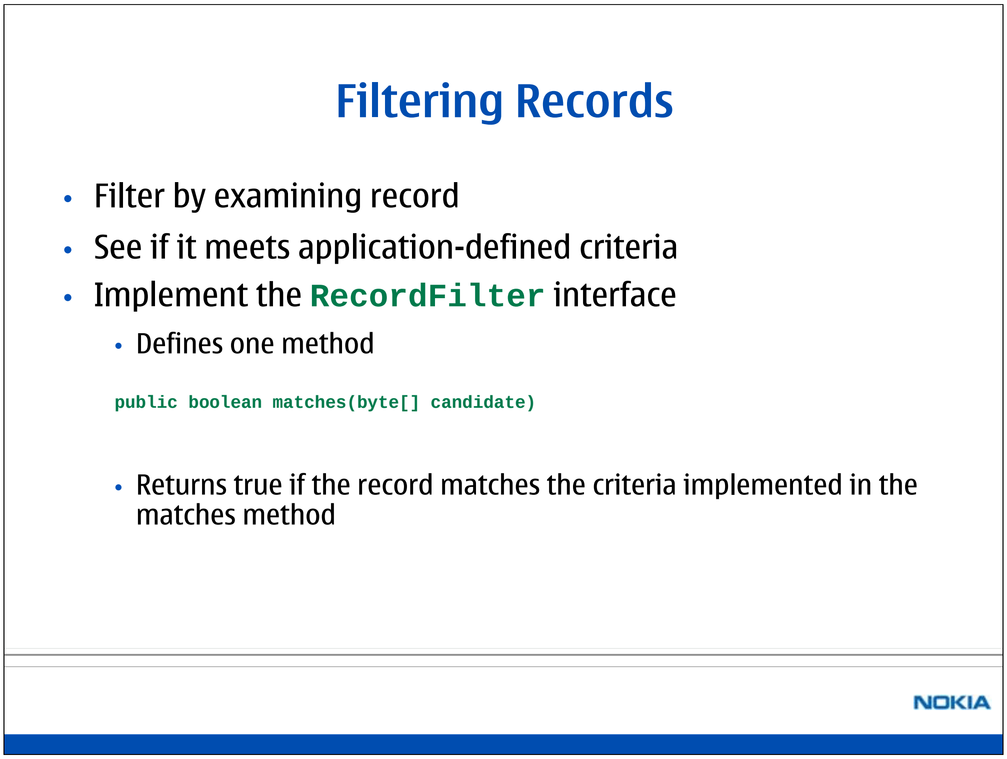
Updating Records

- Use record ID to update record store
- Record stores do not provide locking so be careful if multiple threads are updating at the same time

```
byte[] bytes = “Update”.getBytes();
recordStore.setRecord(1,bytes,0,bytes.length);
```

Updating a particular record involves getting a handle for that record and setting new information.

We can update records with new information by using the record ID of the record we want to update. We provide the record ID and then make sure that the data is in the correct format to allow the update to proceed.   This is done simply enough by converting the data to a byte array as demonstrated in the slide.



Filtering Records

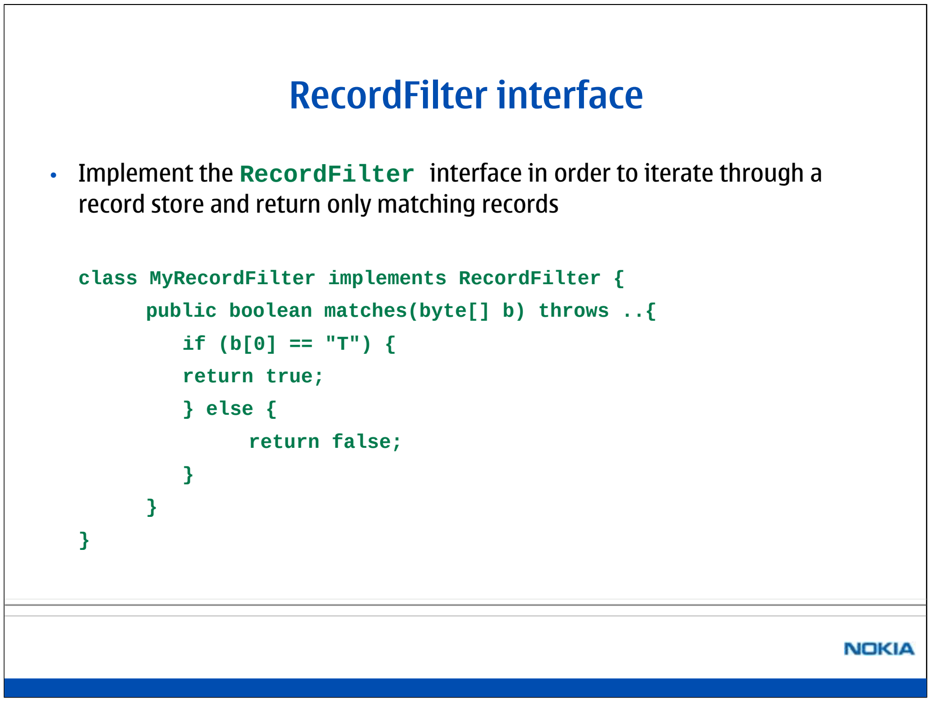
- Filter by examining record
- See if it meets application-defined criteria
- Implement the `RecordFilter` interface
  - Defines one method

```
public boolean matches(byte[] candidate)
```

  - Returns true if the record matches the criteria implemented in the matches method

To be able to filter records, you need to be able to check a record for certain criteria. The interface RecordFilter is included in the RMS package to facilitate this. To use it, you should implement the interface, and provide an implementation for the matches method.

The matches method should be implemented to return true if the record whose content is passed to it meets the filter criterion and false if it does not.



RecordFilter interface

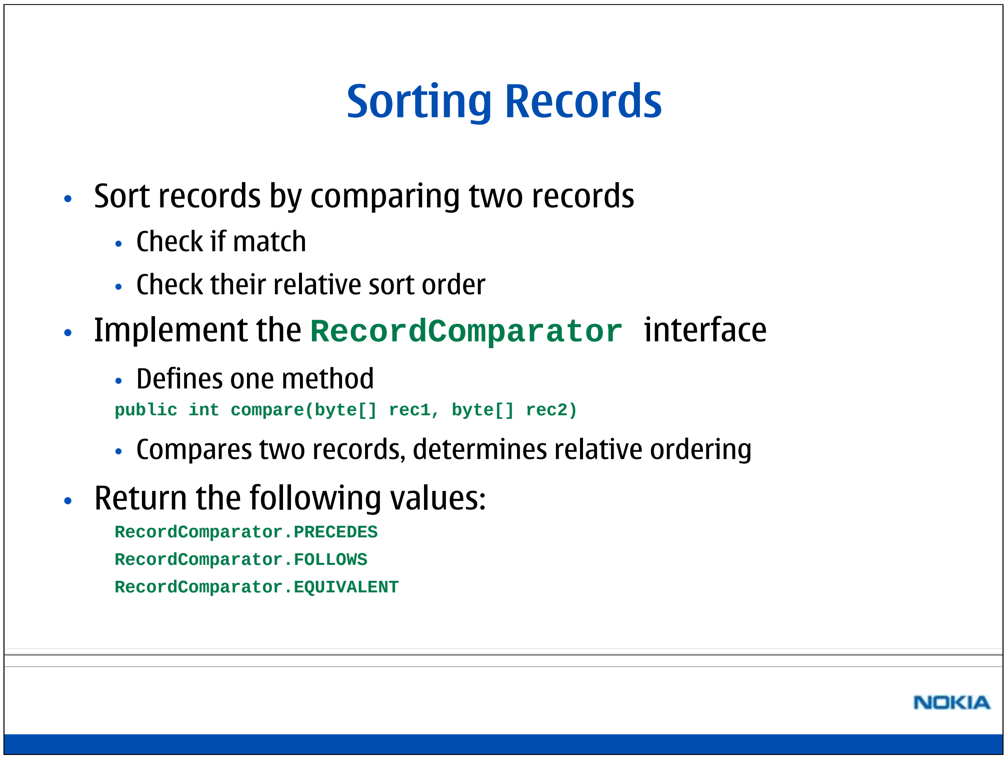
- Implement the `RecordFilter` interface in order to iterate through a record store and return only matching records

```
class MyRecordFilter implements RecordFilter {
      public boolean matches(byte[] b) throws ..{
         if (b[0] == "T") {
         return true;
         } else {
              return false;
         }
      }
}
```

Implementing filters requires that the programmer create a filter class that implements the filtering logic. First we need to create a class the implements the RecordFilter logic:

```
class MyFilter implements RecordFilter {
   public boolean matches(byte[] matchArg) {
      String str = new String(matchArg);
      if (str.indexOf(“found”) != -1) {
        return true;
      } else {
        return false;
      }
   }
}
```

Now the implemented interface method in the above example will return true if there is a substring called “found" in any of the records.



Sorting Records

- Sort records by comparing two records
  - Check if match
  - Check their relative sort order
- Implement the `RecordComparator` interface
  - Defines one method

```
public int compare(byte[] rec1, byte[] rec2)
```

  - Compares two records, determines relative ordering
- Return the following values:

```
RecordComparator.PRECEDES
RecordComparator.FOLLOWS
RecordComparator.EQUIVALENT
```

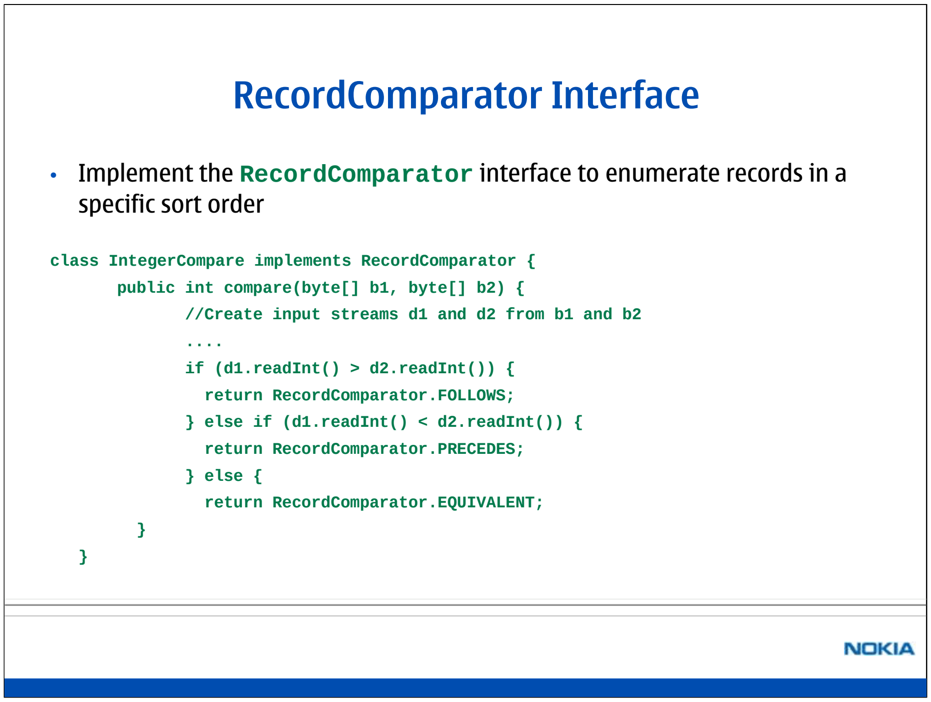
To be able to sort records, you need to be able to determine relative orders between them. The interface RecordComparator is included in the RMS package to facilitate this. To use it, you should implement the interface, and provide an implementation for the compare method.

The compare method should return a value to indicate the ordering of the two records being compared. The constants FOLLOWS, PRECEDES, and EQUIVALENT are defined in the RecordComparator interface and have the following meanings:

•FOLLOWS: Its value is 1 and means the left parameter follows the right parameter in terms of search or sort order.

•PRECEDES: Its value is -1 and means the left parameter precedes the right parameter in terms on search or sort order.

•EQUIVALENT: Its value is 0 and means the two parameters are the same.



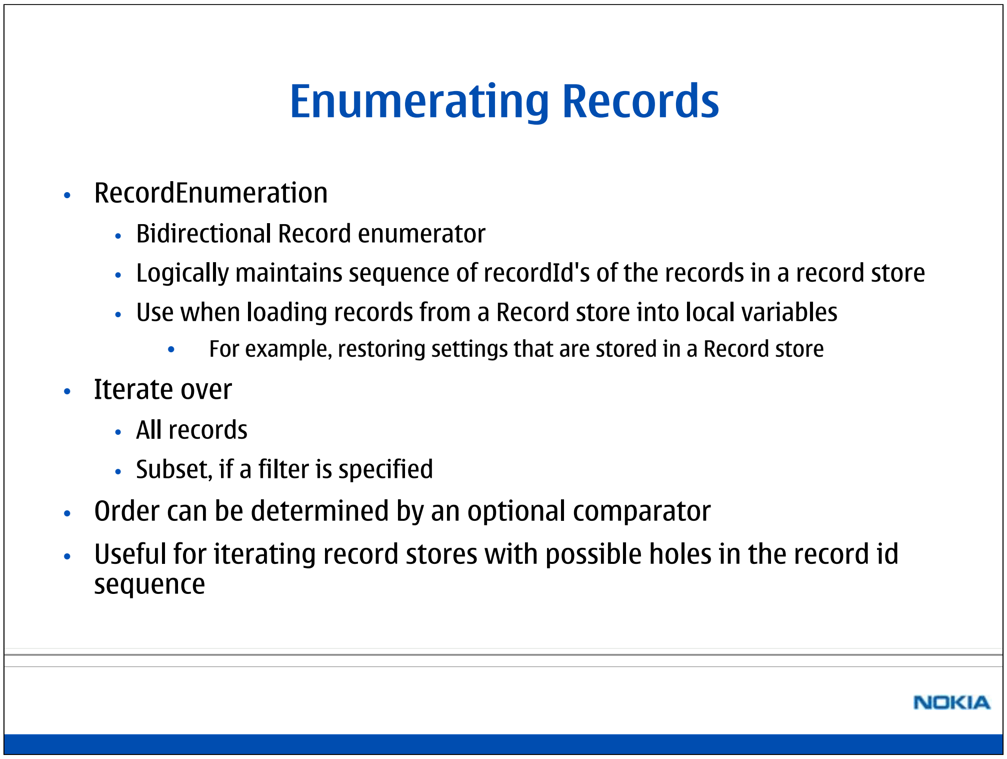
RecordComparator Interface

- Implement the `RecordComparator` interface to enumerate records in a specific sort order

```
class IntegerCompare implements RecordComparator {
       public int compare(byte[] b1, byte[] b2) {
              //Create input streams d1 and d2 from b1 and b2
              ....
              if (d1.readInt() > d2.readInt()) {
                return RecordComparator.FOLLOWS;
              } else if (d1.readInt() < d2.readInt()) {
                return RecordComparator.PRECEDES;
              } else {
                return RecordComparator.EQUIVALENT;
         }
   }
```

Similarly to filtering, we can use the RecordComparator class to create custom routines for sorting the result set. We do this by first implementing the RecordComparator interface.

We've used greater than and less than comparison operators to keep the example simple and in a real application you'll have the ability to defines complex sorts (and filters) simply by implementing the interface and returning the correct RecordComparator return values.



Enumerating Records

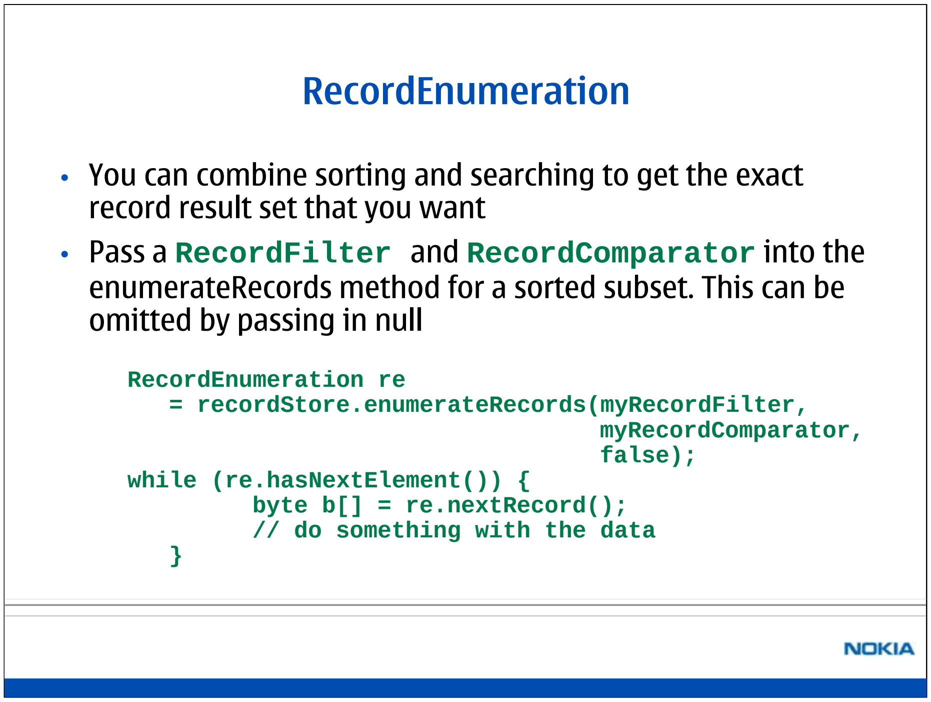
- RecordEnumeration
  - Bidirectional Record enumerator
  - Logically maintains sequence of recordId's of the records in a record store
  - Use when loading records from a Record store into local variables
    - For example, restoring settings that are stored in a Record store
- Iterate over
  - All records
  - Subset, if a filter is specified
- Order can be determined by an optional comparator
- Useful for iterating record stores with possible holes in the record id sequence

A RecordEnumeration is like a java.util.Enumeration, in that it allows you to iterate over a collection of objects. It is, however, more powerful than Enumeration because you can traverse the collection either forward or backward, and you can change direction at any time.

They are particularly useful for iterating record stores with possible holes in the record id sequence. Record Ids are not recycled thus if we delete a record that we've already inserted and then try to access that record, then a javax.microedition.rms.InvalidRecordIDException is thrown. This exception will occur regardless of whether the record store has been “committed” or not as RMS does not support the notion of transactions and rollbacks. The following code will throw and exception even though we have not closed the record store:

```
// remove a record that we inserted
try {
             recordStore.deleteRecord(10);
} catch (Exception e) {}

// access it – this will throw an exception
byte[] bytes[aValidLength];
try {
             recStore.getRecord(10, bytes, 0);
} catch {InvalidRecordIDException e) {}
```

So how can we manage tasks such as iterating through a record store with possible “holes” in the record ID sequence, without resorting to potentially complicated code that manages thrown exceptions? The answer is to use the RecordEnumeration class.



RecordEnumeration

- You can combine sorting and searching to get the exact record result set that you want
- Pass a `RecordFilter` and `RecordComparator` into the enumerateRecords method for a sorted subset. This can be omitted by passing in null

```
RecordEnumeration re
   = recordStore.enumerateRecords(myRecordFilter,
                                  myRecordComparator,
                                  false);
while (re.hasNextElement()) {
         byte b[] = re.nextRecord();
         // do something with the data
   }
```

In order to get a RecordEnumeration, use the following RecordStore method:

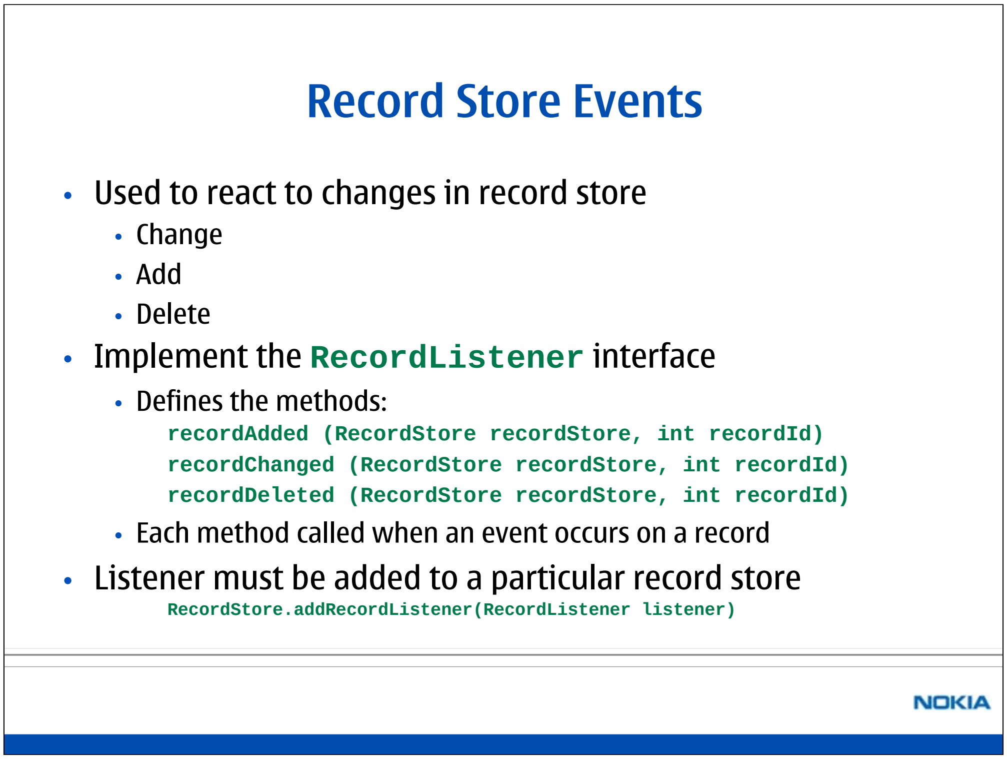
```
public RecordEnumeration enumerateRecords(RecordFilter filter,
RecordComparator comparator, boolean keepUpdated)
```

RecordFilter and RecordComparator are interfaces that define methods that allow you to exclude records from the enumeration and determine the order in which the records are returned, respectively. If the filter argument is null, then all records are included, while a null comparator causes the order of the records to be undefined. The following call, therefore, returns a RecordEnumeration containing all of the records of the RecordStore in no particular order:

```
RecordEnumeration enum = recordStore.enumerateRecords(null, null, false);
```

The keepUpdated argument determines whether the enumeration is static or dynamic. If this argument is false, the enumeration represents a snapshot of the state of the RecordStore when enumerateRecords() is called. If keepUpdated is true, however, changes in the content of the RecordStore will be visible through the enumeration (unless the changes involve records that are excluded by the filter). It is more efficient to create an enumeration with keepUpdated set to false, because it can be expensive to keep the enumeration in step with the underlying RecordStore.

All access to records is no longer done with direct record IDs, instead it is done with valid elements in the record store, so the programmer need not be concerned about trying to access a record that has been deleted.



Record Store Events

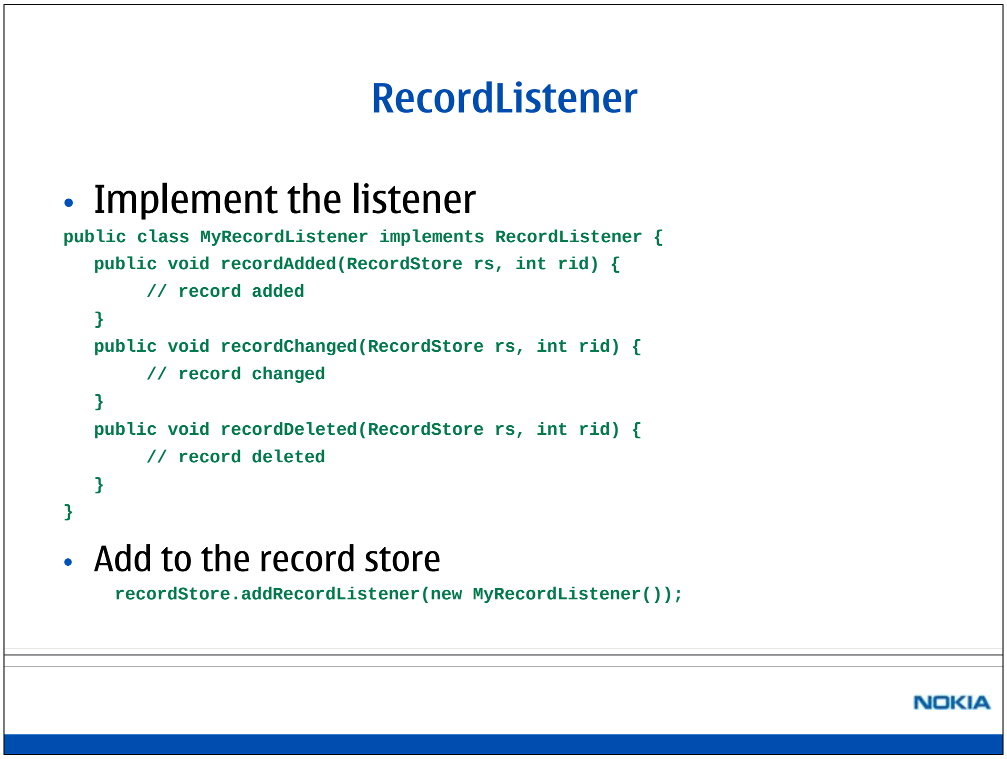
- Used to react to changes in record store
  - Change
  - Add
  - Delete
- Implement the `RecordListener` interface
  - Defines the methods:

    ```
    recordAdded (RecordStore recordStore, int recordId)
    recordChanged (RecordStore recordStore, int recordId)
    recordDeleted (RecordStore recordStore, int recordId)
    ```

  - Each method called when an event occurs on a record
- Listener must be added to a particular record store

  ```
  RecordStore.addRecordListener(RecordListener listener)
  ```

The RecordListener interface defines callbacks that are called whenever a record is added, changed or deleted. The programmer simply overrides the callback methods in the interface and then registers the listener with the record store. Note that even if a record store is closed, its Listeners are still active until the record store is actually deleted.



RecordListener

- Implement the listener

```
public class MyRecordListener implements RecordListener {
   public void recordAdded(RecordStore rs, int rid) {
        // record added
   }
   public void recordChanged(RecordStore rs, int rid) {
        // record changed
   }
   public void recordDeleted(RecordStore rs, int rid) {
        // record deleted
   }
}
```

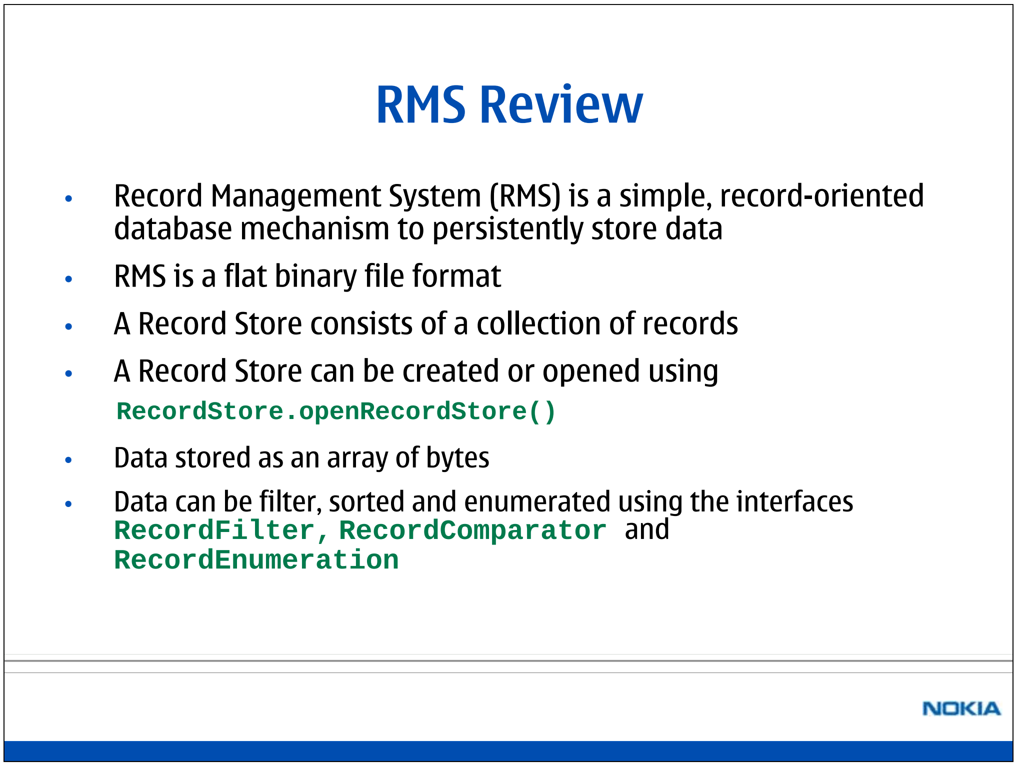
- Add to the record store

```
recordStore.addRecordListener(new MyRecordListener());
```

Each method in the record listener can be overridden so that code can be written to react to each type of event.

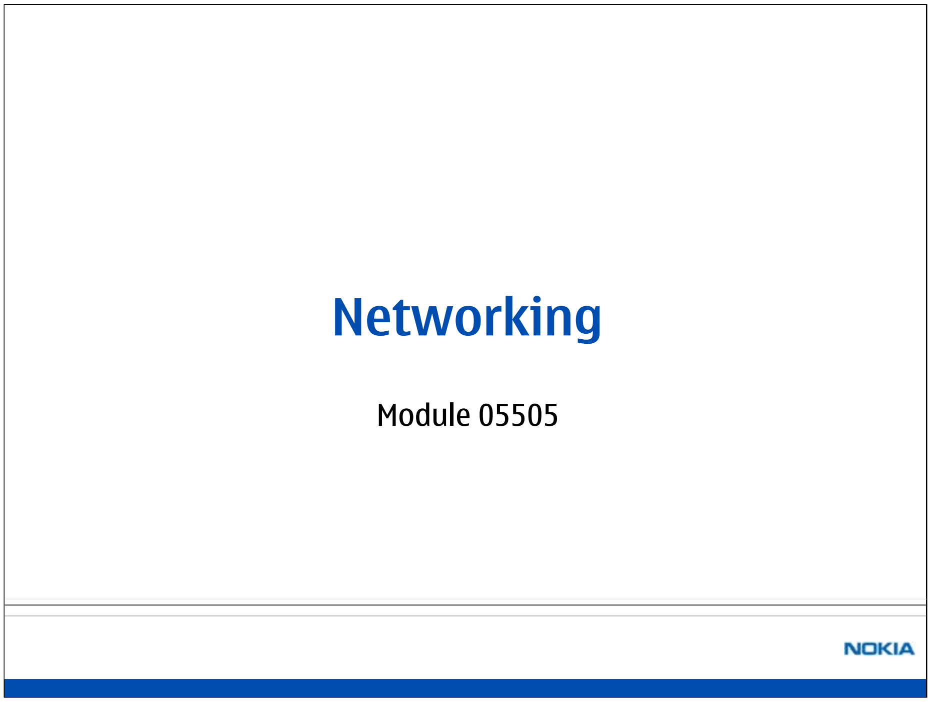
One usage of the record listener could be to keep a log of all changes made to a particular record store and update its representation of the records. For example, the code below shows implementations of the methods of RecordListener, each of which logs any calls made to them to a log file, then updates an internal list of the records

```
public void recordAdded(RecordStore rs, int rid) {
             // record added
             log(“Record” + rid + “ Added in Record store “ +
rs.getName());
             addRecordInList(rs, rid);
             //...
}
public void recordChanged(RecordStore rs, int rid) {
             // record added
             log(“Record” + rid + “ Changed in Record store “ +
rs.getName());
             changeRecordInList(rs, rid);
             //...
}
public void recordDeleted(RecordStore rs, int rid) {
             // record added
             log(“Record” + rid + “ Deleted in Record store “ +
rs.getName());
             deleteRecordInList(rs, rid);
             //...
}
```



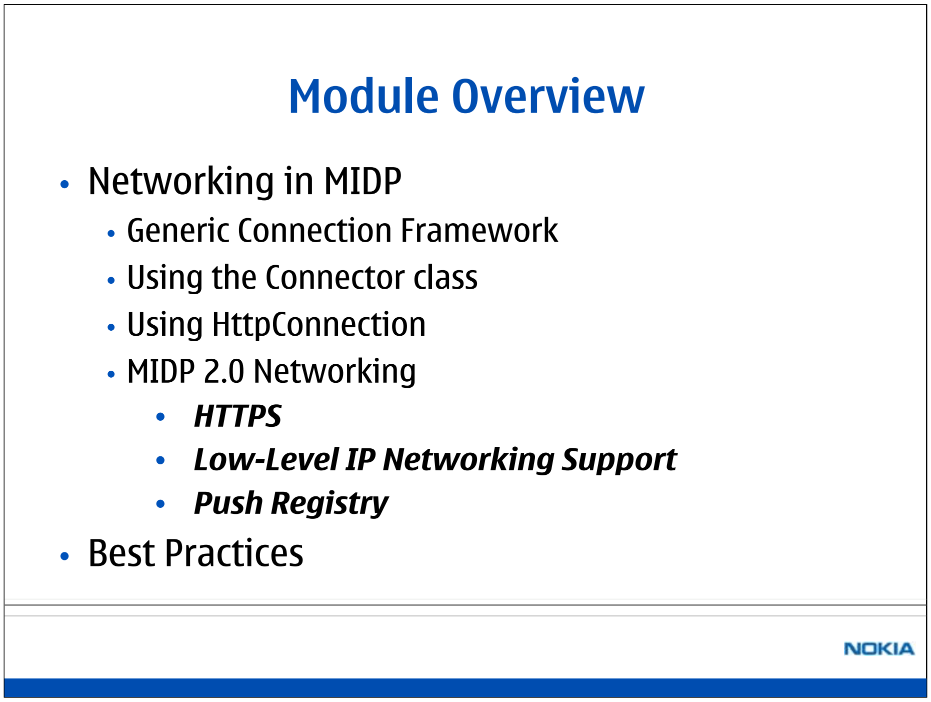
RMS Review

- Record Management System (RMS) is a simple, record-oriented database mechanism to persistently store data
- RMS is a flat binary file format
- A Record Store consists of a collection of records
- A Record Store can be created or opened using `RecordStore.openRecordStore()`
- Data stored as an array of bytes
- Data can be filter, sorted and enumerated using the interfaces `RecordFilter`, `RecordComparator` and `RecordEnumeration`



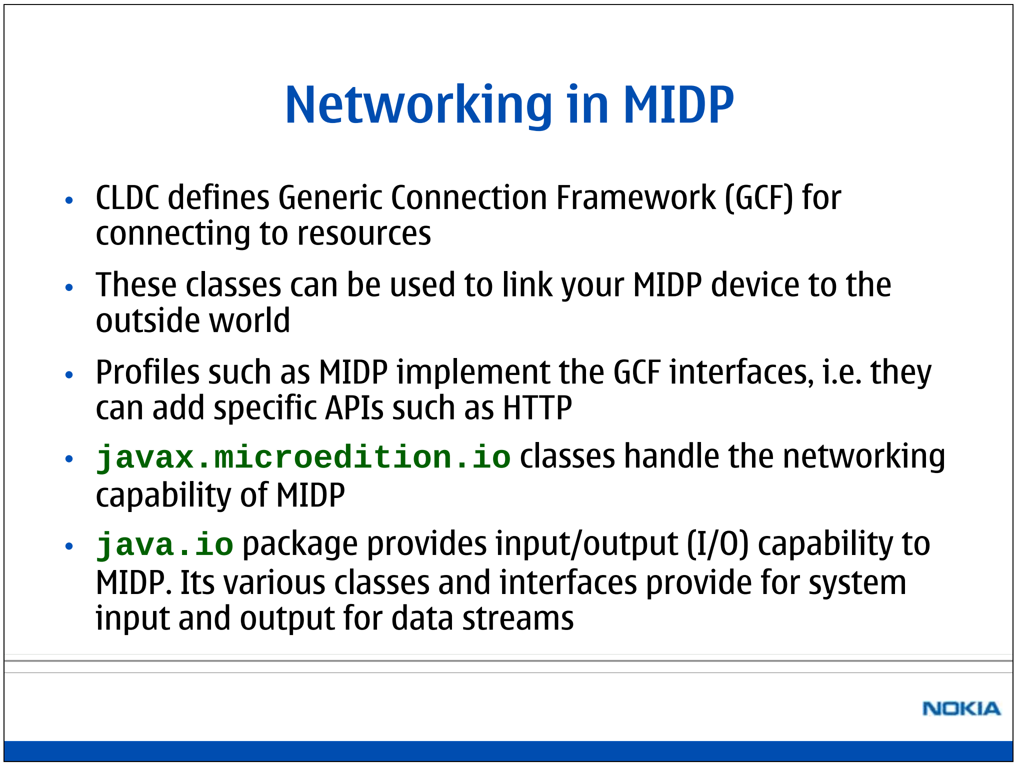
Networking

Module 05505



Module Overview

- Networking in MIDP
  - Generic Connection Framework
  - Using the Connector class
  - Using HttpConnection
  - MIDP 2.0 Networking
    - ***HTTPS***
    - ***Low-Level IP Networking Support***
    - ***Push Registry***
- Best Practices



Networking in MIDP

- CLDC defines Generic Connection Framework (GCF) for connecting to resources
- These classes can be used to link your MIDP device to the outside world
- Profiles such as MIDP implement the GCF interfaces, i.e. they can add specific APIs such as HTTP
- `javax.microedition.io` classes handle the networking capability of MIDP
- `java.io` package provides input/output (I/O) capability to MIDP. Its various classes and interfaces provide for system input and output for data streams

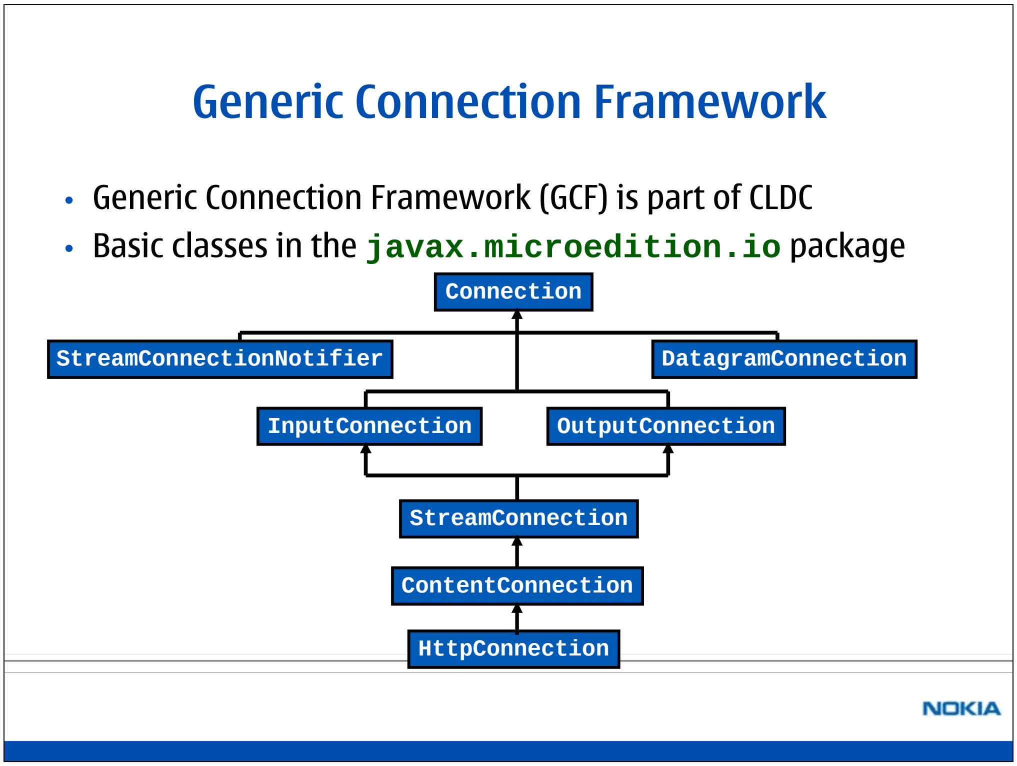
NOKIA

Java ME provides networking features that extend the resources available on a network into the mobile space. It is now possible to get up-to-the-minute stock quotes or updated currency exchange rates on a mobile phone.

The `javax.microedition.io` ***classes and interfaces handle the networking capability of the Mobile Information Device Profile (MIDP).***

The `java.io` ***package, on the other hand, provides input/output (I/O) capability to MIDP. Its various classes and interfaces provide for system input and output for data streams. This Java ME package, a subset of the Java SE*** `java.io` ***package, handles data I/O at a low level.***

The most critical aspect of Java ME network connectivity is communication between a mobile device and Web server. This is essentially a client/server mechanism in which the mobile device acts as a Web client and is capable of interfacing with enterprise systems, databases, corporate intranets, and the Internet.



Generic Connection Framework

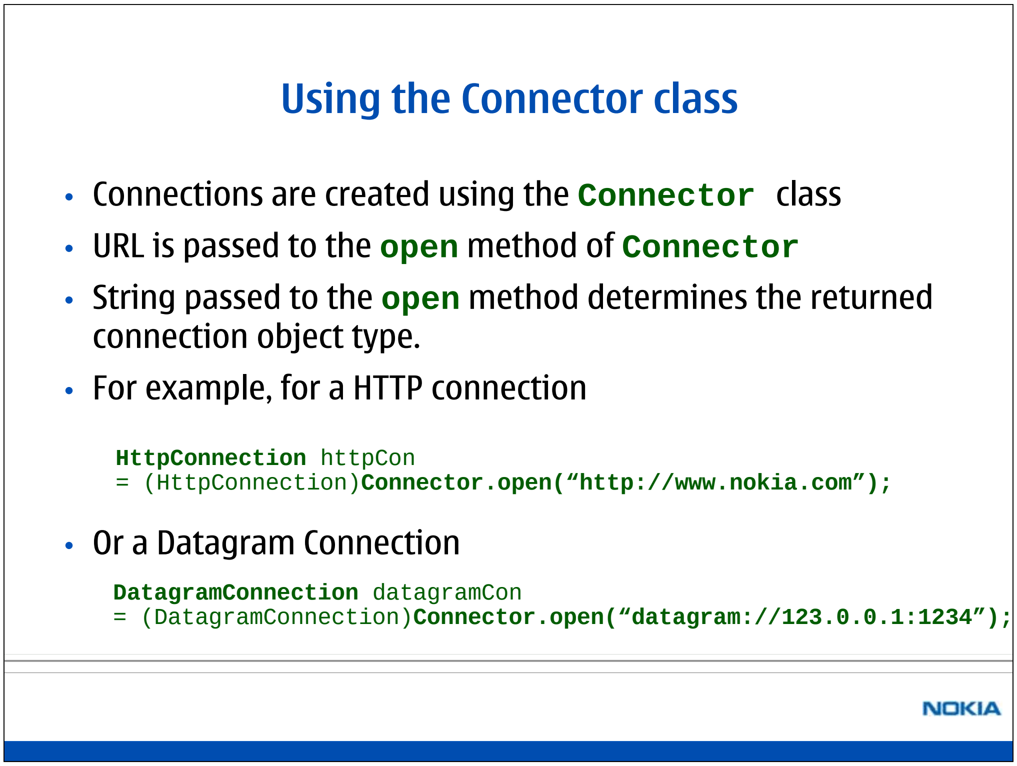
- Generic Connection Framework (GCF) is part of CLDC
- Basic classes in the `javax.microedition.io` package

The setting up of Network Connections is managed by a group of classes that are part of the CLDC specifications and define what is called the Generic Connection Framework (GCF). GCF defines a set of interfaces for connections, it is the responsibility of the specific profiles such as MIDP API to implement those connections. MIDP 1.0 added for example the HttpConnection, whilst MIDP 2.0 added HttpsConnection.

The idea of the generic connection framework is to define the abstractions that cover the general aspects of the networking and file I/O in the form of Java interfaces. This architecture enables support for a broad range of handheld devices, and leaves the actual implementations of these interfaces to individual device manufacturers. A device manufacturer chooses which interface to implement in its particular MIDPs based on the actual capabilities of its devices.

The general aspects defined by the Java interfaces take the form of the following basic types of communication:

- ***Basic serial input (defined by*** `javax.microedition.io.InputConnection`***)***
- ***Basic serial output (defined by*** `javax.microedition.io.OutputConnection`***)***
- ***Datagram communications (defined by*** `javax.microedition.io.DatagramConnection`***)***
- ***A sockets communications notification mechanism (defined by*** `javax.microedition.io.StreamConnectionNotifier`***) for client-server communication***
- ***Basic HTTP communication (defined by*** `javax.microedition.io.HttpConnection`***) with a Web server***



Using the Connector class

- Connections are created using the **Connector** class
- URL is passed to the **open** method of **Connector**
- String passed to the **open** method determines the returned connection object type.
- For example, for a HTTP connection

```
HttpConnection httpCon
= (HttpConnection)Connector.open(“http://www.nokia.com”);
```

- Or a Datagram Connection

```
DatagramConnection datagramCon
= (DatagramConnection)Connector.open(“datagram://123.0.0.1:1234”);
```

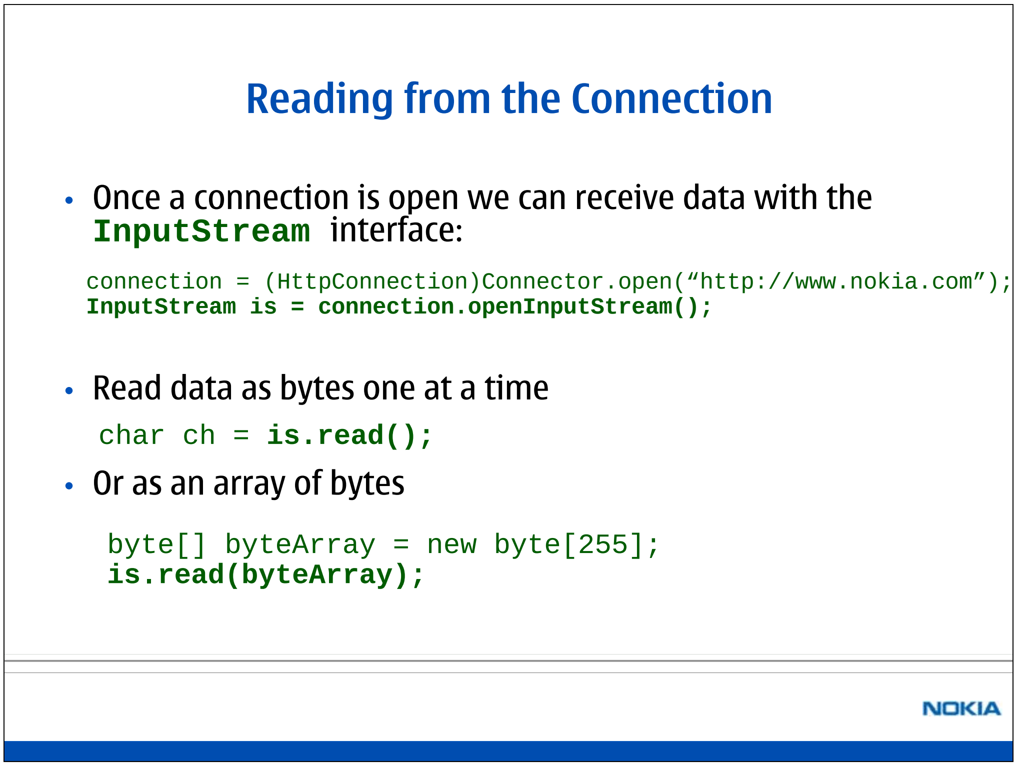
The developer establishes a network connection via the static method from the Connector class, which is packaged in javax.microedition.io. The Connector class returns an object that implements one of the connection interfaces which is determined by the connection string that is used to create the connection. Thus it is the responsibility of the programmer to use the correct connection string and cast to the correct connection interface type.

```
HttpConnection c = (HttpConnection) Connector.open ("http://www.forum.nokia.com/");
```

The above example, while very small, illustrates several concepts mentioned above. The developer used the GCF connection string format which always follows the form PROTOCOL:ADDRESS:PROPERTIES to establish the HTTP connection.

Note: A few words about connection strings. While the form is the same across all devices supporting CLDC, the actual format could be different - e.g. Number of slashes, use semi-colon or colon - for different resources thus consult the platform specific documentation to determine which connection strings are valid and what the format to use.

Once the connection is open the developer knows from the connection string what interface to cast to. So in the above code example we established a HTTP connection and thus cast the Connection object to the HttpConnection interface which implements ContentConnection interface defined in CLDC.



Reading from the Connection

- Once a connection is open we can receive data with the **InputStream** interface:

```
connection = (HttpConnection)Connector.open(“http://www.nokia.com”);
InputStream is = connection.openInputStream();
```

- Read data as bytes one at a time

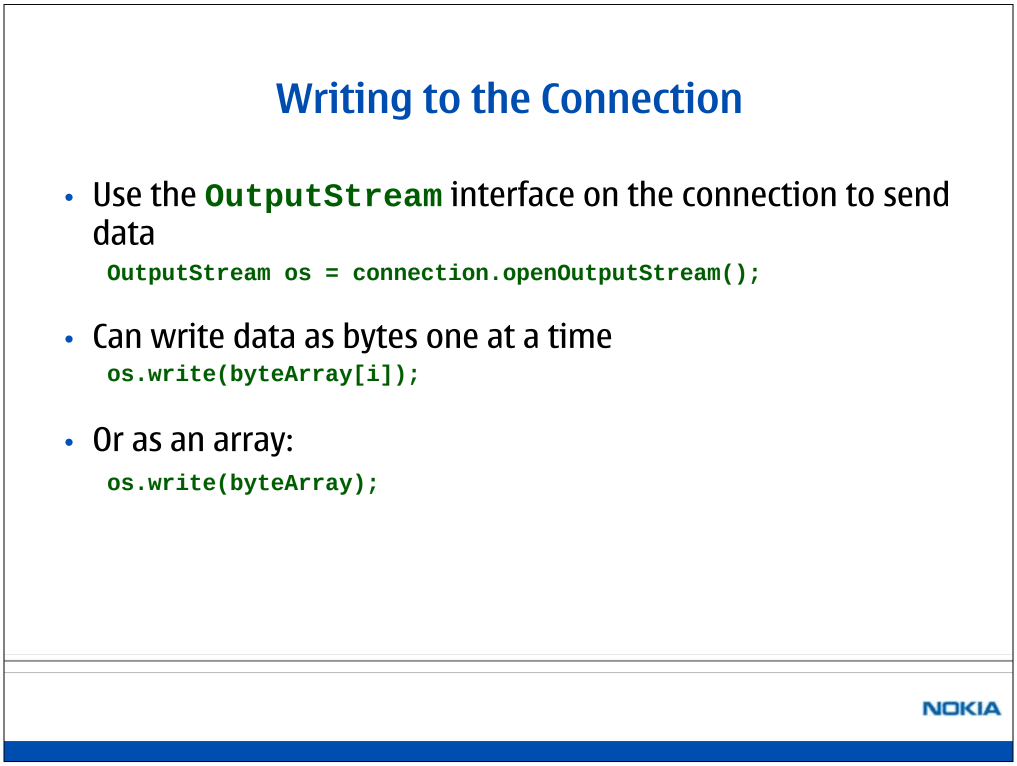
```
char ch = is.read();
```

- Or as an array of bytes

```
byte[] byteArray = new byte[255];
is.read(byteArray);
```

Once the connection is set up, there are several methods available to manage the connection. Perhaps foremost on the programmer's mind is how to send and receive data across the established network connection. Once the connection is established, we need to open up the input or output stream depending on whether or not we need to read or write data from the connection object. The I/O stream is then used to dispatch data from the open connection string. Generally, you'll declare some sort of buffer to hold the data that you want the connection stream manage. Here is a brief code example of how to get data from a HTTP connection:

```
StringBuffer buff = new StringBuffer();
try {
   HttpConnection conn = (HttpConnection)Connector.open
            (“http://www.forum.nokia.com”);
   InputStream is = conn.openInputStream();
   char ch;
   while ((ch = is.read()) != -1) {
      buff.append((char)ch);
   }
   is.close();
   conn.close();
}
```



Writing to the Connection

- Use the `OutputStream` interface on the connection to send data

```
OutputStream os = connection.openOutputStream();
```

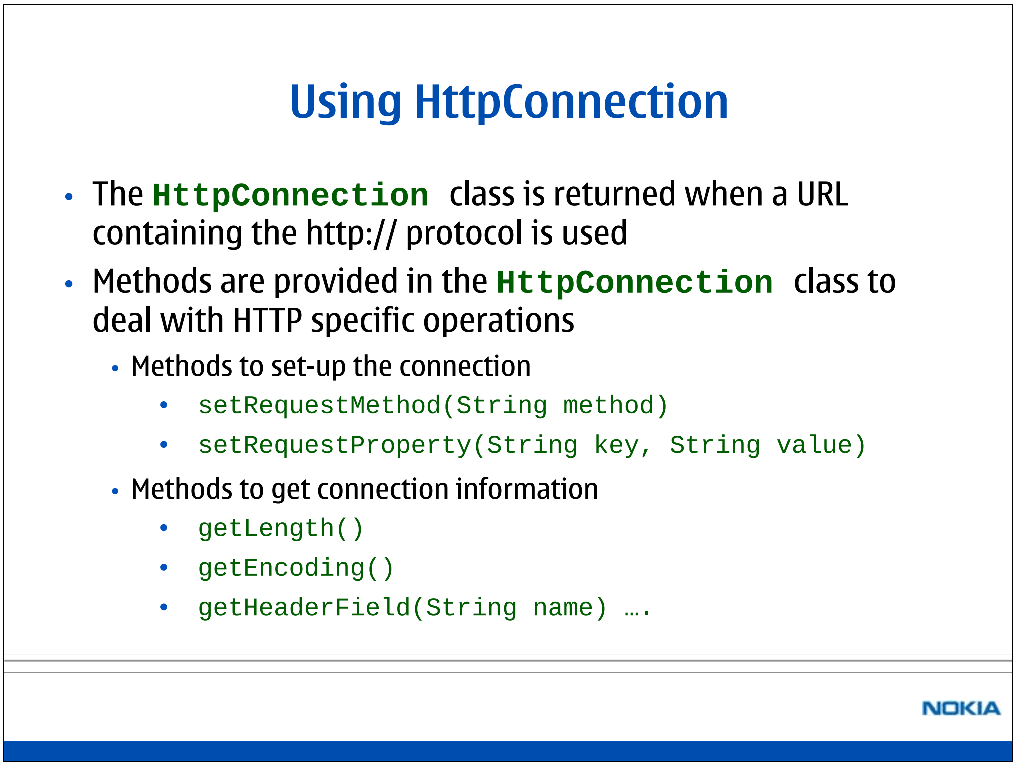
- Can write data as bytes one at a time

```
os.write(byteArray[i]);
```

- Or as an array:

```
os.write(byteArray);
```

Writing to a connection is very easy. Its basically the exact opposite of reading from a connection, i.e. use the openOutputStream() method to retrieve an OutputStream object and then use the write methods to write the byte you want to send.



Using HttpConnection

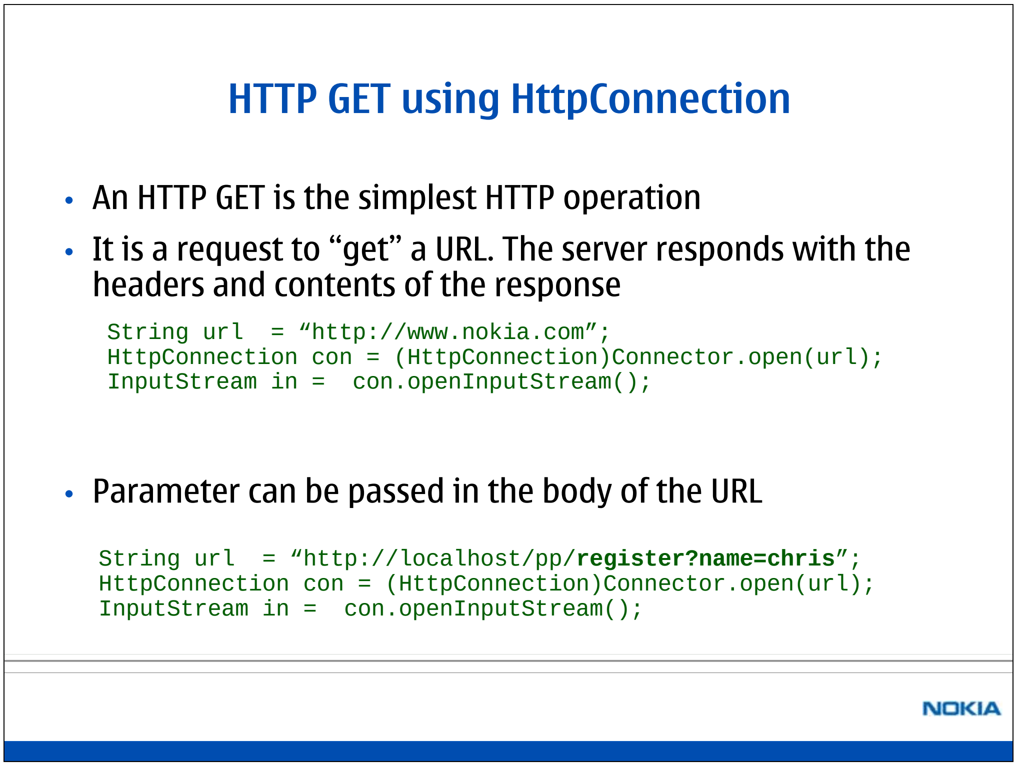
- The `HttpConnection` class is returned when a URL containing the http:// protocol is used
- Methods are provided in the `HttpConnection` class to deal with HTTP specific operations
  - Methods to set-up the connection
    - `setRequestMethod(String method)`
    - `setRequestProperty(String key, String value)`
  - Methods to get connection information
    - `getLength()`
    - `getEncoding()`
    - `getHeaderField(String name) ….`

If we looked at the Javadoc for the HttpConnection class we would see that it contains methods and constants to deal with HTTP specific properties. HTTP is built around requests and responses. A client sends a request to server, and the server responds. Requests and responses have 2 parts, headers and contents. Another thing to consider with HTTP is the parameters that can be sent by the client. Parameters are simple name and value pairs. For example, a client may send a “name” parameter with a value of “chris” to a server.

When setting up a HttpConnection, there are two methods that can be invoked before sending or receiving data to determine the headers and contents of your request. These are setRequestMethod and setRequestProperty

setRequestMethod is used to specify the method for the URL request. This can be HttpConnection.GET, HttpConnection.POST or HttpConnection.HEAD. If you don't call this method, the default is set to GET. The specification says that the Get should be used if you are just making an enquiry to this particular URL, whilst Post should be used if the data you are sending to the URL is storing or updating data. Head should be used if you only want the header data sent back. With a Get, the parameters can be added to the end of the URL, where as with Post the parameters are passed as the body of the request, using the Outputstream.

The setRequestProperty method is used to set a request property in the header of the Http request. Properties such as Content-type and Content-length can be set here.



HTTP GET using HttpConnection

- An HTTP GET is the simplest HTTP operation
- It is a request to “get” a URL. The server responds with the headers and contents of the response

```
String url  = “http://www.nokia.com”;
HttpConnection con = (HttpConnection)Connector.open(url);
InputStream in =  con.openInputStream();
```

- Parameter can be passed in the body of the URL

```
String url  = “http://localhost/pp/register?name=chris”;
HttpConnection con = (HttpConnection)Connector.open(url);
InputStream in =  con.openInputStream();
```

The simplest HTTP operation is GET. This is what happens when you type a URL into your browser; the browser GETs the URL from the server, which responds with the headers and content.

Parameters are added to the end of the URL in encoded form when using a GET request, for example
`http://localhost/pp/register?name=chris`

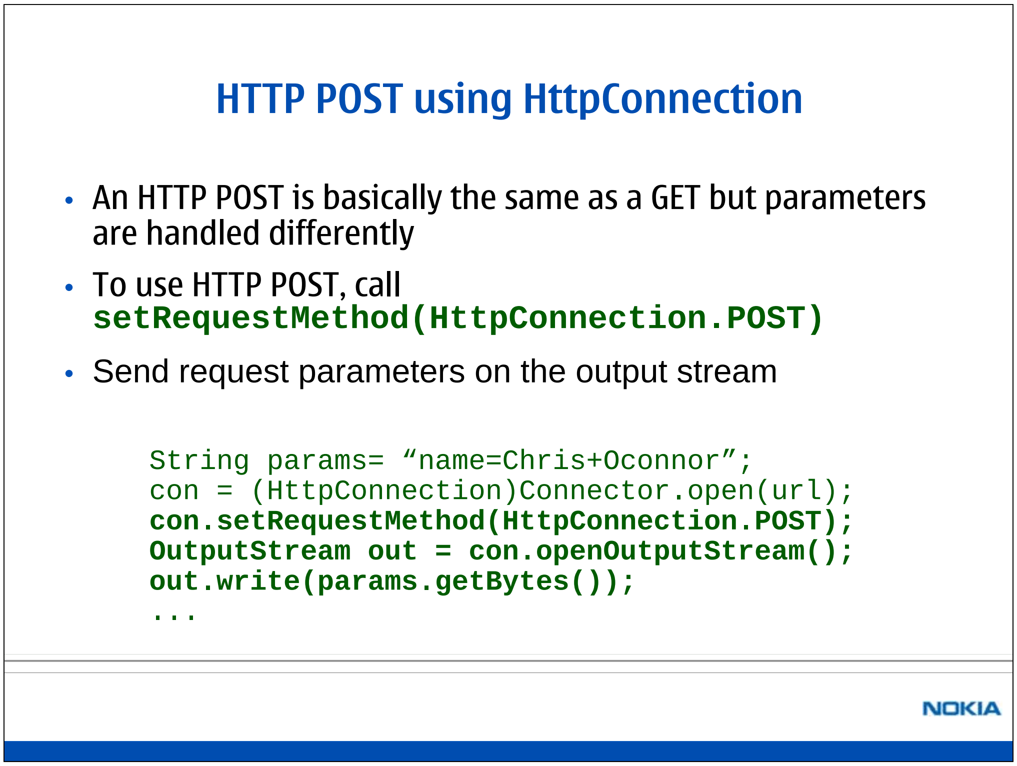
Additional name and value pairs can be added, separated by ampersands:
`http://localhost/pp/register?name=chris&age=26`

Loading data from a server is very simple, particularly if you’re performing a HTTP GET. Simply pass a URL to the Connector’s static open() method. The returned Connection will probably be an implementation of HttpConnection, but you can just treat it as an InputConnection. Then get the corresponding InputStream and the data.

```
String url = “http://www.nokia.com”
InputConnection ic = (InputConnection)Connector.open(url);
InputStream in = ic.openInputStream();
ic.close();
```

As mention earlier, with HTTP GET, all parameters are passed to the server in the body of the URL. Before appending your parameters to the URL, however, you should encode them using the java.net.URLEncoder class. The rule for encoding are relatively simple:

- ***The space character is converted to a plus (+) sign***
- ***The following characters remain unchanged: lowercase letters ‘a’ through to ‘z’, uppercase letters ‘A’ through to ‘Z’, the numbers 0 through to 9, the period (.), the hyphen (-), the asterisk (*) and the underscore (_).***
- ***All other characters are converted into “%xy”, where “xy” is a hexadecimal number that represents the low eight bits of the character***



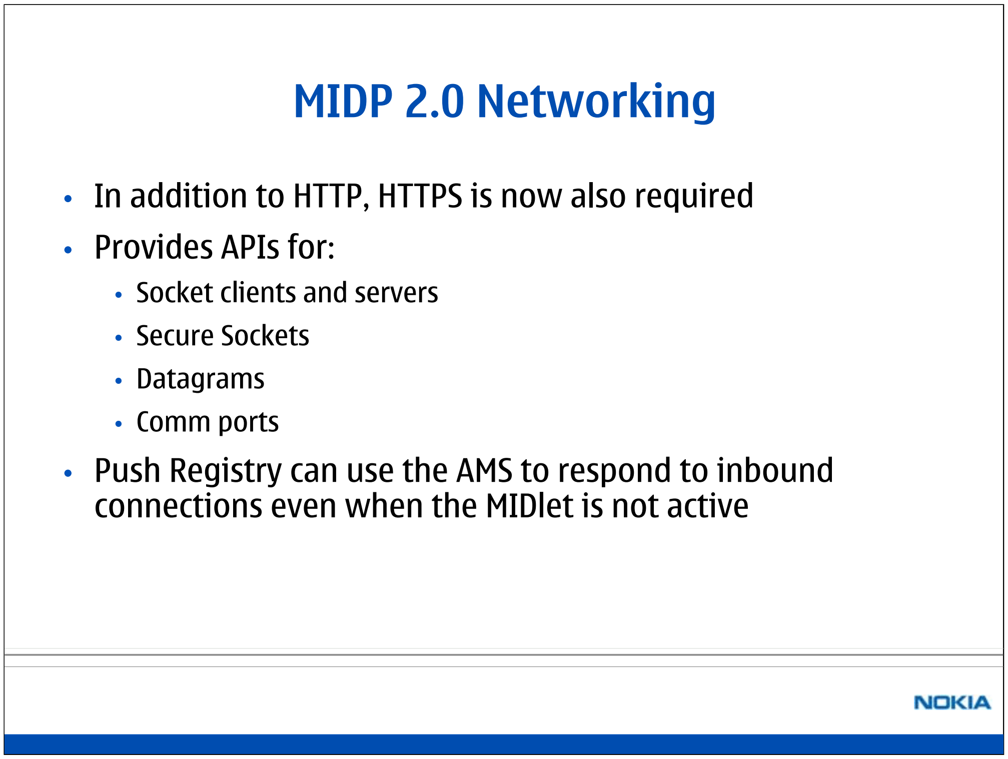
HTTP POST using HttpConnection

- An HTTP POST is basically the same as a GET but parameters are handled differently
- To use HTTP POST, call **`setRequestMethod(HttpConnection.POST)`**
- Send request parameters on the output stream

```
String params= “name=Chris+Oconnor”;
con = (HttpConnection)Connector.open(url);
con.setRequestMethod(HttpConnection.POST);
OutputStream out = con.openOutputStream();
out.write(params.getBytes());
...
```

Posting a form is a little more complicated on the Personal Profile application side. In particular, there are request headers that need to be set in HttpConnection before the server is contacted. The process works like this:

1. ***Obtain an HttpConnection from Connector’s open() method***
2. ***Modify the header fields of the request. In particular, you need to change the request method by calling setRequestMethod(), and you should set the “Content-length” header by calling setRequestProperty(). This is the length of the parameters you will be sending***
3. ***Obtain the output stream for the HttpConnection by calling openOutputStream(). This sends the request headers to the server***
4. ***Send the request parameters on the output stream returned from the HttpConnection.***
5. ***Read the response from the server from the input stream HttpConnection’s openInputStream() method***



MIDP 2.0 Networking

- In addition to HTTP, HTTPS is now also required
- Provides APIs for:
  - Socket clients and servers
  - Secure Sockets
  - Datagrams
  - Comm ports
- Push Registry can use the AMS to respond to inbound connections even when the MIDlet is not active

While the only new requirement of MIDP 2.0 is the HTTPS API, the API also provides and recommends the implementation of sockets, datagrams and serial port access. The SocketConnection and SecureSocketConnection APIs provide raw sockets, which don't have the overhead of HTTP connections. The UDPDatagramConnection provides an API for connectionless and very lightweight datagram functionality. An abstraction of serial ports is provided by the CommConnection API. Finally, inbound connections can be dispatched by the SocketServerConnection API and by registering inbound connections using the PushRegistry.

Secure Connections

Probably the biggest difference in MIDP 2.0 networking is the concept of secure connections, which introduced three new classes SecureConnection, HttpsConnection and SecurityInfo as described above. Security is accomplished using encrypted protocols implemented either as SSL (version 3.0), TLS (version 1.0), or wTLS. You can use the ssl:// or https:// connection strings to affect the type of connection you want like this:

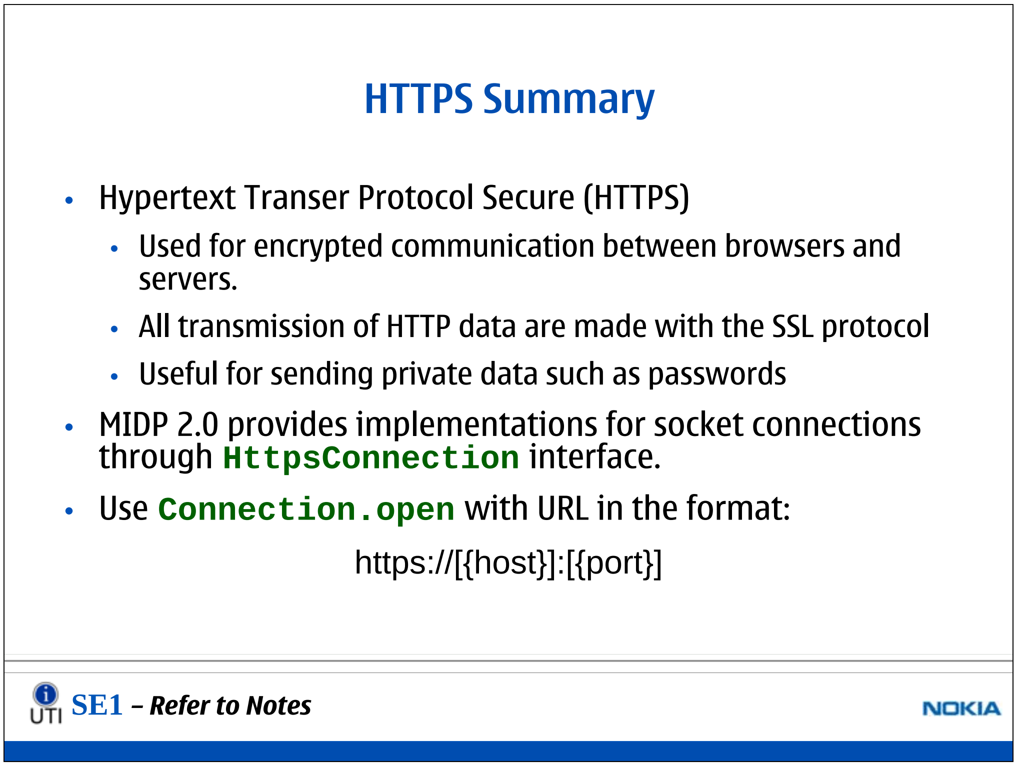
```
SecureConnection secConn = (SecureConnection) Connector.open("ssl://myserver:445");
secConn.setSocketOption(SocketConnection.LINGER, 5);
InputStream is = socketConn.openInputStream();
OutputStream os = socketConn.openOutputStream();
os.write("X".getBytes());
while (int i != -1) {
        i = is.read();
}
is.close();
os.close();
secConn.close();
```

To write code that opens a HTTPS connection, we could write similar code like this:

```
HttpsConnection conn = Connector.open(https://myserver);
InputStream is = conn.openDataInputStream();
if (conn.getResponseCode() == HttpConnection.HTTP_OK) {
int len = (int) conn.getLength(); // check if length is avail
if (len > 0) {
        byte buf[] = new byte[len];
        is.readFully(buf);
} else {
        while (int i != -1) {
                i = is.read(); // char at a time }
        }
}
is.close();
conn.close();
```

Finally, to find out more information about how your secure connection is in fact being implemented you can get an instance to the SecurityInfo method and call the appropriate methods like this:

```
SecurityInfo secInf = secConn.getSecurityInfo();
String protocol = secInf.getProtocolName();
String version = secInf.getProtocolVersion();
String certificate = secInf.getServerCertificate();
String cipher = secInf.getCipherSuite();
```



HTTPS Summary

- Hypertext Transer Protocol Secure (HTTPS)
  - Used for encrypted communication between browsers and servers.
  - All transmission of HTTP data are made with the SSL protocol
  - Useful for sending private data such as passwords
- MIDP 2.0 provides implementations for socket connections through `HttpsConnection` interface.
- Use `Connection.open` with URL in the format:

https://[{host}]:[{port}]

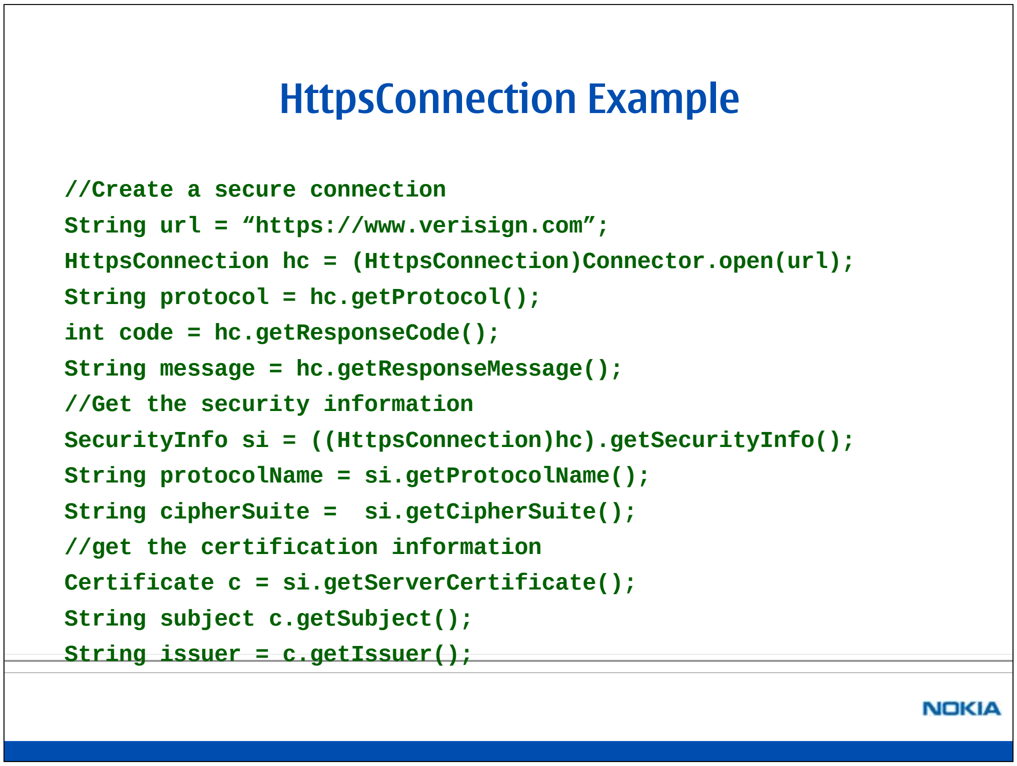
SE1 – ***Refer to Notes***

HTTPS is the secure version of HTTP a request-response protocol in which the parameters of the request must be set before the request is sent.

MIDP 2.0 provides implementations for socket connections through HttpsConnection interface

SE1 - Privacy security and data integrity,must be assured. Encryption (if the system supports it) must be used to send password and / or personal data

When sending personal data and/or passwords you should use HttpsConnection



HttpsConnection Example

```
//Create a secure connection
String url = “https://www.verisign.com”;
HttpsConnection hc = (HttpsConnection)Connector.open(url);
String protocol = hc.getProtocol();
int code = hc.getResponseCode();
String message = hc.getResponseMessage();
//Get the security information
SecurityInfo si = ((HttpsConnection)hc).getSecurityInfo();
String protocolName = si.getProtocolName();
String cipherSuite =  si.getCipherSuite();
//get the certification information
Certificate c = si.getServerCertificate();
String subject c.getSubject();
String issuer = c.getIssuer();
```

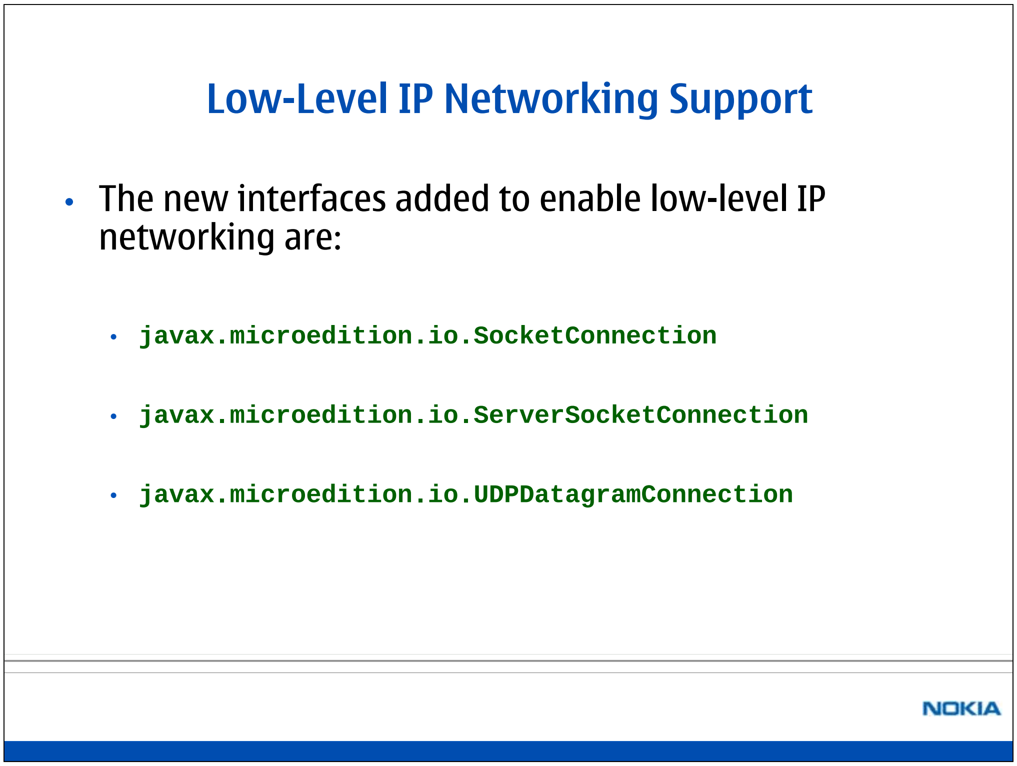
A secure connection can be created using the Connector.open method using a URL with the protocol “https” for example “https://www.verisign.com” and casting the returned objects to HttpsConnection.

The example above creates a secure connection, then gets information about this secure connection.

Information such as the protocol name, protocol version and details of the encryption algorithms can be obtained from the SecurityInfo object. The SecurityInfo object is obtained from the HttpsConnection object by calling the getSecurityInfo method. Developers should note that the getSecurityInfo method is not a method of the base class Connection, therefore, to use this method you must ensure to cast a secure Connection object to a HttpsConnection object when using https as the protocol.

Certificate information can be obtained by calling the getServerCertificate method on the SecurityInfo object. This provides identifying information about the server, and information about the certificate authority that issued it.

The only network protocol the MIDP 1.0 specification requires is HTTP. MIDP 2.0 also requires HTTPS, which is basically HTTP over the Secure Sockets Layer (SSL). SSL is a socket protocol that encrypts data sent over the network and provides authentication for the socket endpoints. Although many MIDP 1.0 implementations support HTTPS, application developers cannot rely on its availability. MIDP 2.0 provides a stable, consistent foundation for wireless applications that deal with money or sensitive information.



Low-Level IP Networking Support

- The new interfaces added to enable low-level IP networking are:
  - `javax.microedition.io.SocketConnection`
  - `javax.microedition.io.ServerSocketConnection`
  - `javax.microedition.io.UDPDatagramConnection`

MIDP 2.0 has added the interfaces SocketConnection, ServerSocketConnection and UDPDatagramConnection for low-level IP networking.

A call such as Connector.open("socket://host:port") returns a SocketConnection,

A call such as Connector.open("socket://:port") returns a ServerSocketConnection.

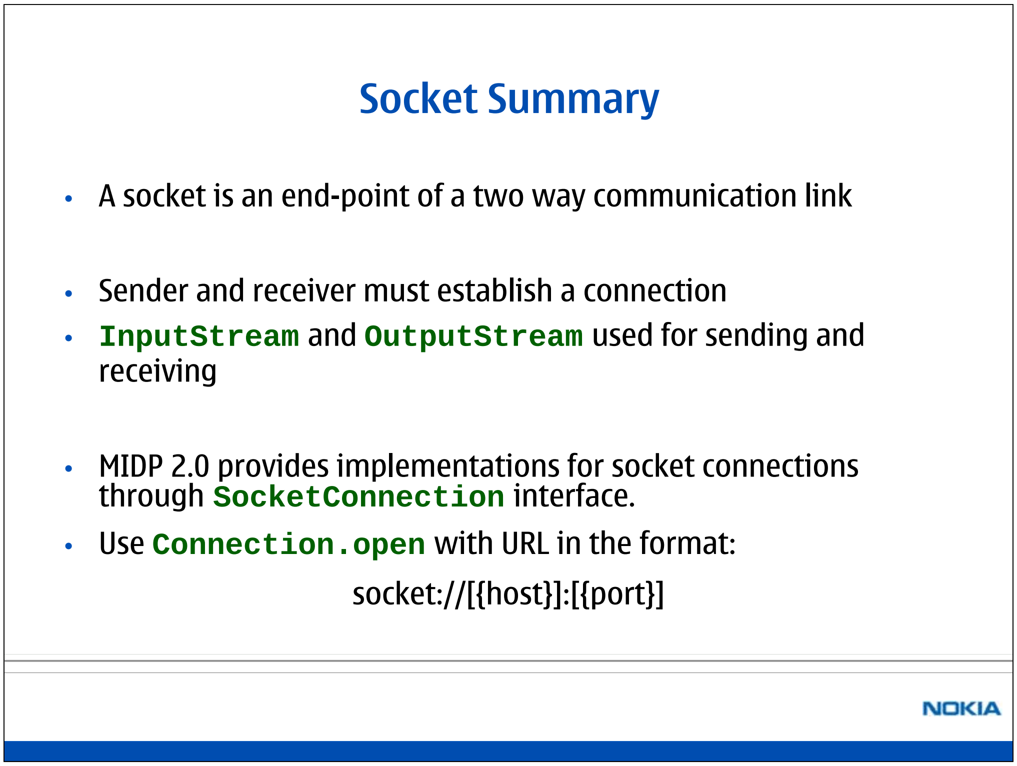
A MIDlet should specify a host when requesting an outbound client connection and omit the host when requesting an inbound server connection.

If you leave out the port parameter when obtaining a server socket – as in Connector.open("socket://") – an available port number is assigned dynamically.

If this is done, you can use the getLocalPort method to discover the assigned port number, and the getLocalAddress method to discover the local address to which the socket is bound.

A call such as Connector.open("datagram://host:port") returns a UDPDatagramConnection.

Note, however, that the UDP protocol is transaction-oriented, and delivery and duplicate protection are not guaranteed. Therefore, if your applications require ordered, reliable delivery and streams of data, you should use TCP/IP stream connections.



Socket Summary

- A socket is an end-point of a two way communication link
- Sender and receiver must establish a connection
- `InputStream` and `OutputStream` used for sending and receiving
- MIDP 2.0 provides implementations for socket connections through `SocketConnection` interface.
- Use `Connection.open` with URL in the format:

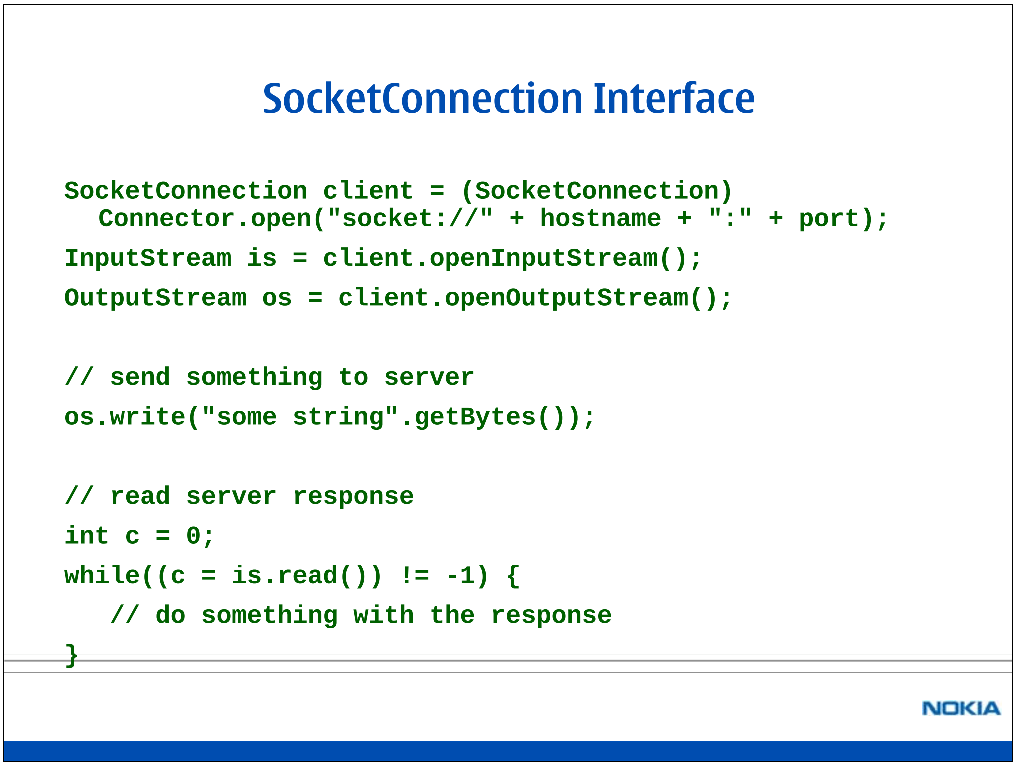
socket://[{host}]:[{port}]

A socket is one end-point of a two-way communication link between programs running on the network. A socket connection is the most basic low-level reliable communication mechanism between a wireless device and a remote server or between two wireless devices. The socket communication capability provided with some of the mobile devices in Java ME enables a variety of client/server applications.

MIDP 2.0 provides implementations for socket connections through SocketConnection interface. To use a socket, the sender and receiver that are communicating must first establish a connectionbetween their sockets. One will be listening for a request for a connection, and the other will be asking for a connection. Once two sockets have been connected, they may be used for transmitting data in either direction.

Network programming using sockets is very straightforward in Java ME. The process works as follows:

1. *A socket connection is opened with a remote server or another wireless device using Connector.open().*
2. *InputStream or OutputStream is created from the socket connection for sending or receiving data packets.*
3. *Data can be sent to and received from the remote server via the socket connection by performing read or write operations on the InputStream or OutputStream object.*
4. *The socket connection and input or output streams must be closed before exiting the program.*



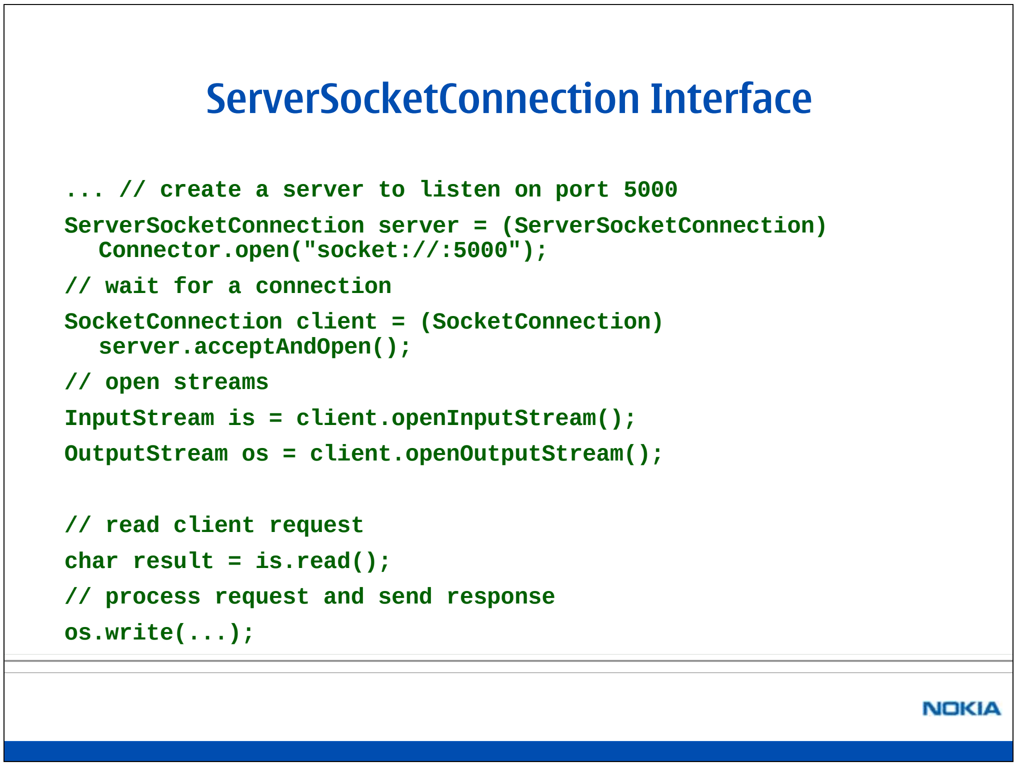
SocketConnection Interface

```
SocketConnection client = (SocketConnection)
  Connector.open("socket://" + hostname + ":" + port);
InputStream is = client.openInputStream();
OutputStream os = client.openOutputStream();

// send something to server
os.write("some string".getBytes());

// read server response
int c = 0;
while((c = is.read()) != -1) {
   // do something with the response
}
```

The SocketConnection interface defines the socket stream connection. You use it when writing MIDlets that access TCP/IP servers.

A socket is one end-point of a two-way communication link between programs running on the network. As we can see in the example above, once a socket is established the stream can be read from using an InputStream, or written to, using an OutputStream.

Before a client can request a socket connection to a listener, the listener must be listening to a designated port. A socket listener application can be created using a ServerSocketConnection.



ServerSocketConnection Interface

```
... // create a server to listen on port 5000
ServerSocketConnection server = (ServerSocketConnection)
   Connector.open("socket://:5000");
// wait for a connection
SocketConnection client = (SocketConnection)
   server.acceptAndOpen();
// open streams
InputStream is = client.openInputStream();
OutputStream os = client.openOutputStream();

// read client request
char result = is.read();
// process request and send response
os.write(...);
```

The ServerSocketConnection interface defines the server socket stream connection. You use it when requesting an inbound server connection.

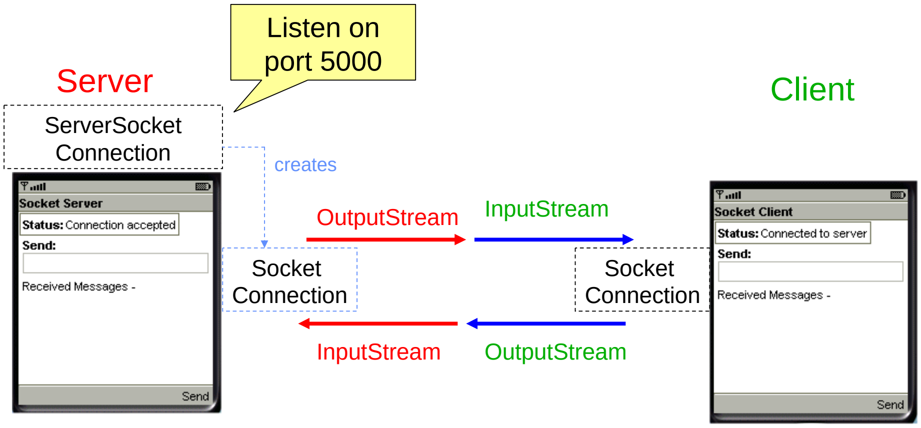
Once a ServerSocketConnection is established, the socket listener waits for client to attempt a connection using the acceptAndOpen() method.

The server host can be discovered using getLocalAddress() method.

Developers should remember that if they open a server socket and want to know the source address of the client connecting to the server socket, they should call the getAddress() method on the SocketConnection's object created once acceptAndOpen() function returns. For example, this can be done in the following way:

```
ServerSocketConnection ssc = (ServerSocketConnection)
Connector.open("socket://:5000");

// Wait for a connection.
SocketConnection sc = (SocketConnection) ssc.acceptAndOpen();

String remoteAddress = sc.getAddress();
```

Socket Connection Example



Listen on port 5000

Server

ServerSocket Connection

creates

Socket Connection

OutputStream

InputStream

InputStream

OutputStream

Socket Connection

Client

Socket Server

Status: Connection accepted

Send:

Received Messages -

Socket Client

Status: Connected to server

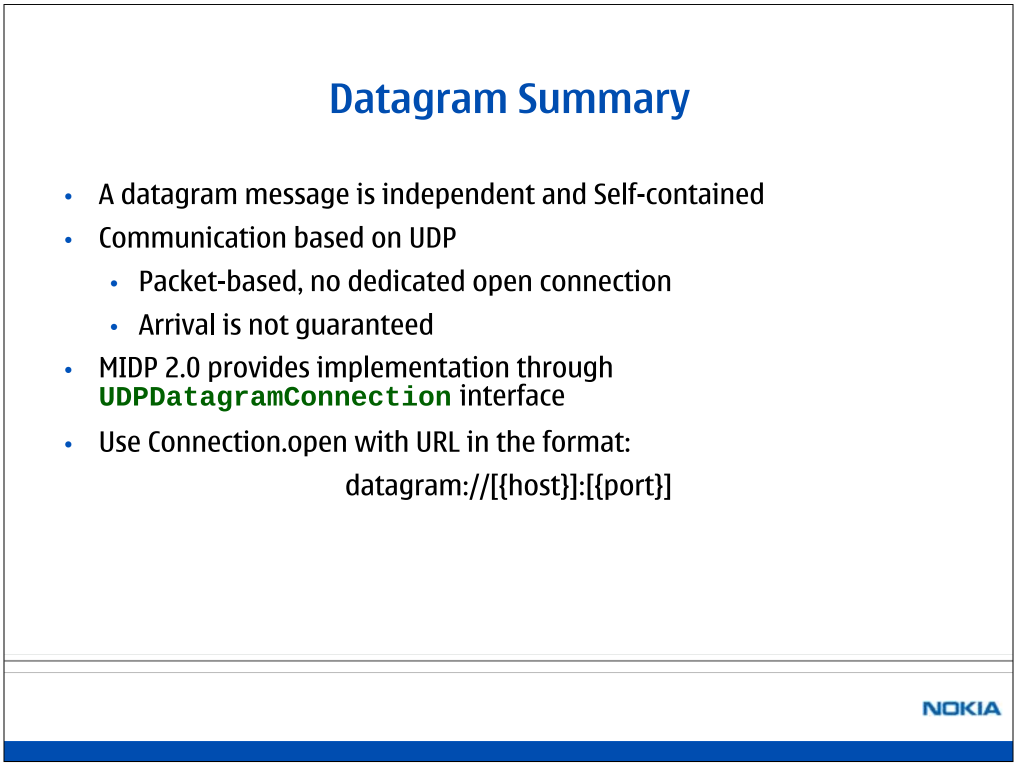
Send:

Received Messages -

The slide shows when the server is started, its waits for a connection using ServerSocketConnection by listening on port 5000.

When the client is started, its uses a SocketConnection to attempt to connect on the same port number. When the ServerSocketConnection accepts and opens the connection, it creates a SocketConnection on the Server side. The input streams and output streams are then opened on the server and client terminals

Messages can then be exchanged between the two terminals by writing bytes onto the output stream and reading bytes on the input stream.



Datagram Summary

- A datagram message is independent and Self-contained
- Communication based on UDP
  - Packet-based, no dedicated open connection
  - Arrival is not guaranteed
- MIDP 2.0 provides implementation through `UDPDatagramConnection` interface
- Use Connection.open with URL in the format:

  datagram://[{host}]:[{port}]

A datagram is an independent, self-contained message sent over the network; the datagram's arrival, arrival time, and content are not guaranteed. It is a packet-based communication mechanism. Unlike stream-based communication, packet-based communication is connectionless, which means that no dedicated open connection exists between the sender and the receiver.

Datagram communication is based on UDP. The sender builds a datagram packet with destination information (an Internet address and a port number) and sends it out. Lower-level network layers do not perform any sequencing, error checking, or acknowledgement of packets. So, there is no guarantee that a data packet will arrive at its destination. The server might never receive your initial datagram—moreover, if it does, its response might never reach your wireless device. Because UDP is a not a guaranteed-delivery protocol, it is not suitable for applications such as FTP that require reliable transmission of data.

MIDP 2.0 provides implementation through UDPDatagramConnection interface

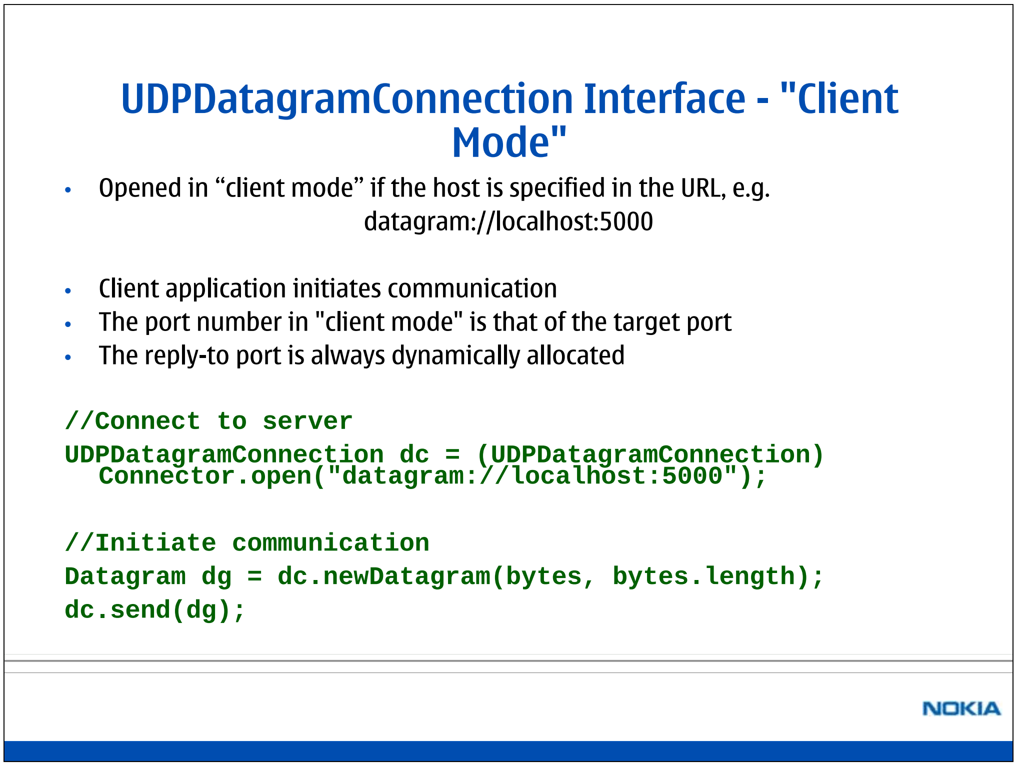
Here are the typical steps for using datagram communication in MIDlet applications:

1. ***Establish a datagram connection.***
2. ***Construct a send datagram object with a message body and a destination address.***
3. ***Send the datagram message out through the established datagram connection.***
4. ***Construct a receive datagram object with a pre-allocated buffer.***
5. ***Wait to receive the message through the established connection using the allocated datagram buffer.***
6. ***Free up the datagram connection after use.***

The following are rules of thumb for choosing a datagram size:

- ***Never exceed the maximum allowable packet size. The maximum allowable packet size can be obtained by using the method getMaximumLength() in the UDPDatagramConnection interface***

- ***If the wireless network is very reliable and most of the data transmitted will arrive at the destination, use a bigger packet size. The bigger the packet size, the more efficient the data transfer, because the datagram header causes significant overhead when the packet size is too small.***



UDPDatagramConnection Interface - "Client Mode"

- Opened in “client mode” if the host is specified in the URL, e.g.

  datagram://localhost:5000

- Client application initiates communication
- The port number in "client mode" is that of the target port
- The reply-to port is always dynamically allocated

```
//Connect to server
UDPDatagramConnection dc = (UDPDatagramConnection)
  Connector.open("datagram://localhost:5000");

//Initiate communication
Datagram dg = dc.newDatagram(bytes, bytes.length);
dc.send(dg);
```

Like other types of connections, a UDPDatagramConnection connection is created with the open method in Connector. The connect string is in this format:

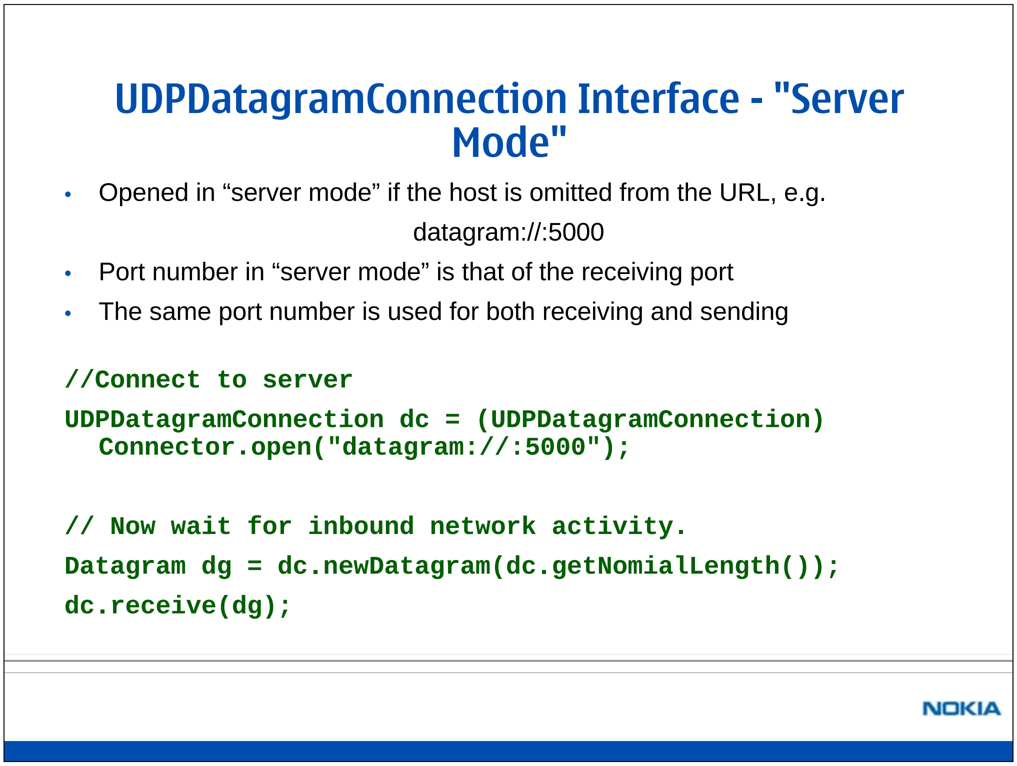
datagram://[{host}]:{port}

In the connect string, the port field is required; it specifies the target port with a host. The host field is optional; it specifies the target host. If the host field is specified in the connection string, the connection is created in “client" mode and initiates communication by sending out a datagram.

The UDPDatagramConnection interface defines a datagram connection which knows it's local end point address.

The Datagram interface is a generic interface that provides a placeholder for a datagram message. A Datagram object can then be sent or received through a UDPDatagramConnection.

The Datagram class extends from the DataInput and DataOutput classes in the java.io package. These classes provide methods for the necessary read and write operations to the binary data stored in the datagram's buffer.



UDPDatagramConnection Interface - "Server Mode"

- Opened in “server mode” if the host is omitted from the URL, e.g.

datagram://:5000

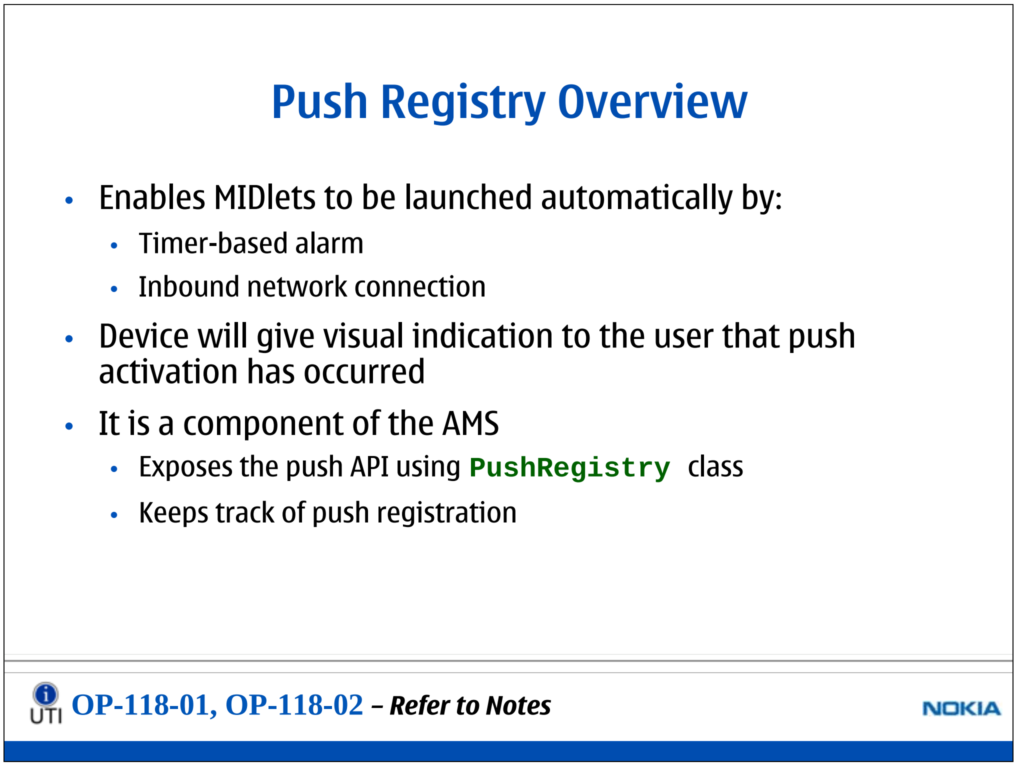
- Port number in “server mode” is that of the receiving port
- The same port number is used for both receiving and sending

```
//Connect to server
UDPDatagramConnection dc = (UDPDatagramConnection)
  Connector.open("datagram://:5000");

// Now wait for inbound network activity.
Datagram dg = dc.newDatagram(dc.getNomialLength());
dc.receive(dg);
```

If the host is omitted from URL string used in the Connector.open method, the connection is opened in server mode, and must then wait for inbound network activity before it can send a reply to a client.

Once a datagram is received from the client, the server can call the method getAddress() on the Datagram object. This can be used to send a reply back to the client by passing the address into the construction of a datagram, for example.

```
String address = receivedDatagram.getAddress();
Datagram sentDatagram = datagramConnection.newDatagram(bytes, bytes.length,
address);
datagramConnection.send(sentDatagram);
```



Push Registry Overview

- Enables MIDlets to be launched automatically by:
  - Timer-based alarm
  - Inbound network connection
- Device will give visual indication to the user that push activation has occurred
- It is a component of the AMS
  - Exposes the push API using `PushRegistry` class
  - Keeps track of push registration

UTI **OP-118-01, OP-118-02** – ***Refer to Notes***

NOKIA

The push registry enables inbound network connections or timer-based alarms to wake up a MIDlet, so that the application can then act asynchronously on the information it receives. This is an alternative to using polling based synchronous techniques, which increase resource use or latency.

MIDlets can set themselves up to be launched automatically. This can be done in one of two ways:

- ***Timer based activation - The MIDlet is scheduled to launch after a certain period, then sleeps after finishing its task. An example of this could be a MIDlet that synchronises with a server every hour.***
- ***Network activation – The MIDlet is woken up by an inbound network connection. An example application of this technique could be a MIDlet that is woken up by a newly received email. This MIDlet then processes the email and sleeps, waiting for the next email to be received.***

The push registry is part of the Application Management system (AMS). This is device specific software that is responsible for the installation, execution and removal of applications. It maintains the list of inbound network connections and timer alarms and lists of MIDlets that consumer network connections and timer alarms.

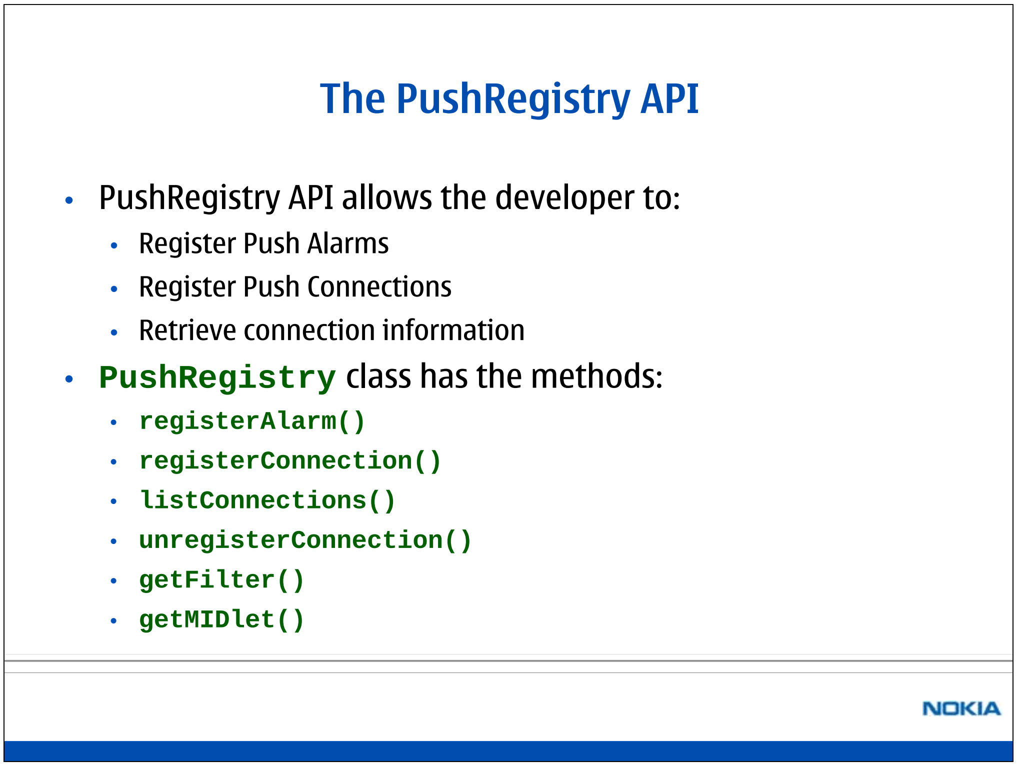
An API is provided so that alarms and connections can be registered with the AMS push registry. This is encapsulated within a single class javax.microedition.io.PushRegistry

OP-118-01 - All registered alarms and connections must be activatable under test.

Your application should register alarms using PushRegistry.registerAlarm is destroyApp. Connections can be registered statically in the JAD file using the MIDlet-Push attribute or dynamically using PushRegistry.registerConnection.

OP-118-02 - All push-activated MIDlets must show some visual indication to the user that push activation has occurred.

When the Java Application Manager (JAM) auto-launches a push MIDlet the device will automatically ask permission from the user. The following details are presented to the user: The name of the MIDlet



The PushRegistry API

- PushRegistry API allows the developer to:
  - Register Push Alarms
  - Register Push Connections
  - Retrieve connection information
- `PushRegistry` class has the methods:
  - `registerAlarm()`
  - `registerConnection()`
  - `listConnections()`
  - `unregisterConnection()`
  - `getFilter()`
  - `getMIDlet()`

The PushRegistry API allows you to register push alarms and connections, and to retrieve information about push connections. A typical push registry maintains lists of connection and alarm registrations in both memory and persistent storage. It is encapsulated within a single class javax.microedition.io.PushRegistry

The PushRegistry API allows the developer to register a MIDlet for push events (inbound network connections and time-based alarms), discover whether the MIDlet was activated by an inbound connection, and retrieve push-specific information for a particular connection.

The push related methods that are exposed by the PushRegistry class are:

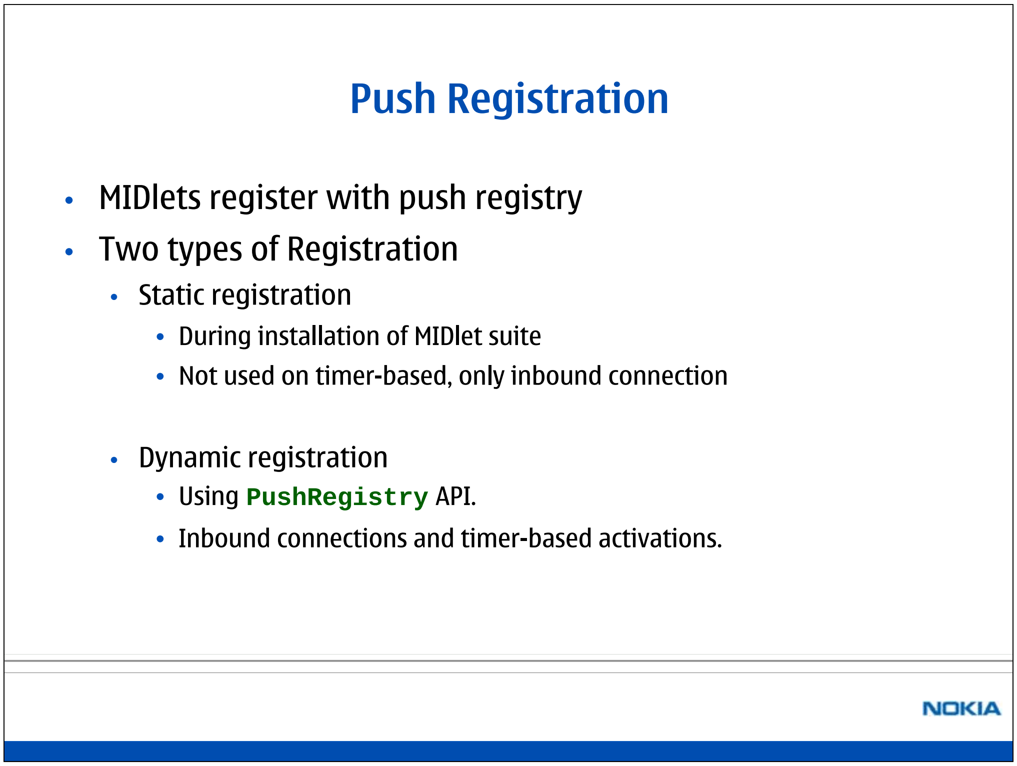
registerAlarm(String midlet, long time) - Registers a timer-based alarm to launch the MIDlet. The MIDlet class name and time at which this MIDlet should be executed are past as the parameters to this method. The MIDlet named in this method must be registered in the descriptor file/jar file manifest. The format of the time parameter is the same as that returned by the call Date.getTime(), i.e. the number of milliseconds since January 1, 1970.

*registerConnection(String connection, String midlet, String filter) – Registers a push connection. The parameters are: A connection string (in the format <protocol>://<host>:<port>), the MIDlet class name and a filter string, in the form of a connection URL, to determine which senders are allowed to cause the MIDlet to be launched. The filter string is dependant on the protocol being used, however the wildcard symbols * and ? can be used. For example, is the filter string was "*", connections from any source will be accepted.*

listConnections(boolean available) - Returns the list of registered push connections for the MIDlet suite. This returns an array of Strings representing the registered connections. The boolean parameter for this method determines whether the method returns connections with input available, or just the complete list connections.

unregisterConnection(String connection) – Unregisters a push connection. This method returns a boolean to indicate whether the unregistration was successful.

getFilter(String connection) - Returns the filter for a specific connection. The filter indicates which senders are allowed to cause the MIDlet to be launched



Push Registration

- MIDlets register with push registry
- Two types of Registration
  - Static registration
    - During installation of MIDlet suite
    - Not used on timer-based, only inbound connection
  - Dynamic registration
    - Using `PushRegistry` API.
    - Inbound connections and timer-based activations.

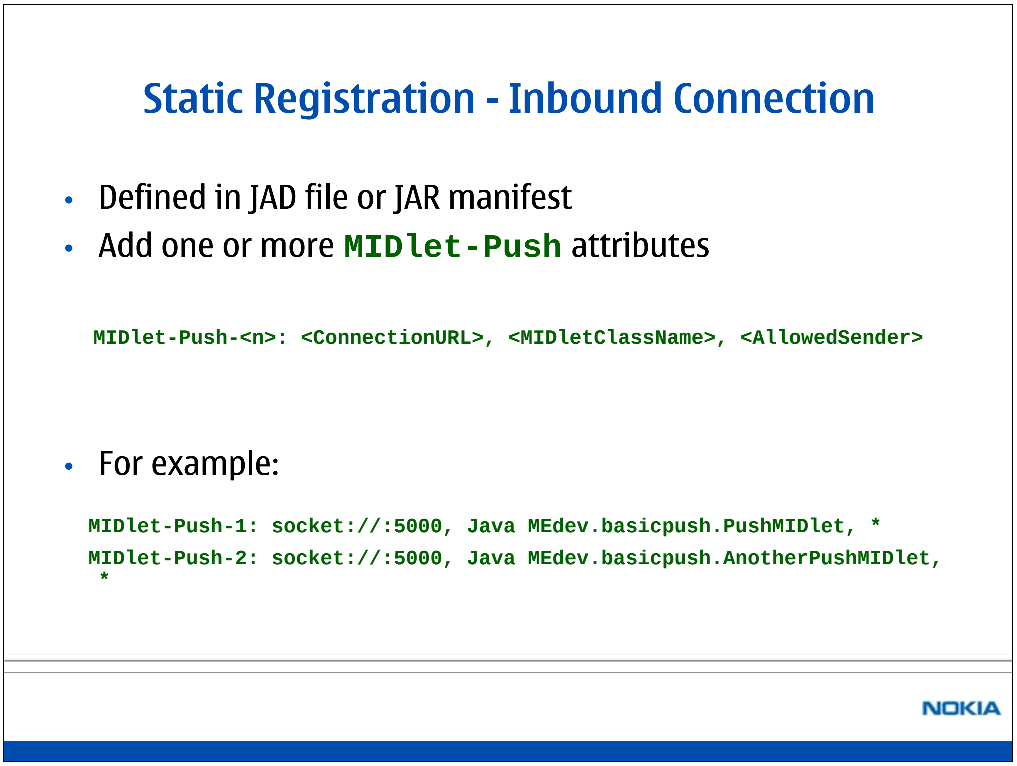
To become push-enabled, MIDlets must register with the push registry, using one of two types of registration:

- *Static Registration - Registrations of static connections occur during the installation of the MIDlet suite. You specify them by listing MIDlet-Push attributes in the MIDlet suite's JAD file or JAR manifest. The installation will fail if you attempt to register an address that's already bound. Uninstalling a MIDlet suite automatically unregisters the connection.*

- *Dynamic Registration - You register dynamic connections and timer alarms at runtime, using the PushRegistry API.*

In some cases you may want to register a static connection conditionally, or to ensure that push exceptions will not preclude your MIDlet suite from installing. In such cases you can use the PushRegistry API to register your static connection and catch any IOExceptions or SecurityExceptions thrown. Typically, however, you'll register a static connection using the JAD file, and let the system unregister it when the MIDlet suite is uninstalled.

Note that, while you can use either method to register inbound connections, timer-based activation can be registered only at runtime.



Static Registration - Inbound Connection

- Defined in JAD file or JAR manifest
- Add one or more `MIDlet-Push` attributes

```
MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>
```

- For example:

```
MIDlet-Push-1: socket://:5000, Java MEdev.basicpush.PushMIDlet, *
MIDlet-Push-2: socket://:5000, Java MEdev.basicpush.AnotherPushMIDlet,
 *
```

Static registrations are defined by listing one or more MIDlet-Push attributes in the JAD file or JAR manifest. The AMS performs static registration when the MIDlet suite is installed. Similarly, when the MIDlet suite is uninstalled, the AMS automatically unregisters all its associated push registrations.

The format of the MIDlet-Push attribute is...

MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>

...where:

MIDlet-Push-<n> is the property name that identifies push registration, and where <n> is a number starting from 1; for example, MIDlet-Push-1. Note that multiple push entries are allowed.

<ConnectionURL> is a URL connection string that identifies the inbound endpoint to register, in the same URL format used when invoking Connector.open(). For example, socket://:5000 reserves an inbound server socket connection on port 5000.

<MIDletClassName> is the fully qualified class name of the MIDlet to be activated when network activity in <ConnectionURL> is detected; for example, Java MEdeveloper.basicpush.PushMIDlet.

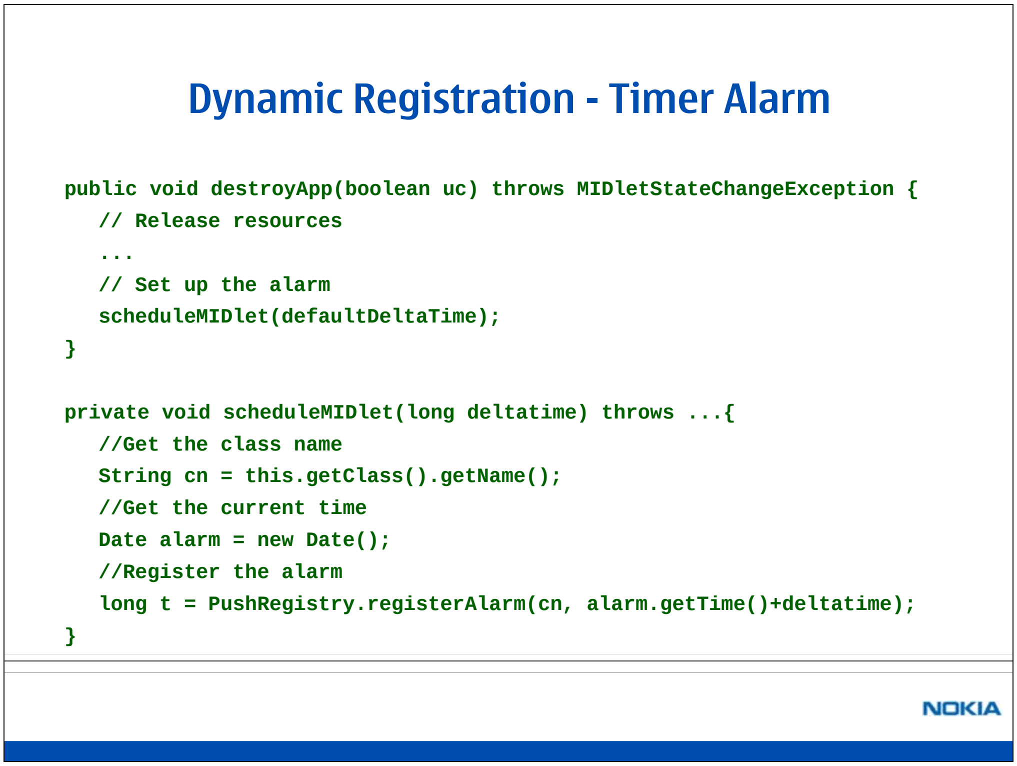
<Allowed-Sender> is a filter used to restrict the servers that can activate <MIDletClassName>. You can use wildcards; a * indicates one or more characters and a ? indicates one character. For example, 192.168.1.190, or 192.168.1.*, or 192.168.19?.1, or simply *.

The following shows a JAD file with a static registration of an inbound socket connection on port 5000, with no address filtering, which activates MIDlet Java MEdeveloper.basicpush.PushMIDlet.

```
MIDlet-1: PushMIDlet,,Java MEdeveloper.basicpush.PushMIDlet
MIDlet-2: WMAMIDlet,,Java MEdeveloper.wma.WMAMIDlet
MIDlet-Name: MyMIDletSuite
MIDlet-Vendor: Sun Microsystems, Inc.
MIDlet-Version: 1.0
MIDlet-Jar-Size: 4735
MIDlet-Jar-URL: basicpush.jar
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
MIDlet-Push-1: socket://:5000, Java MEdeveloper.basicpush.PushMIDlet, *
MIDlet-Permissions: javax.microedition.io.PushRegistry, javax.microedition.io.Connector.serversocket
```

If the requested local address <ConnectionURL> is already in use, the installation will fail; how this failure is presented to the user depends on the vendor and the implementation.

Note the MIDlet-Permissions property, which is used to request permissions for the MIDlet suite. This example requests



Dynamic Registration - Timer Alarm

```
public void destroyApp(boolean uc) throws MIDletStateChangeException {
   // Release resources
   ...
   // Set up the alarm
   scheduleMIDlet(defaultDeltaTime);
}

private void scheduleMIDlet(long deltatime) throws ...{
   //Get the class name
   String cn = this.getClass().getName();
   //Get the current time
   Date alarm = new Date();
   //Register the alarm
   long t = PushRegistry.registerAlarm(cn, alarm.getTime()+deltatime);
}
```

To schedule a MIDlet launch by the AMS the MIDlet invokes the PushRegistry.registerAlarm() method, passing as arguments the fully qualified class name of the MIDlet to launch, and the time for the launch. Passing a time of zero disables the alarm. Note that only one outstanding alarm per MIDlet is supported, and invoking this method overwrites any previously scheduled alarm.

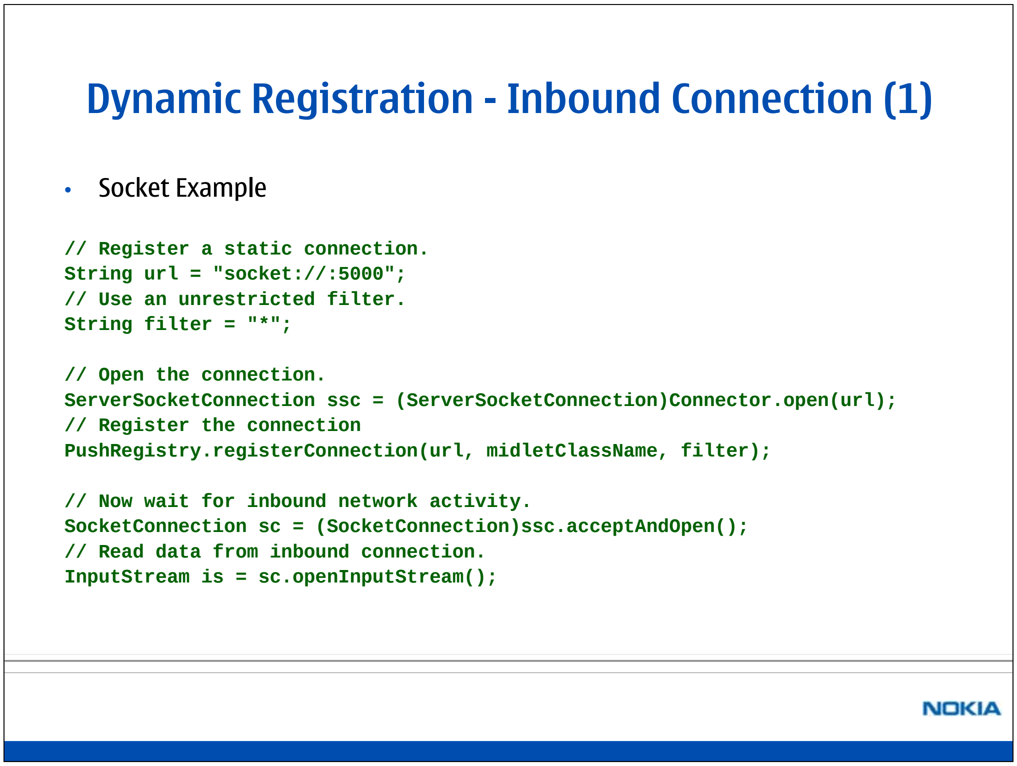
In this code sample, a helper method uses the registerAlarm() method to schedule the launch of the currently executing MIDlet

If the call to registerAlarm() overwrites a previous timer, it returns that timer's scheduled time; if not, it returns 0.

The following call can be made to launch the MIDlet after two hours:

```
scheduleMIDlet(2*60*60*1000);
```

A MIDlet that requires push alarms must schedule them before it exits, so is dealt with in the destroyApp method. When destroyApp() is invoked, it release resources (connection, threads, etc), and then schedules the push alarm for the future launch of the MIDlet before exiting.



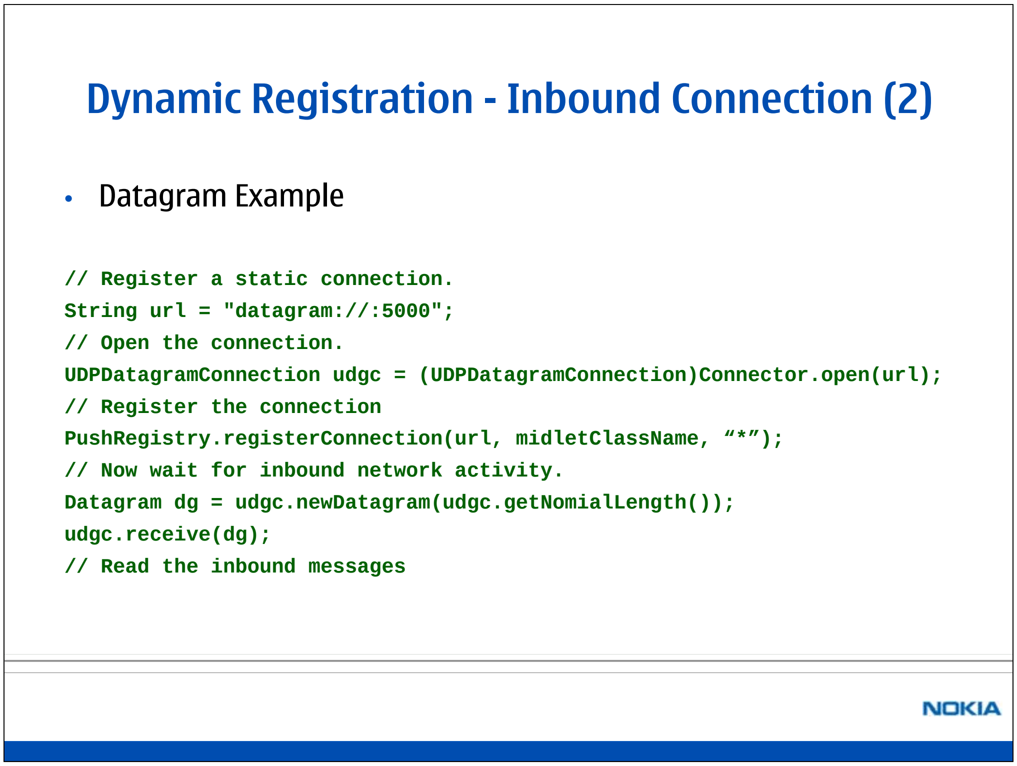
Dynamic Registration - Inbound Connection (1)

- Socket Example

```
// Register a static connection.
String url = "socket://:5000";
// Use an unrestricted filter.
String filter = "*";

// Open the connection.
ServerSocketConnection ssc = (ServerSocketConnection)Connector.open(url);
// Register the connection
PushRegistry.registerConnection(url, midletClassName, filter);

// Now wait for inbound network activity.
SocketConnection sc = (SocketConnection)ssc.acceptAndOpen();
// Read data from inbound connection.
InputStream is = sc.openInputStream();
```

The code shows how to register an inbound connection, using a user-defined local port. The MIDlet calls registerConnection() to register the newly created inbound (server) socket connection.

The connection is registered after the Connector.open method is called so that when this MIDlet exits (is destroyed), the AMS can activate the MIDlet when network activity is detected. The AMS will remember the registered URL even when the MIDlet is not active.

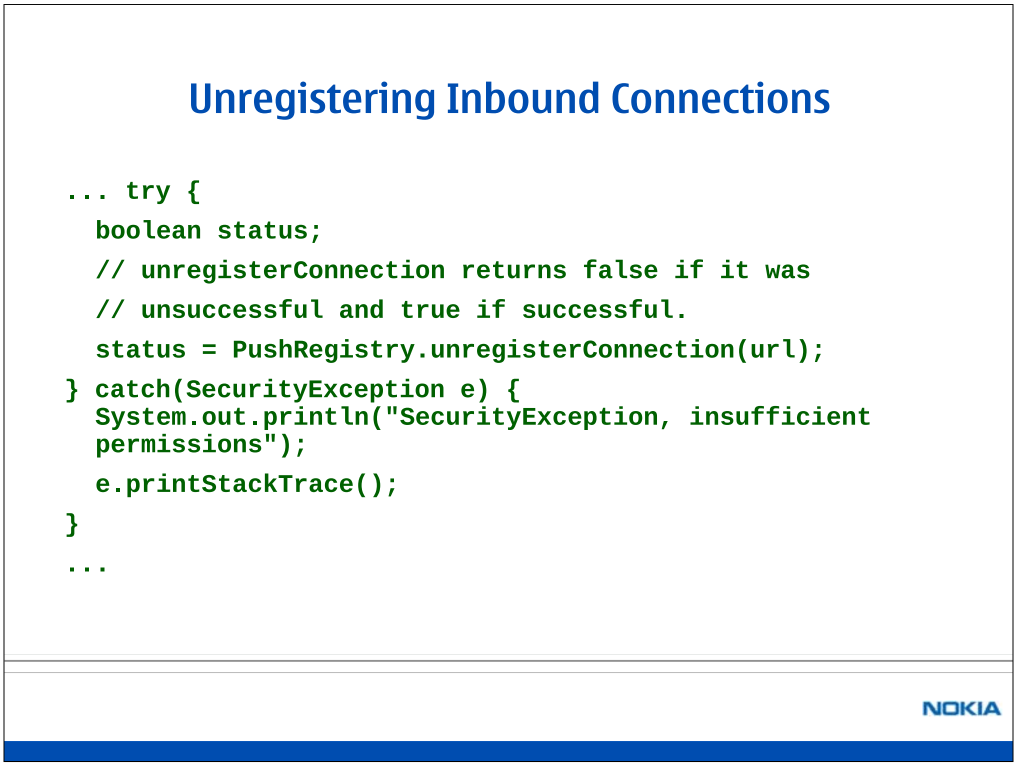


Dynamic Registration - Inbound Connection (2)

- Datagram Example

```
// Register a static connection.
String url = "datagram://:5000";
// Open the connection.
UDPDatagramConnection udgc = (UDPDatagramConnection)Connector.open(url);
// Register the connection
PushRegistry.registerConnection(url, midletClassName, “*”);
// Now wait for inbound network activity.
Datagram dg = udgc.newDatagram(udgc.getNomialLength());
udgc.receive(dg);
// Read the inbound messages
```

Registering an inbound datagram connection is done much the same way as the Socket example on the previous slide. You just use a DatagramConnection and a Datagram instead

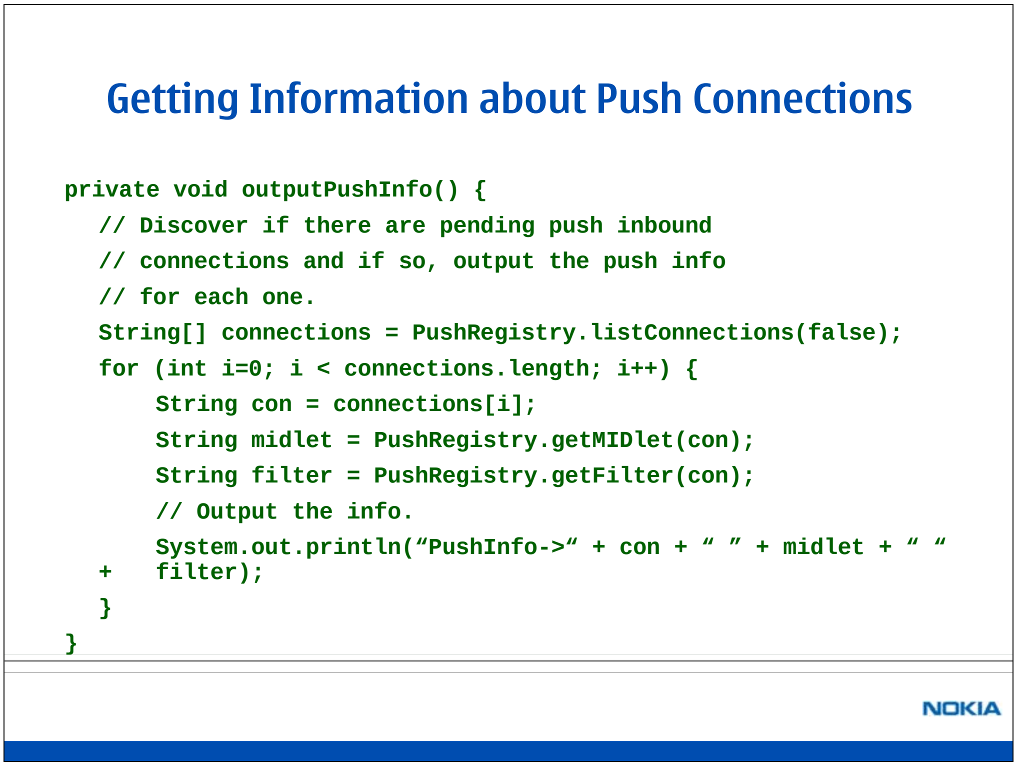


Unregistering Inbound Connections

```
... try {
  boolean status;
  // unregisterConnection returns false if it was
  // unsuccessful and true if successful.
  status = PushRegistry.unregisterConnection(url);
} catch(SecurityException e) {
  System.out.println("SecurityException, insufficient
  permissions");
  e.printStackTrace();
}
...
```

Because the AMS maintains the registration even after the MIDlet exits, it's important that the MIDlet unregister the connection when it's no longer needed. To unregister an inbound connection, use the unregisterConnection() method.

The method returns true if it was successful and false if it fails to unregister the connection, for example if the argument is null, or it specifies a connection that hasn't been registered. The method throws SecurityException if the specified MIDlet has been registered by another MIDlet suite.



Getting Information about Push Connections

```
private void outputPushInfo() {
  // Discover if there are pending push inbound
  // connections and if so, output the push info
  // for each one.
  String[] connections = PushRegistry.listConnections(false);
  for (int i=0; i < connections.length; i++) {
      String con = connections[i];
      String midlet = PushRegistry.getMIDlet(con);
      String filter = PushRegistry.getFilter(con);
      // Output the info.
      System.out.println("PushInfo->" + con + " " + midlet + " "
  +   filter);
  }
}
```

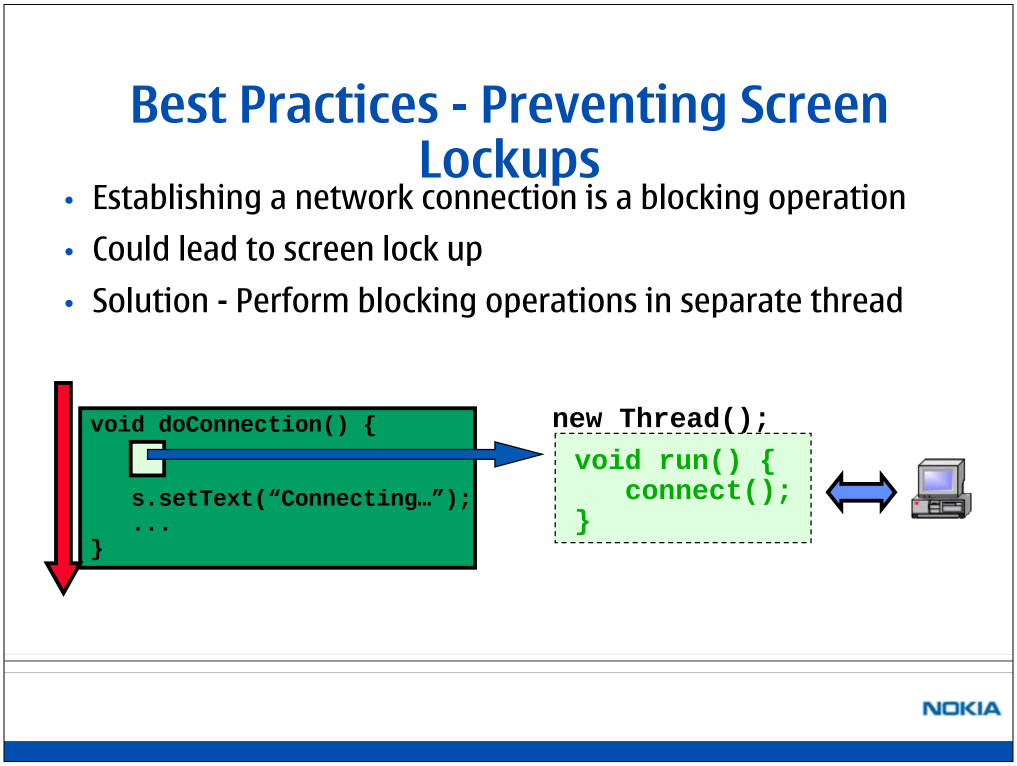
The push registry provides two methods to retrieve information about a registered connection - specifically, the information defined by the MIDlet-Push property or by a call to registerConnection():

- *getMIDlet() retrieves the MIDlet responsible for a particular connection.*
- *getFilter() retrieves the filter for a particular connection.*

The PushRegistry.listConnections() method allows you to discover all the inbound connections registered by the MIDlet suite. You can also use it to discover whether the MIDlet was activated by an incoming connection.

If you pass false to listConnections(), the method returns a string array that identifies all the inbound connections registered by the MIDlet suite, but if you pass a true argument the method reports only the registered connections with available data - indicating that MIDlet activation was due to incoming network data .

This method calls listConnection() with a false argument, then for each connection it retrieves and reports the associated MIDlet and filter information.



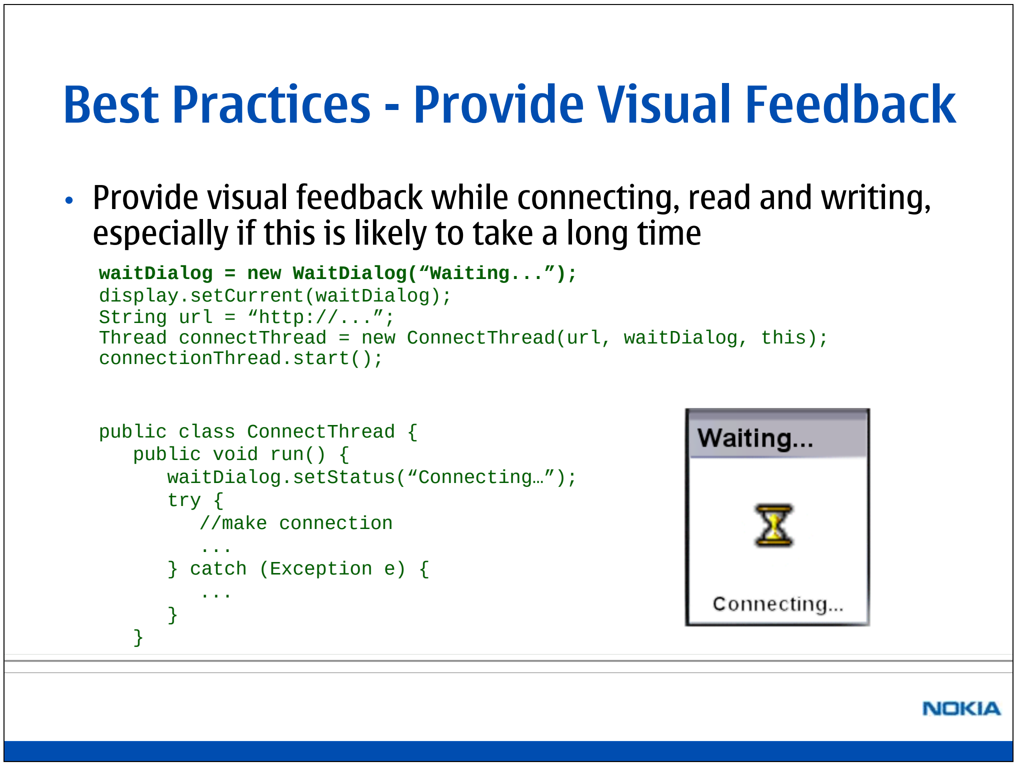
Best Practices - Preventing Screen Lockups

- Establishing a network connection is a blocking operation
- Could lead to screen lock up
- Solution - Perform blocking operations in separate thread

```
void doConnection() {

   s.setText("Connecting…");
   ...
}
```

```
new Thread();
void run() {
   connect();
}
```

To avoid hang-ups in the main UI thread is to put all lengthy or potentially blocking operations in separate threads. Network calls can block the user interface from redrawing or responding to user input until the call returns. In some cases having the user interface and a network connection on the same thread can cause a device to deadlock.



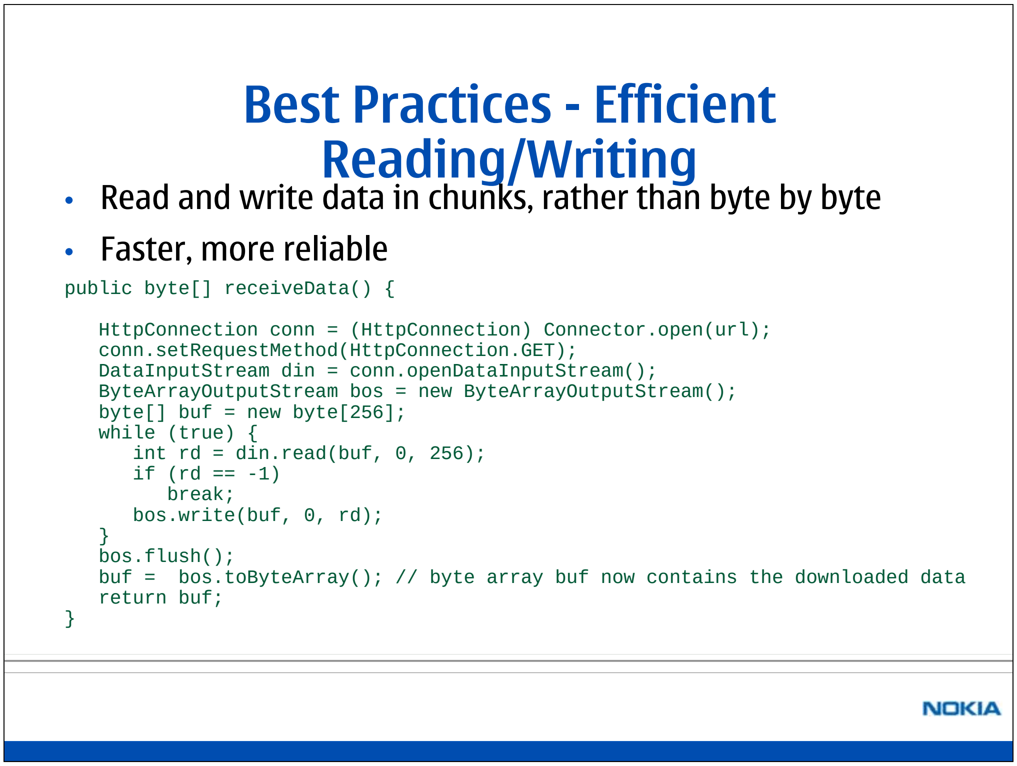
Best Practices - Provide Visual Feedback

- Provide visual feedback while connecting, read and writing, especially if this is likely to take a long time

```
waitDialog = new WaitDialog(“Waiting...”);
display.setCurrent(waitDialog);
String url = “http://...”;
Thread connectThread = new ConnectThread(url, waitDialog, this);
connectionThread.start();


public class ConnectThread {
   public void run() {
      waitDialog.setStatus(“Connecting…”);
      try {
         //make connection
         ...
      } catch (Exception e) {
         ...
      }
   }
```

Since the devices that MIDP application execute on could have limited resources, it likely that the network connection hardware on the device could also be limited. Keeping this in mind, when any network connections are made from the device, it may take longer than say 3 seconds for the connection to be made. To indicate to the user of the application that it is making connection, or doing something that could take a while, you should provide visual feedback.

Visual feedback could be in the form of a animated sand timer, a clock, or simply some text on the screen explaining the current status of the connection.



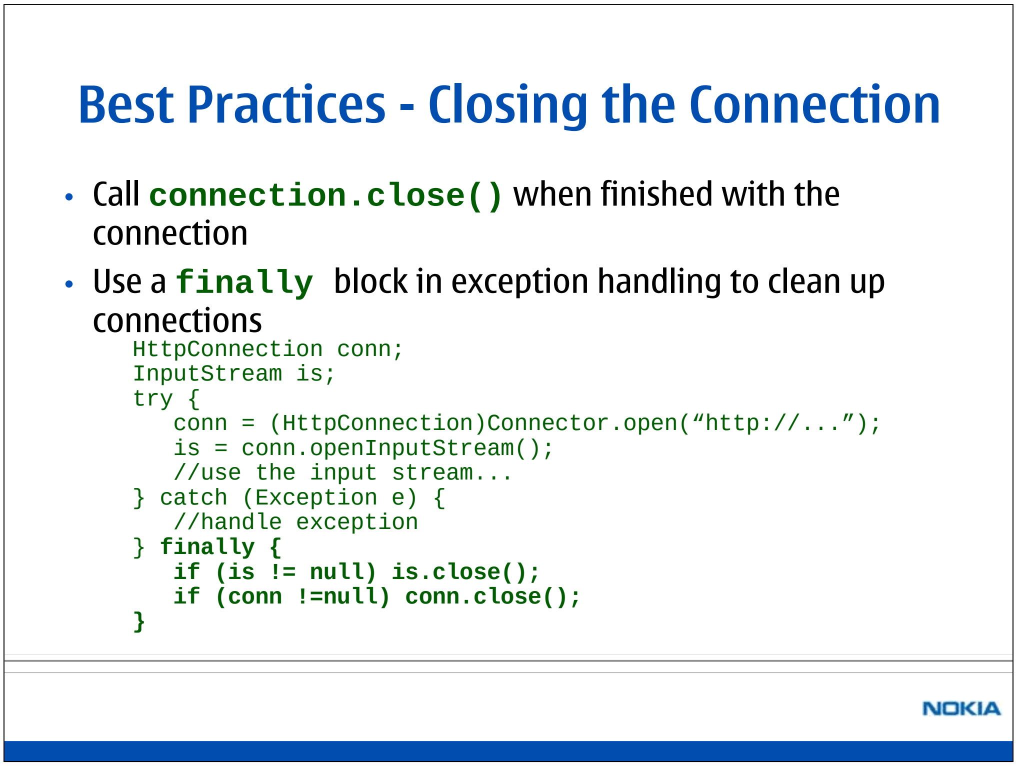
Best Practices - Efficient Reading/Writing

- Read and write data in chunks, rather than byte by byte
- Faster, more reliable

```
public byte[] receiveData() {

   HttpConnection conn = (HttpConnection) Connector.open(url);
   conn.setRequestMethod(HttpConnection.GET);
   DataInputStream din = conn.openDataInputStream();
   ByteArrayOutputStream bos = new ByteArrayOutputStream();
   byte[] buf = new byte[256];
   while (true) {
      int rd = din.read(buf, 0, 256);
      if (rd == -1)
         break;
      bos.write(buf, 0, rd);
   }
   bos.flush();
   buf =  bos.toByteArray(); // byte array buf now contains the downloaded data
   return buf;
}
```

To ensure the data your application is sending and receiving over the network is done quickly and in a reliable way, the data should be handled in chunks, rather than byte by byte. The code below shows how to send an array of bytes over a HttpConnection by using a DataOutputStream and its method write(byte[] b, int offset, int length)

```
public String sendData(byte[] data) throw IOException {
             HttpConnection hc = (HttpConnection)Connector.open(url,
Connector.WRITE);
             hc.setRequestMethod(HttpConnection.POST);
             DataOutputStream dos = hc.openDataOutputStream();
             dos.write(data, 0, data.length);
             dos.flush();
             dos.close();
}
```



Best Practices - Closing the Connection

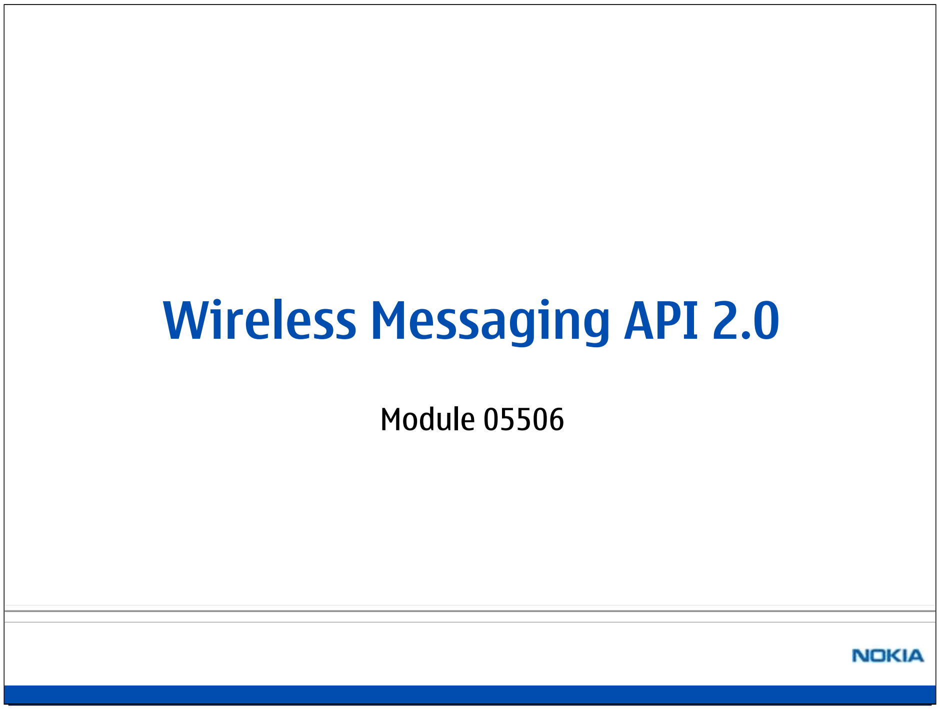
- Call **`connection.close()`** when finished with the connection
- Use a **`finally`** block in exception handling to clean up connections

```
HttpConnection conn;
InputStream is;
try {
   conn = (HttpConnection)Connector.open(“http://...”);
   is = conn.openInputStream();
   //use the input stream...
} catch (Exception e) {
   //handle exception
} finally {
   if (is != null) is.close();
   if (conn !=null) conn.close();
}
```

You should remember to close any connection once you have finished using it. This prevents resource leak, which could potentially cause your device to become unstable. To guarantee that the close() method is called in a block of code you should consider using the finally construct.

The finally construct enables code to execute whether or not an exception occurred. Using finally is good to maintain the internal state of an object and to clean up non-memory resources.

A finally block ensures the close method is executed whether or not an exception is thrown from within the try block. Therefore, the close method is guaranteed to be called before the method exits. You are then sure the socket is closed and you have not leaked a resource.



Wireless Messaging API 2.0

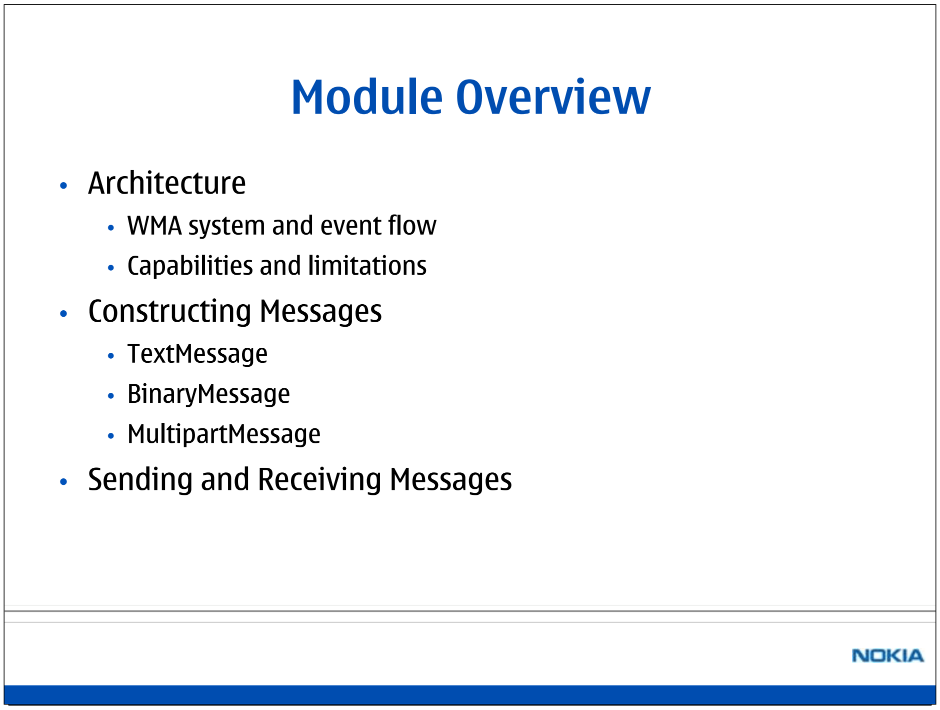
Module 05506

The WMA (Wireless Messaging API) as defined in JSR 120 enables the applications developer to use a set of APIs for mobile communications via SMS. SMS messages follow the same conventions as normal SMS messages that your phone can already send (incl. binary data) with a few caveats which are discussed below.

WMA 2.0 as defined in JSR 205 add the ability to send MMS.

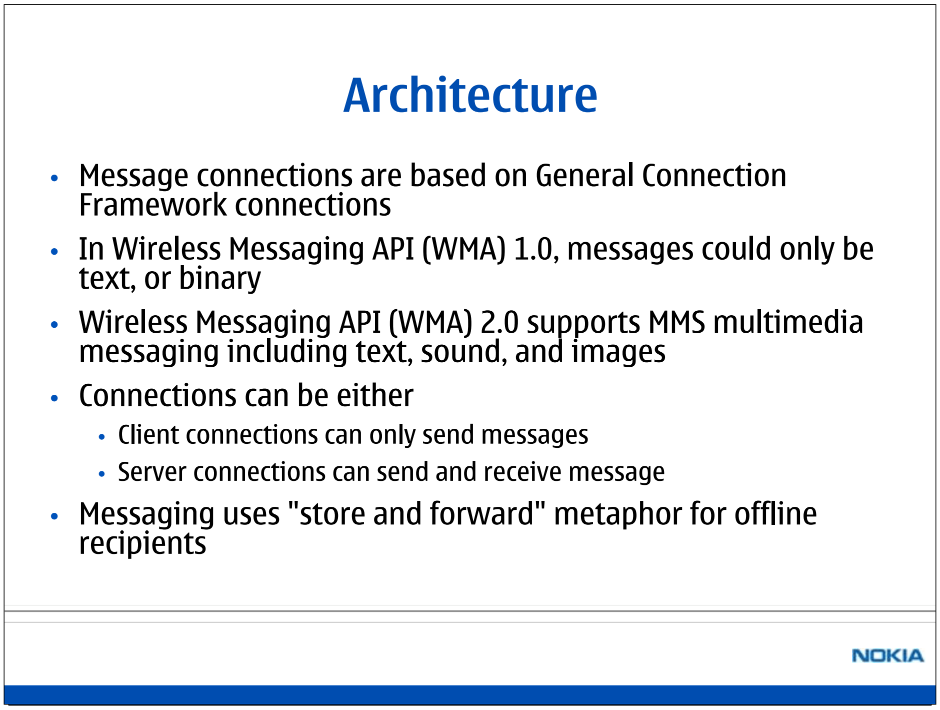
The WMA is also conceptually similar to the Nokia SMS API which will eventually become a deprecated API for future generations of phones that support Java ME, however for backwards compatibility the developer can structure the MIDlet to support both APIs. This technique is described fully in the accompanying lab for this module.

The core functionality of the WMA is to allow the developer to construct SMS messages and send and receive them. MIDlets that employ the WMA can either act as servers or clients (senders or receivers) or both as will be explained in this course and demonstrated in the lab accompanying this module.



Module Overview

- Architecture
  - WMA system and event flow
  - Capabilities and limitations
- Constructing Messages
  - TextMessage
  - BinaryMessage
  - MultipartMessage
- Sending and Receiving Messages



Architecture

- Message connections are based on General Connection Framework connections
- In Wireless Messaging API (WMA) 1.0, messages could only be text, or binary
- Wireless Messaging API (WMA) 2.0 supports MMS multimedia messaging including text, sound, and images
- Connections can be either
  - Client connections can only send messages
  - Server connections can send and receive message
- Messaging uses "store and forward" metaphor for offline recipients

The WMA is based on the Generic Construction Framework which presents a coherent way to access all manner of data using a simple string text format resembling a URL. WMA connections are opened and closed in a similar fashion to UDP messages however there are some subtle differences. UDP datagrams are only binary and WMA messages can be text or binary. Also WMA messages use a store and forward mechanism for message dispatch because the message recipient could be offline at the time of initial message dispatch. WMA connections can be opened in server and client mode. In client mode the GCF connection URL points to the device that will receive the message and the client connection can only send messages. In server mode the connection is opened by specifying the URL as an endpoint to listen for messages and the connection thereof can be used for both receiving and sending messages. Message formats consist of an address part and a data part and the data format can be binary as indicated above and are managed by specific interfaces that are part of the WMA API.

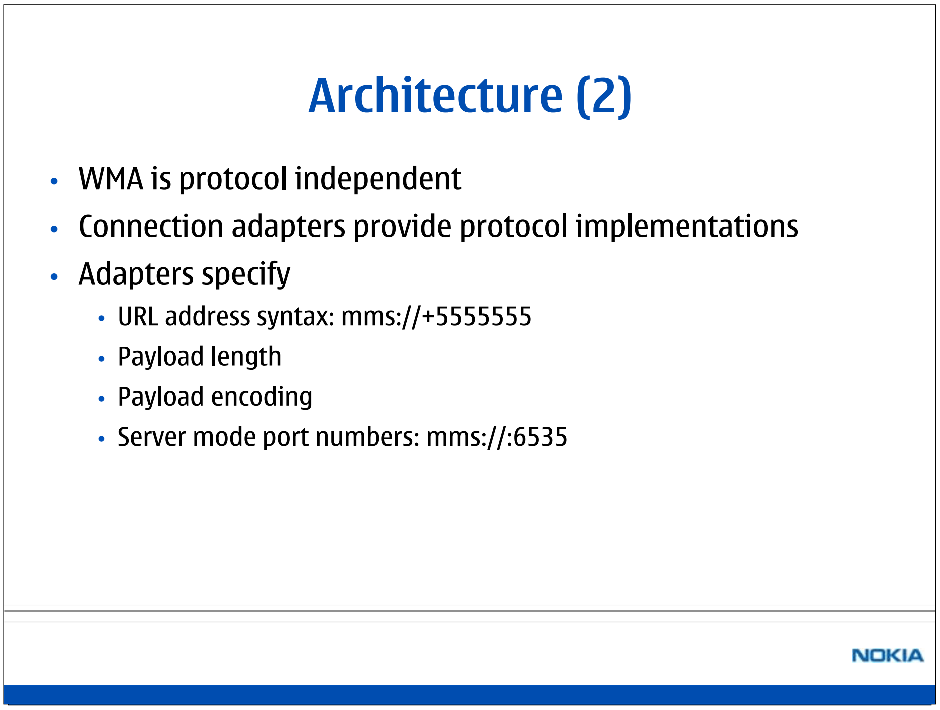
An important part of using the WMA is the concept of connection adapters. Adapters are vendor specific implementations of a connection protocol and determine the way a connection URL is formatted – i.e. the connection URL scheme. There are different types of adapters for different types of connections and the important issue to note is the format of the connection URL must be in accordance with the specification for the protocol scheme as implemented by the connection adapter. For example any string that contains

sms://

as part of the connection URL describes the connection scheme to send a SMS message. So to create a message connection we can do it like this:

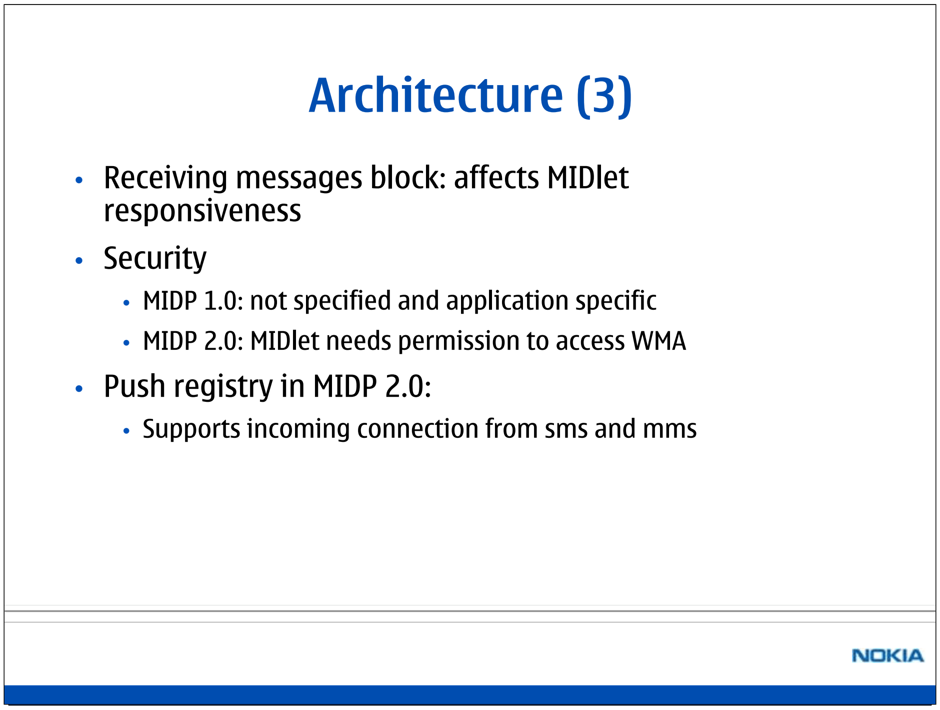
```
MessageConnection conn = (MessageConnection)Connection.open(“sms://+3725555555”);
```

Applications need permission to send and receive messages. In MIDP 1.0 there is no formal mechanism to specify this and can be set at the user level within the MIDlet. In MIDP 2.0 the MIDlet suite must have the appropriate permissions to access the WMA connection API. Care must be taken to catch the SecurityException exception if the appropriate permissions are not in place.



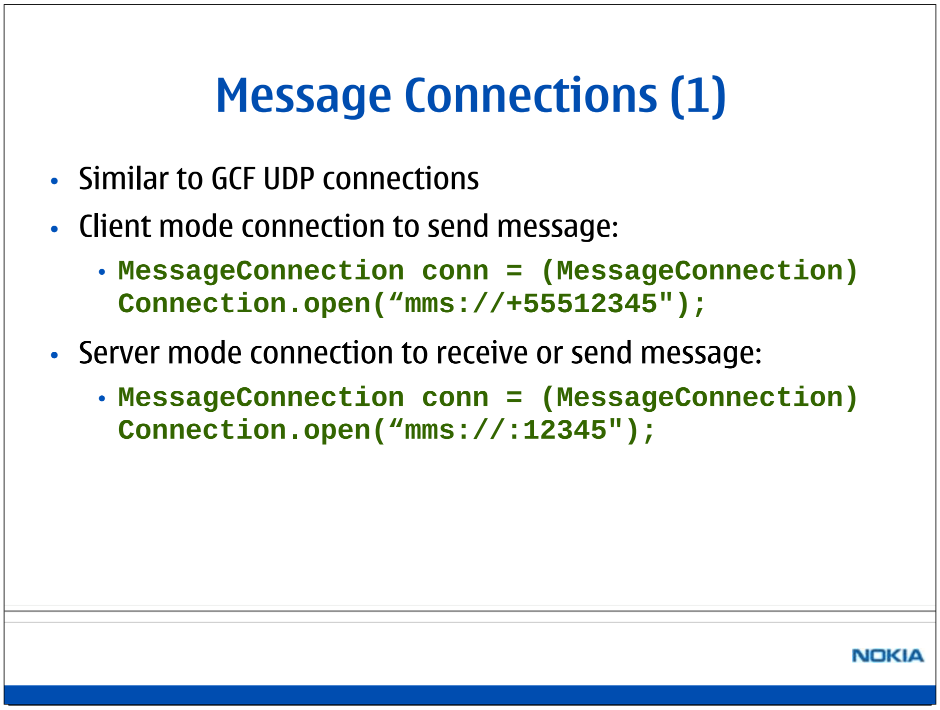
Architecture (2)

- WMA is protocol independent
- Connection adapters provide protocol implementations
- Adapters specify
  - URL address syntax: mms://+5555555
  - Payload length
  - Payload encoding
  - Server mode port numbers: mms://:6535



Architecture (3)

- Receiving messages block: affects MIDlet responsiveness
- Security
  - MIDP 1.0: not specified and application specific
  - MIDP 2.0: MIDlet needs permission to access WMA
- Push registry in MIDP 2.0:
  - Supports incoming connection from sms and mms



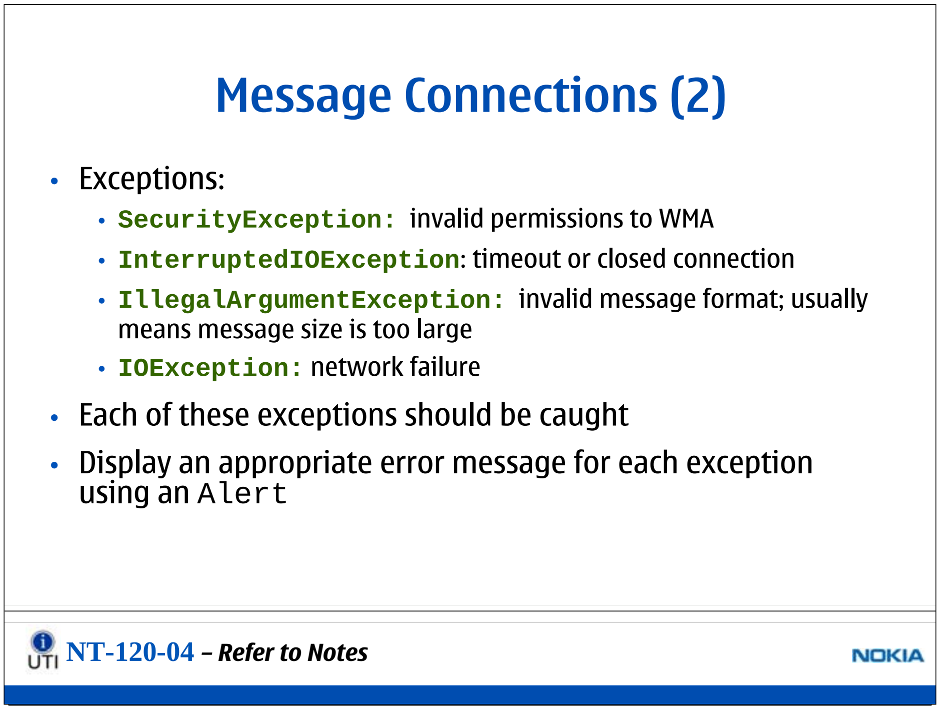
Message Connections (1)

- Similar to GCF UDP connections
- Client mode connection to send message:

  -
    ```
    MessageConnection conn = (MessageConnection)
    Connection.open(“mms://+55512345");
    ```
- Server mode connection to receive or send message:

  -
    ```
    MessageConnection conn = (MessageConnection)
    Connection.open(“mms://:12345");
    ```

NOKIA



Message Connections (2)

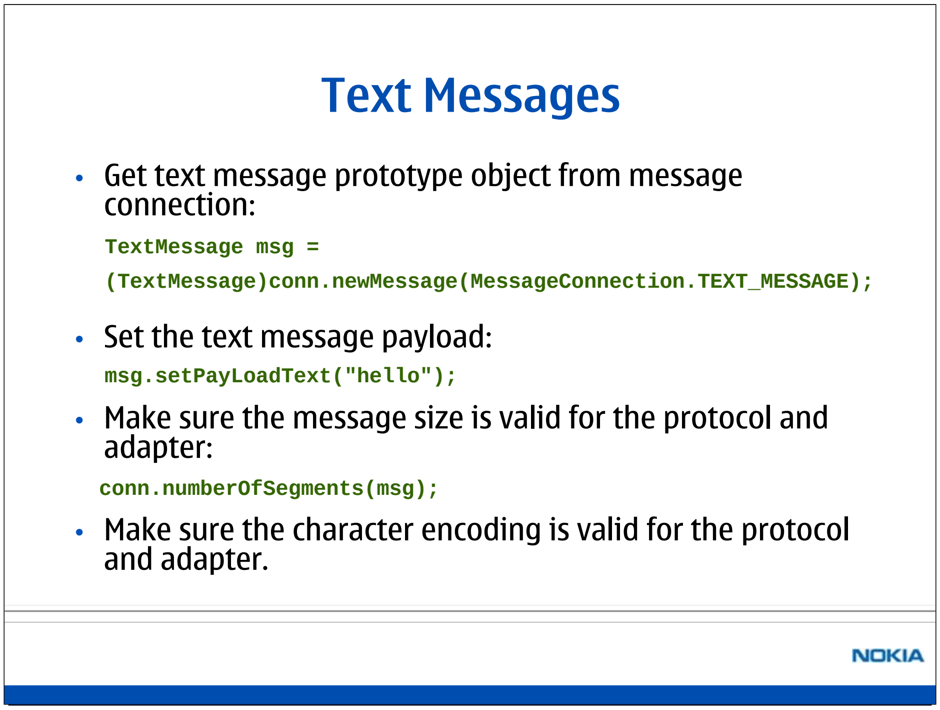
- Exceptions:
  - `SecurityException:` invalid permissions to WMA
  - `InterruptedIOException`: timeout or closed connection
  - `IllegalArgumentException:` invalid message format; usually means message size is too large
  - `IOException:` network failure
- Each of these exceptions should be caught
- Display an appropriate error message for each exception using an `Alert`

UTI **NT-120-04** – ***Refer to Notes***

NOKIA

NT-120-04 - Verify the application presents an accurate and appropriate error message to user, if the handset is unable to send the SMS message due to external factors (e.g. network connectivity, etc).

As the slide explains, there a quite a few exceptions to handle when dealing with the MessageConnection interface. To present and accurate and appropriate error message to the user, your application needs to handle each individual exception that is thrown by the methods of the MessageConnection interface. You should avoid wrapping all exception handing up into a general IOException or Exception. When an exception is thrown, your application should report an error message using an Alert.



Text Messages

- Get text message prototype object from message connection:

```
TextMessage msg =
(TextMessage)conn.newMessage(MessageConnection.TEXT_MESSAGE);
```

- Set the text message payload:

```
msg.setPayLoadText("hello");
```

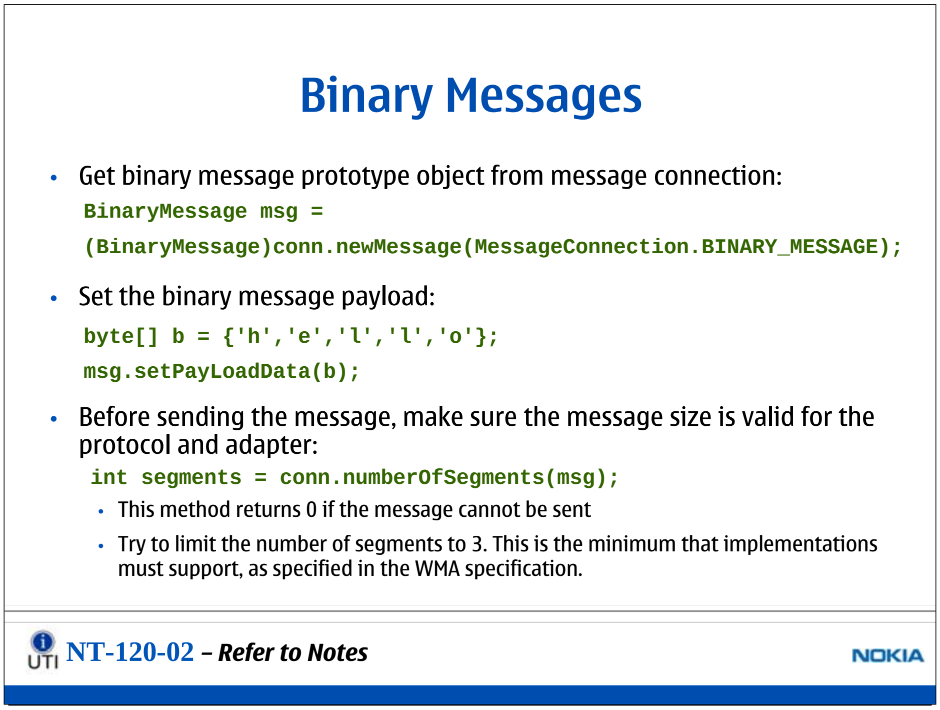
- Make sure the message size is valid for the protocol and adapter:

```
conn.numberOfSegments(msg);
```

- Make sure the character encoding is valid for the protocol and adapter.

Messages are constructed by first determining the type of message to send – binary or text – and then obtaining a connection to the remote device. Once that is established the payload is set and the message can be sent.

```
TextMessage tmsg =
(TextMessage)conn.newMessage(MessageConnection.TEXT_MESSAGE);
             tmsg.setPayloadText(“my message”);  // construct a TEXT
messages
```



Binary Messages

- Get binary message prototype object from message connection:

```
BinaryMessage msg =
(BinaryMessage)conn.newMessage(MessageConnection.BINARY_MESSAGE);
```

- Set the binary message payload:

```
byte[] b = {'h','e','l','l','o'};
msg.setPayLoadData(b);
```

- Before sending the message, make sure the message size is valid for the protocol and adapter:

```
int segments = conn.numberOfSegments(msg);
```

  - This method returns 0 if the message cannot be sent
  - Try to limit the number of segments to 3. This is the minimum that implementations must support, as specified in the WMA specification.

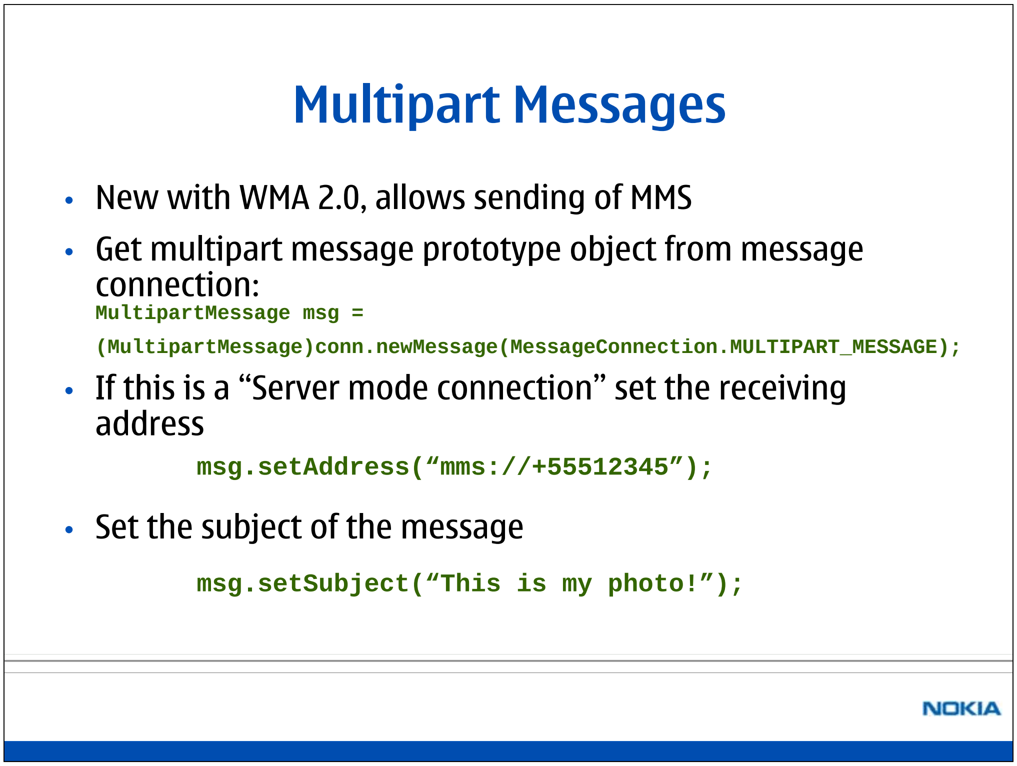
UTI **NT-120-02** – ***Refer to Notes***

NOKIA

```
byte[] ba = {'m','e','s','s'.'a','g','e'};
             BinaryMessage bmsg =
(BinaryMessage)conn.newMessage(MessageConnection.BINARY_MESSAGE);
             bmsg.setPayloadData(ba);  // construct a BINARY messages
```

NT-120-02 - Verify the message is formatted appropriately if being sent to the handset mailbox.

As the slide explains, before attempting to send a message, ensure to check the message size is valid for the protocol and adapter. This can be done by using the numberOfSegments method in the MessageConnection class. This method returns the number of protocol segments that are needed to send this message. It returns 0 if the message cannot be sent using the underlying protocol. The WMA specification mandates that implementations much support a minimum of 3 segments, so to maintain portability, try to the limit the number of segments in your application to 3.



Multipart Messages

- New with WMA 2.0, allows sending of MMS
- Get multipart message prototype object from message connection:

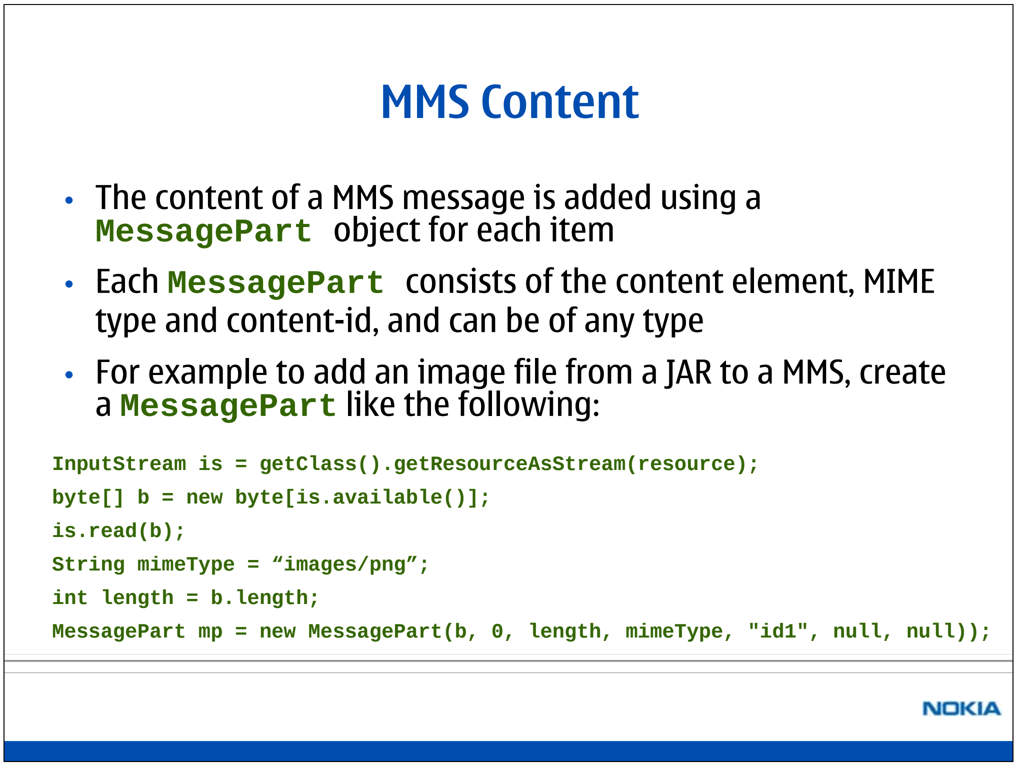
```
MultipartMessage msg =
(MultipartMessage)conn.newMessage(MessageConnection.MULTIPART_MESSAGE);
```

- If this is a “Server mode connection” set the receiving address

```
        msg.setAddress(“mms://+55512345”);
```

- Set the subject of the message

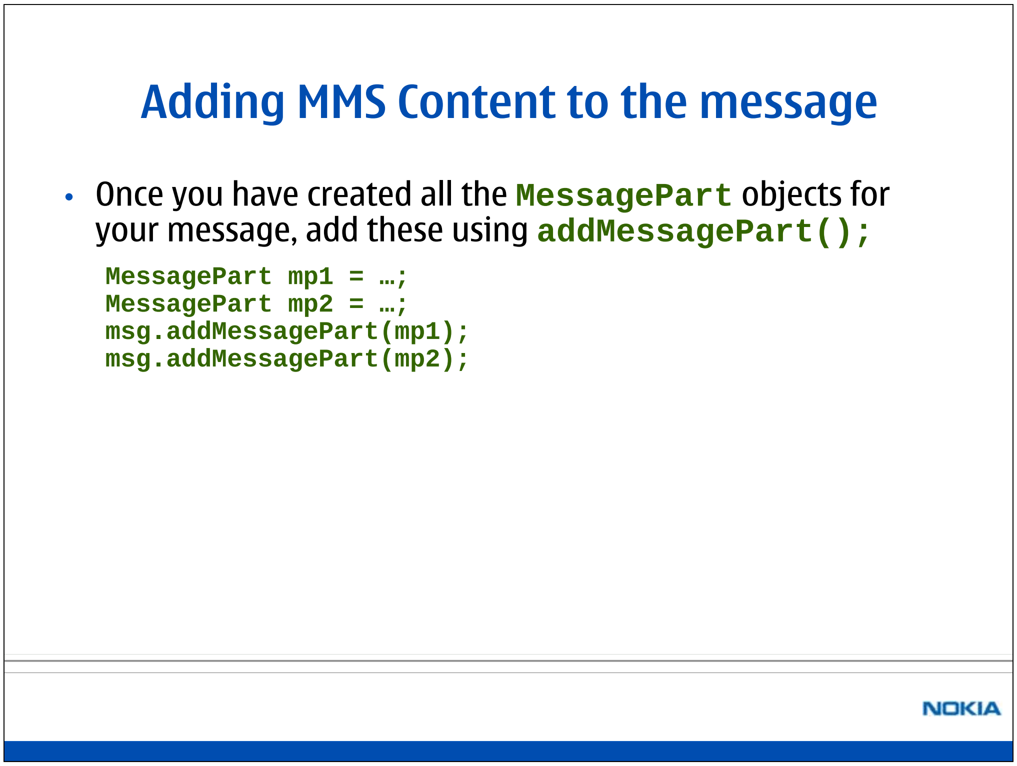
```
        msg.setSubject(“This is my photo!”);
```



MMS Content

- The content of a MMS message is added using a `MessagePart` object for each item
- Each `MessagePart` consists of the content element, MIME type and content-id, and can be of any type
- For example to add an image file from a JAR to a MMS, create a `MessagePart` like the following:

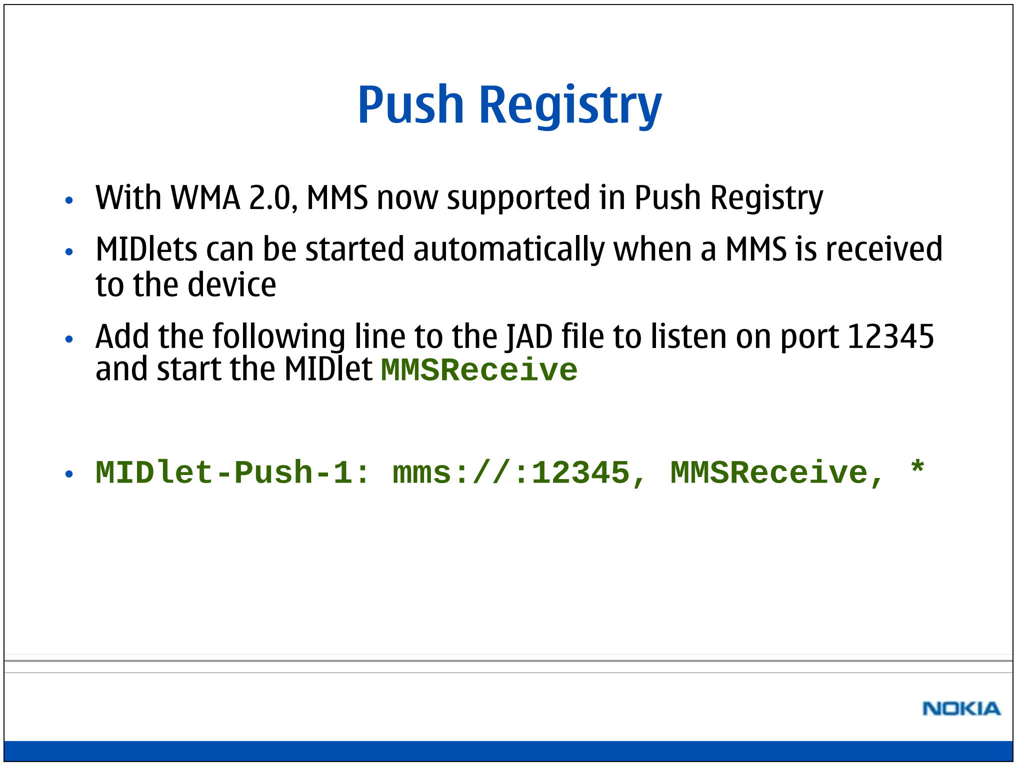
```
InputStream is = getClass().getResourceAsStream(resource);
byte[] b = new byte[is.available()];
is.read(b);
String mimeType = “images/png”;
int length = b.length;
MessagePart mp = new MessagePart(b, 0, length, mimeType, "id1", null, null));
```



Adding MMS Content to the message

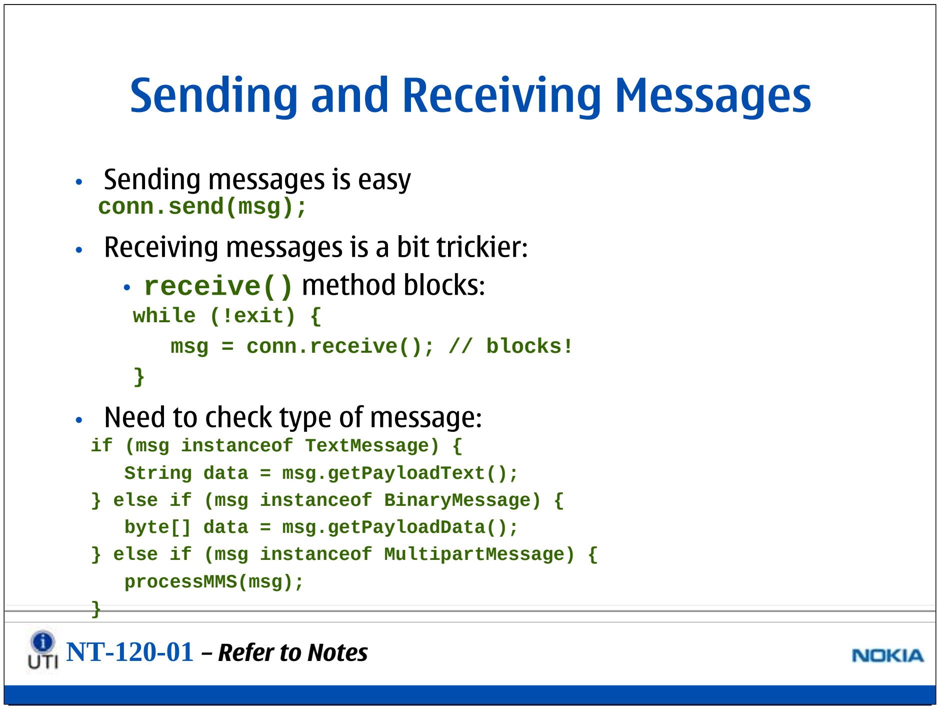
- Once you have created all the `MessagePart` objects for your message, add these using `addMessagePart();`

```
MessagePart mp1 = …;
MessagePart mp2 = …;
msg.addMessagePart(mp1);
msg.addMessagePart(mp2);
```



Push Registry

- With WMA 2.0, MMS now supported in Push Registry
- MIDlets can be started automatically when a MMS is received to the device
- Add the following line to the JAD file to listen on port 12345 and start the MIDlet `MMSReceive`
- `MIDlet-Push-1: mms://:12345, MMSReceive, *`



Sending and Receiving Messages

- Sending messages is easy

```
conn.send(msg);
```

- Receiving messages is a bit trickier:
  - `receive()` method blocks:

```
while (!exit) {
   msg = conn.receive(); // blocks!
}
```

- Need to check type of message:

```
if (msg instanceof TextMessage) {
   String data = msg.getPayloadText();
} else if (msg instanceof BinaryMessage) {
   byte[] data = msg.getPayloadData();
} else if (msg instanceof MultipartMessage) {
   processMMS(msg);
}
```

NT-120-01 – ***Refer to Notes***

NOKIA

Messages are sent in a straightforward manner by calling the send method on the connection after the payload has been set.

```
          conn.send(); // sends message
```

You should also confirm that the message has been sent correctly by displaying an information screen

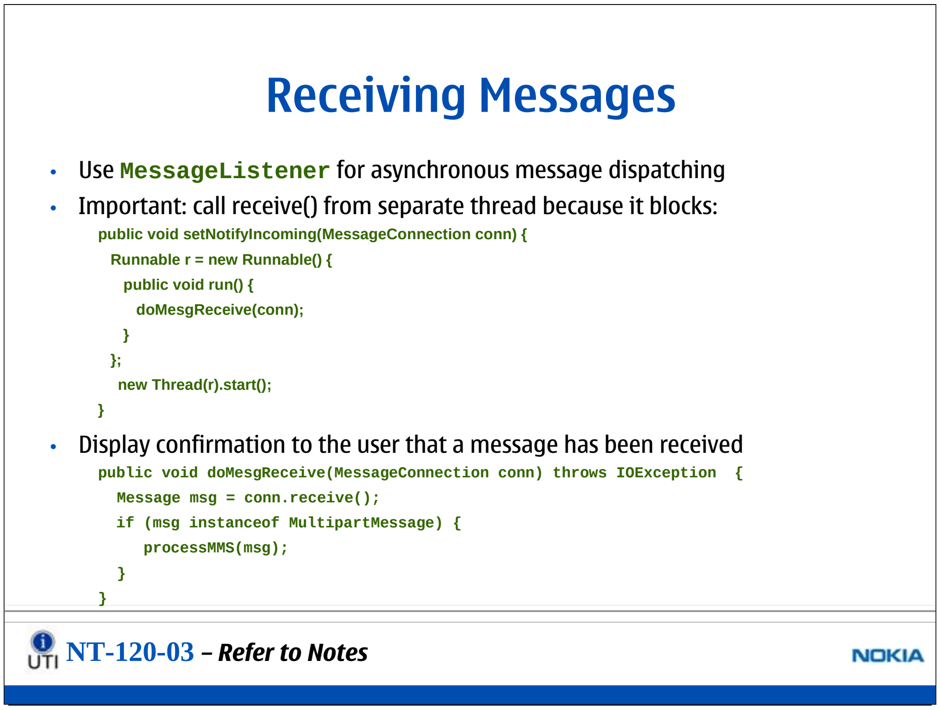
```
public void displayMessageSentConfirmation(TextMessage msg, String to) {
   Alert alert = new Alert("SMS Sent!");
   alert.setString("\nTo :" + to);
   alert.addCommand(okCommand);
   alert.setCommandListener(this);
   display.setCurrent(alert);
}
```

So putting together the above code snippets, the following illustrates how to send a text message, confirm to the user the message has been sent and to trap the appropriate exceptions – individual exceptions are discussed elsewhere in the course.

```
try {
   MessageConnection conn = (MessageConnection)Connection.open(“sms://+3725555555”);
   TextMessage tmsg = (TextMessage)conn.newMessage(MessageConnection.TEXT_MESSAGE);
   tmsg.setPayloadText(“my message”);  // construct a TEXT messages
   conn.send(); // sends messages in the message queue.
   displayMessageSentConfirmation(tmsg, “+3725555555”);
} catch (IOException e) {
} catch (InterruptedIOException e) {
} catch (SecurityException e) {
} catch (IllegalArgumentException) {
} catch (Exception e) {
}
```

NT-120-01 - Verify MIDlet successfully sends the SMS message. This test verifies that the WMA method is used correctly.

As the slide shows, when the connection.send(msg) method has completed, you should display an information screen to tell the user that the SMS has been sent correctly. Use an Alert class to do this, possibly including details such as the senders address and the current time etc.



Receiving Messages

- Use `MessageListener` for asynchronous message dispatching
- Important: call receive() from separate thread because it blocks:

```
public void setNotifyIncoming(MessageConnection conn) {
  Runnable r = new Runnable() {
    public void run() {
      doMesgReceive(conn);
    }
  };
   new Thread(r).start();
}
```

- Display confirmation to the user that a message has been received

```
public void doMesgReceive(MessageConnection conn) throws IOException  {
   Message msg = conn.receive();
   if (msg instanceof MultipartMessage) {
       processMMS(msg);
   }
}
```

NT-120-03 – ***Refer to Notes***

Messages can be received by a server either in polling mode or as asynchronous callbacks via an event listener interface. In either case – esp polling – the server can be set up as a separate thread so as not to block the application. As explained above, a connection is determined to be a server connection if the connection is opened with the URL describing an endpoint and not the address of a recipient. Endpoints are protocol dependent and describe the port that server should listen to for incoming messages.

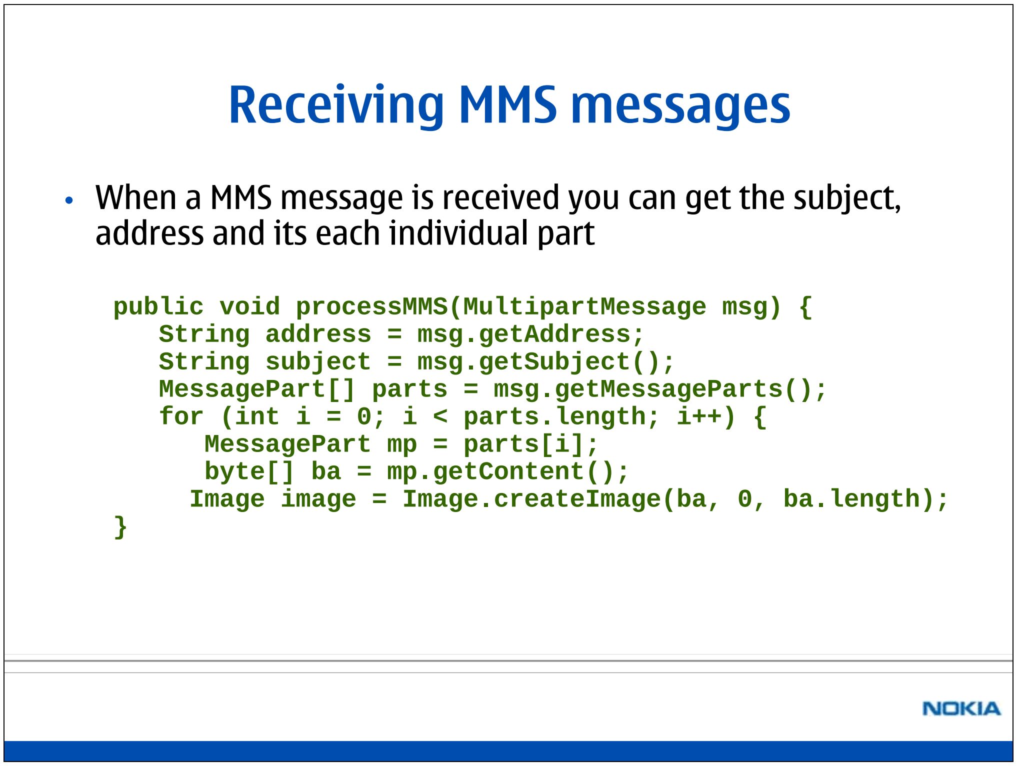
```
MessageConnection conn = (MessageConnection)Connection.open(“sms://6535”);
```

Care should be taken to check for the type of message – either binary or text – to determine the appropriate code to unpack the message.

```
Messages msg = conn.receive();
if (msg instanceof TextMessage) {
               // process text message
} else {
               // process as binary message
}
```

You should provide indicate to the user that a message has been received

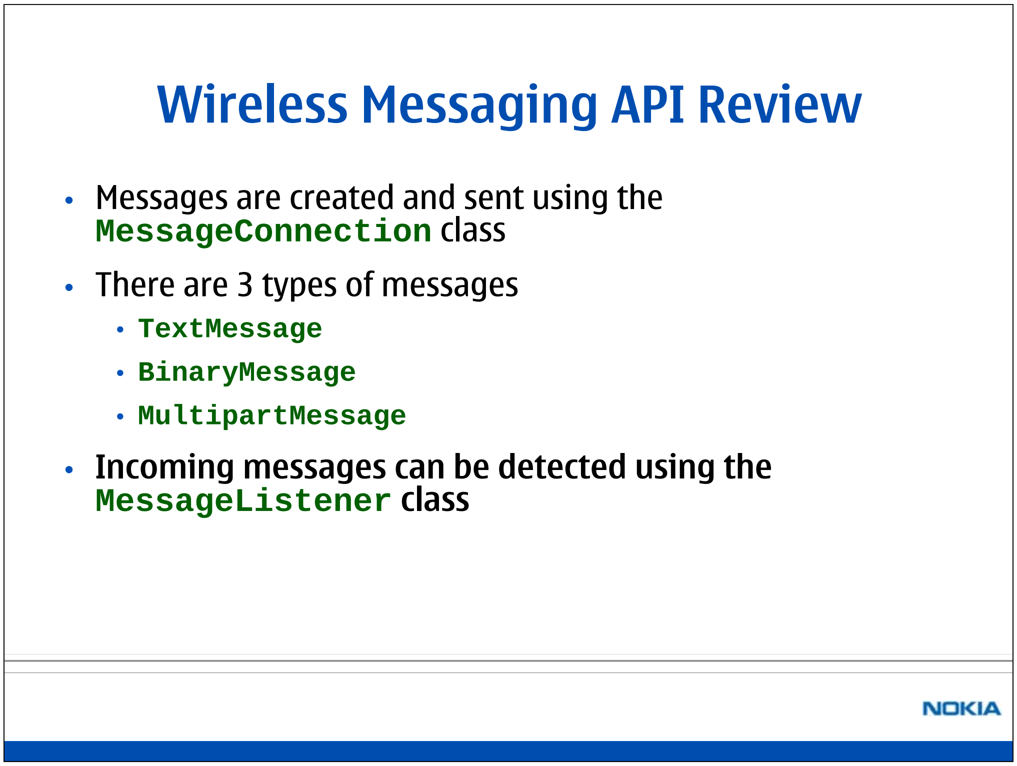
```
public void displayMessageReceivedConfirmation (TextMessage msg) {
    receivedMessage = msg;
    Alert alert = new Alert("SMS Received!");
    alert.setString("\nFrom :" + msg.getAddress());
    alert.addCommand(readCommand);
    alert.addCommand(cancelCommand);
    alert.setCommandListener(this);
    display.setCurrent(alert);
  }
```



Receiving MMS messages

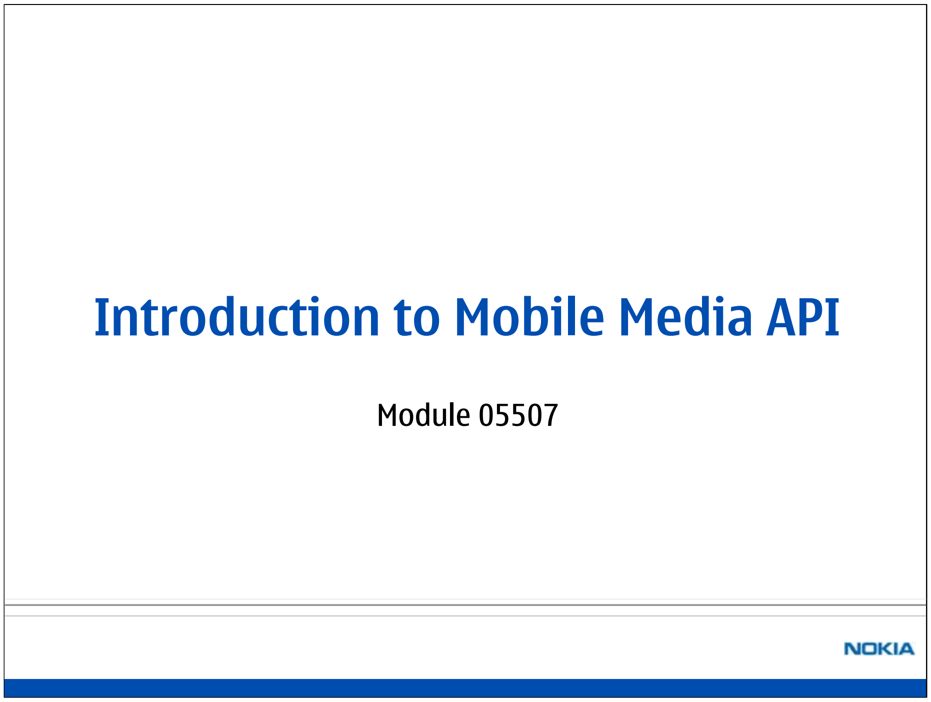
- When a MMS message is received you can get the subject, address and its each individual part

```
public void processMMS(MultipartMessage msg) {
   String address = msg.getAddress;
   String subject = msg.getSubject();
   MessagePart[] parts = msg.getMessageParts();
   for (int i = 0; i < parts.length; i++) {
      MessagePart mp = parts[i];
      byte[] ba = mp.getContent();
     Image image = Image.createImage(ba, 0, ba.length);
}
```



Wireless Messaging API Review

- Messages are created and sent using the `MessageConnection` class
- There are 3 types of messages
  - `TextMessage`
  - `BinaryMessage`
  - `MultipartMessage`
- Incoming messages can be detected using the `MessageListener` class



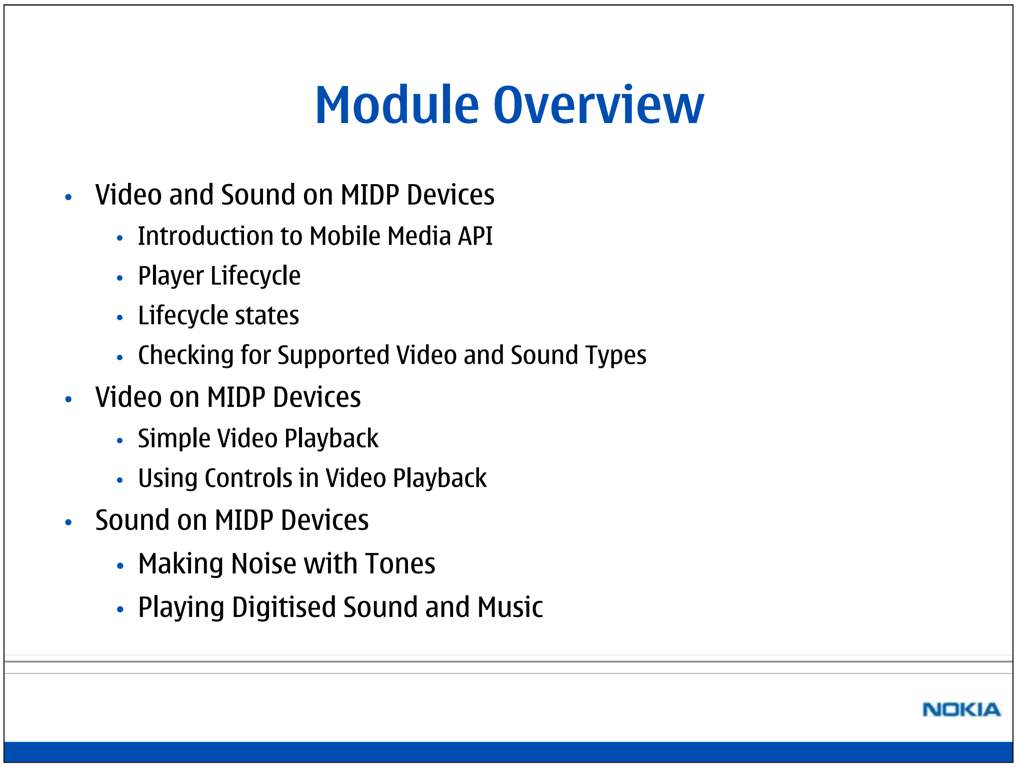
Introduction to Mobile Media API

Module 05507

With the availability of many different formats for rich media, the MIDlet programmer can use the Mobile Media API (MMA) to access several rich media formats in a unified manner. The API is flexible enough so that media can be accessed as either streamed across a network or directly from a JAR file. In addition to standard supported media protocols, custom protocol handlers can also be implemented as well.

After completing this module the developer should be able to:

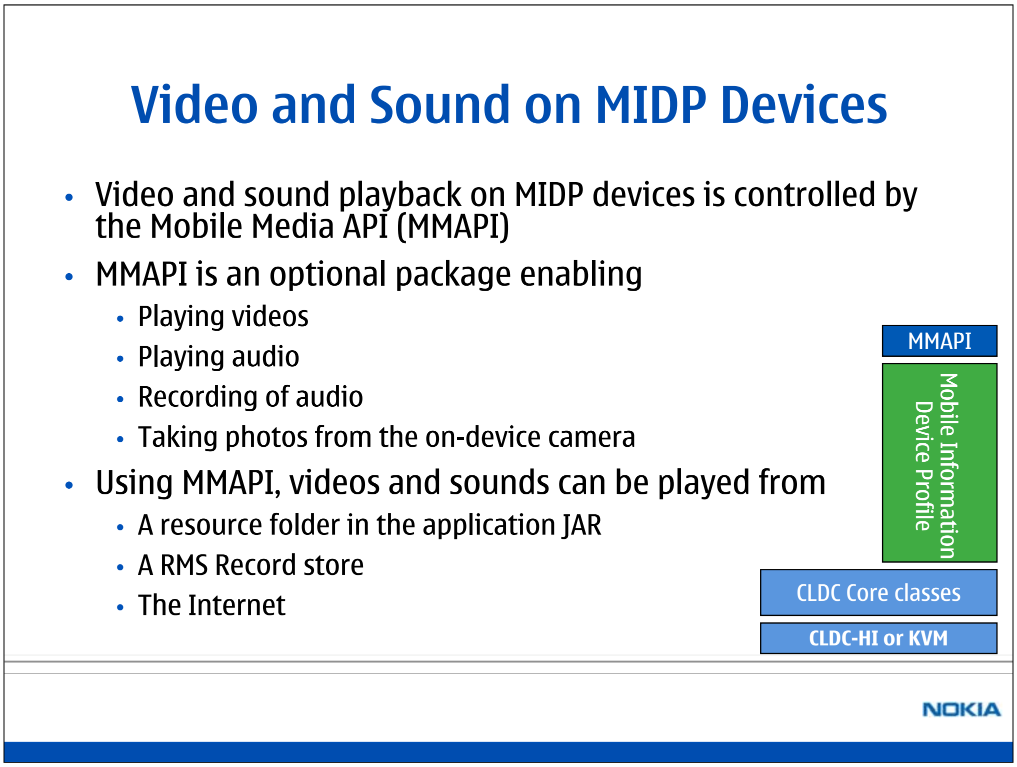
- ***Render rich media***
- ***Control how media is rendered***
- ***Dispatch media events***



Module Overview

- Video and Sound on MIDP Devices
  - Introduction to Mobile Media API
  - Player Lifecycle
  - Lifecycle states
  - Checking for Supported Video and Sound Types
- Video on MIDP Devices
  - Simple Video Playback
  - Using Controls in Video Playback
- Sound on MIDP Devices
  - Making Noise with Tones
  - Playing Digitised Sound and Music

NOKIA

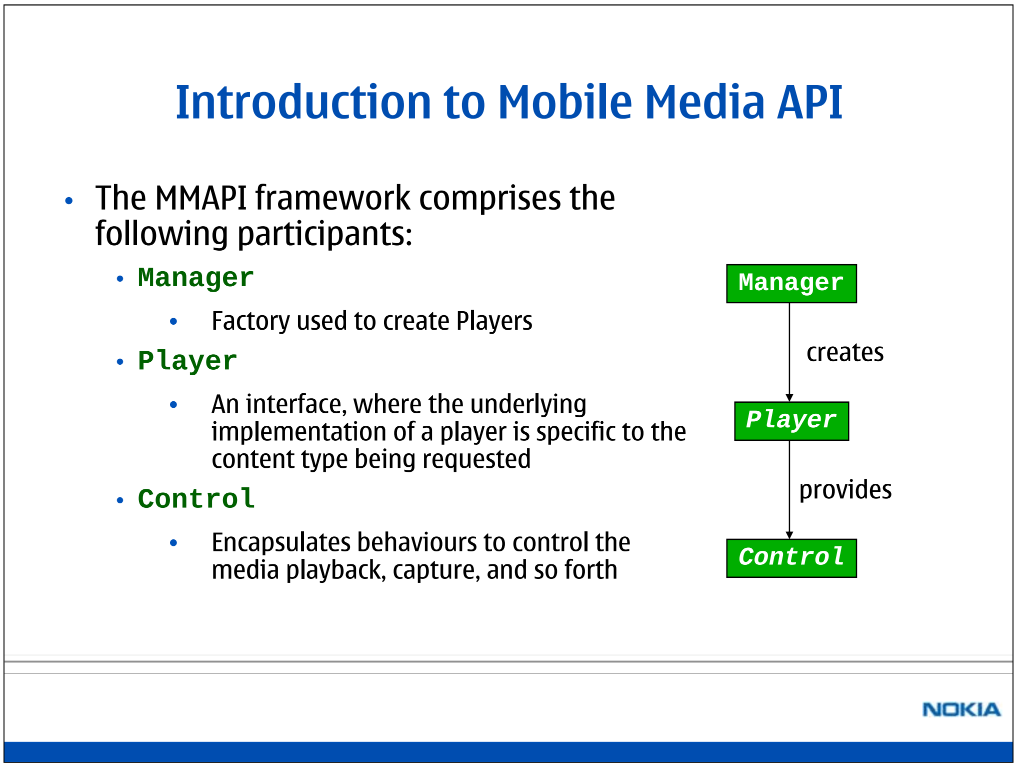


Video and Sound on MIDP Devices

- Video and sound playback on MIDP devices is controlled by the Mobile Media API (MMAPI)
- MMAPI is an optional package enabling
  - Playing videos
  - Playing audio
  - Recording of audio
  - Taking photos from the on-device camera
- Using MMAPI, videos and sounds can be played from
  - A resource folder in the application JAR
  - A RMS Record store
  - The Internet

MIDlets are resource constrained in that they run on devices with little memory, processing power and limited visual and audio capabilities. The Mobile Media Api can be used to access several media formats in a unified and efficient manner, making the most of these constrained resources.

It is flexible, in that you can load media over a network connection, or directly from the JAR, and it is designed to allow further development of new media protocols.



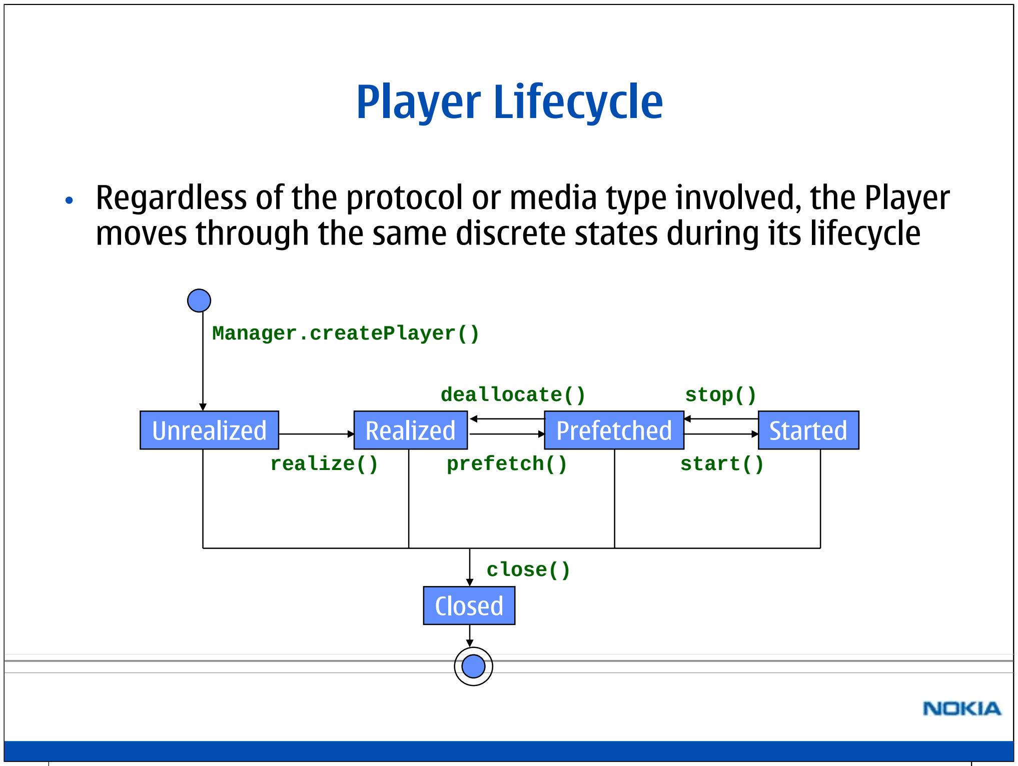
Introduction to Mobile Media API

- The MMAPI framework comprises the following participants:
  - **`Manager`**
    - Factory used to create Players
  - **`Player`**
    - An interface, where the underlying implementation of a player is specific to the content type being requested
  - **`Control`**
    - Encapsulates behaviours to control the media playback, capture, and so forth

Manager
creates
Player
provides
Control
NOKIA

The MMA consists of three packages: javax.microedition.media, javax.microedition.media.control, and javax.microedition.media.protocol. The first two packages contain all the classes and interfaces needed to access rich media. The javax.microedition.media.protocol package is concerned with implementing custom protocol handlers and will not be covered here.

The javax.microedition.media classes and interfaces define how to access different media types whereas javax.microedition.media.control implements interfaces and classes to control how the media is played. The timing of how to play media consists of first obtaining a suitable player from the Manger class. Once the Manager creates a specific media player from a description of the media – called the Data Source, the player can be used how the media itself is actually played. Different controls, such as Volume Control, can be implemented by the player to control how the media itself is managed.



Player Lifecycle

- Regardless of the protocol or media type involved, the Player moves through the same discrete states during its lifecycle

Manager.createPlayer()
Unrealized
realize()
Realized
deallocate()
prefetch()
Prefetched
stop()
start()
Started
close()
Closed

NOKIA

When the Player is created it starts out in the UNREALIZED state. In the above code example, after the Player finds its data it is in the REALIZED state. Once the player has enough data to start playing, the Player is in the PREFETCHED state. Once the player starts rendering media to the device it is in the STARTED state. Finally, we can close the Player and put it in the CLOSED state. The reason for the five different states is that there are potentially time consuming operations that a Player has to go through before it can start rendering media, thus methods are provided to transition the player from state-to-state in order to allow the programmer to provide a mechanism for feedback and add controls that enhances the Players capabilities for rich media. For example, here's how to add a Volume Control to the player:

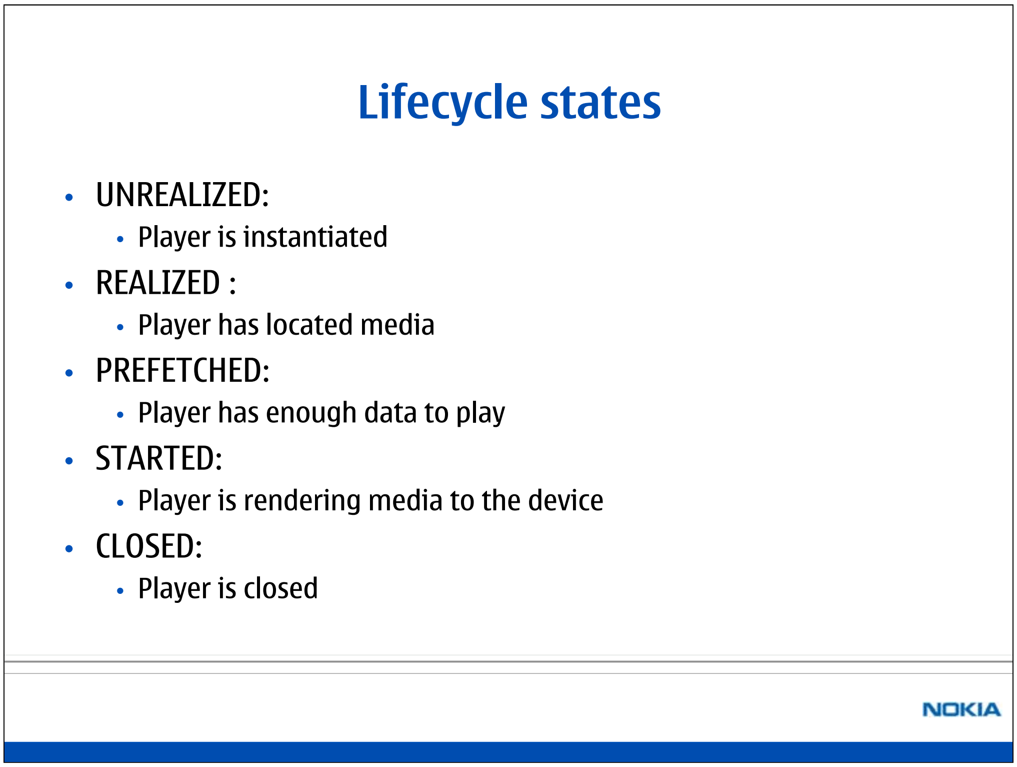
```
Player player;
VolumeControl volCtrl;
try {
        player = Manager.createPlayer(“http://myserver/mymusic.wav”);
        player.realize();
        // once the player is realized we can add the volume control
        volCtrl =  (VolumeControl) p.getControl(“VolumeControl”);
        volCtrl.setVolume(50);
        p.prefetch(); // grab enough data to start
        p.start();
} catch (IOException e) {
} catch (MediaException e) {
}
```

If we want to play video, we can do it much the same way as for music except we now use a Video controller:

```
Player player;
VideoControl vidCtrl;
try {
        player = Manager.createPlayer(“http://myserver/myvideo.3gp”);
        player.realize();
        vidCtrl =  (VolumeControl) p.getControl(“VideoControl”);
        player.prefetch();
        player.start();
} catch (IOException e) {
} catch (MediaException e) {
}
```

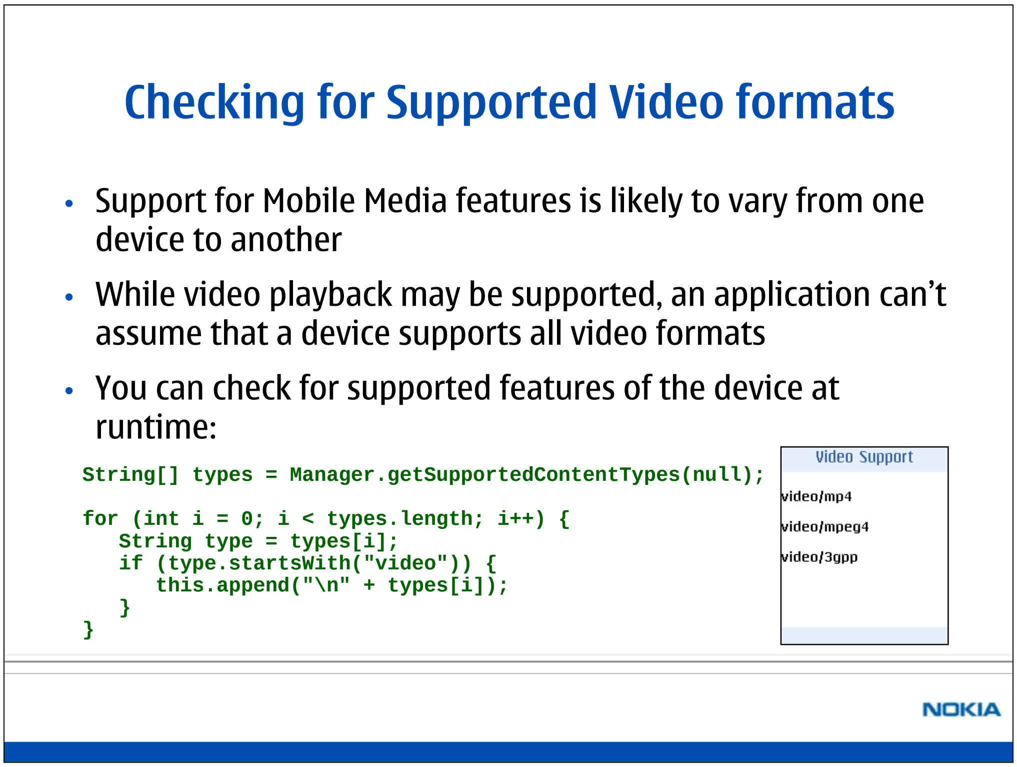
Finally, some Nokia devices have a built-in camera. To access the camera we can use a special resource format capture://video and we can use the getSnapshot method to take a still image of a frame of video like this:

```
Player player;
VideoControl vidCtrl;
try {
        player = Manager.createPlayer(“capture://video”);
        player.realize();
        // once the player is realized we can add the volume control
        vidCtrl =  (VideoControl) p.getControl(“VideoControl”);
        player.prefetch(); // grab enough data to start
        player.start();
} catch (IOException e) {
```



Lifecycle states

- UNREALIZED:
  - Player is instantiated
- REALIZED :
  - Player has located media
- PREFETCHED:
  - Player has enough data to play
- STARTED:
  - Player is rendering media to the device
- CLOSED:
  - Player is closed



Checking for Supported Video formats

- Support for Mobile Media features is likely to vary from one device to another
- While video playback may be supported, an application can't assume that a device supports all video formats
- You can check for supported features of the device at runtime:

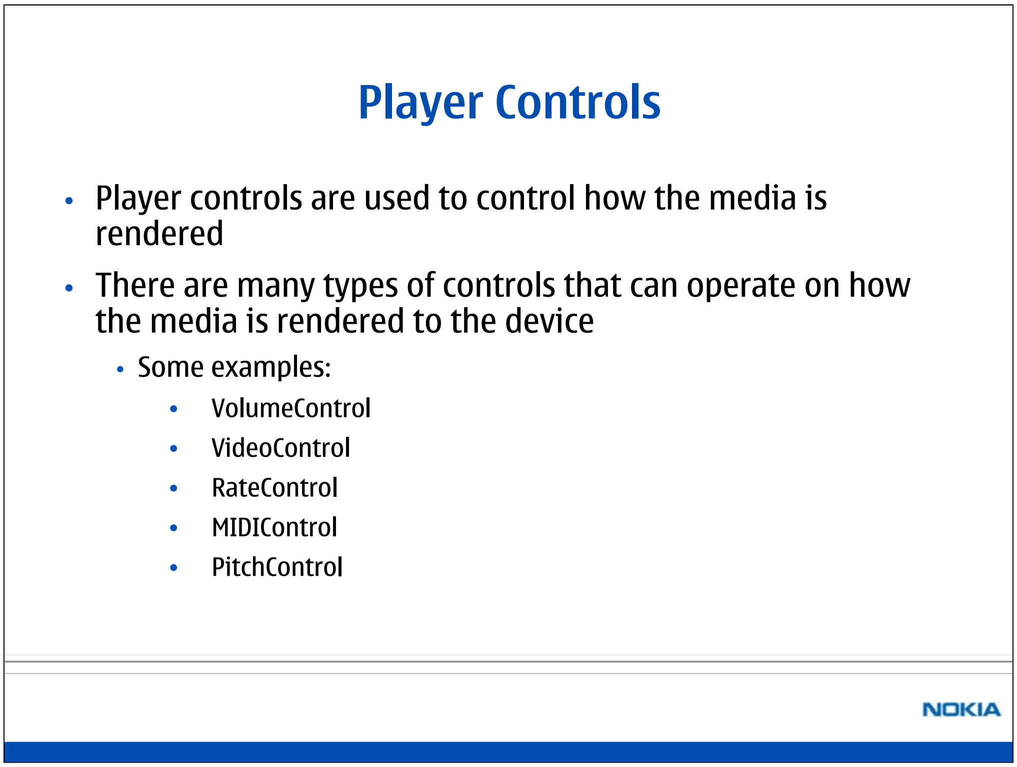
```
String[] types = Manager.getSupportedContentTypes(null);

for (int i = 0; i < types.length; i++) {
   String type = types[i];
   if (type.startsWith("video")) {
      this.append("\n" + types[i]);
   }
}
```

Video Support

video/mp4

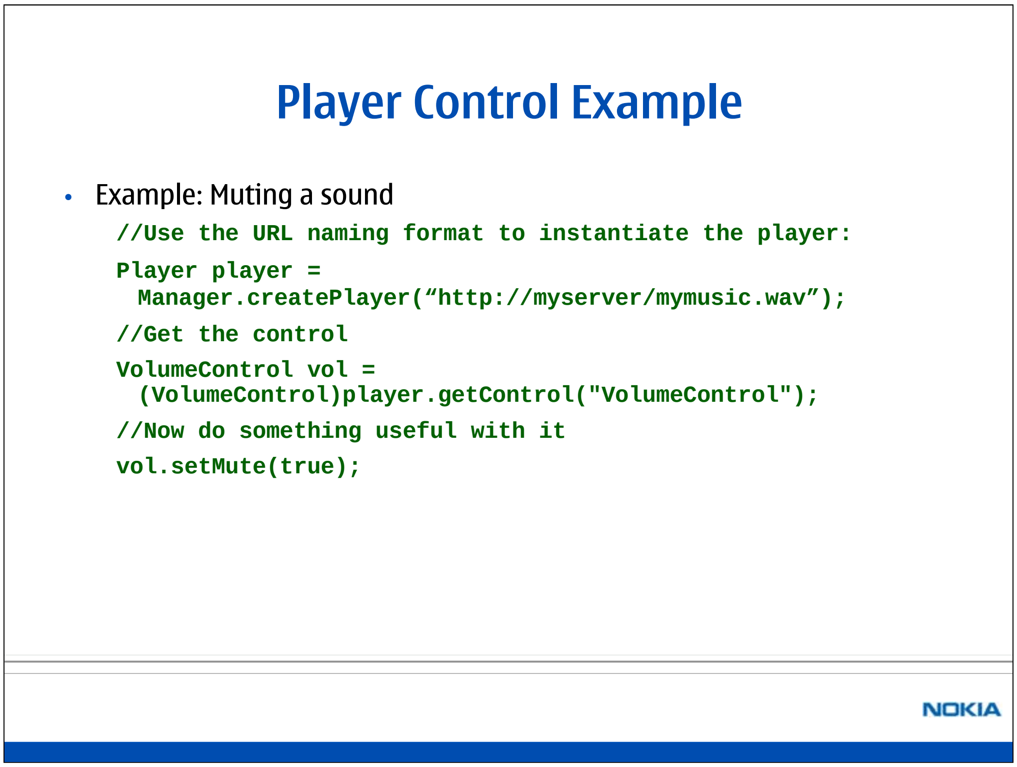
video/mpeg4

video/3gpp



Player Controls

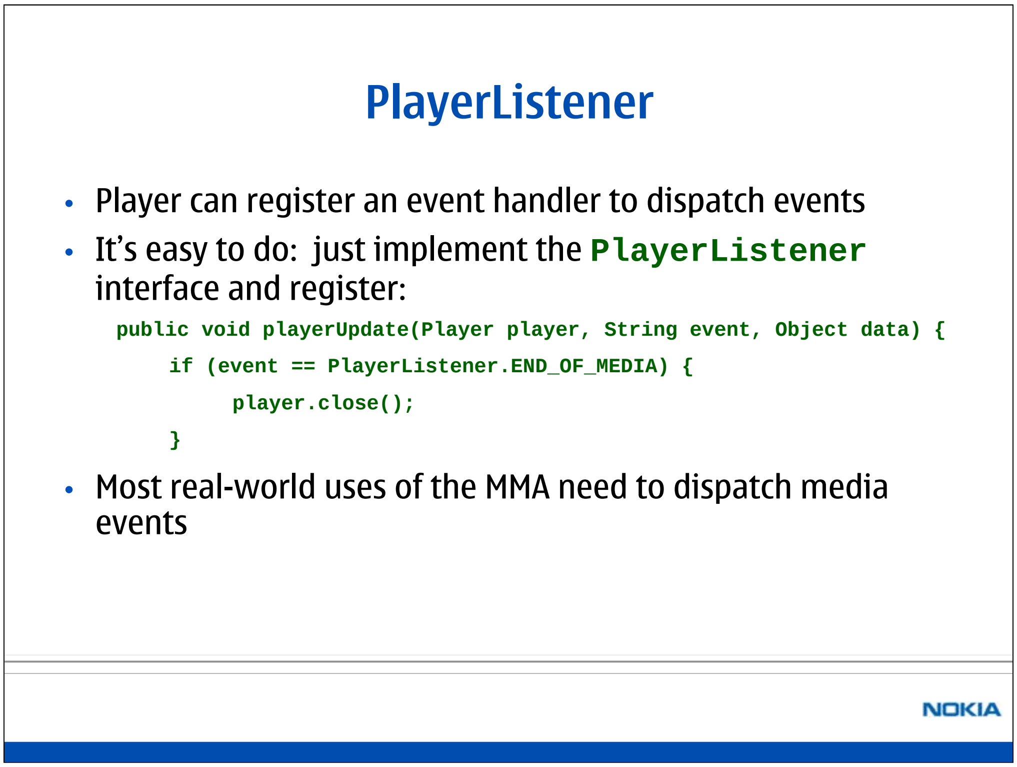
- Player controls are used to control how the media is rendered
- There are many types of controls that can operate on how the media is rendered to the device
  - Some examples:
    - VolumeControl
    - VideoControl
    - RateControl
    - MIDIControl
    - PitchControl



Player Control Example

- Example: Muting a sound

```
//Use the URL naming format to instantiate the player:
Player player =
  Manager.createPlayer(“http://myserver/mymusic.wav”);
//Get the control
VolumeControl vol =
  (VolumeControl)player.getControl("VolumeControl");
//Now do something useful with it
vol.setMute(true);
```



PlayerListener

- Player can register an event handler to dispatch events
- It's easy to do: just implement the `PlayerListener` interface and register:

```
public void playerUpdate(Player player, String event, Object data) {
    if (event == PlayerListener.END_OF_MEDIA) {
         player.close();
    }
```

- Most real-world uses of the MMA need to dispatch media events

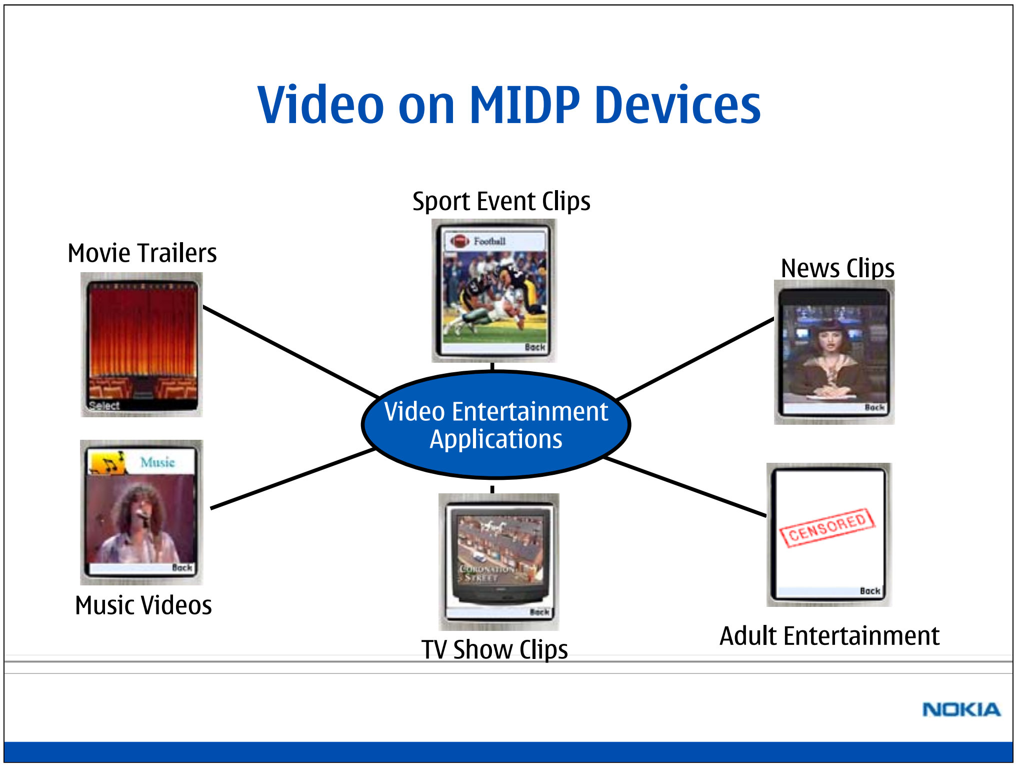
Media Events

As your player is rendering media you can monitor and dispatch events as they occur with the PlayerListener interface. Simply implement the PlayerListener interface and call Player.addPlayerListener() to register your listener. Then implement the playerUpdate() method of the PlayerListener interface to dispatch media events. At it's most basic you can determine when your media is finished rendering and then close the player to free up resources:

```
public void playerUpdate(Player player, String event, Object data) {
       if (event == PlayerListener.END_OF_MEDIA) {
              player.stop();
       }
}
```

The playerUpdate method is used to check what event was fired by the Player object, and act accordingly. For example, if the player has stopped playing the media data, an END_OF_MEDIA event is fired. In the example in the slide, you can see that when this event is detected, the application closes the player object. Other events that could be fired are

- ***STARTED - fired when the player has begun to play the media data***
- ***STOPPED - fired when the player has stopped playing the media data***
- ***CLOSED - fired when the player has been closed***



Video on MIDP Devices

Sport Event Clips

Movie Trailers

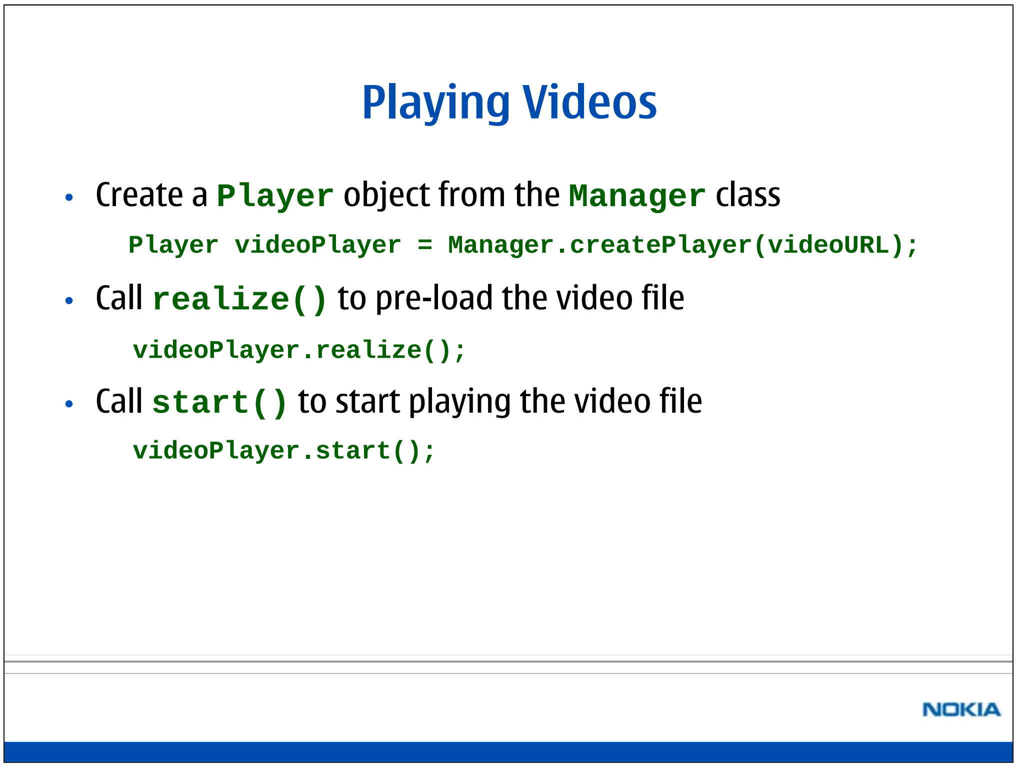
News Clips

Video Entertainment Applications

Music Videos

TV Show Clips

Adult Entertainment



Playing Videos

- Create a **`Player`** object from the **`Manager`** class

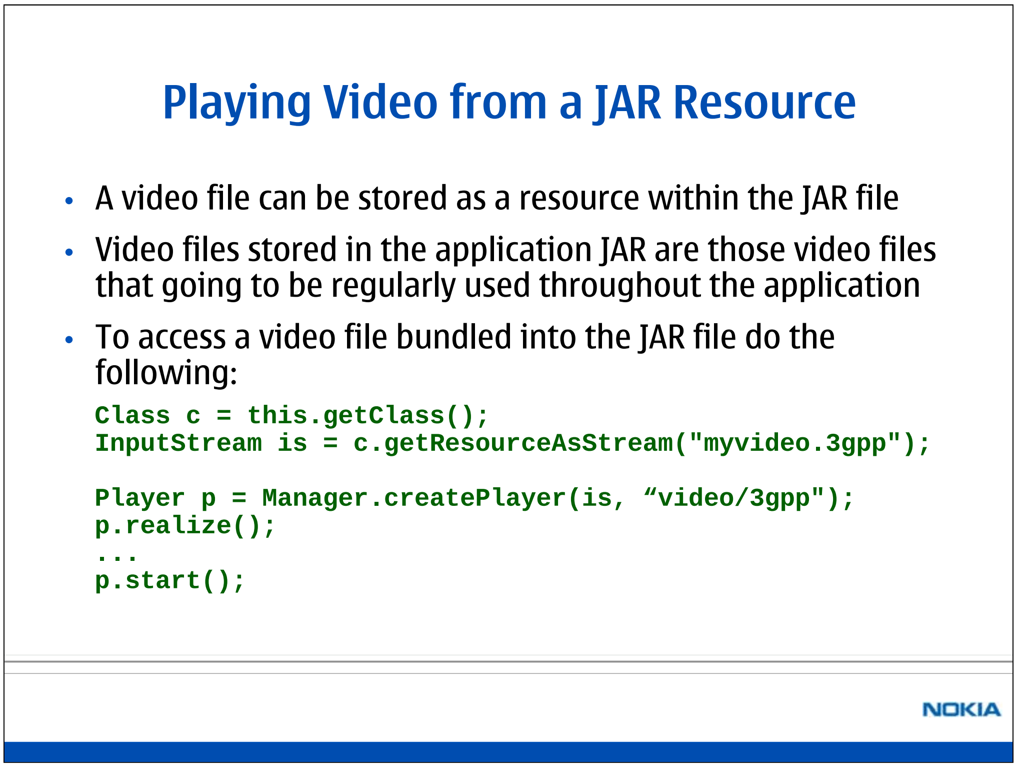
```
Player videoPlayer = Manager.createPlayer(videoURL);
```

- Call **`realize()`** to pre-load the video file

```
videoPlayer.realize();
```

- Call **`start()`** to start playing the video file

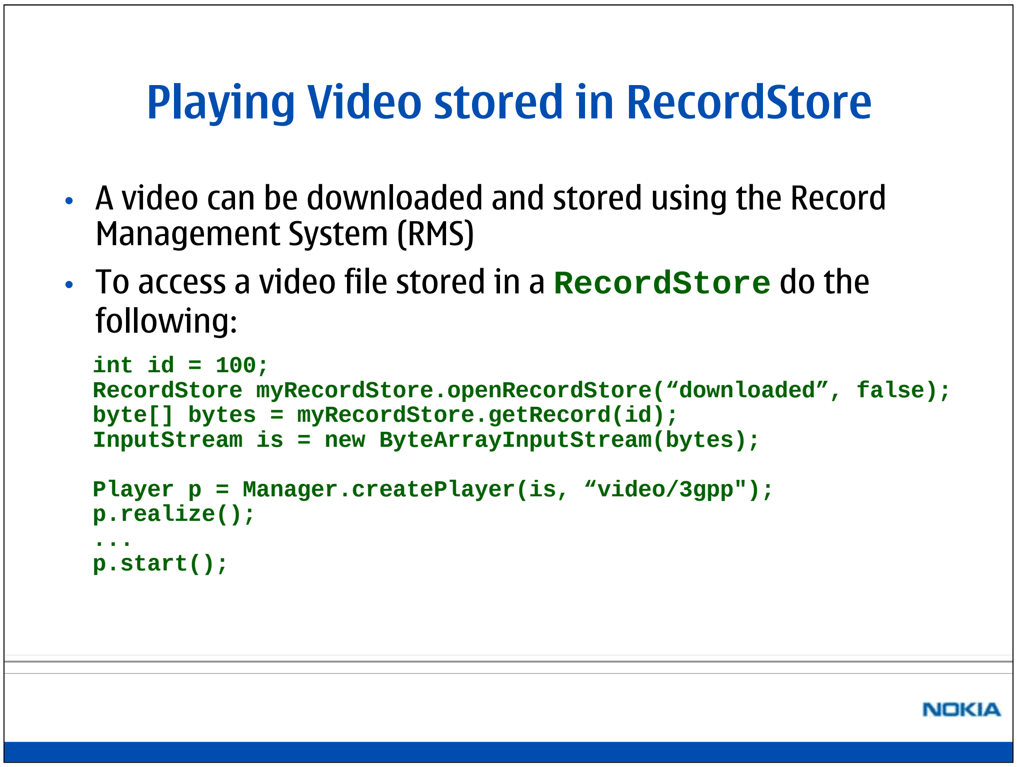
```
videoPlayer.start();
```



Playing Video from a JAR Resource

- A video file can be stored as a resource within the JAR file
- Video files stored in the application JAR are those video files that going to be regularly used throughout the application
- To access a video file bundled into the JAR file do the following:

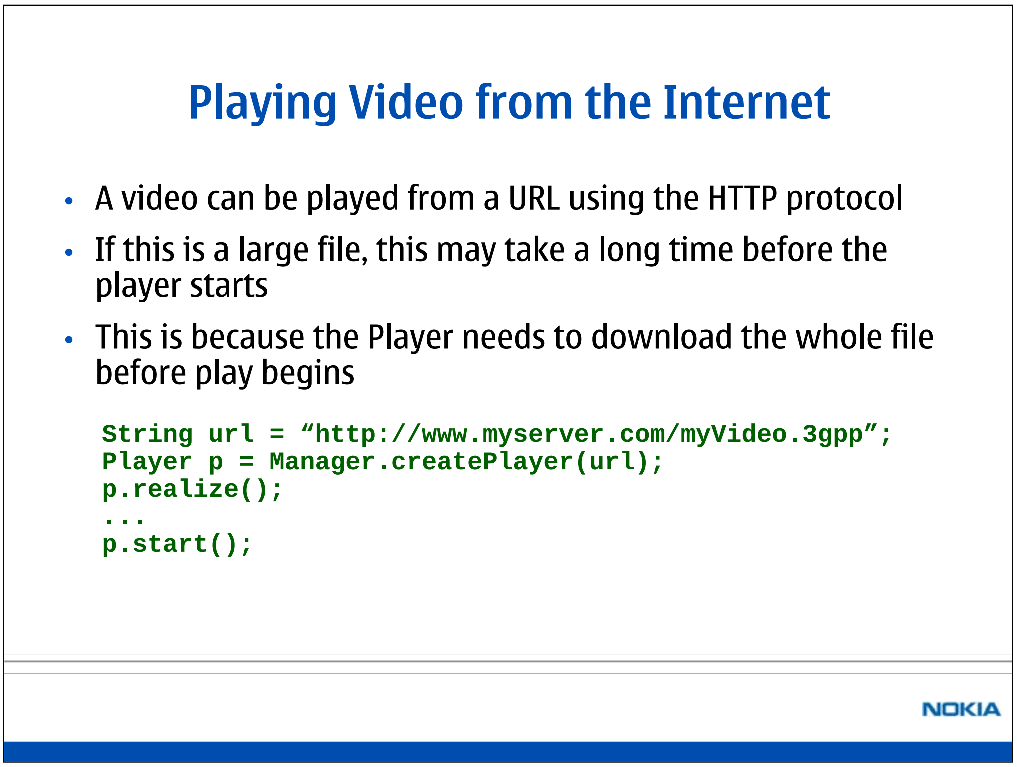
```
Class c = this.getClass();
InputStream is = c.getResourceAsStream("myvideo.3gpp");

Player p = Manager.createPlayer(is, “video/3gpp");
p.realize();
...
p.start();
```



Playing Video stored in RecordStore

- A video can be downloaded and stored using the Record Management System (RMS)
- To access a video file stored in a `RecordStore` do the following:

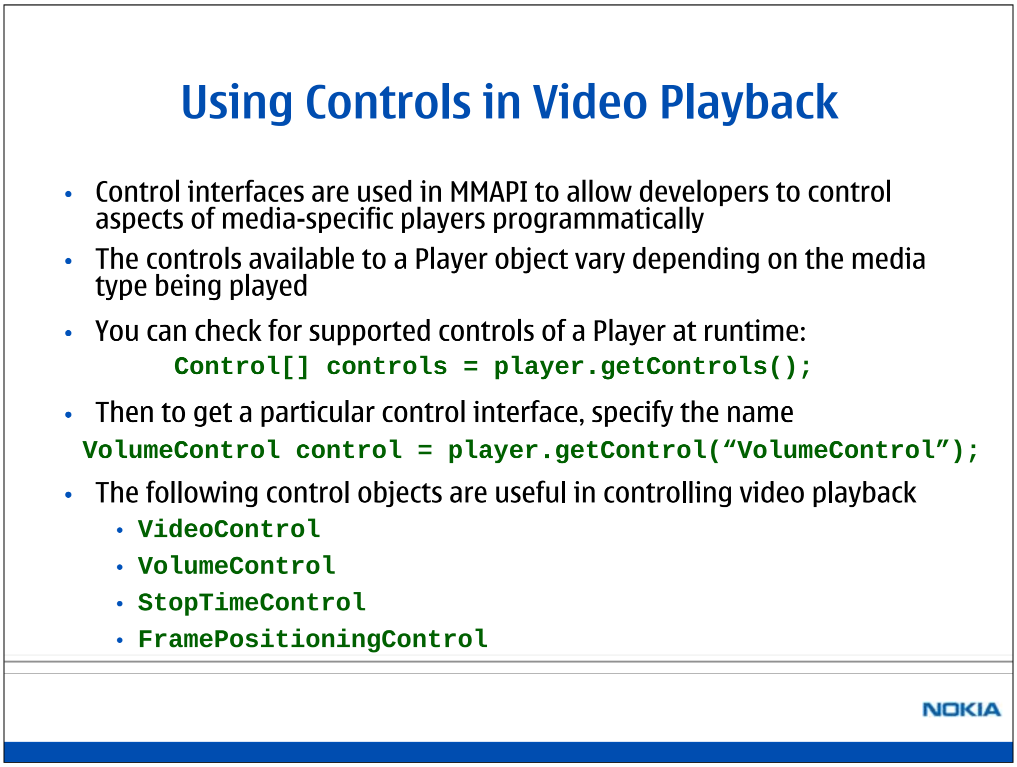
```
int id = 100;
RecordStore myRecordStore.openRecordStore(“downloaded”, false);
byte[] bytes = myRecordStore.getRecord(id);
InputStream is = new ByteArrayInputStream(bytes);

Player p = Manager.createPlayer(is, “video/3gpp");
p.realize();
...
p.start();
```



Playing Video from the Internet

- A video can be played from a URL using the HTTP protocol
- If this is a large file, this may take a long time before the player starts
- This is because the Player needs to download the whole file before play begins

```
String url = “http://www.myserver.com/myVideo.3gpp”;
Player p = Manager.createPlayer(url);
p.realize();
...
p.start();
```



Using Controls in Video Playback

- Control interfaces are used in MMAPI to allow developers to control aspects of media-specific players programmatically
- The controls available to a Player object vary depending on the media type being played
- You can check for supported controls of a Player at runtime:

```
      Control[] controls = player.getControls();
```

- Then to get a particular control interface, specify the name

```
VolumeControl control = player.getControl(“VolumeControl”);
```

- The following control objects are useful in controlling video playback
  - `VideoControl`
  - `VolumeControl`
  - `StopTimeControl`
  - `FramePositioningControl`

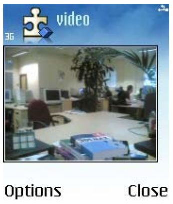
Displaying the Video on a Form

- Once you have loaded the video, you can use the **VideoControl** interface to display it on a **Form**
- Get the **VideoControl** interface

```
VideoControl vidCtrl = player.getControl(“VideoControl”);
```

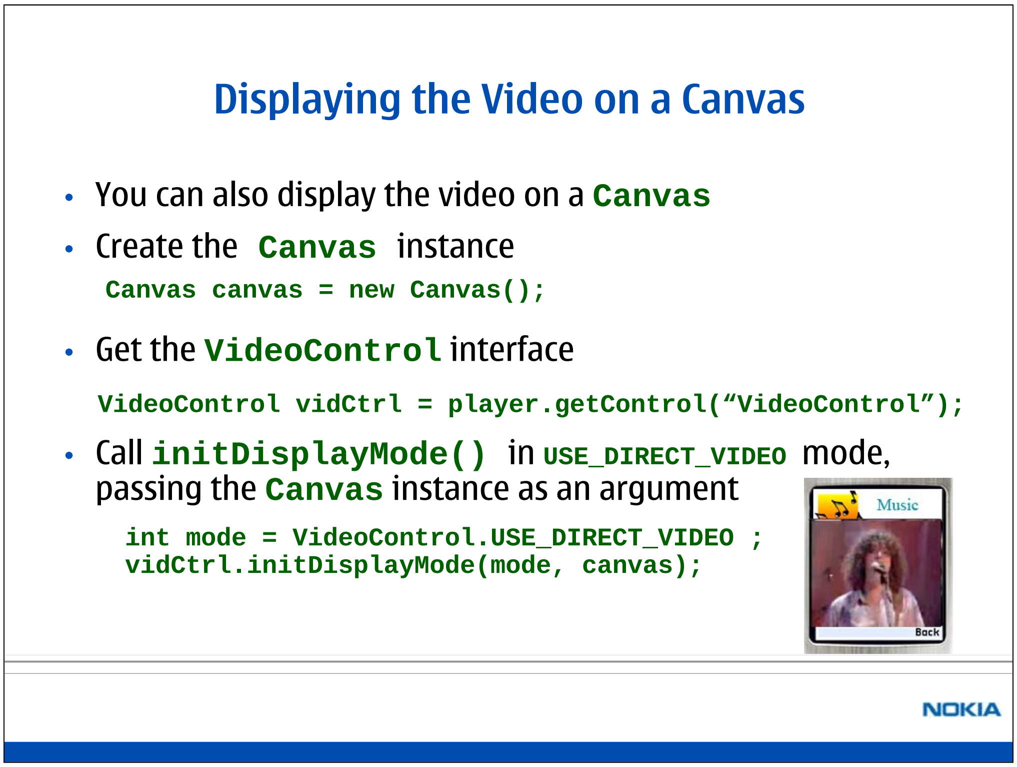
- Call **initDisplayMode()** in **USE_GUI_PRIMITIVE** mode to get an **Item**

```
int mode = VideoControl.USE_GUI_PRIMITIVE;
Item videoItem
   = (Item)vidCtrl.initDisplayMode(mode, null);
```



- Append the **Item** to the form

```
Form form = new Form(“video”);
form.append(videoItem);
```



Displaying the Video on a Canvas

- You can also display the video on a **`Canvas`**
- Create the **`Canvas`** instance

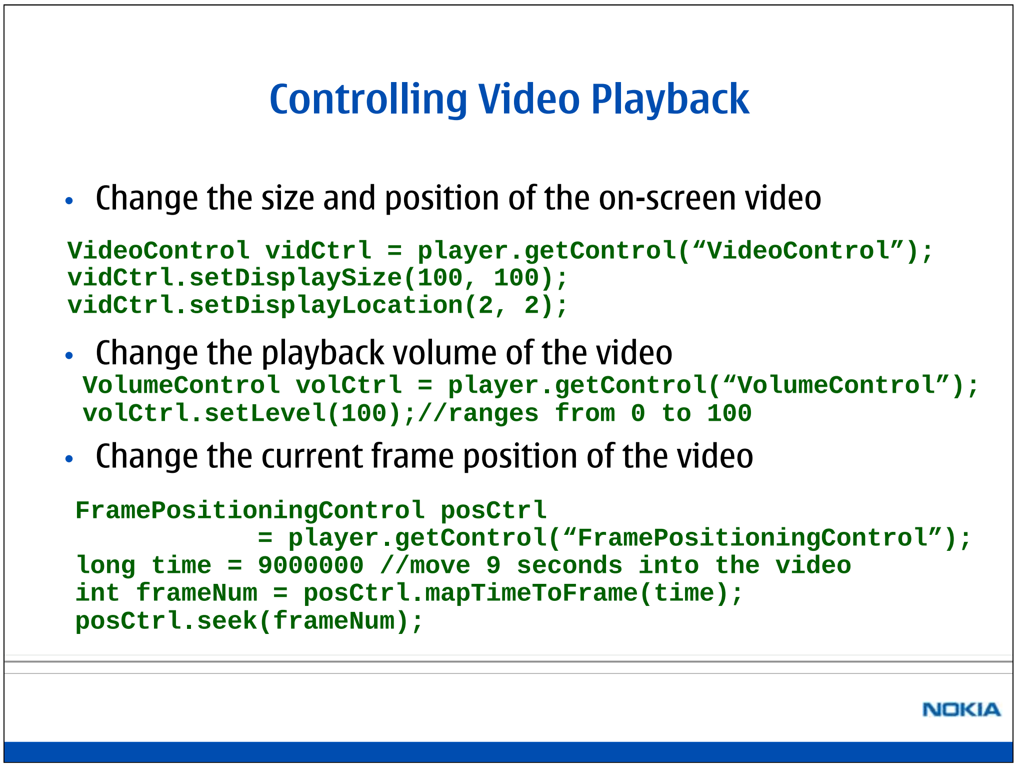
```
Canvas canvas = new Canvas();
```

- Get the **`VideoControl`** interface

```
VideoControl vidCtrl = player.getControl(“VideoControl”);
```

- Call **`initDisplayMode()`** in **`USE_DIRECT_VIDEO`** mode, passing the **`Canvas`** instance as an argument

```
int mode = VideoControl.USE_DIRECT_VIDEO ;
vidCtrl.initDisplayMode(mode, canvas);
```



Controlling Video Playback

- Change the size and position of the on-screen video

```
VideoControl vidCtrl = player.getControl(“VideoControl”);
vidCtrl.setDisplaySize(100, 100);
vidCtrl.setDisplayLocation(2, 2);
```

- Change the playback volume of the video

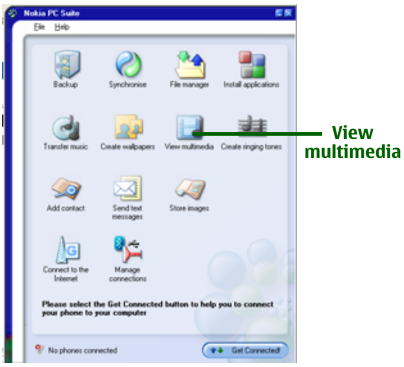
```
VolumeControl volCtrl = player.getControl(“VolumeControl”);
volCtrl.setLevel(100);//ranges from 0 to 100
```

- Change the current frame position of the video

```
FramePositioningControl posCtrl
            = player.getControl(“FramePositioningControl”);
long time = 9000000 //move 9 seconds into the video
int frameNum = posCtrl.mapTimeToFrame(time);
posCtrl.seek(frameNum);
```

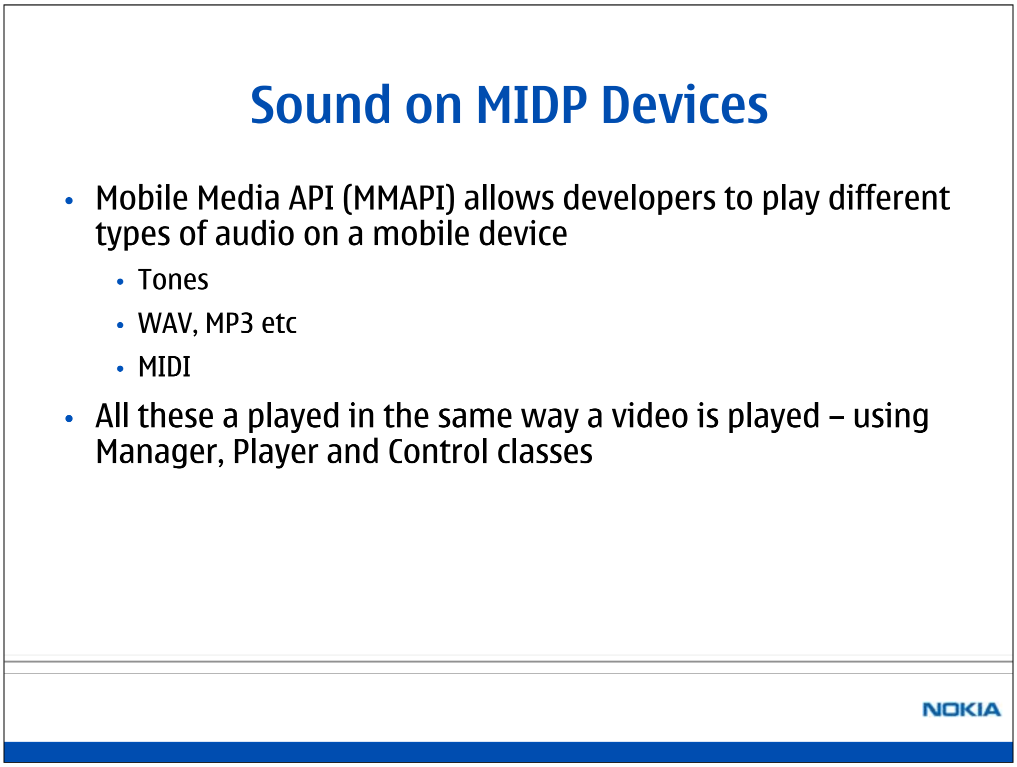
Converting Video files into 3gp format

- Nokia PC Suite includes a tool to convert any video file into a 3gp format
- This allows you to play the video on your mobile device
- Dowload Nokia PC Suite from http://www.forum.nokia.com
- Select View multimedia
- Select File -> Open to load your original file
- Select File -> Save As
- Select .3gp as the Save as type



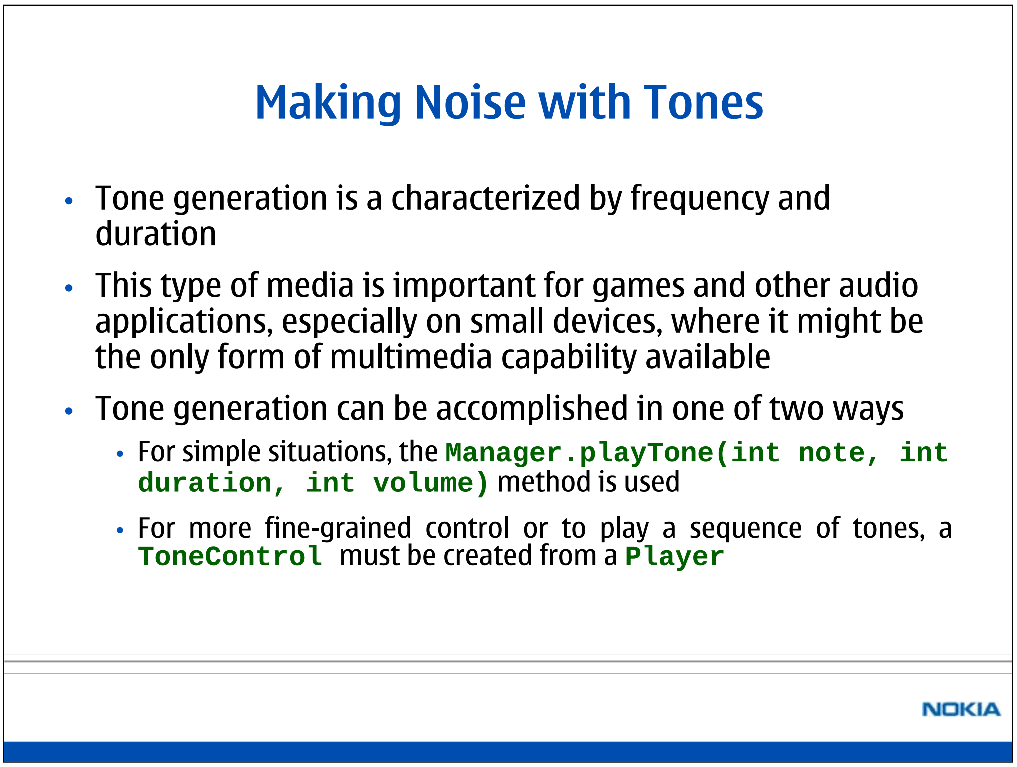
Please select the Get Connected button to help you to connect your phone to your computer

View multimedia



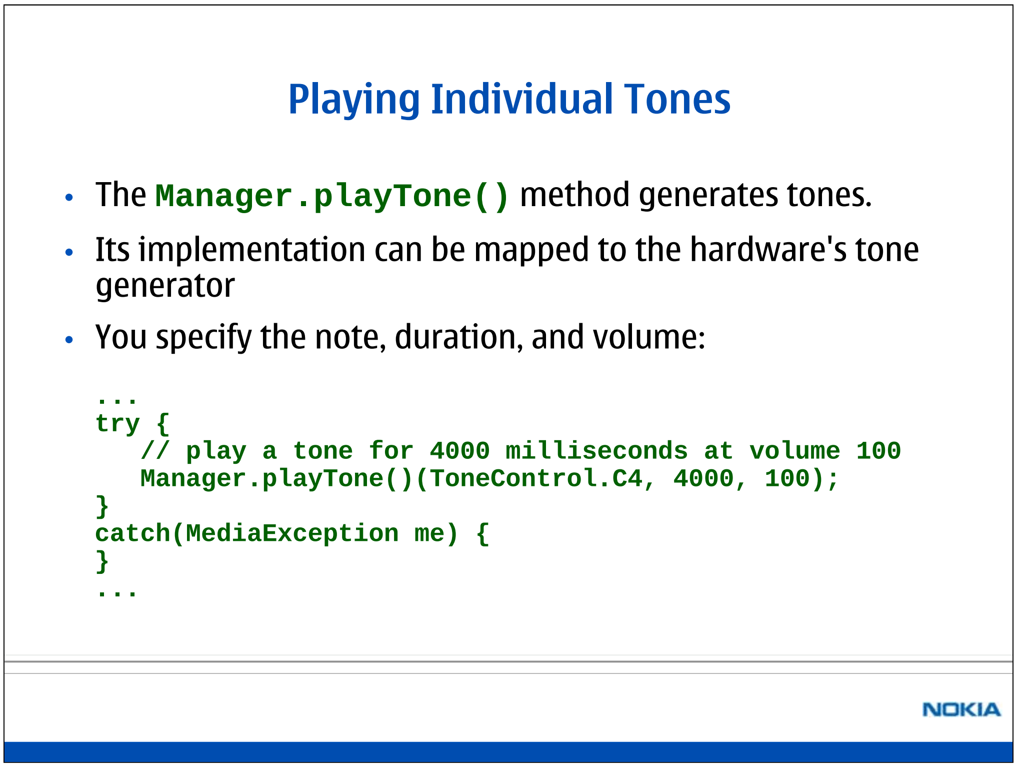
Sound on MIDP Devices

- Mobile Media API (MMAPI) allows developers to play different types of audio on a mobile device
    - Tones
    - WAV, MP3 etc
    - MIDI
- All these a played in the same way a video is played – using Manager, Player and Control classes



Making Noise with Tones

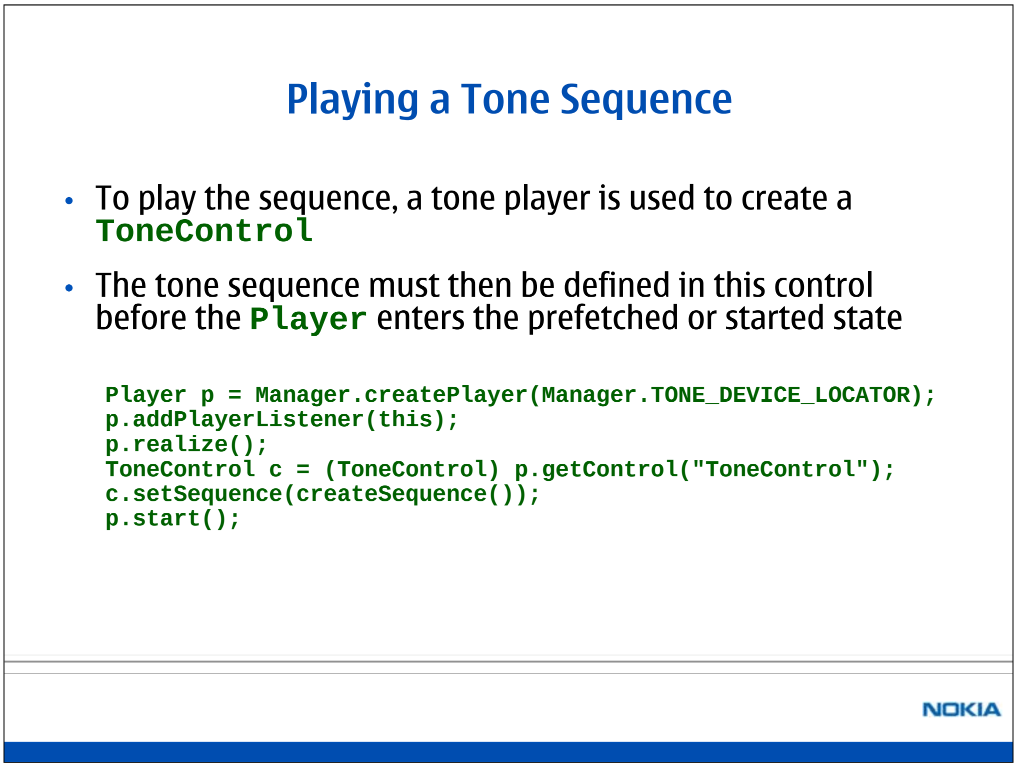
- Tone generation is a characterized by frequency and duration
- This type of media is important for games and other audio applications, especially on small devices, where it might be the only form of multimedia capability available
- Tone generation can be accomplished in one of two ways
  - For simple situations, the `Manager.playTone(int note, int duration, int volume)` method is used
  - For more fine-grained control or to play a sequence of tones, a `ToneControl` must be created from a `Player`



Playing Individual Tones

- The `Manager.playTone()` method generates tones.
- Its implementation can be mapped to the hardware's tone generator
- You specify the note, duration, and volume:

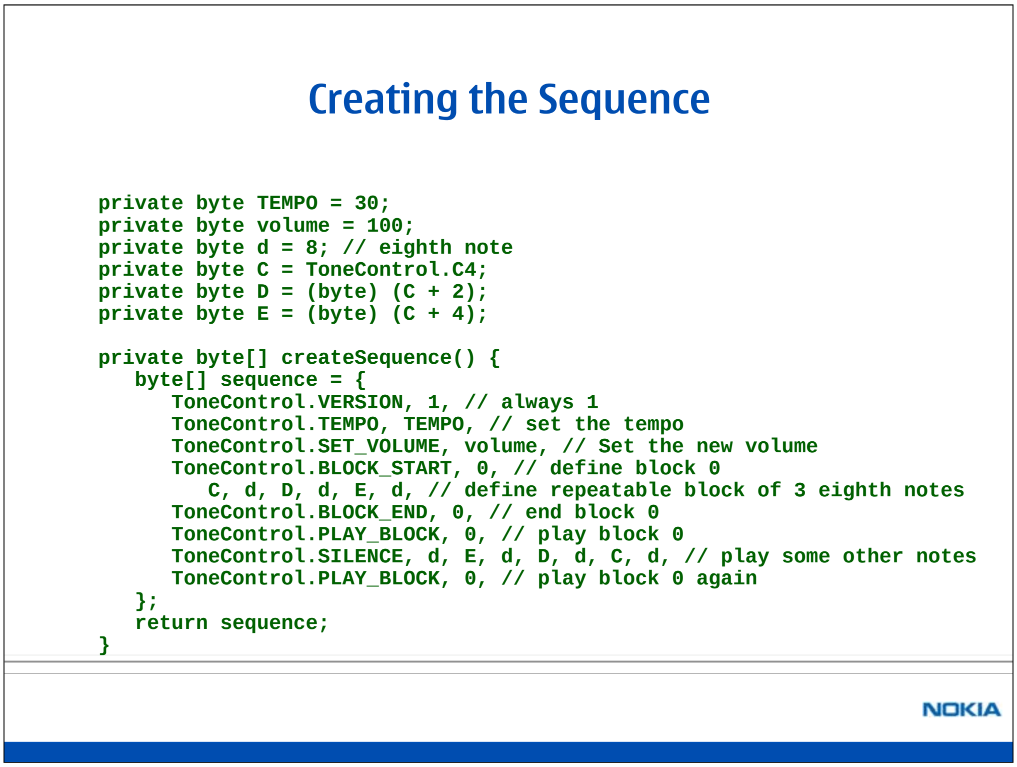
```
...
try {
   // play a tone for 4000 milliseconds at volume 100
   Manager.playTone()(ToneControl.C4, 4000, 100);
}
catch(MediaException me) {
}
...
```



Playing a Tone Sequence

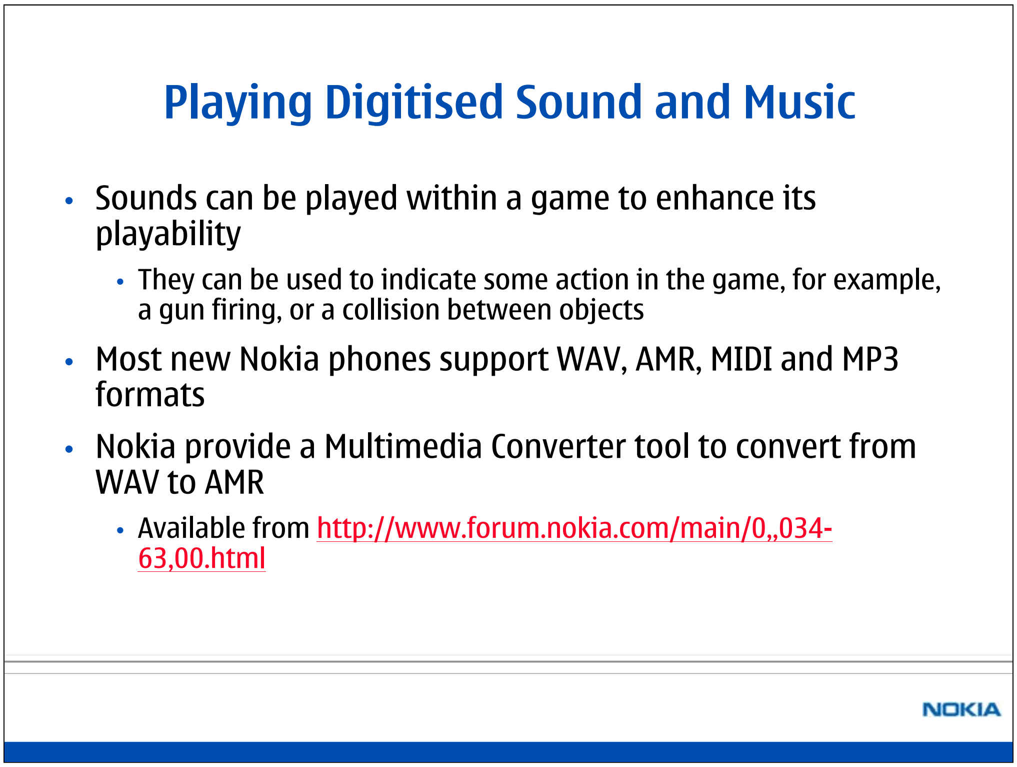
- To play the sequence, a tone player is used to create a `ToneControl`
- The tone sequence must then be defined in this control before the `Player` enters the prefetched or started state

```
Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
p.addPlayerListener(this);
p.realize();
ToneControl c = (ToneControl) p.getControl("ToneControl");
c.setSequence(createSequence());
p.start();
```



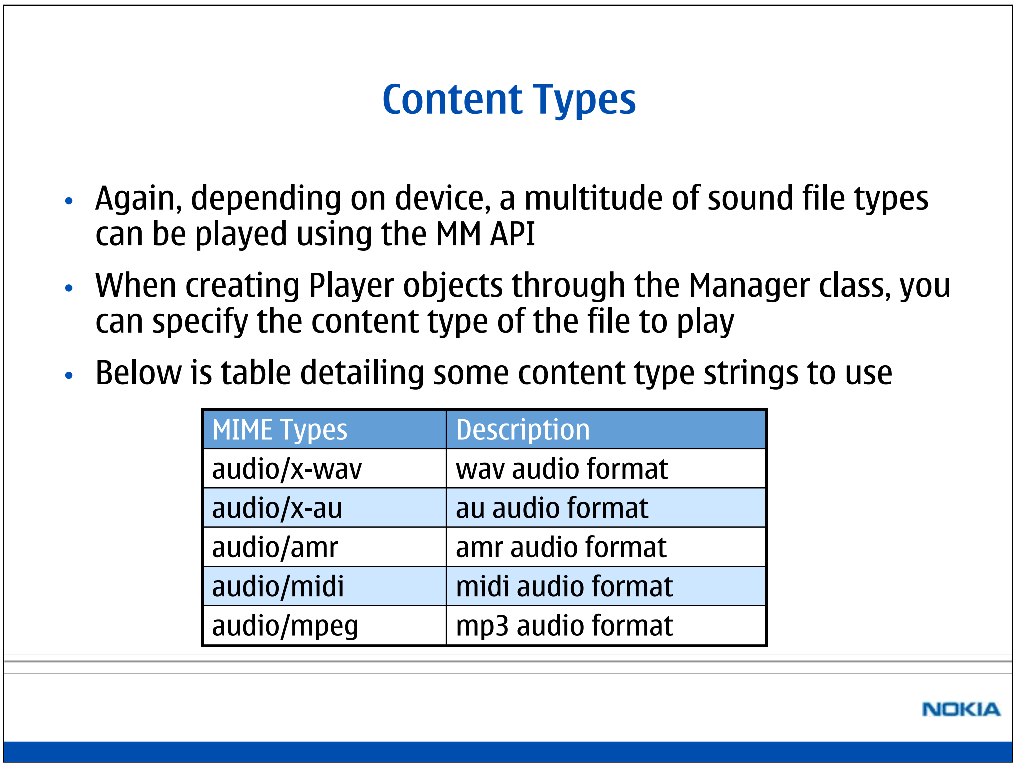
Creating the Sequence

```
private byte TEMPO = 30;
private byte volume = 100;
private byte d = 8; // eighth note
private byte C = ToneControl.C4;
private byte D = (byte) (C + 2);
private byte E = (byte) (C + 4);

private byte[] createSequence() {
   byte[] sequence = {
      ToneControl.VERSION, 1, // always 1
      ToneControl.TEMPO, TEMPO, // set the tempo
      ToneControl.SET_VOLUME, volume, // Set the new volume
      ToneControl.BLOCK_START, 0, // define block 0
         C, d, D, d, E, d, // define repeatable block of 3 eighth notes
      ToneControl.BLOCK_END, 0, // end block 0
      ToneControl.PLAY_BLOCK, 0, // play block 0
      ToneControl.SILENCE, d, E, d, D, d, C, d, // play some other notes
      ToneControl.PLAY_BLOCK, 0, // play block 0 again
   };
   return sequence;
}
```



Playing Digitised Sound and Music

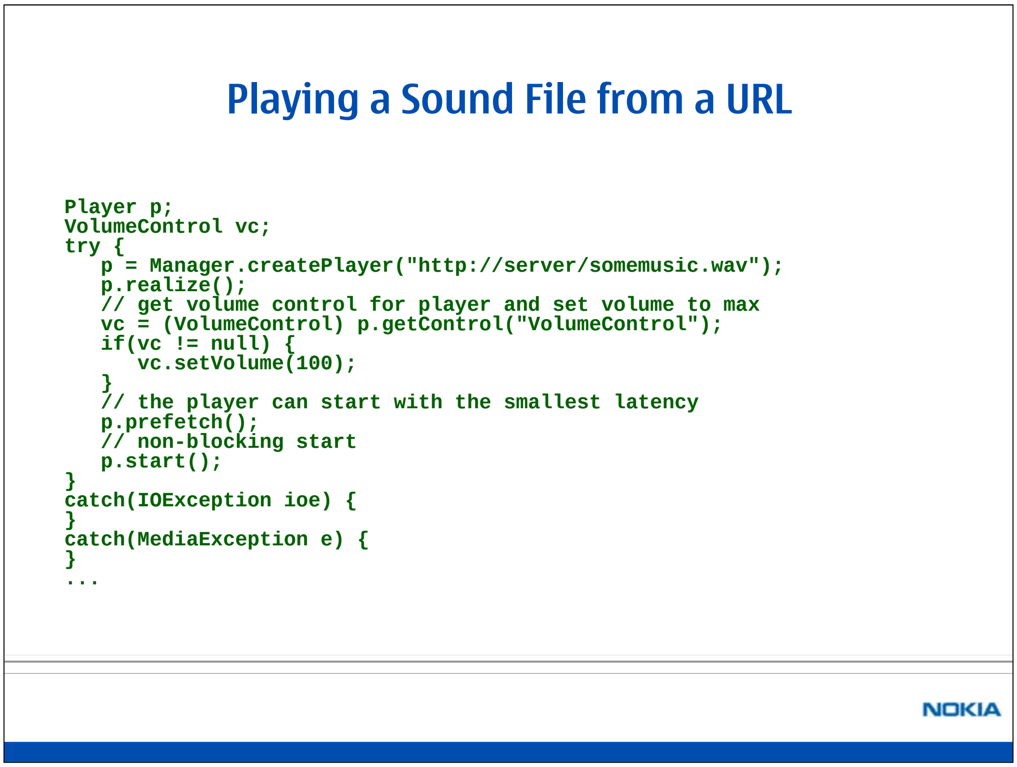
- Sounds can be played within a game to enhance its playability
  - They can be used to indicate some action in the game, for example, a gun firing, or a collision between objects
- Most new Nokia phones support WAV, AMR, MIDI and MP3 formats
- Nokia provide a Multimedia Converter tool to convert from WAV to AMR
  - Available from http://www.forum.nokia.com/main/0,,034-63,00.html



Content Types

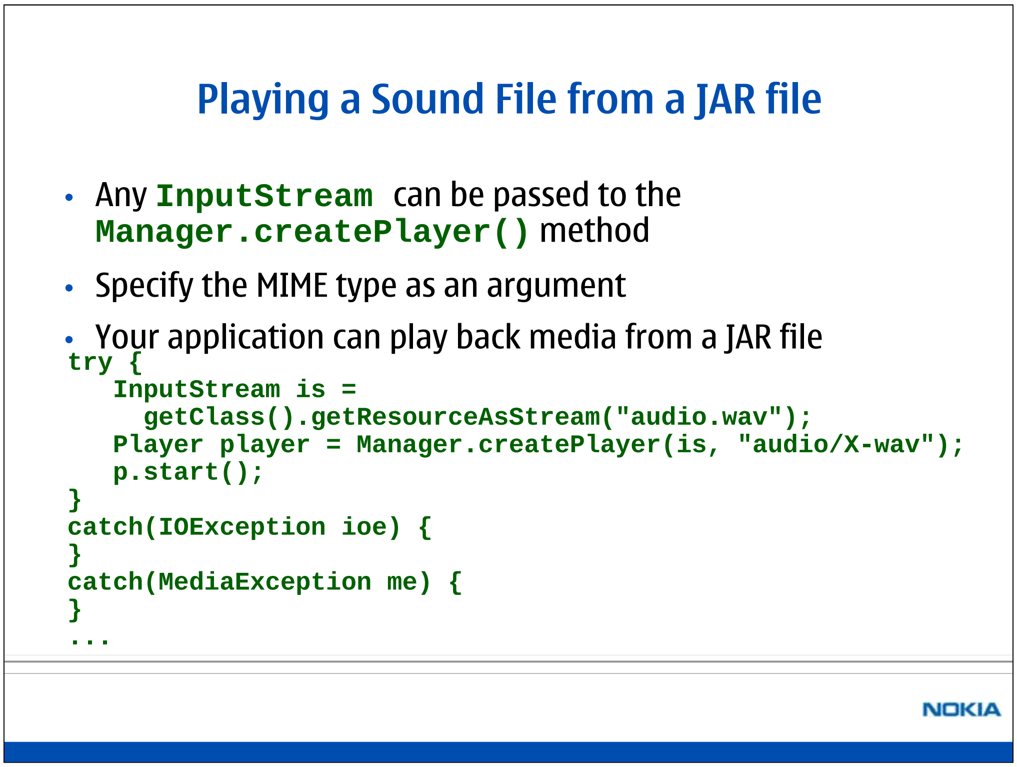
- Again, depending on device, a multitude of sound file types can be played using the MM API
- When creating Player objects through the Manager class, you can specify the content type of the file to play
- Below is table detailing some content type strings to use

MIME Types	Description
audio/x-wav	wav audio format
audio/x-au	au audio format
audio/amr	amr audio format
audio/midi	midi audio format
audio/mpeg	mp3 audio format



Playing a Sound File from a URL

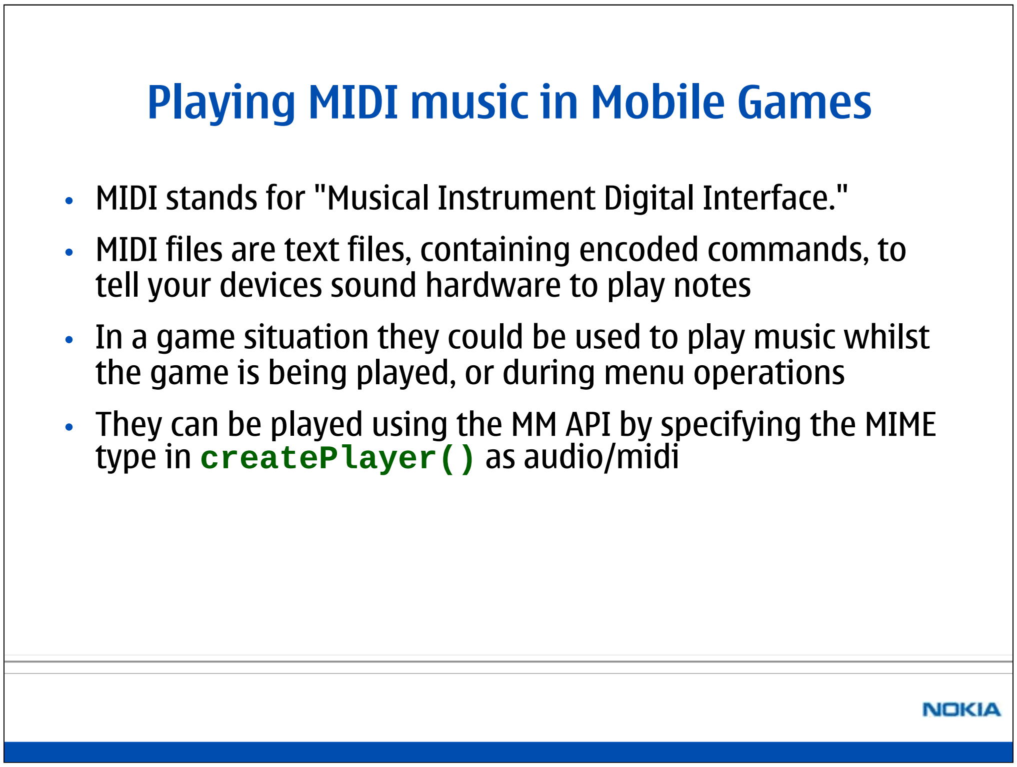
```
Player p;
VolumeControl vc;
try {
   p = Manager.createPlayer("http://server/somemusic.wav");
   p.realize();
   // get volume control for player and set volume to max
   vc = (VolumeControl) p.getControl("VolumeControl");
   if(vc != null) {
      vc.setVolume(100);
   }
   // the player can start with the smallest latency
   p.prefetch();
   // non-blocking start
   p.start();
}
catch(IOException ioe) {
}
catch(MediaException e) {
}
...
```



Playing a Sound File from a JAR file

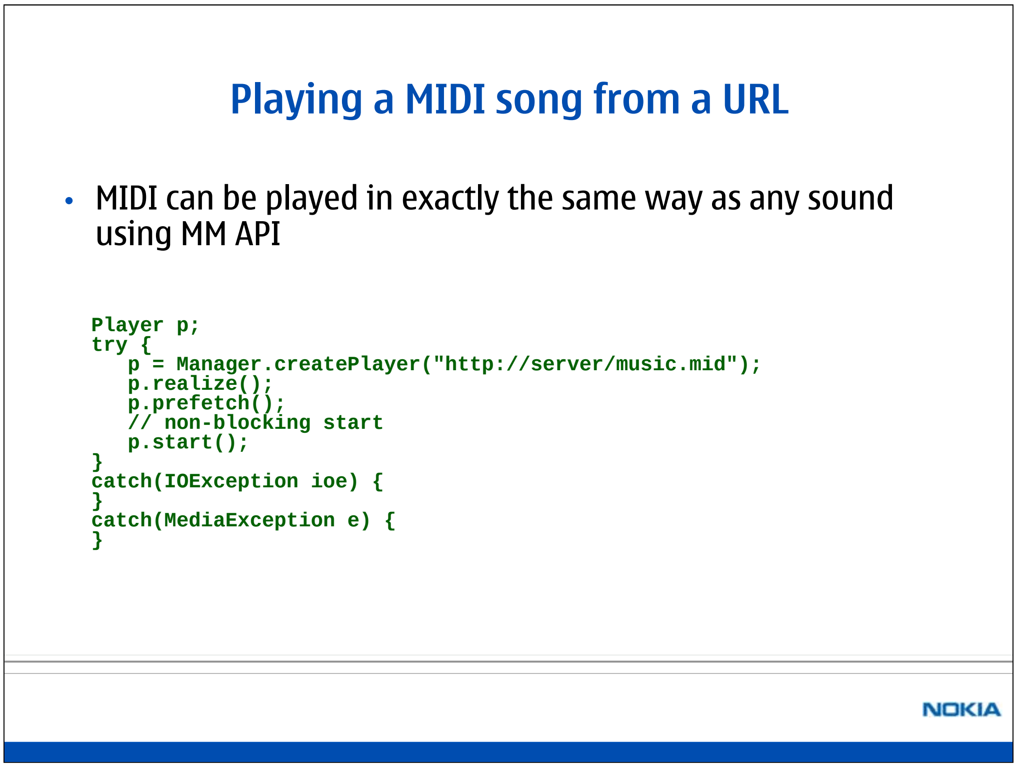
- Any `InputStream` can be passed to the `Manager.createPlayer()` method
- Specify the MIME type as an argument
- Your application can play back media from a JAR file

```
try {
   InputStream is =
     getClass().getResourceAsStream("audio.wav");
   Player player = Manager.createPlayer(is, "audio/X-wav");
   p.start();
}
catch(IOException ioe) {
}
catch(MediaException me) {
}
...
```



Playing MIDI music in Mobile Games

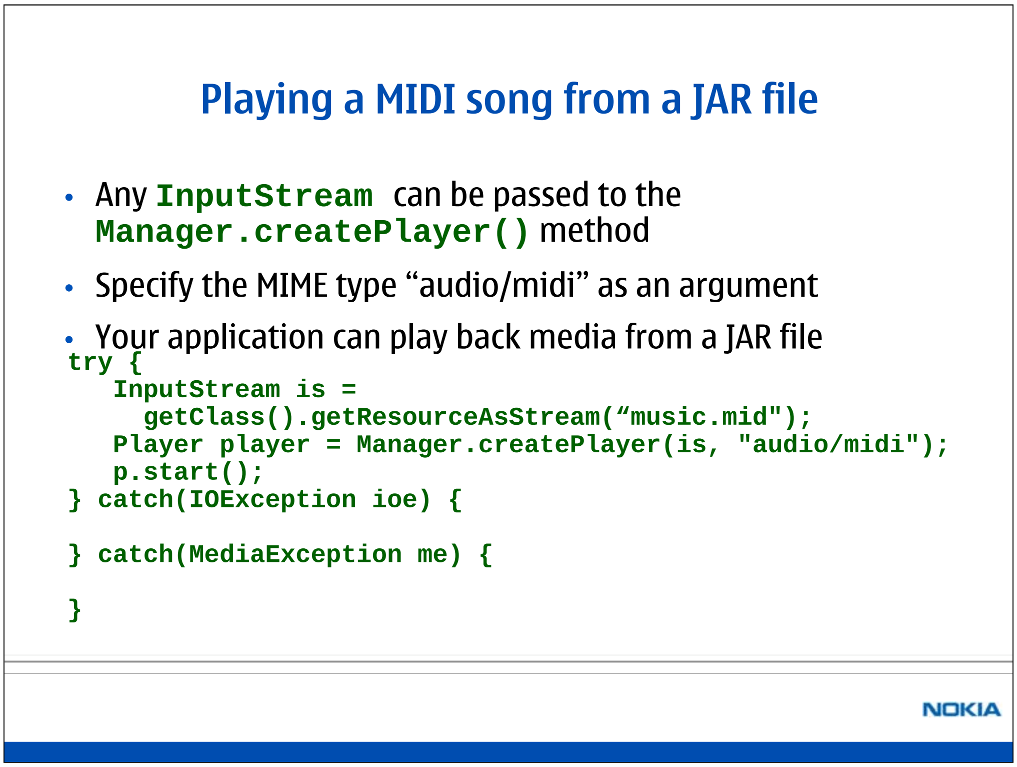
- MIDI stands for "Musical Instrument Digital Interface."
- MIDI files are text files, containing encoded commands, to tell your devices sound hardware to play notes
- In a game situation they could be used to play music whilst the game is being played, or during menu operations
- They can be played using the MM API by specifying the MIME type in `createPlayer()` as audio/midi



Playing a MIDI song from a URL

- MIDI can be played in exactly the same way as any sound using MM API

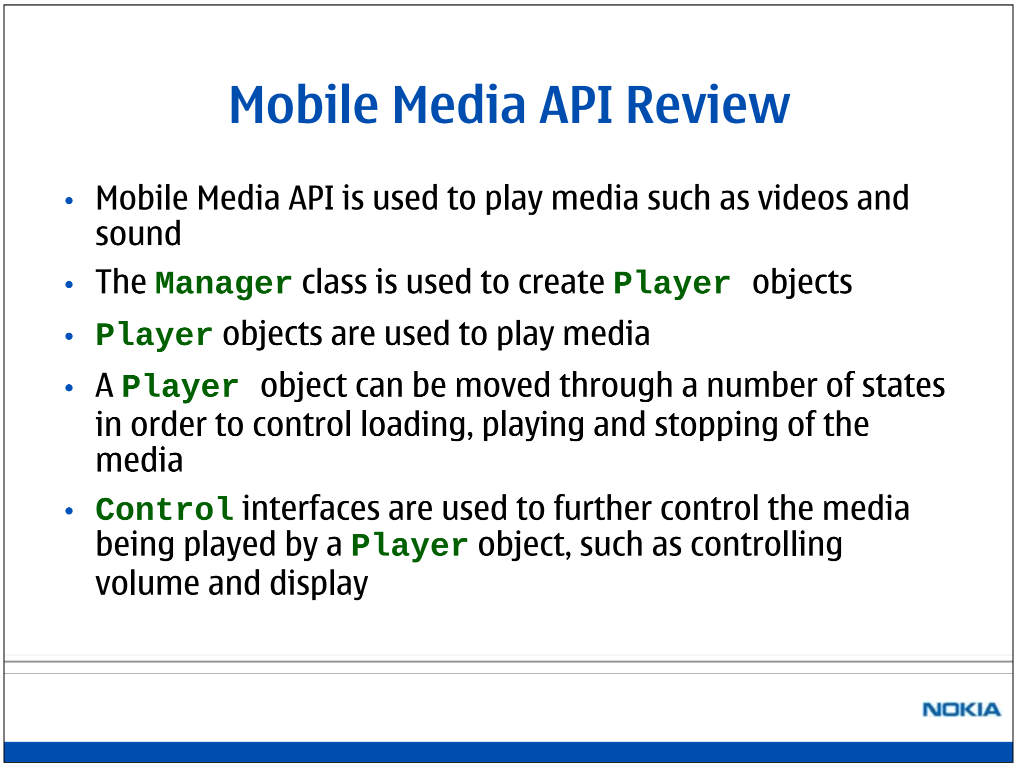
```
Player p;
try {
   p = Manager.createPlayer("http://server/music.mid");
   p.realize();
   p.prefetch();
   // non-blocking start
   p.start();
}
catch(IOException ioe) {
}
catch(MediaException e) {
}
```



Playing a MIDI song from a JAR file

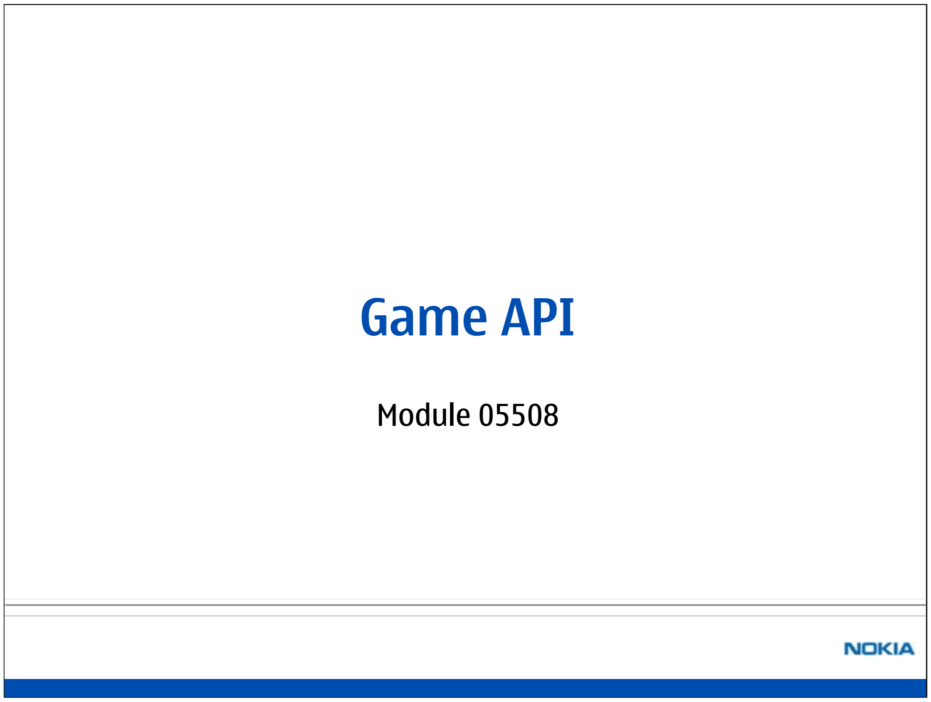
- Any `InputStream` can be passed to the `Manager.createPlayer()` method
- Specify the MIME type “audio/midi” as an argument
- Your application can play back media from a JAR file

```
try {
   InputStream is =
     getClass().getResourceAsStream(“music.mid");
   Player player = Manager.createPlayer(is, "audio/midi");
   p.start();
} catch(IOException ioe) {

} catch(MediaException me) {

}
```



Mobile Media API Review

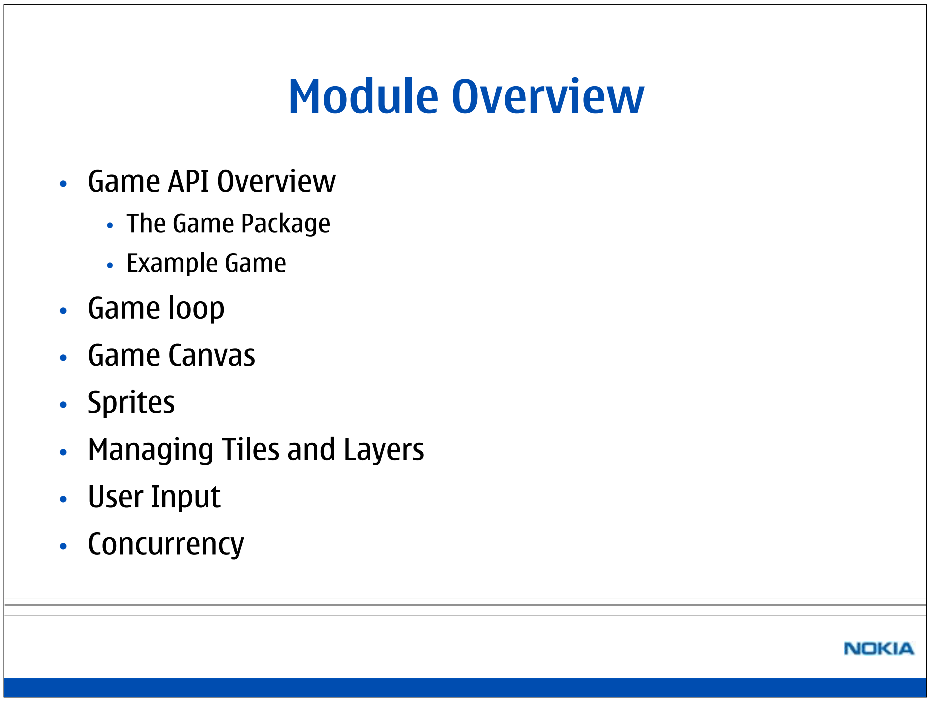
- Mobile Media API is used to play media such as videos and sound
- The `Manager` class is used to create `Player` objects
- `Player` objects are used to play media
- A `Player` object can be moved through a number of states in order to control loading, playing and stopping of the media
- `Control` interfaces are used to further control the media being played by a `Player` object, such as controlling volume and display



Game API

Module 05508

MIDP games have certain requirements that the developer needs to address in order to program a successful game. First and foremost is the ability to manage graphics in an efficient manner. Both MIDP 1.0 and 2.0 provide facilities for doing this and as we'll learn later in the module, the developer's job is a lot simpler with MIDP 2.0 so there is a lot to look forward to with the new MIDP 2.0 capable devices.



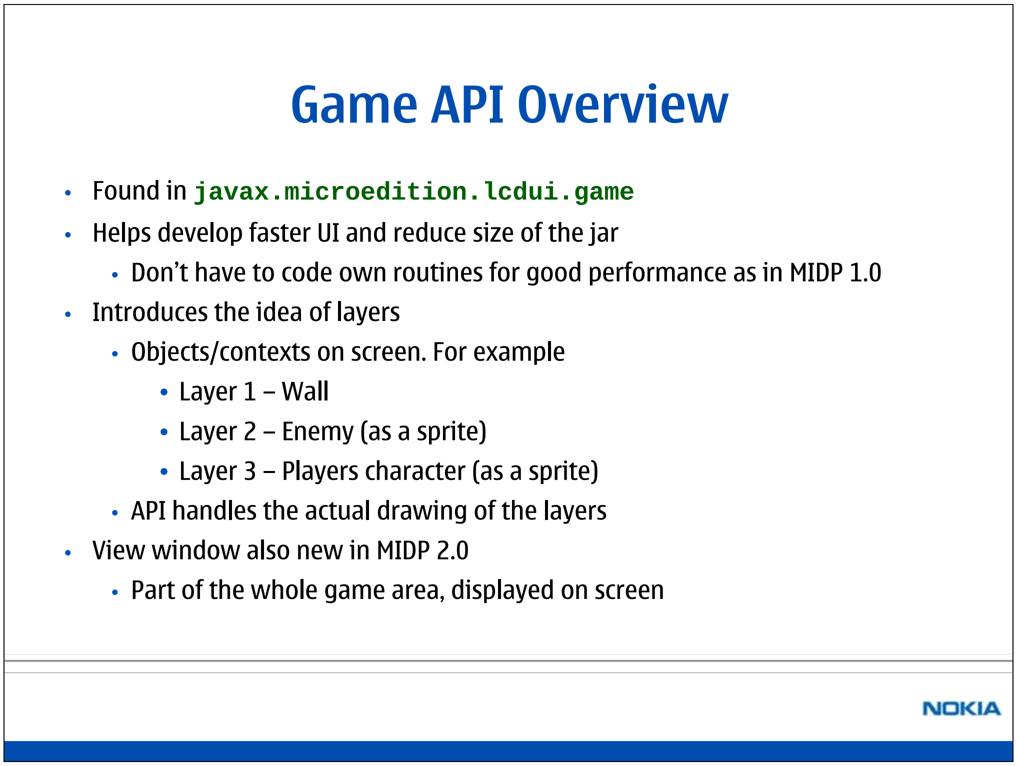
Module Overview

- Game API Overview
  - The Game Package
  - Example Game
- Game loop
- Game Canvas
- Sprites
- Managing Tiles and Layers
- User Input
- Concurrency

2-D Games

This module is about developing 2-D games. While it is certainly possible to create 3-D games with MIDP 1.0/2.0, it requires quite a lot of extra work especially since JSR 184 (Java ME 3-D scene graph) is not yet widely available on targets.

2-D games are mostly about managing bitmaps. We can use bitmaps to represents objects in the game world and to do animations which are called sprites. Other important aspects include programming in the game physics, collision detection and how non-player entities react – i.e. the AI of the game system. We'll cover all this below and the accompanying lab will show you how to create a simple yet functional game the uses the information from this module.



Game API Overview

- Found in `javax.microedition.lcdui.game`
- Helps develop faster UI and reduce size of the jar
  - Don't have to code own routines for good performance as in MIDP 1.0
- Introduces the idea of layers
  - Objects/contexts on screen. For example
    - Layer 1 – Wall
    - Layer 2 – Enemy (as a sprite)
    - Layer 3 – Players character (as a sprite)
  - API handles the actual drawing of the layers
- View window also new in MIDP 2.0
  - Part of the whole game area, displayed on screen

MIDP 2.0 includes a new Game API that simplifies writing 2D games. The API contains only five classes in the javax.microedition.lcdui.game package. These five classes provide two valuable capabilities:

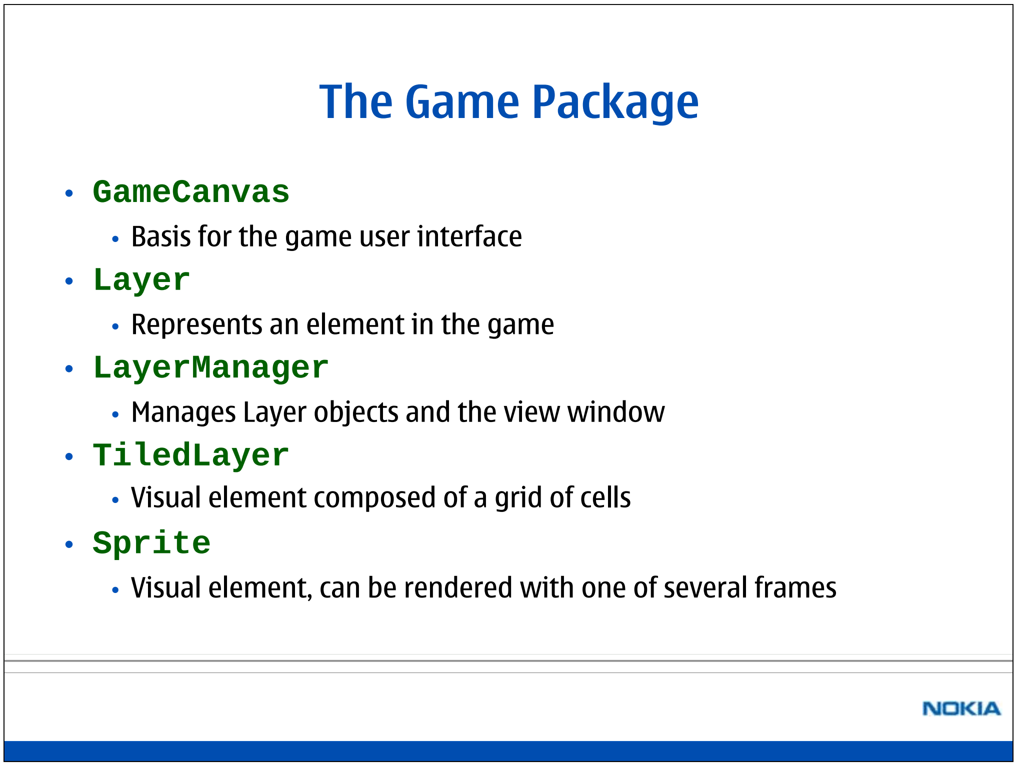
- ***A new canvas class makes it possible to paint a screen and respond to input in the body of a game loop, instead of relying on the system's paint and input threads.***
- ***A powerful and flexible layer API makes it easy to build complex scenes efficiently.***

The Game API helps developers to develop faster user interfaces with better usability that save device's resources, like memory. The Game API also helps by reducing the size of the jar file. With MIDP 1.0 game developers had to do their own graphics routines to gain good performance and programmability. This had the disadvantage of increasing the size of the jar file (and possibly some poor code). By taking the routines closer to the MIDP implementation the jar file isn't so big and the implementation of the routines is good. And developers can trust the fact that all MIDP 2.0 devices have the Game API.

The basic idea of the Game API is that the game screen consists of layers.

The wall could be on one layer, the road on another layer, and the player's character on another (as a sprite). All these layers can be handled separately and the API even handles the actual drawing of the layers.

Also the game area is often larger than the screen and scrolling the screen in the game code can be a big job. The Game API provides a view window, which is a view of the whole game area. The view window can be moved easily and points in the view window can be referenced as points on the actual screen



The Game Package

- `GameCanvas`
  - Basis for the game user interface
- `Layer`
  - Represents an element in the game
- `LayerManager`
  - Manages Layer objects and the view window
- `TiledLayer`
  - Visual element composed of a grid of cells
- `Sprite`
  - Visual element, can be rendered with one of several frames

The Game API can be found in javax.microedition.lcdui.game. There are five new classes in the API and they are GameCanvas, Layer, LayerManager, Sprite, and TiledLayer.

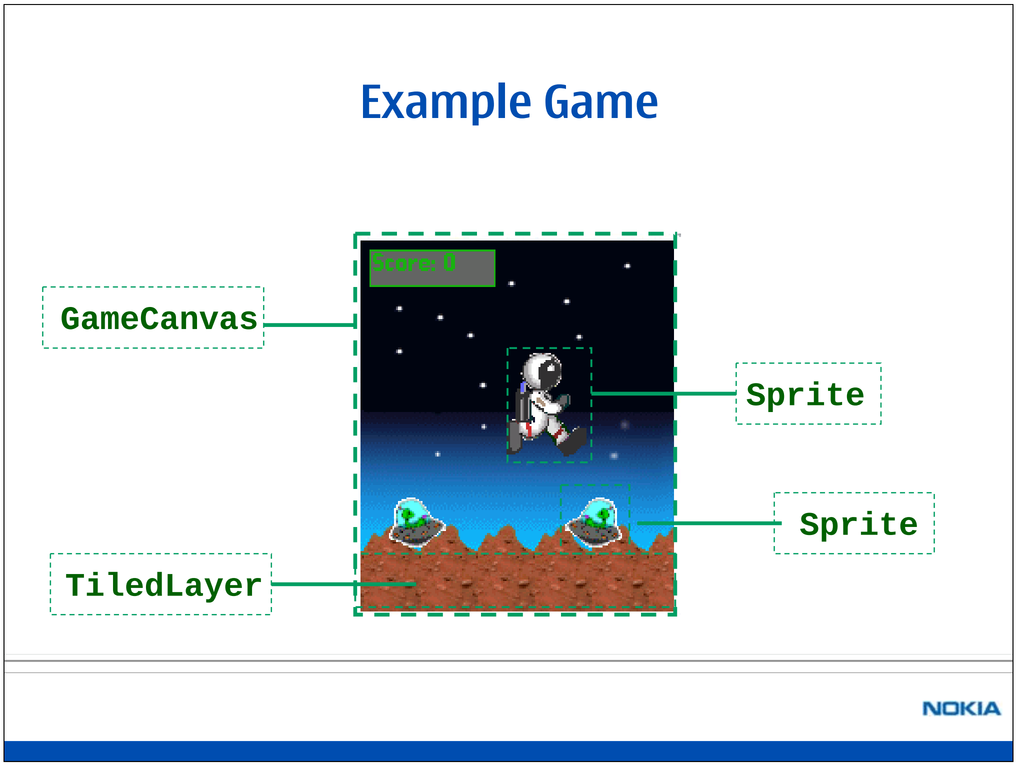
The GameCanvas is an abstract class that provides the basis for the game user interface. The class has two benefits over the Canvas class; it has an off-screen buffer and it has functionality to get the states of the physical keys of the device.

The Layer is an abstract class that represents an element in the game. Sprite and TiledLayer are inherited from Layer. Layer is mostly used by its subclasses.

The LayerManager manages several Layer objects and draws them on the screen in specific order.

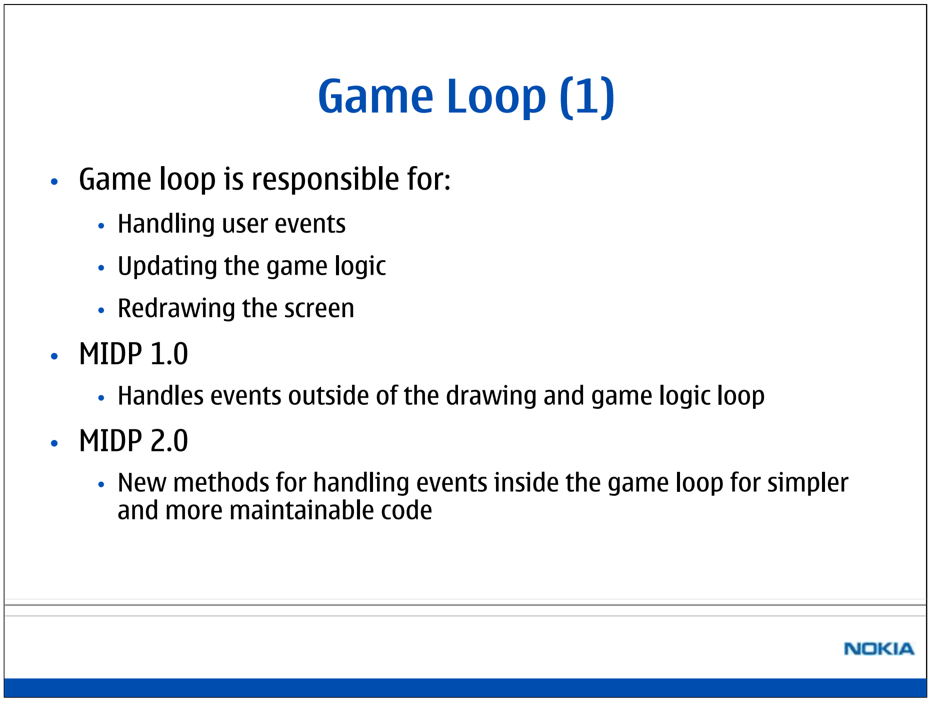
The TiledLayer is mainly meant for backgrounds, roads or other larger areas. TiledLayer consists of a grid of cells, which can be filled with images or tiles. So the background or scenery is built of small images.

The Sprite is a Layer that may contain several frames stored in an Image. One Image contains e.g. four images of walking dog. With Sprite we can use parts of the image as frames and make a sequence of frames to create motion etc. The Sprite also has functionality for checking collisions between other Sprites or TiledLayers.



Example Game

The slide shows elements of a game, and how they correspond to the classes used to create them.

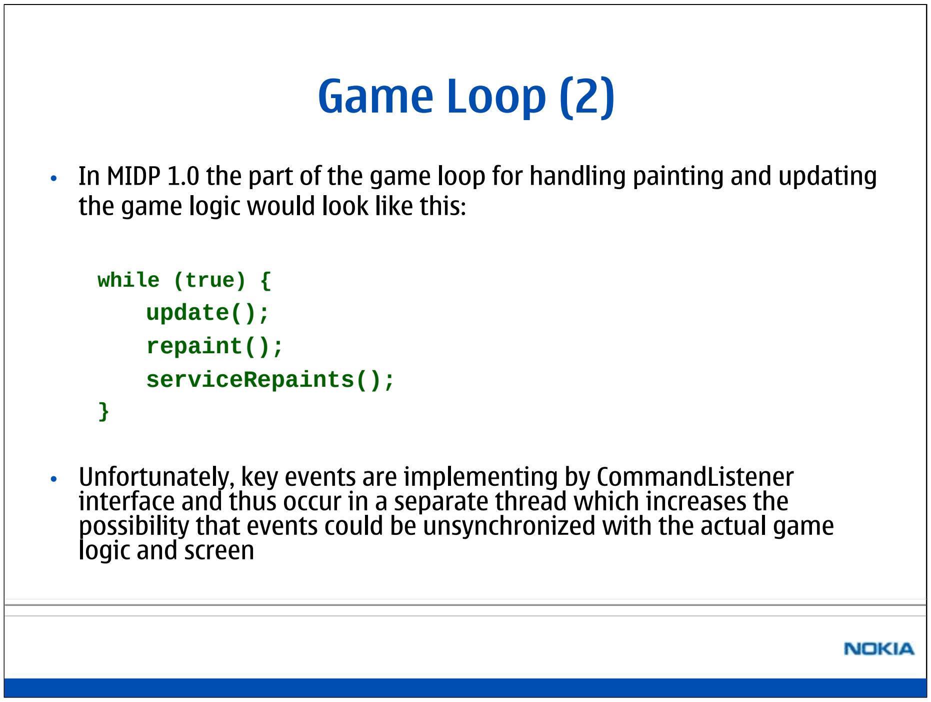


Game Loop (1)

- Game loop is responsible for:
  - Handling user events
  - Updating the game logic
  - Redrawing the screen
- MIDP 1.0
  - Handles events outside of the drawing and game logic loop
- MIDP 2.0
  - New methods for handling events inside the game loop for simpler and more maintainable code

Writing games for the MIDP platform is fun and simple. If you follow the design guidelines herein, your games should run fast and be easy to maintain and extend. In MIDP 2.0, development is radically simplified through the Game API. Specifically management of sprites, tiling, and simplified main game through the Game API allow the developer to focus all attention on game play instead of having to write a lot of supporting code as is required in MIDP 1.0.

All the APIs required to create MIDlet games are contained in the javax.microedition.lcdui and javax.microedition.lcdui.game (MIDP 2.0) packages. The classes and interfaces in the packages thereof allow for creation of images, drawing primitives and utility functions for managing onscreen graphics.

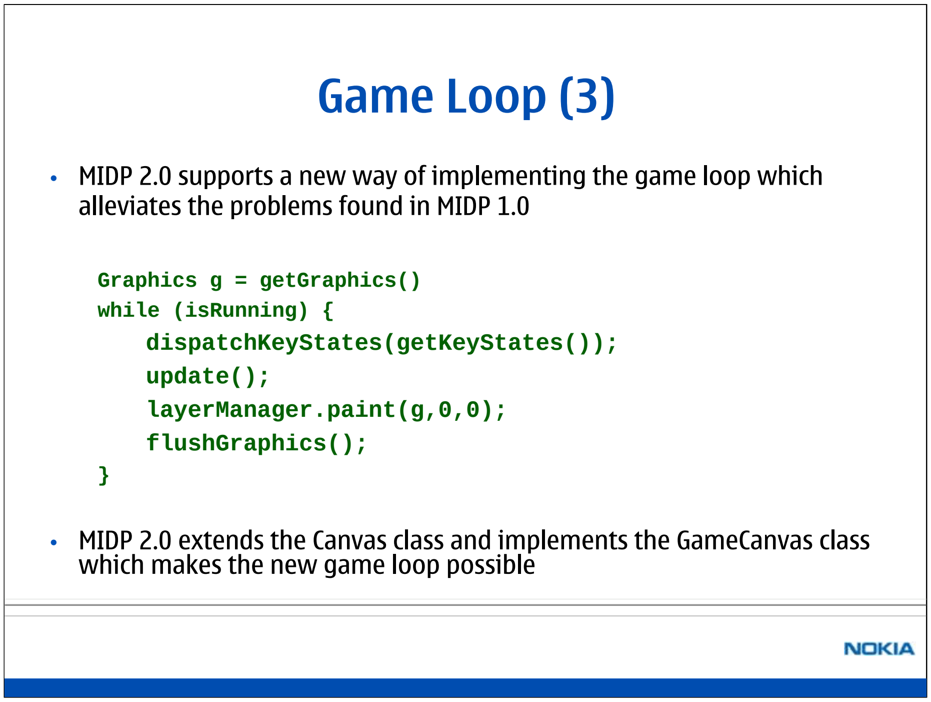


Game Loop (2)

- In MIDP 1.0 the part of the game loop for handling painting and updating the game logic would look like this:

```
while (true) {
    update();
    repaint();
    serviceRepaints();
}
```

- Unfortunately, key events are implementing by CommandListener interface and thus occur in a separate thread which increases the possibility that events could be unsynchronized with the actual game logic and screen

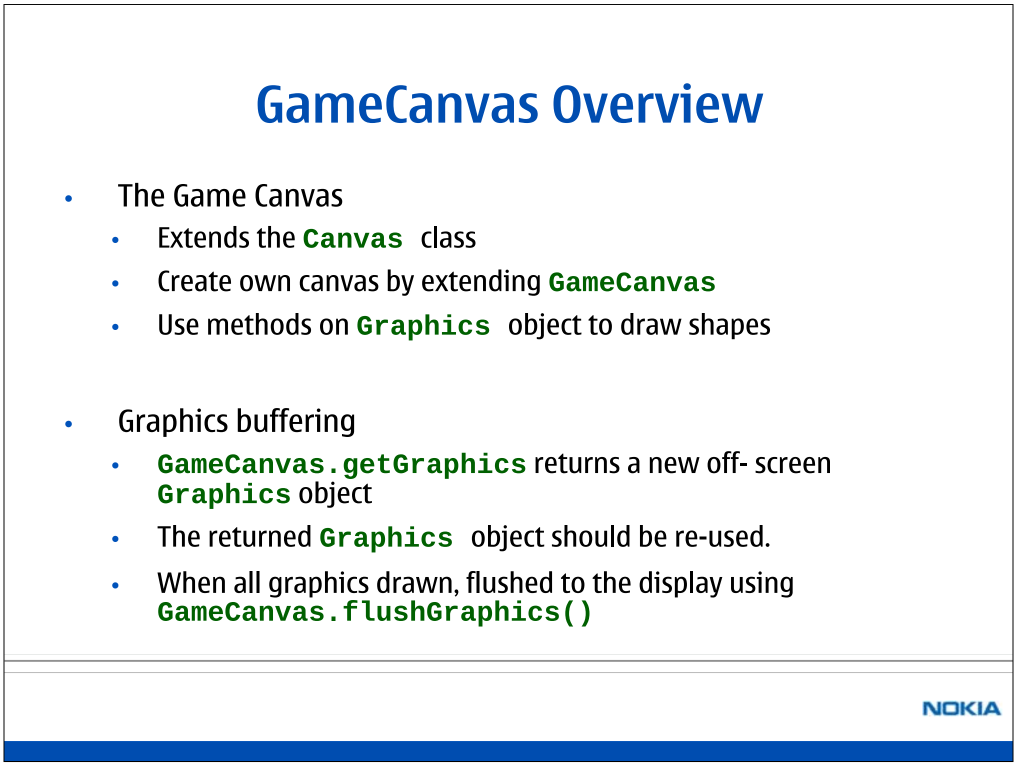


Game Loop (3)

- MIDP 2.0 supports a new way of implementing the game loop which alleviates the problems found in MIDP 1.0

```
Graphics g = getGraphics()
while (isRunning) {
    dispatchKeyStates(getKeyStates());
    update();
    layerManager.paint(g,0,0);
    flushGraphics();
}
```

- MIDP 2.0 extends the Canvas class and implements the GameCanvas class which makes the new game loop possible



GameCanvas Overview

- The Game Canvas
  - Extends the `Canvas` class
  - Create own canvas by extending `GameCanvas`
  - Use methods on `Graphics` object to draw shapes
- Graphics buffering
  - `GameCanvas.getGraphics` returns a new off- screen `Graphics` object
  - The returned `Graphics` object should be re-used.
  - When all graphics drawn, flushed to the display using `GameCanvas.flushGraphics()`

The GameCanvas class represents the area of the screen that the device has allotted to the game. The GameCanvas class differs from its superclass Canvas in two important ways: graphics buffering and the ability to query key states. Both of these changes give the game developer enhanced control over precisely when the program deals with events such as keystrokes and repainting the screen. Handling user input will be covered in subsequent sections of this lesson.

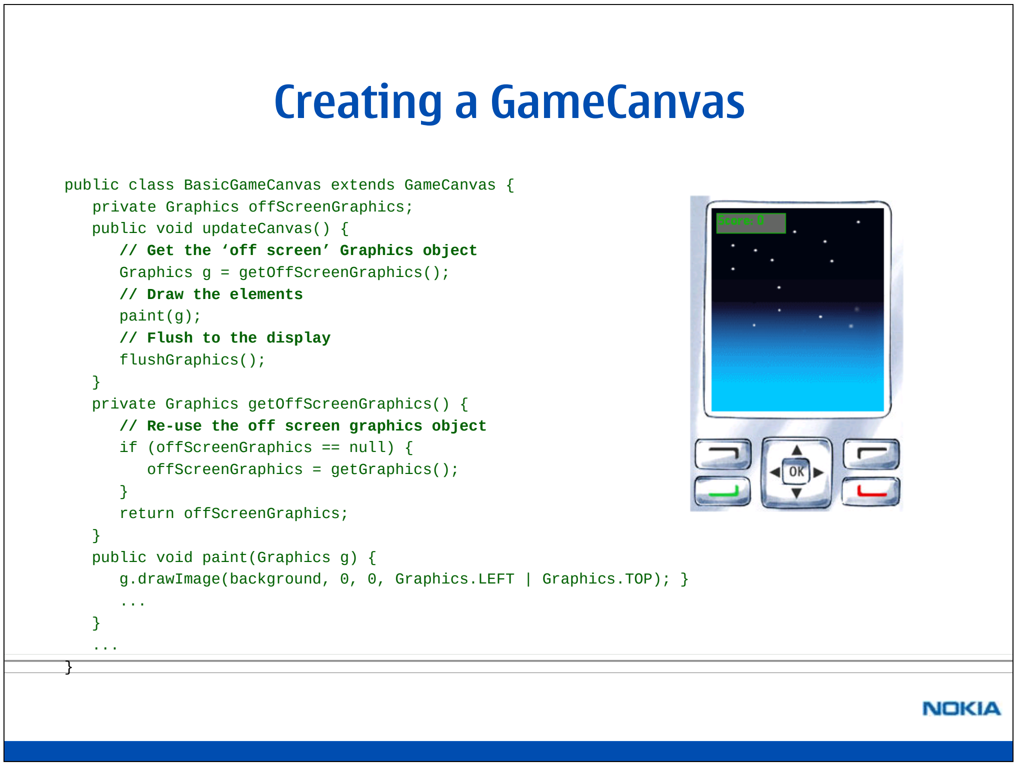
Graphics buffering reduces the delay of drawing to the screen by providing an off-screen Graphics object to create graphics behind the scenes, and then flush these to the screen all at once. This improves performance when drawing shapes to the canvas and reduces flickering when displaying animation.

The getGraphics() method creates and returns a new Graphics object every time it is called, therefore it is recommended to obtain the Graphics objects before the game begins, and the re-use them when updating the canvas.

For example, if a game canvas requires one Graphics object, this could be stored as an class variable offScreenGraphics after the getGraphics() method is called. Each time the canvas needs to be updated, this class variable could be used instead of creating a new instance of the Graphics object.

The flushGraphics() method uses the dedicated off-screen buffer provided in the GameCanvas class, and flushes this to the display. To make applications even more efficient, GameCanvas provides versions of the flushGraphics() method which allow the developer to repaint a subset of the screen if it is known that only part has changed.

flushGraphics(int x, int y, int width, int height) flushes a specific region of the off-screen buffer. This is specified by the arguments x, y, width and height. x and y being the left and top edge of a region of a size specified by the width and height arguments.



Creating a GameCanvas

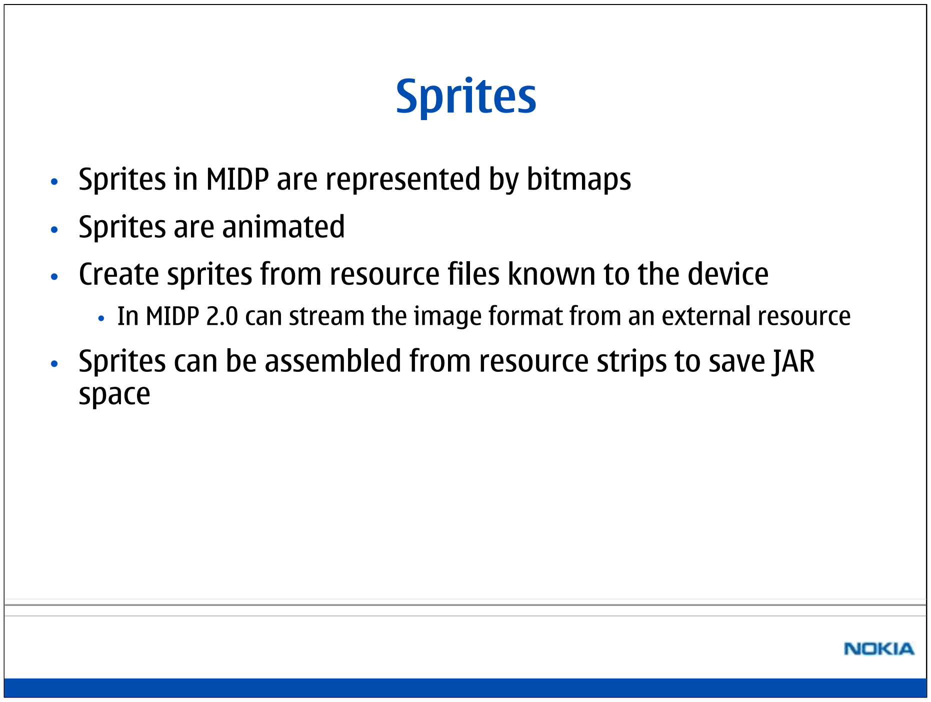
```
public class BasicGameCanvas extends GameCanvas {
   private Graphics offScreenGraphics;
   public void updateCanvas() {
      // Get the 'off screen' Graphics object
      Graphics g = getOffScreenGraphics();
      // Draw the elements
      paint(g);
      // Flush to the display
      flushGraphics();
   }
   private Graphics getOffScreenGraphics() {
      // Re-use the off screen graphics object
      if (offScreenGraphics == null) {
         offScreenGraphics = getGraphics();
      }
      return offScreenGraphics;
   }
   public void paint(Graphics g) {
      g.drawImage(background, 0, 0, Graphics.LEFT | Graphics.TOP); }
      ...
   }
   ...
}
```

A game canvas can be created by extending the GameCanvas class. The example in the slide shows how the getGraphics() and flushGraphics() methods are used to implement the off-screen buffering feature of the GameCanvas class. Also shown, in the code, is the technique of re-using the off-screen graphics object by assigning this to a local variable in the game canvas.

In the example, all drawing using the Graphics object is encapsulated within one method, namely paint(Graphics g).

The update() method is called by the main game thread which calls the paint(Graphics g) method to display the graphics on screen. This method gets a Graphics object, uses this to paint the elements, and then flushes the graphics to the display.

In the slide code, the Graphics object is returned by the method getOffScreenGraphics(). This method simple stores the Graphics object returned by calling the GameCanvas method getGraphics() in a local variable, and then returns this local variable. This is done so that the off-screen Graphics object is re-used.



Sprites

- Sprites in MIDP are represented by bitmaps
- Sprites are animated
- Create sprites from resource files known to the device
  - In MIDP 2.0 can stream the image format from an external resource
- Sprites can be assembled from resource strips to save JAR space

NOKIA

Since most graphical games use sprites it's important to know how they are created. Sprites are bitmaps that can be used for creating objects in a scene and for doing animations. Typically the bitmaps are created with a program such as Paint and the bitmaps are stored in a file known to the target device such as PNG. Animations can be done using a program such as Blender and then the keyframes are stored individually as separate bitmaps or can be stripped together as a single bitmap. The process of assignining keyframes to produce the illusion of animation is known as interpolation and we'll learn how to do that in the Lab for this course. Enough theory – here's how to create a sprite:

```
try {
   Image image = Image.createImage(“/myimage.png”);
}
catch (Exception e) {
}
```

To move and or position the sprite on the Canvas we can override the paint() method in the Canvas to draw the sprite like this:

```
protected void paint(Graphics g) {
   g.drawImage(image, xpos, ypos, Graphics.BOTTOM | Graphics.HCENTER);
}
```

MIDP 2.0

In MIDP 2.0 we create the image like above, however we add another step that creates an Sprite object:

```
try {
   Image image = Image.createImage(“/myimage.png”);
}
catch (Exception e) {
}
Sprite sprite = new Sprite(image, 10, 10);
```

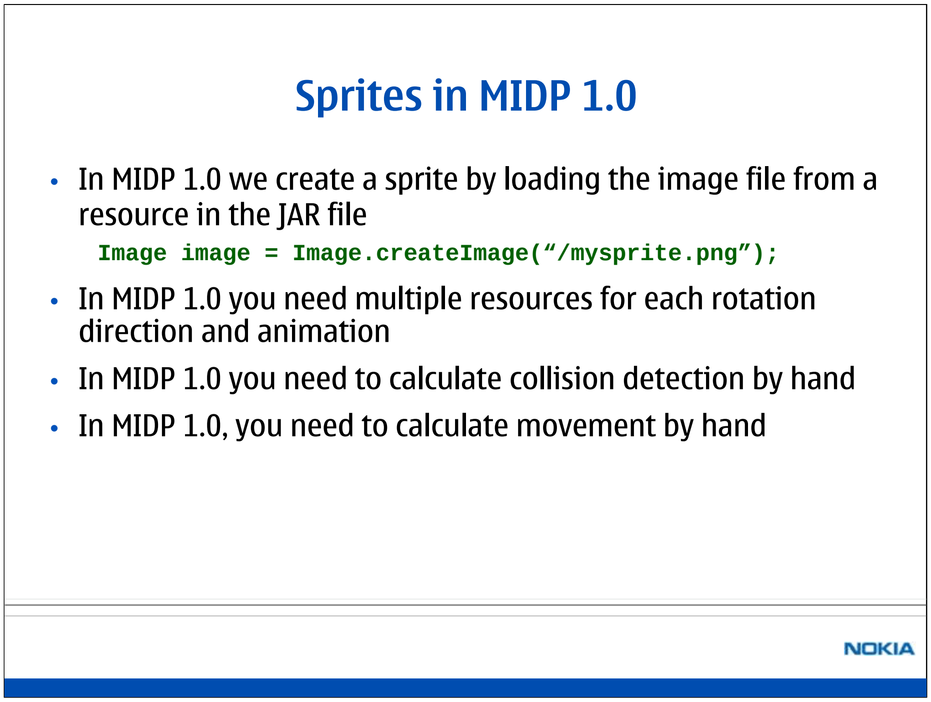
The advantage with using the Sprite class is that you can position the sprite using the move() and setPositions() methods:

```
sprite.move(xpos, ypos);
```

Note that sprites are only available from the GameCanvas class which is a new class in MIDP 2.0. The GameCanvas allows the developer to get the state of keys of the target device at any time and has a double-buffered framebuffer for flicker-free animation.

Drawing sprites in MIDP 2.0 is a bit different than in MIDP 1.0 and for that you need to understand Layers and the LayerManager class which is covered in the next section. However, the basic code is simple enough and looks like this:

```
LayerManager layerManager = new LayerManager();
layerManager.append(sprite);
Graphics g = getGraphics(); // from GameCanvas
layerManager.paint(g,0,0);
```

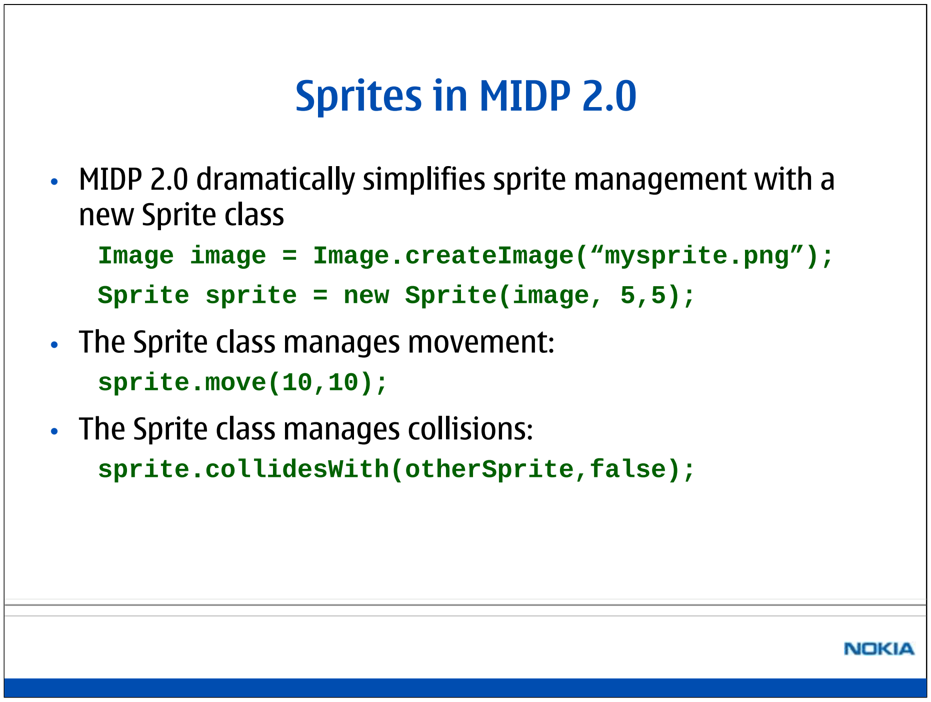


Sprites in MIDP 1.0

- In MIDP 1.0 we create a sprite by loading the image file from a resource in the JAR file

```
Image image = Image.createImage(“/mysprite.png”);
```

- In MIDP 1.0 you need multiple resources for each rotation direction and animation
- In MIDP 1.0 you need to calculate collision detection by hand
- In MIDP 1.0, you need to calculate movement by hand



Sprites in MIDP 2.0

- MIDP 2.0 dramatically simplifies sprite management with a new Sprite class

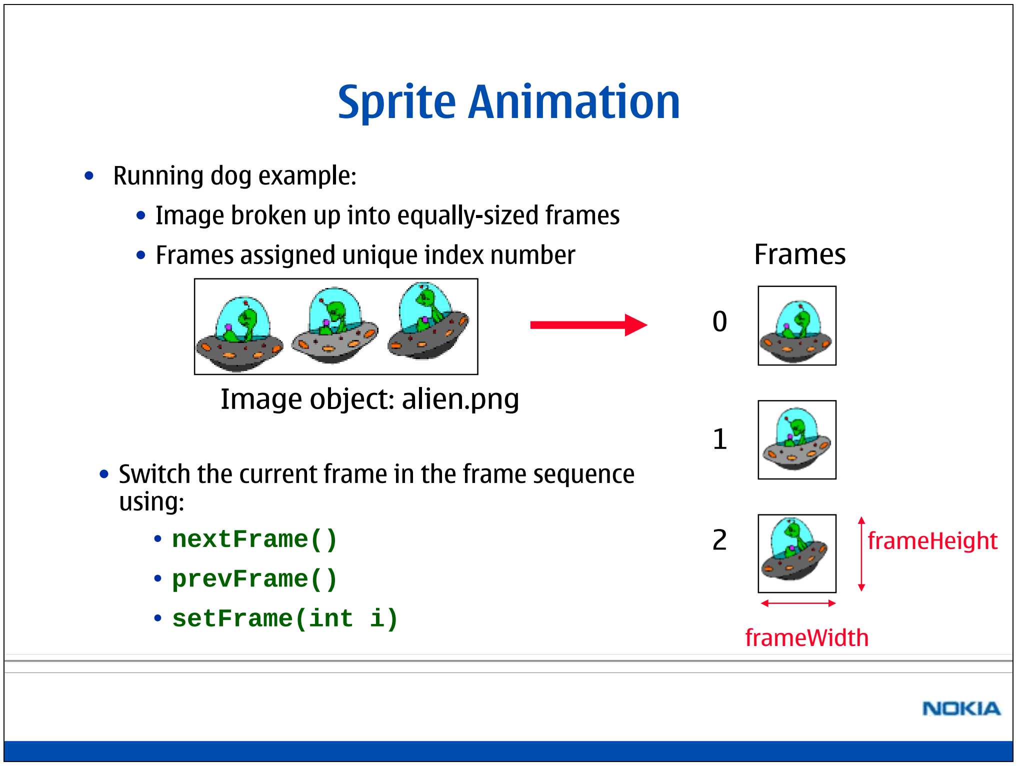
```
Image image = Image.createImage(“mysprite.png”);
Sprite sprite = new Sprite(image, 5,5);
```

- The Sprite class manages movement:

```
sprite.move(10,10);
```

- The Sprite class manages collisions:

```
sprite.collidesWith(otherSprite,false);
```



Sprite Animation

- Running dog example:
  - Image broken up into equally-sized frames
  - Frames assigned unique index number

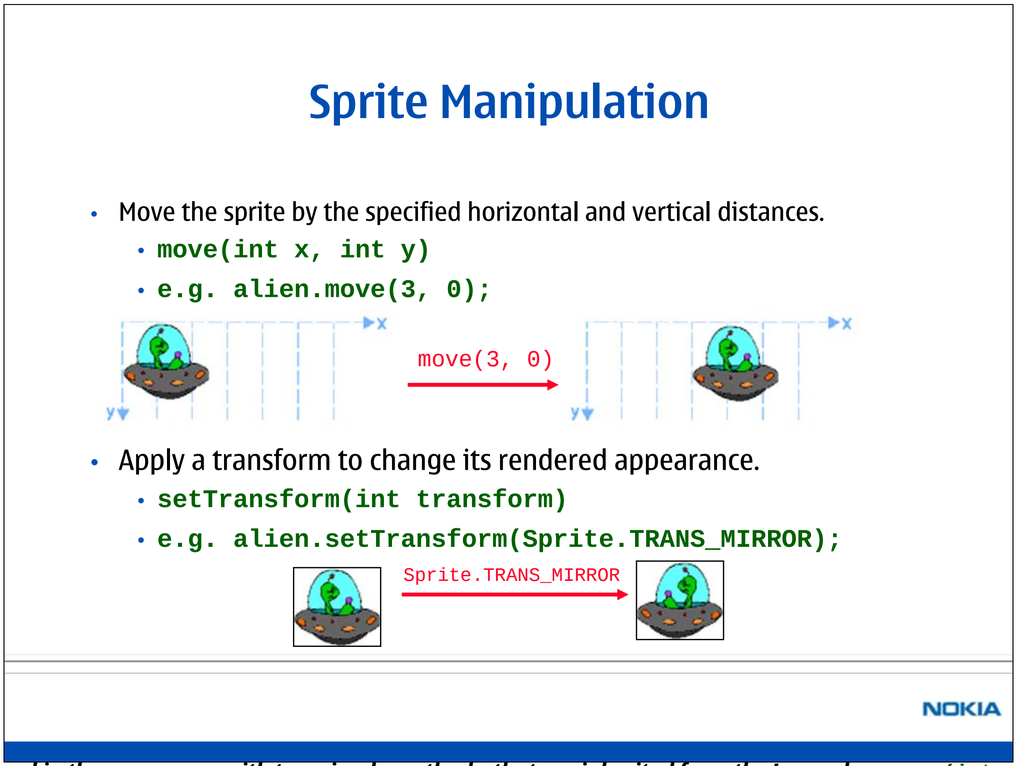
Image object: alien.png

- Switch the current frame in the frame sequence using:
  - `nextFrame()`
  - `prevFrame()`
  - `setFrame(int i)`

The frames used to create an animated Sprite are provided by a single image file, loaded using an Image object. The Sprite class separates the image into each individual frame of a width and height specified in its constructor and assigns a number to each, starting at 0.

The sequence in which the frames are displayed defaults to the order they where drawn in the original image, however, this can be overridden using the method setFrameSequence(int[] sequence), passing to the method an array of integers, representing the order of the frame numbers.

To achieve the effect of animation, the current frame being displayed on the sprite must be changed. This is done by calling the methods nextFrame() and prevFrame(). These methods simply select the next or previous frame in the frame sequence. If the nextFrame() method is called at the end of the sequence, the first frame in the sequence is displayed. If the prevFrame() method is called at the start of the sequence, the last frame is displayed.



Sprite Manipulation

- Move the sprite by the specified horizontal and vertical distances.
  - `move(int x, int y)`
  - `e.g. alien.move(3, 0);`

- Apply a transform to change its rendered appearance.
  - `setTransform(int transform)`
  - `e.g. alien.setTransform(Sprite.TRANS_MIRROR);`

A Sprite can be moved in the game area with two simple methods, that are inherited from the Layer class, `move(int x, int y) and setPosition(int x, int y)`*.*

Also, transforms can be applied to a Sprite to change its rendered appearance. This is achieved using the setTransform(int transform) method.

The move(...) method takes two parameters, the distance to move along the x axis and the distance the move along the y axis. The example in the slide moves the sprite to the right, along the x axis. To move the sprite to the left, along the x axis, the code would be

```
alien.move(-3, 0);
```

i.e. setting the distance to move along the x axis to a negative value.

The setPosition(...) method takes two parameters, the horizontal position and vertical position. For example, the dog sprite was currently at the position (5,5) and we want to translate it for the position (10,10) we use the code

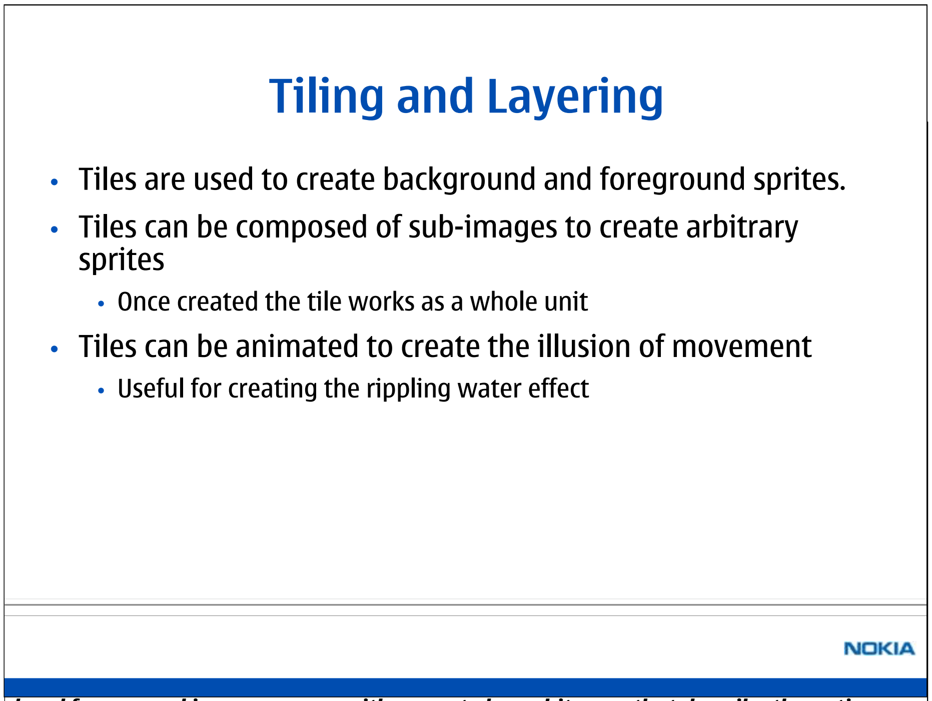
```
alien.setPosition(10, 10);
```

The setTransform(...) method takes one parameter, the desired transform for this Sprite. Transform constants, that this method can be called with, are stored in the Sprite class itself. In the slide example, the sprite is reflected about its vertical centre by calling the method with the constant Sprite.TRANS_MIRROR

Developers should note that applying a transform to a Sprite may result in its width and height being swapped, for example if it is rotated 90 degrees.

The other transforms that can be applied to a sprite are detailed below:

- *TRANS_NONE – No transform should be applied to the Sprite. This can be used reset the Sprite to its original state as it was constructed from the Image object.*
- *TRANS_ROT90 – This rotates the Sprite 90 degrees clockwise.*
- *TRANS_ROT180 – This rotates the Sprite 180 degrees clockwise*
- *TRANS_ROT270 – This rotates the Sprite 270 degrees clockwise*
- *TRANS_MIRROR – This reflects the Sprite about its vertical centre*
- *TRANS_MIRROR_ROT90 – This reflects the Sprite about its vertical centre then rotates it 90 degrees clockwise*
- *TRANS_MIRROR_ROT180 – This reflects the Sprite about its vertical centre then rotates it 180 degrees clockwise*
- *TRANS_MIRROR_ROT270 – This reflects the Sprite about its vertical centre then rotates it 270 degrees clockwise*



Tiling and Layering

- Tiles are used to create background and foreground sprites.
- Tiles can be composed of sub-images to create arbitrary sprites
  - Once created the tile works as a whole unit
- Tiles can be animated to create the illusion of movement
  - Useful for creating the rippling water effect

When we create background and foreground images we can either create large bitmaps that describe the entire scene or we can use a technique called tiling to keep our JAR file size down and allow us some flexibility as well.

We can create tiles by first loading our resource bitmap:

```
try {
   Image image = Image.createImage("/mytile.png");
}
catch (Exception e) {
}
```

Then we can iterate over the area of our Canvas where we want the tile to appear. For example, we can override the Canvas paint method and do the following:

```
void protected paint(Graphics g) {
   for (int i =0; i < getWidth(); i+= image.getWidth()) {
      g.drawImage(image, i, yPos, Graphics.TOP | Graphics.LEFT)
```

The above would give us a way to draw a background using a single sprite. What about if we wanted to use multiple sprites to create a varied landscape? If you want to do that, then we can create a static double array with offsets controlling which sprites should be drawn and at which position. For example if we create two bitmaps and refer to them with integer Ids 1 and 2, we can describe our tile like this:

```
Image images[] = new Images[2];
try {
   images[0] = Image.createImage("/myimage1.png");
   images[1] = Image.createImage("/myimage2.png");
} catch (Exception e) {
}
int[][] background = {{0,1,1,1,0,0},{1,1,0,0,0,1}};
```

Note that in the above example the positioning, X and Y, of the images is determined by actual width and height of the sprite. You could just as easily input the X and Y coordinates into a double array to control the placement of the sprites and make things even more interesting by using random placement. Both these techniques are covered further in the accompanying lab.

Finally, we can discuss layering which is assigning which sprites are to be considered as whole objects -- even when connected together as tiles – and abstracting the ordering of the respective sprites. MIDP 1.0 does not include explicit layering functionality, thus if we want to draw sprites using layering abstractions then we can follow the simple rule: draw sprites in background to foreground order.

MIDP 2.0

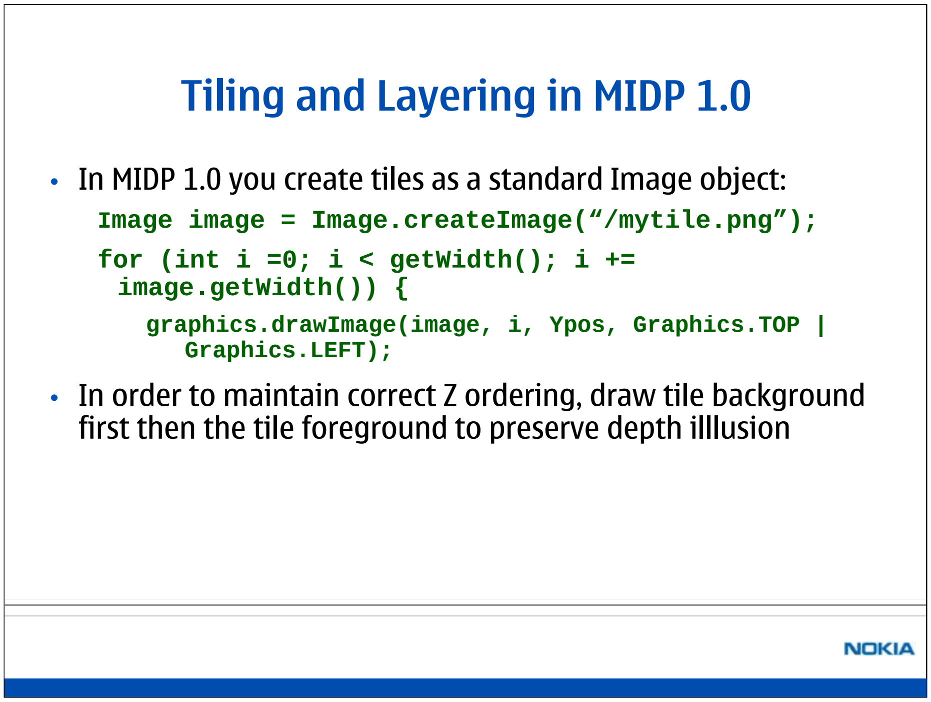
Performing tiling in MIDP 2.0 is far simpler than the above MIDP 1.0 examples. To create a tile in MIDP 2.0 we can tack advantage of the LayerManager class that works with the GameCanvas to use TiledLayers:

```
Image image;
try {
   image = Image.createImage("/tiles.png");
} catch (Exception e) {}
LayerManager layer = new LayerManager();
TiledLayer tiledLayer = new TiledLayer(COLS, ROWS, image, Xsz, Ysz);
```

Once the TiledLayer object is created then you can use the fillCell() or fillCells method to place the tiles where you want them:

```
tiledLayer.fillCells(0,0,COLS,ROWS, 1);
tiledLayer.fillCells(10,10,20,10,2);
layer.append(tiledLayer);
```

Once you're satisfied with where you want the tiles to appear, you can call the append() method to append the tiles in the background or use insert() method to place the tiles anywhere within the layer manager.

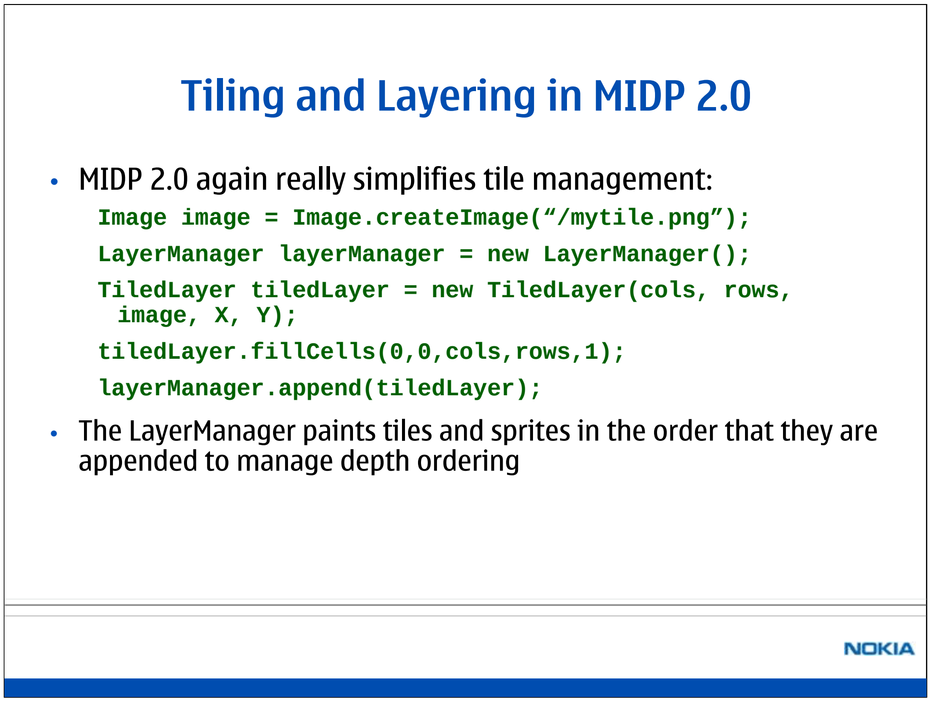


Tiling and Layering in MIDP 1.0

- In MIDP 1.0 you create tiles as a standard Image object:

```
Image image = Image.createImage(“/mytile.png”);
for (int i =0; i < getWidth(); i +=
 image.getWidth()) {
   graphics.drawImage(image, i, Ypos, Graphics.TOP |
     Graphics.LEFT);
```

- In order to maintain correct Z ordering, draw tile background first then the tile foreground to preserve depth illlusion

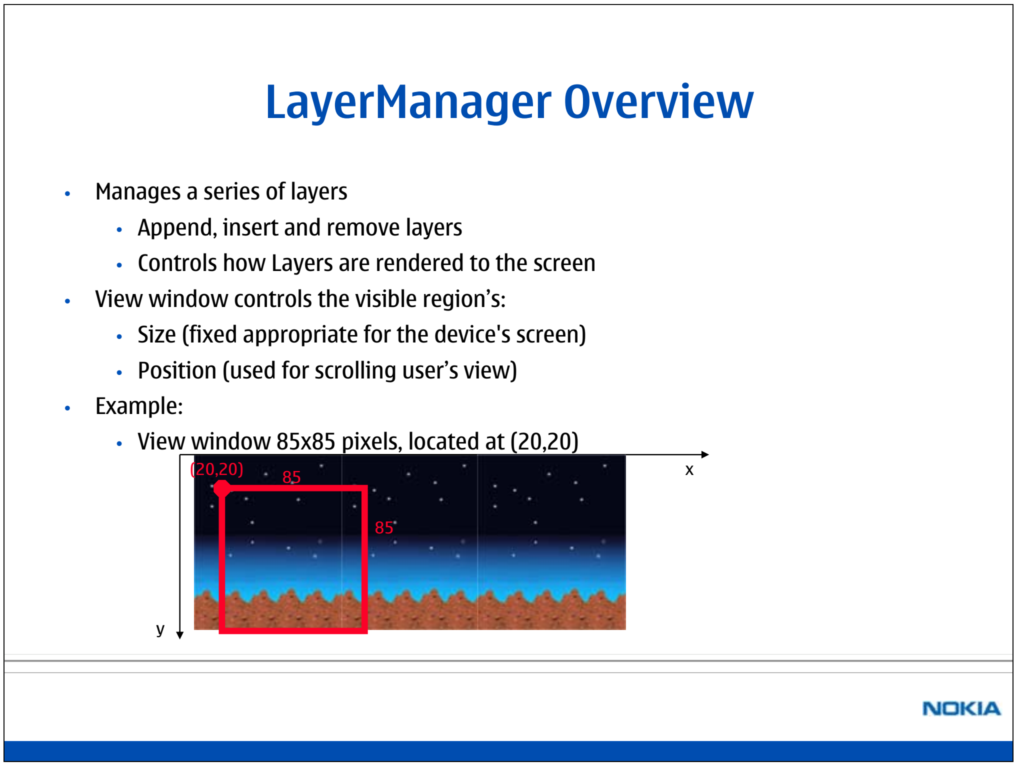


Tiling and Layering in MIDP 2.0

- MIDP 2.0 again really simplifies tile management:

```
Image image = Image.createImage(“/mytile.png”);
LayerManager layerManager = new LayerManager();
TiledLayer tiledLayer = new TiledLayer(cols, rows,
  image, X, Y);
tiledLayer.fillCells(0,0,cols,rows,1);
layerManager.append(tiledLayer);
```

- The LayerManager paints tiles and sprites in the order that they are appended to manage depth ordering



LayerManager Overview

- Manages a series of layers
  - Append, insert and remove layers
  - Controls how Layers are rendered to the screen
- View window controls the visible region’s:
  - Size (fixed appropriate for the device's screen)
  - Position (used for scrolling user’s view)
- Example:
  - View window 85x85 pixels, located at (20,20)

The LayerManager class helps to organize all of the graphical layers needed for a game. Layers can be appended, inserted and removed from the LayerManager. The first layer to be appended to the LayerManager is the last to painted on screen, i.e. it is the top most layer and is in front of all the other layers. Probably the most useful aspect of the LayerManager class is that a virtual screen can be created that is much larger than the actual screen and then the developer can choose which section of it will appear on the actual screen at any point. This is known as the View window.

The view window controls the size of the region that is visible on screen, and its position relative to the LayerManager's coordinate system. Effects such as scrolling the user’s view can be achieved by changing the position of the view window.

For example, the view window in the slide is currently located at the position (20, 20). To scroll the user’s view to the right, to position of the view could be set to (60, 20), i.e. moves along the x axis.

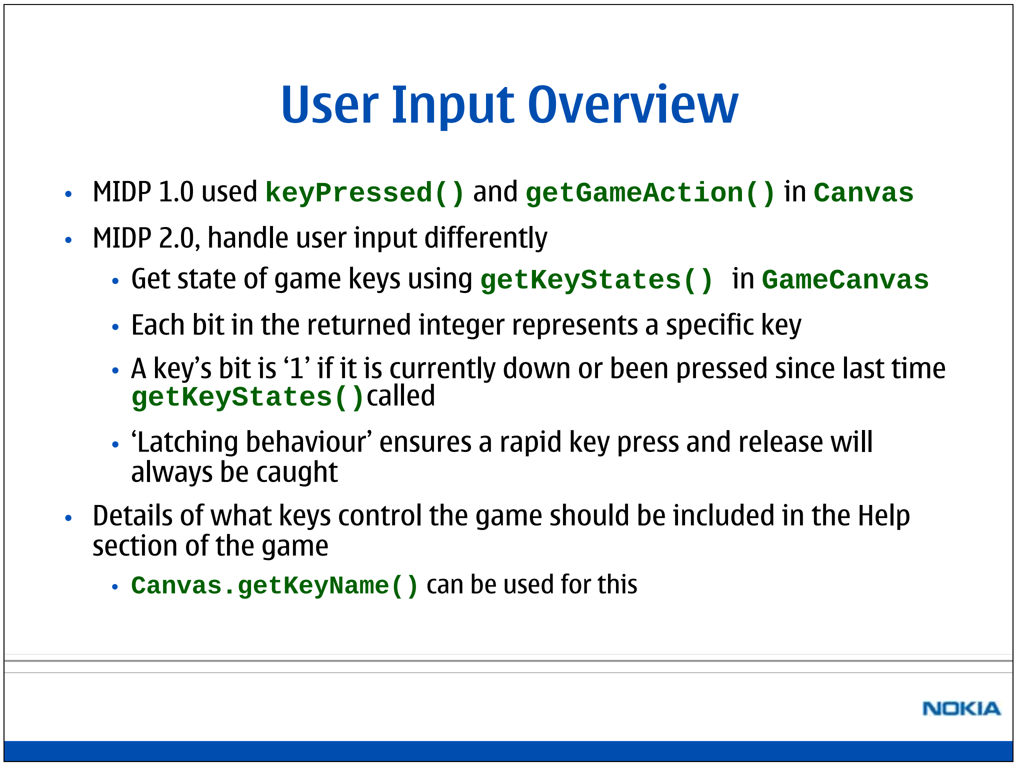
The size of the view window controls how large the user's view will be, and is usually fixed at a size that is appropriate for the device's screen.

Allowing the possibility of a virtual screen that is much larger than the actual screen is extremely helpful for games on devices with very small screens. It saves the developer huge amounts of time and effort if, for example, a game involves a player exploring an elaborate maze.

However, having a large virtual screen and an actual screen means that the developer has to deal with two separate coordinate systems.

Layers need to be placed in the LayerManager according the LayerManager’s coordinate system. The Graphics object of the GameCanvas has its own coordinate system.

So keep in mind that the method LayerManager.paint(Graphics g, int x, int y) paints the layer on the screen according to the coordinates of the GameCanvas whereas the method LayerManager.setViewWindow(int x, int y, int width, int height) sets the visible rectangle of the LayerManager in terms of the LayerManager's coordinate system.



User Input Overview

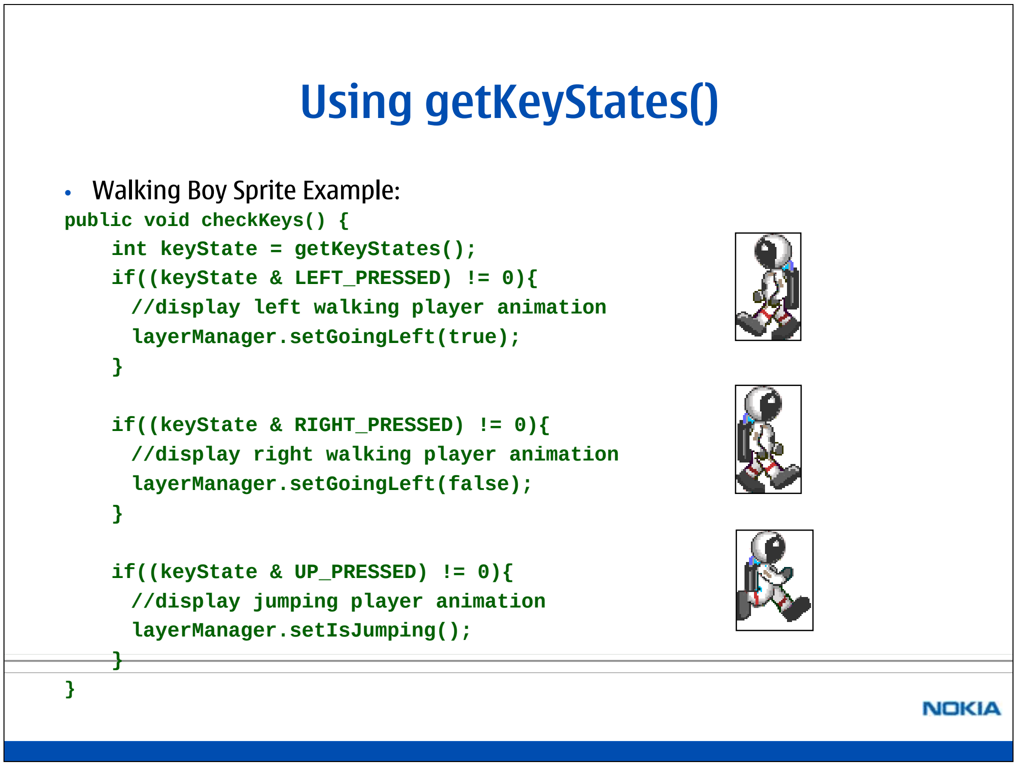
- MIDP 1.0 used `keyPressed()` and `getGameAction()` in `Canvas`
- MIDP 2.0, handle user input differently
  - Get state of game keys using `getKeyStates()` in `GameCanvas`
  - Each bit in the returned integer represents a specific key
  - A key's bit is '1' if it is currently down or been pressed since last time `getKeyStates()`called
  - 'Latching behaviour' ensures a rapid key press and release will always be caught
- Details of what keys control the game should be included in the Help section of the game
  - `Canvas.getKeyName()` can be used for this

GameCanvas allows you to bypass the normal key-event mechanisms so that all game logic can be contained in a single loop by introducing a polling technique to obtain the current states of the device's keys.

In MIDP 1.0, handling user-input meant that you had to wait for the system to call the keyPressed(int keyCode) method in your extended Canvas, and then call getGameAction(int keyCode) to find out what key was pressed. The problem with this is that it was difficult to know what's going on with the other parts of the application when the system calls keyPressed(). For example, if your code in keyPressed() is making updates to the game state at the same time the screen is being rendered in the Canvas's paint() method, the screen may end up looking strange.

GameCanvas gets around this problem by introducing methods to obtain the current states of the device's keys. Instead of waiting for the system indicate a key has been pressed, GameCanvas can determine immediately which keys are pressed by calling the getKeyStates() method. This polling technique can be done just before the screen is updated so that the game state can be suitably updated before the screen is redrawn.

The getKeyStates() method returns an integer containing the key state information. The information returned is in the form of bits, with one bit per key. A key's bit will be 1 if the key is currently down or has been pressed at least once since the last time this method was called. The bit will be 0 if the key is currently up and has not been pressed at all since the last time this method was called. Therefore, to check for a particular key, the developer has to use a bitwise operation with the result and the bit representing a particular key.



Using getKeyStates()

- Walking Boy Sprite Example:

```
public void checkKeys() {
    int keyState = getKeyStates();
    if((keyState & LEFT_PRESSED) != 0){
      //display left walking player animation
      layerManager.setGoingLeft(true);
    }

    if((keyState & RIGHT_PRESSED) != 0){
      //display right walking player animation
      layerManager.setGoingLeft(false);
    }

    if((keyState & UP_PRESSED) != 0){
      //display jumping player animation
      layerManager.setIsJumping();
    }
}
```

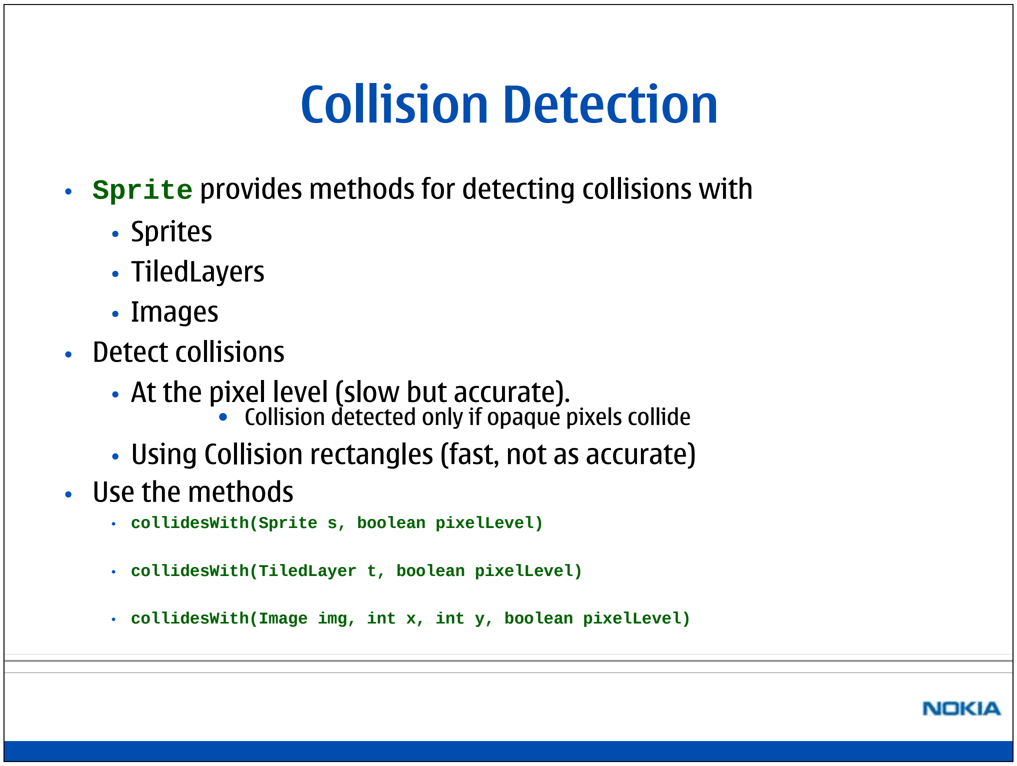
The code in the slide shows a typical way of checking what keys have been pressed on the device. The GameCanvas class includes constants that represent game keys on a device. These can be used with the result of calling the getKeyStates() method to determine which keys have been pressed.

The AND bitwise operator '&' is used to determine whether a particular key has been pressed. This operator returns a 1 if corresponding bits in both operands have a 1, otherwise it returns 0.

For example, in the code in the slide, the first if statement checks if the left key has been pressed. If it had, the getKeyStates() method would have returned '4' (i.e.00000100). Using this result and the constant LEFT_PRESSED (i.e. 00000100) with the AND bitwise operator, results in a value that is not zero.

The game key constants that can be used in the GameCanvas class are:

- `UP_PRESSED`
- `DOWN_PRESSED`
- `LEFT_PRESSED`
- `RIGHT_PRESSED`
- `FIRE_PRESSED`
- `GAME_A_PRESSED`
- `GAME_B_PRESSED`
- `GAME_C_PRESSED`
- `GAME_D_PRESSED`



Collision Detection

- `Sprite` provides methods for detecting collisions with
  - Sprites
  - TiledLayers
  - Images
- Detect collisions
  - At the pixel level (slow but accurate).
    - Collision detected only if opaque pixels collide
  - Using Collision rectangles (fast, not as accurate)
- Use the methods
  - `collidesWith(Sprite s, boolean pixelLevel)`
  - `collidesWith(TiledLayer t, boolean pixelLevel)`
  - `collidesWith(Image img, int x, int y, boolean pixelLevel)`

An important aspect of writing games is the ability to check when on screen objects collide with other on screen objects. For example, in a game involving a maze, the sprites need to stay within the walls of the maze, and so if sprites collide with the walls, the game needs prevent them from travelling through it. Also, it may be important in some games to detect if one moving sprite has collided with another moving sprite. The Sprite class provides methods the detect whether its has collided with Sprites, TiledLayer or Images.

Each Sprite has a default collision rectangle. This is a bounding rectangle that is used for collision detection purposes. The default rectangle is location at 0,0 on the sprite and has the same dimensions as the sprite.

In the methods provided to detect collisions, the developer can decided whether the detection algorithms should just check if the collision rectangles intersect, or whether the actual opaque pixels collide.

In the collidesWith methods, if the pixel level boolean is set to true, the detection algorithms check whether opaque pixels have collided. Choosing this option means detection is accurate and precise, but processing may be slow.

If the pixel level boolean is set to false, only the collision rectangles are used. Choosing this option means the detection will not be as accurate, but may increase the speed of processing the detection.

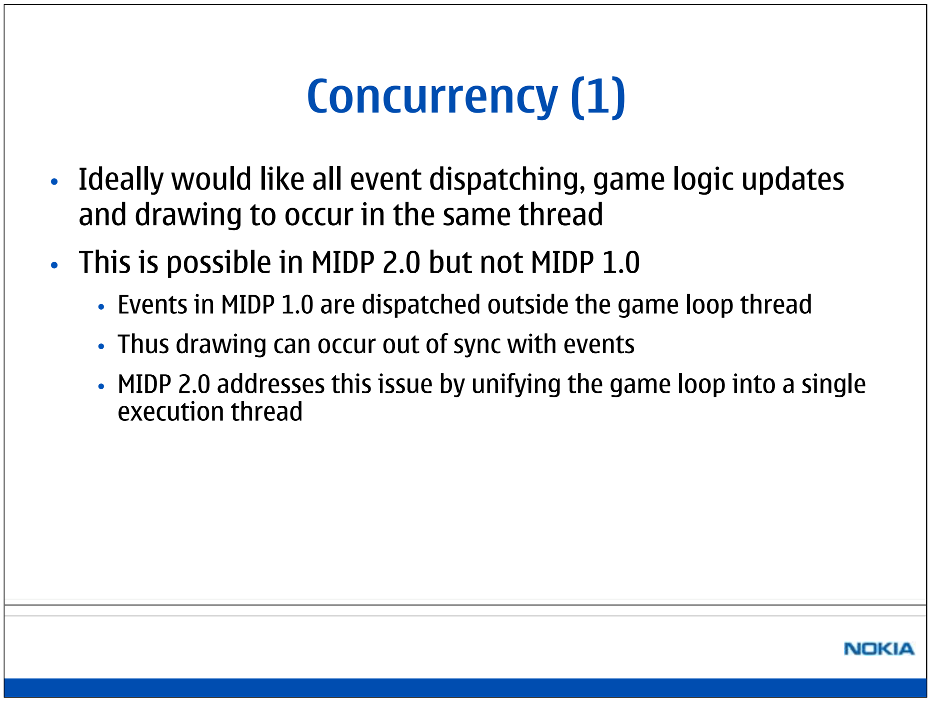
Pixel level collision detection is useful if the objects being detected for collision are not 'square' or 'rectangular' shape, as sprites are only classed as have collided if there edges have actually intersected.

Collision rectangle detection is useful if game processing speed is important, and the sprite being detected for collision are 'square' or 'rectangular' shape.

The collision rectangle for a particular sprite can be defined by using the method

```
defineCollisionRectangle(int x, int y, int width, int height)
```

Using this method is useful if the developer needs to specify collision detection for just an small

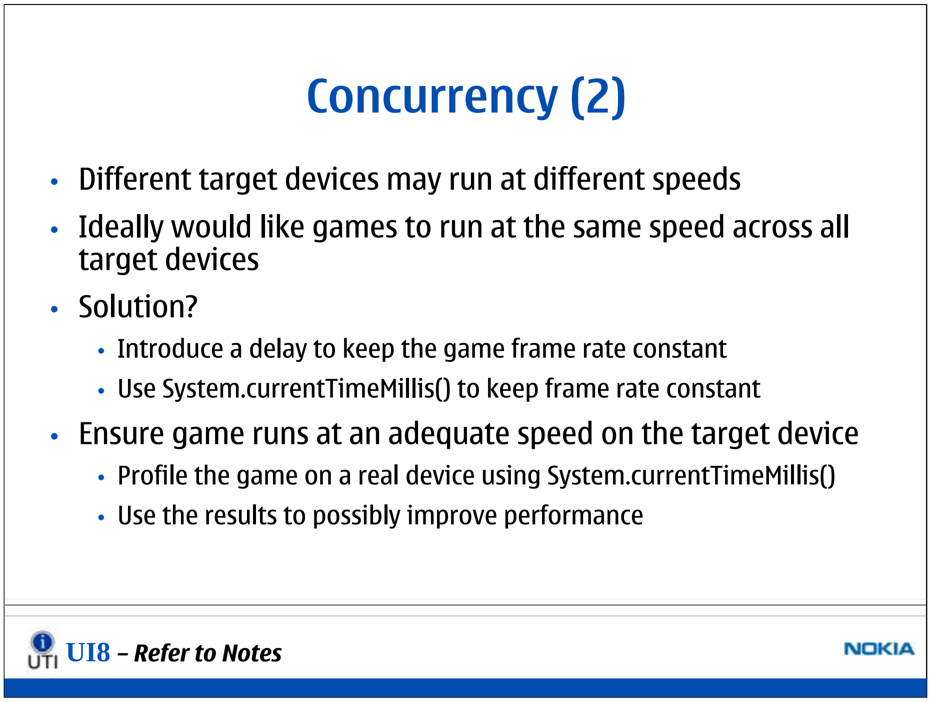


Concurrency (1)

- Ideally would like all event dispatching, game logic updates and drawing to occur in the same thread
- This is possible in MIDP 2.0 but not MIDP 1.0
  - Events in MIDP 1.0 are dispatched outside the game loop thread
  - Thus drawing can occur out of sync with events
  - MIDP 2.0 addresses this issue by unifying the game loop into a single execution thread

Most games will have a separate thread that manages the game loop. You can do this by implementing the Runnable interface in your game canvas. The purpose of the game loop is usually fairly simple: you check for user events – e.g. key presses, update the sprites on the canvas, then redraw the canvas. One thing to note however is that you generally want to game play speed to be consistent over platforms because that speed is what you are testing the game play against. So the way to do that is to add a check to make sure that each tick of the game loop occurs at the same speed:

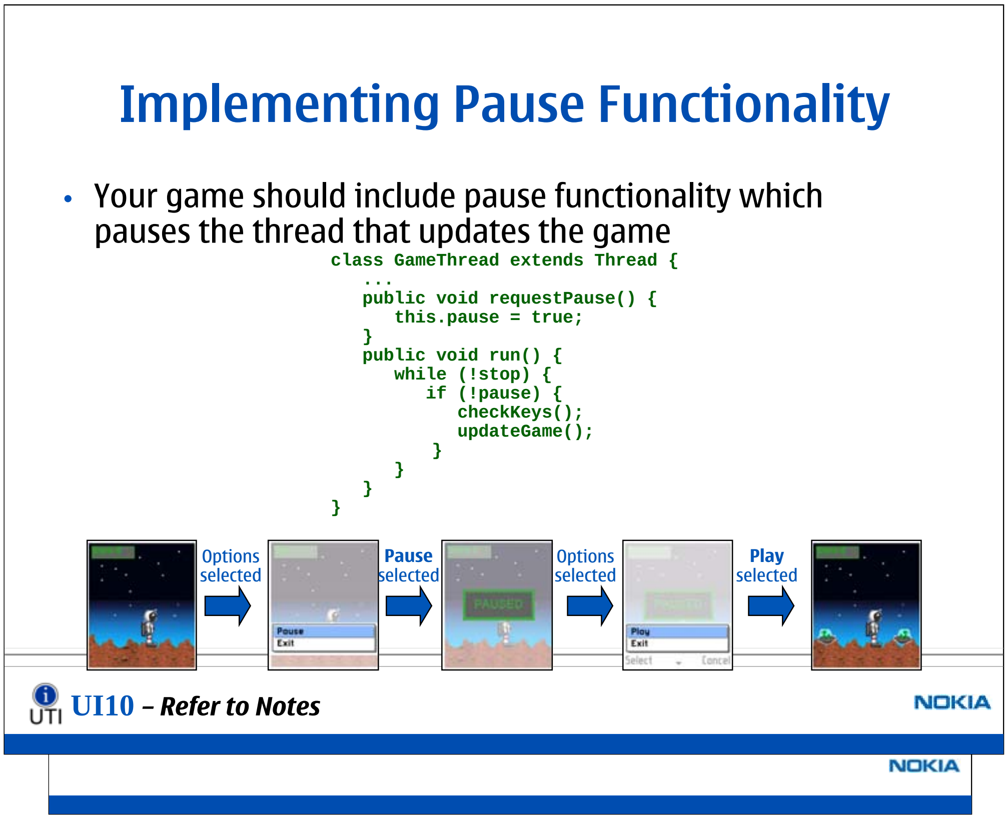
```
public final void run() {
   while (isRunning) {
      time = System.currentTimeMillis();
      repaint();
      serviceRepaints();
      update();
      time = delay – (System.currentTimeMillis() - time);
      try {
        (Thread.sleep((time < 0 ? 0 : time));
      } catch (InterruptedException e) {
      }
   }
}
```



Concurrency (2)

- Different target devices may run at different speeds
- Ideally would like games to run at the same speed across all target devices
- Solution?
  - Introduce a delay to keep the game frame rate constant
  - Use System.currentTimeMillis() to keep frame rate constant
- Ensure game runs at an adequate speed on the target device
  - Profile the game on a real device using System.currentTimeMillis()
  - Use the results to possibly improve performance

UI8 – ***Refer to Notes***



Implementing Pause Functionality

- Your game should include pause functionality which pauses the thread that updates the game

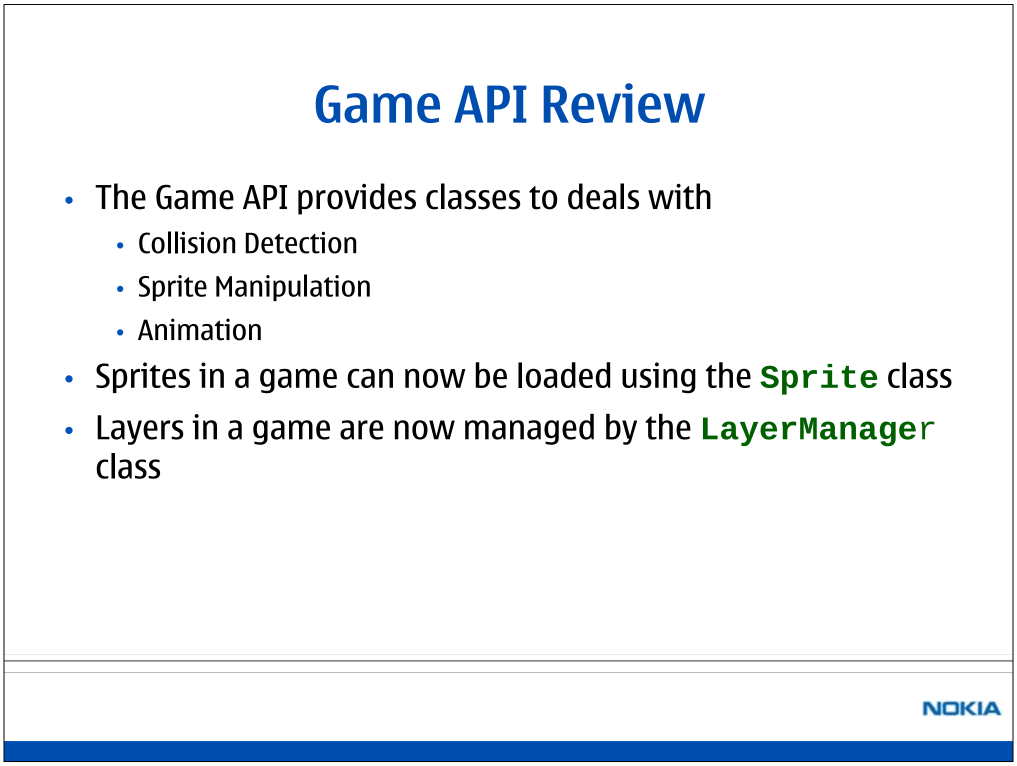
```
class GameThread extends Thread {
   ...
   public void requestPause() {
      this.pause = true;
   }
   public void run() {
      while (!stop) {
         if (!pause) {
            checkKeys();
            updateGame();
         }
      }
   }
}
```

UI10 – Refer to Notes

To be able pause a game, the thread that updates the canvas should be paused. This can be done by setting a variable that the game thread checks occasionally. When the game thread detects that the variable is set, it calls Object.wait(). The paused thread can then be woken up by calling its Object.notify() method.

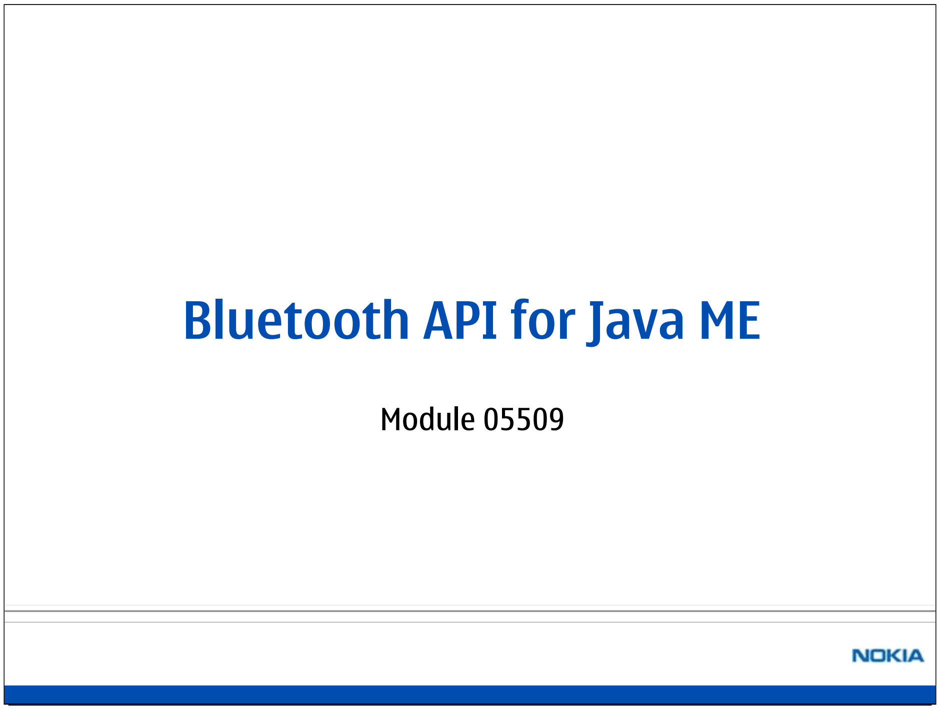
The following code shows this technique. It can be seen that when the pause method is called, a variable pauseThread is set to true. A check is made in the run method. When this variable is true, the run method is calls wait().

```
public class GameThread
    extends Thread {
  private boolean pauseThread = false;
  private GameCanvas canvas;
  public GameThread(GameCanvas canvas) {
    this.canvas = canvas;
  }
  public void pause() {
    this.pauseThread = true;
  }
  public void play() {
    this.pauseThread = false;
  }
  public void run() {
    while (true) {
      canvas.repaint();
      // Check if should wait
      synchronized (this) {
        while (pauseThread) {
          try {
            wait();
          }
          catch (Exception e) {
          }
        }
      }
    }
  }
```



Game API Review

- The Game API provides classes to deals with
  - Collision Detection
  - Sprite Manipulation
  - Animation
- Sprites in a game can now be loaded using the `Sprite` class
- Layers in a game are now managed by the `LayerManager` class

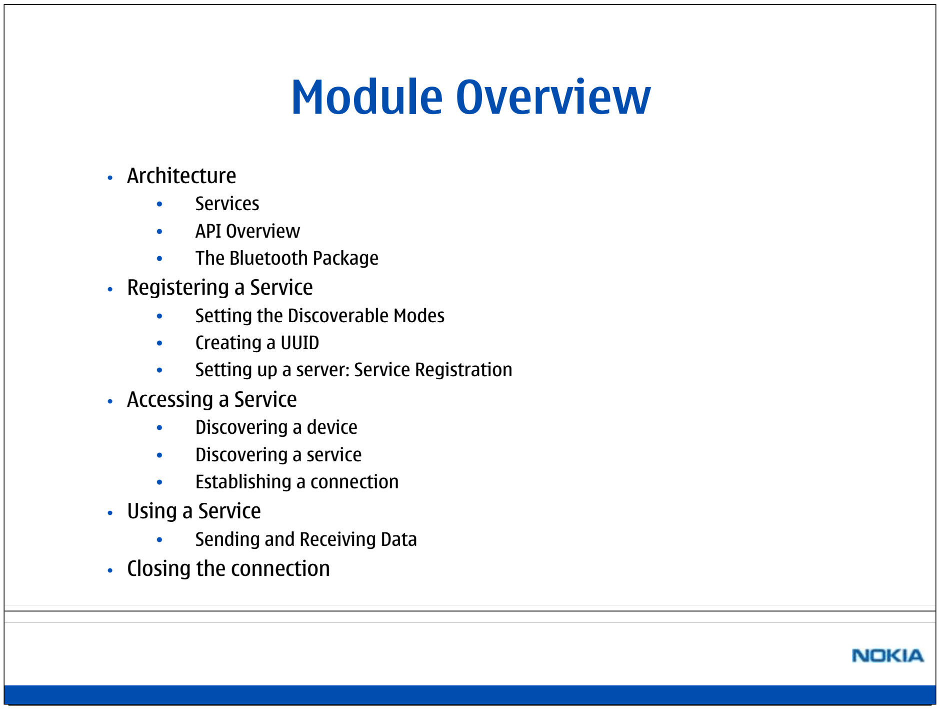


Bluetooth API for Java ME

Module 05509

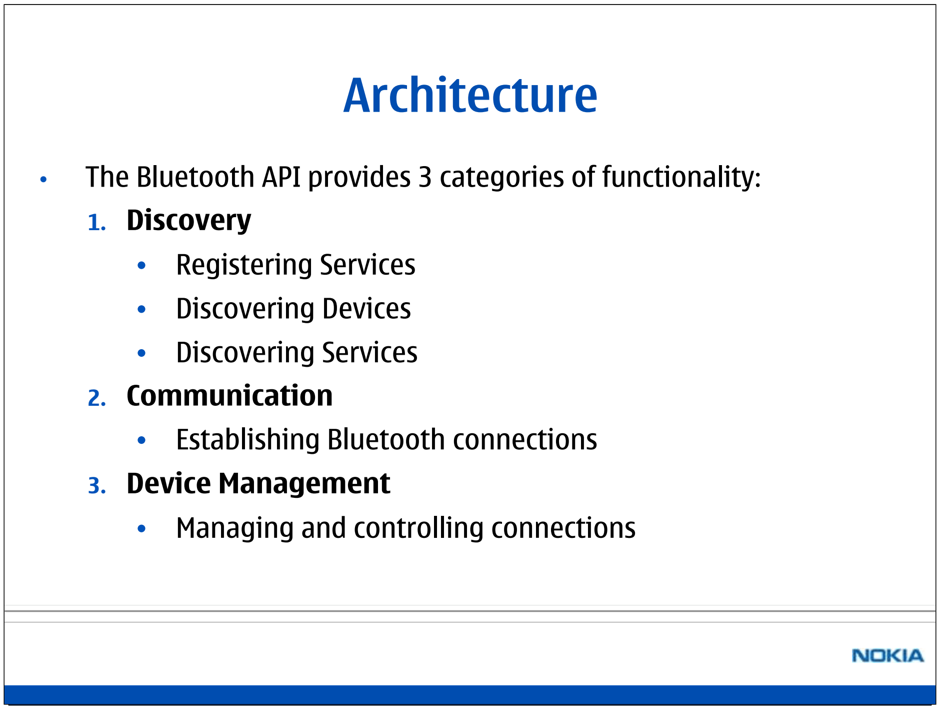
The Bluetooth API, as defined in JSR-82, allows Java technology-enabled devices to integrate into a Bluetooth environment. The goal of JSR-82 was to create a standard API, so that the same code would work on different devices, and different Bluetooth stacks. It hides the complexity of the Bluetooth protocol stack behind a set of Java APIs that allow you to focus on application development rather than the low-level details of Bluetooth. The Java APIs for Bluetooth do not implement the Bluetooth specification, but rather provide a set of APIs to access and control a Bluetooth-enabled device. JSR-82 concerns itself primarily with providing Bluetooth capabilities to Java MIDP devices.

The core functionality of the Bluetooth API is to allow applications to discover Bluetooth devices, search for services on a device and establish a connection between Bluetooth devices.



Module Overview

- Architecture
  - Services
  - API Overview
  - The Bluetooth Package
- Registering a Service
  - Setting the Discoverable Modes
  - Creating a UUID
  - Setting up a server: Service Registration
- Accessing a Service
  - Discovering a device
  - Discovering a service
  - Establishing a connection
- Using a Service
  - Sending and Receiving Data
- Closing the connection



Architecture

- The Bluetooth API provides 3 categories of functionality:
  1. **Discovery**
     - Registering Services
     - Discovering Devices
     - Discovering Services
  2. **Communication**
     - Establishing Bluetooth connections
  3. **Device Management**
     - Managing and controlling connections

Discovery includes Device Discovery, Service Discovery and Service Registration.

Device Discovery is the process of detecting the Bluetooth devices that are in the area. Device discovery requires the local device to broadcast an inquiry request and wait for Bluetooth devices in the area to respond. In order to determine if a device responds to an inquiry request, the Bluetooth specification has defined three different discoverable modes (general, limited, and not discoverable) and two different inquiry types (general and limited). When a device issues a general inquiry, all devices in the general and limited discoverable modes will respond to the request. When a limited inquiry is issued, only those devices that are limited discoverable will respond. If a device is not discoverable, it never responds to an inquiry request.

Once device discovery has identified a list of nearby Bluetooth devices, the next step for a Bluetooth application is to find out what applications or services those devices provide.

A **Service** is an entity that can provide information, perform an action, or control a resource on behalf of another entity.

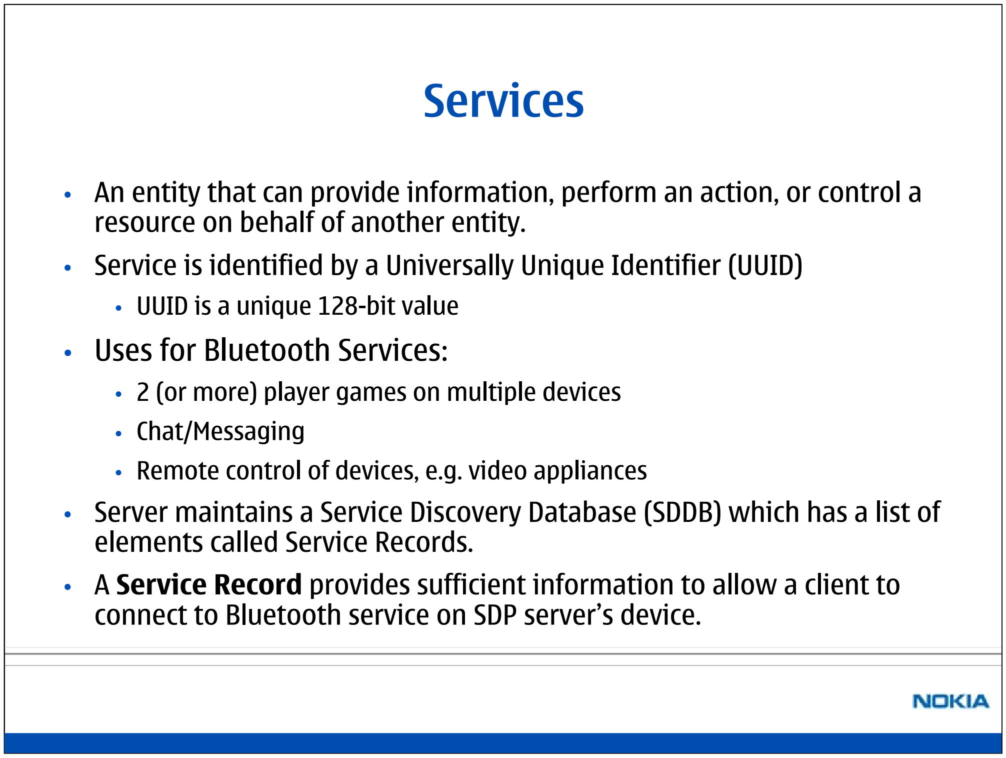
Service Discovery is the process of using the Bluetooth service discovery protocol to find services of interest on nearby Bluetooth devices.

A Server application advertises the service it offers through its service record. A service record describes a Bluetooth service using a set of service attributes. Service attributes may specify how to connect to a service, the name of the service, a textual description of the service, and other helpful information. A Bluetooth profile specification describes the contents of the service records used for that profile.

Service registration is the process of making Service records visible to prospective clients of a Bluetooth service.

Communication includes setting up connections between devices and using those connections for Bluetooth communication between applications. These connections can be made over several different protocols, and include RFCOMM, L2CAP and OBEX.

Device Management allows for managing and controlling Bluetooth connections. It deals with managing local and remote device states and properties. In addition, it facilitates the security aspects of connections.



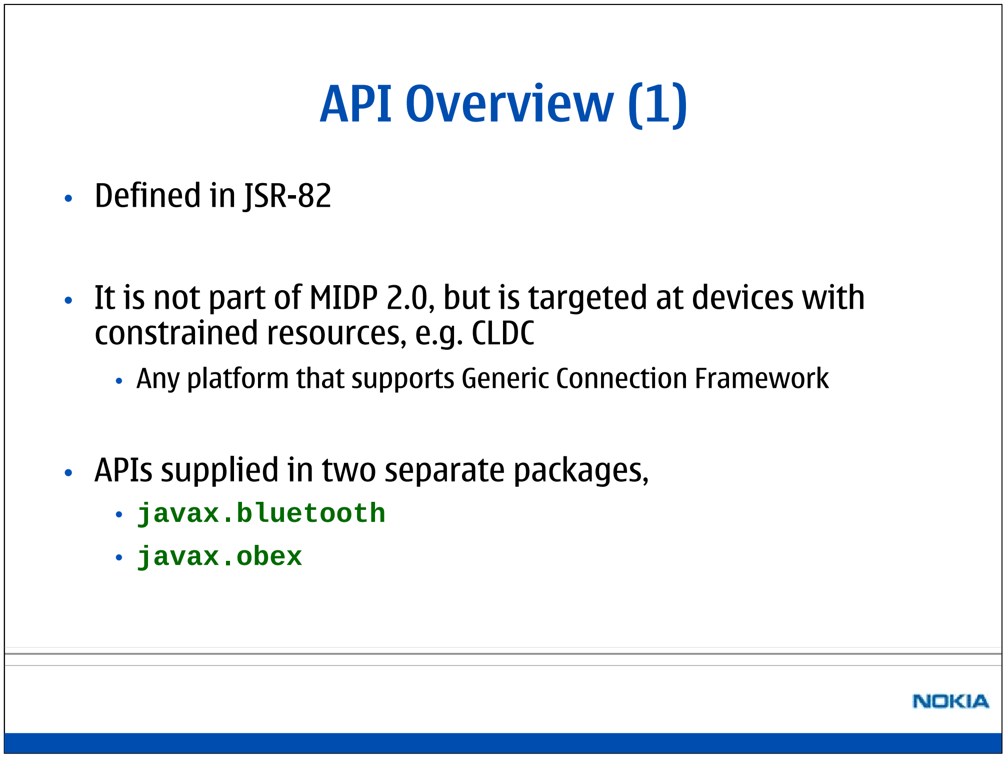
Services

- An entity that can provide information, perform an action, or control a resource on behalf of another entity.
- Service is identified by a Universally Unique Identifier (UUID)
  - UUID is a unique 128-bit value
- Uses for Bluetooth Services:
  - 2 (or more) player games on multiple devices
  - Chat/Messaging
  - Remote control of devices, e.g. video appliances
- Server maintains a Service Discovery Database (SDDB) which has a list of elements called Service Records.
- A **Service Record** provides sufficient information to allow a client to connect to Bluetooth service on SDP server’s device.

A service may be implemented as software, hardware, or a combination of both. A service has attributes that describe a single characteristic of a service. Each service is an instance of a service class, which provides an initial indicator of the capabilities of the service, and defines what other attributes, including their types and semantics, must or can appear in the service record.

All of the information about a service that is maintained by a Service Discovery Protocol (SDP) server is contained within a single service record. A service record consists entirely of a list of service attributes

The SDP allows Bluetooth devices to discover what services other devices may offer. It permits service browsing and searching for specific services. But SDP does not provide access to the services themselves. All information about a service is stored in a single Service Record within the SDP server.



API Overview (1)

- Defined in JSR-82
- It is not part of MIDP 2.0, but is targeted at devices with constrained resources, e.g. CLDC
  - Any platform that supports Generic Connection Framework
- APIs supplied in two separate packages,
  - `javax.bluetooth`
  - `javax.obex`

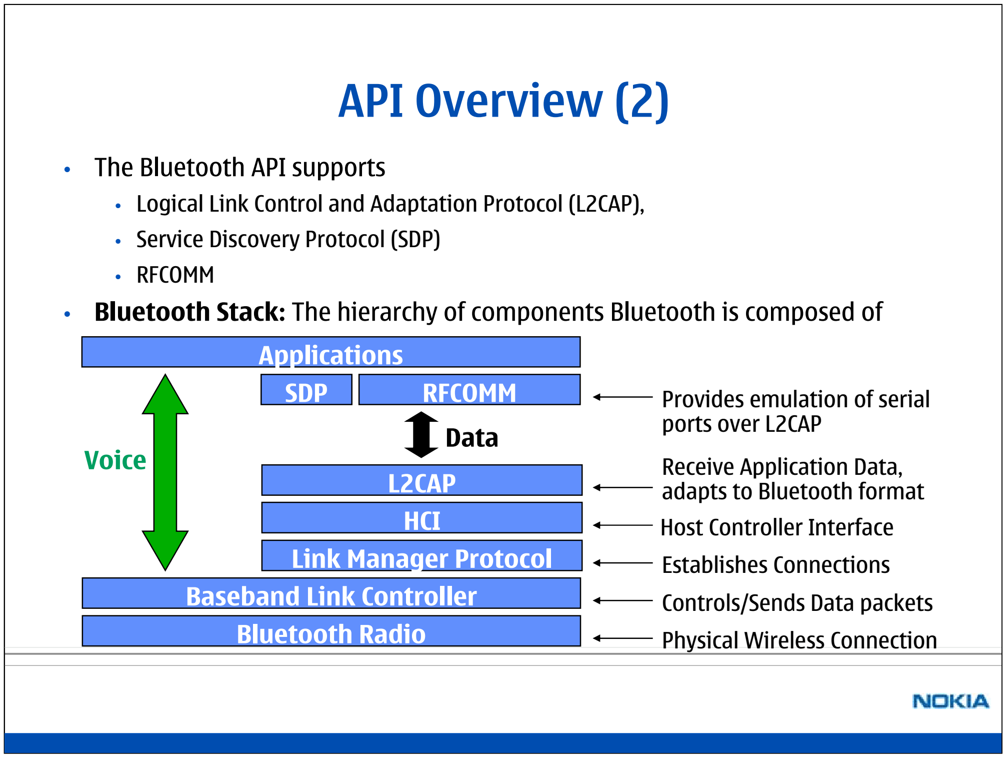
JSR-82 defines a standard set of APIs that will enable an open, third party application development environment for Bluetooth wireless technology. The API is targeted mainly at devices that are limited in processing power and memory, and are primarily battery-operated. Some important features:

- The specification provides basic support for Bluetooth protocols and profiles. It doesn't include specific APIs for all Bluetooth profiles simply because the number of profiles is growing.
- The specification incorporates the OBEX, L2CAP, and RFCOMM communication protocols in the JSR-82 APIs, primarily because all current Bluetooth profiles are designed to use these communication protocols.
- The JSR-82 specification addresses the Generic Access Profile, Service Discovery Application Profile, Serial Port Profile, and Generic Object Exchange Profile.
- The Service Discovery protocol is also supported. JSR-82 defines service registration in detail in order to standardize the registration process for the application programmer.
- JSR 82 standardizes the programming interface for Bluetooth. Its two optional packages can be used with any of the Java ME profiles. The minimum configuration is CLDC. Because CLDC is a subset of CDC, applications using these APIs should work on CDC devices. If the Generic Connection Framework Optional Package for J2SE (JSR 197) is implemented, the JSR 82 APIs should work smoothly with J2SE.

The Java APIs for Bluetooth define two packages that depend on the CLDC javax.microedition.io package:

- javax.bluetooth: core Bluetooth API
- javax.obex: APIs for the Object Exchange (OBEX) protocol

The OBEX APIs are defined independently of the Bluetooth transport layer and packaged separately because the OBEX protocol can be used over several different transmission media - wired, infrared, Bluetooth radio. Each of the above packages represents a separate optional package, which means that a CLDC implementation can include either package or both. MIDP-enabled devices are expected to be the kind of devices to incorporate this specification.



API Overview (2)

- The Bluetooth API supports
  - Logical Link Control and Adaptation Protocol (L2CAP),
  - Service Discovery Protocol (SDP)
  - RFCOMM
- **Bluetooth Stack:** The hierarchy of components Bluetooth is composed of

Applications

SDP | RFCOMM ← Provides emulation of serial ports over L2CAP

Data

Voice

L2CAP ← Receive Application Data, adapts to Bluetooth format

HCI ← Host Controller Interface

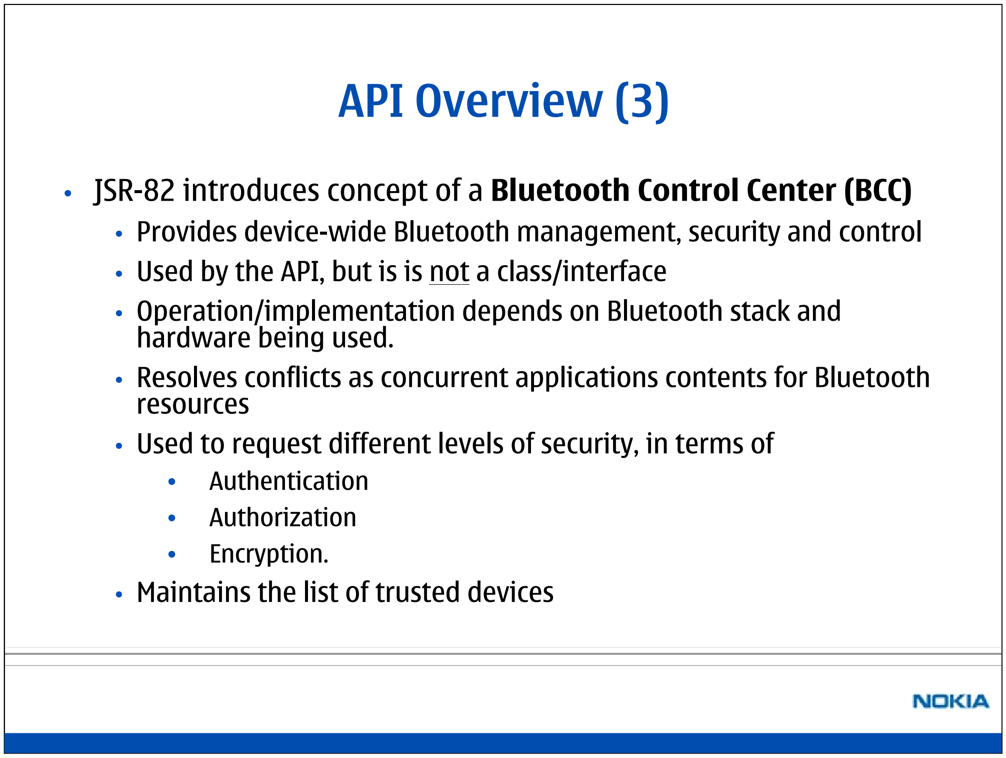
Link Manager Protocol ← Establishes Connections

Baseband Link Controller ← Controls/Sends Data packets

Bluetooth Radio ← Physical Wireless Connection

Like many other communications technologies, Bluetooth is composed of a hierarchy of components, referred to as stack. This stack is shown is the slide. JSR-82 requires that the Bluetooth stack underlying a JSR-82 implementation be qualified for the Generic Access Profile, the Service Discovery Application Profile, and the Serial Port Profile. The stack must also provide access to its Service Discovery Protocol, and to the RFCOMM and L2CAP layers. The Logical Link Control and Adaptation Layer Protocol (L2CAP) is layered over the Baseband Protocol and resides in the data link layer. L2CAP provides connection-oriented and connectionless data services to upper layer protocols with protocol multiplexing capability, segmentation and reassembly operation, and group abstractions. The RFCOMM protocol provides emulation of serial ports over the L2CAP protocol
The responsibilities of the layers in the Bluetooth stack are as follows:

- The radio layer is the physical wireless connection.
- The baseband layer is responsible for controlling and sending data packets over the radio link. It provides transmission channels for both data and voice. The baseband layer maintains Synchronous Connection-Oriented (SCO) links for voice and Asynchronous Connectionless (ACL) links for data
- The Link Manager Protocol (LMP) uses the links set up by the baseband to establish connections and manage piconets. Responsibilities of the LMP also include authentication and security services, and monitoring of service quality.
- The Host Controller Interface (HCI) is the dividing line between software and hardware. The L2CAP and layers above it are currently implemented in software, and the LMP and lower layers are in hardware. The HCI is the driver interface for the physical bus that connects these two components. The HCI may not be required. The L2CAP may be accessed directly by the application, or through certain support protocols provided to ease the burden on application programmers.
- The Logical Link Control and Adaptation Protocol (L2CAP) receives application data and adapts it to the Bluetooth format.



API Overview (3)

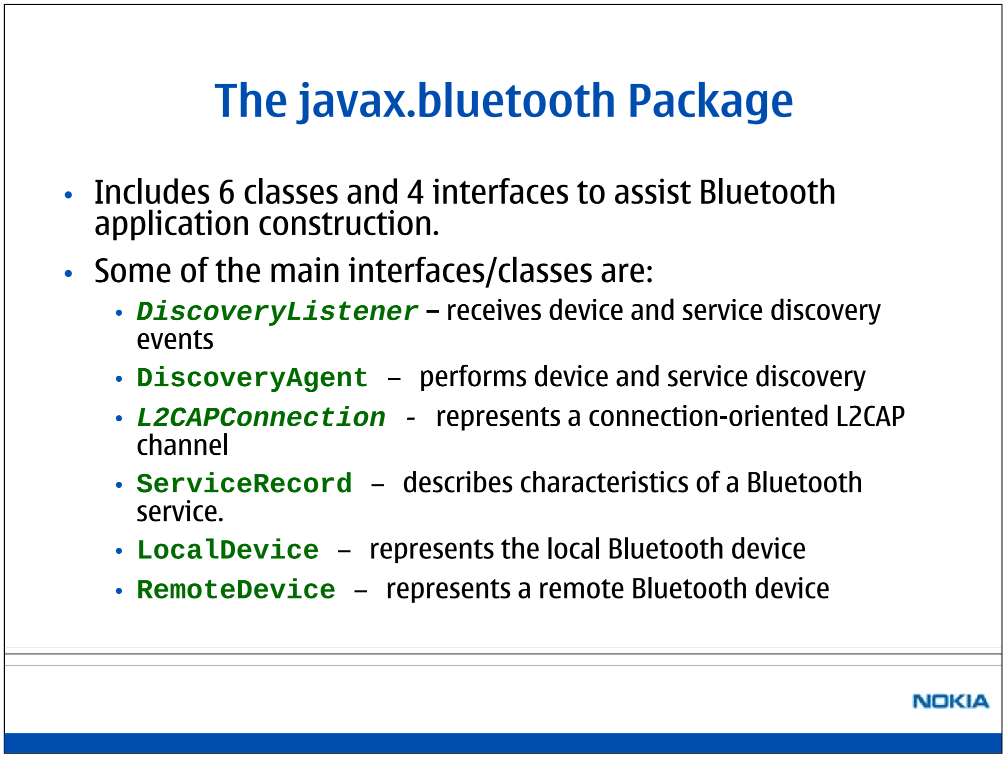
- JSR-82 introduces concept of a **Bluetooth Control Center (BCC)**
  - Provides device-wide Bluetooth management, security and control
  - Used by the API, but is is not a class/interface
  - Operation/implementation depends on Bluetooth stack and hardware being used.
  - Resolves conflicts as concurrent applications contents for Bluetooth resources
  - Used to request different levels of security, in terms of
    - Authentication
    - Authorization
    - Encryption.
  - Maintains the list of trusted devices

The JSR-82 specification introduced the concept of a Bluetooth control center (BCC). The BCC is the central authority in providing device-wide Bluetooth management, security and control; the Bluetooth API is dependent on its presence. The Java ME platform allows multiple Bluetooth applications to run concurrently. The BCC resolves any conflicts that arise as applications contend for Bluetooth resources, request different configurations of the Bluetooth system, or request different levels of security.

The policies enforced by the BCC vary for different types of devices. The Java API specification does not provide specification for the development of the BCC and leaves it to the vendors to provide a BCC in any way they want. The JSR-82 specification only lists the functions of the BCC and leaves the implementation details to the API implementers.

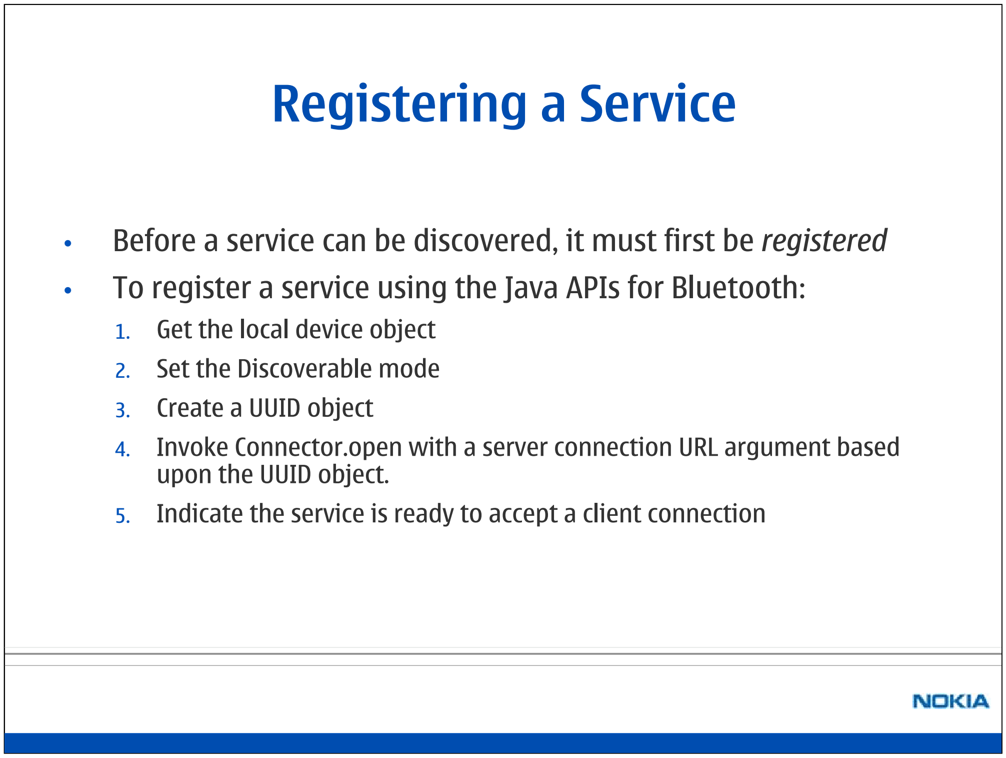
The following are the functionality of the BCC

- Provide a method to handle multiple applications to run and use the Bluetooth protocol stack
- Provide a means to manage and resolve potential conflicts between applications making similar or conflicting calls to the stack.
- Provide a means of security for authentication, authorization and encryption
- Provide a means to determine a list of devices maintained by the device to which it has previously connected and their security access levels
- Provide a means to detect new devices and establish a connection and perform security related tasks.



The javax.bluetooth Package

- Includes 6 classes and 4 interfaces to assist Bluetooth application construction.
- Some of the main interfaces/classes are:
    - *`DiscoveryListener`* – receives device and service discovery events
    - `DiscoveryAgent` – performs device and service discovery
    - *`L2CAPConnection`* - represents a connection-oriented L2CAP channel
    - `ServiceRecord` – describes characteristics of a Bluetooth service.
    - `LocalDevice` – represents the local Bluetooth device
    - `RemoteDevice` – represents a remote Bluetooth device



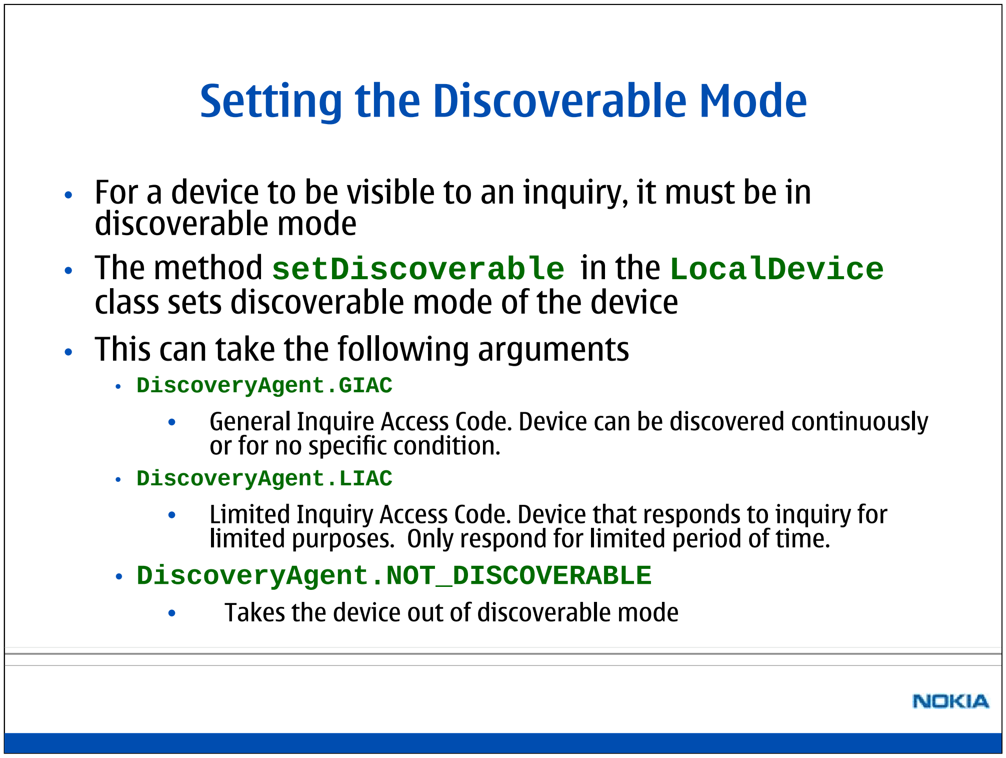
Registering a Service

- Before a service can be discovered, it must first be *registered*
- To register a service using the Java APIs for Bluetooth:
  1. Get the local device object
  2. Set the Discoverable mode
  3. Create a UUID object
  4. Invoke Connector.open with a server connection URL argument based upon the UUID object.
  5. Indicate the service is ready to accept a client connection

Service records are made visible to potential clients of a Bluetooth service through a process called **Service Registration.** A service is registered with a server.

The server is responsible for:

- Creating service record
- Adding service record to server's Service Discovery Database (SDDB)
- Registering security measures associated with connections to clients
- Accepting connections from clients that request service



Setting the Discoverable Mode

- For a device to be visible to an inquiry, it must be in discoverable mode
- The method `setDiscoverable` in the `LocalDevice` class sets discoverable mode of the device
- This can take the following arguments
  - `DiscoveryAgent.GIAC`
    - General Inquire Access Code. Device can be discovered continuously or for no specific condition.
  - `DiscoveryAgent.LIAC`
    - Limited Inquiry Access Code. Device that responds to inquiry for limited purposes. Only respond for limited period of time.
  - `DiscoveryAgent.NOT_DISCOVERABLE`
    - Takes the device out of discoverable mode

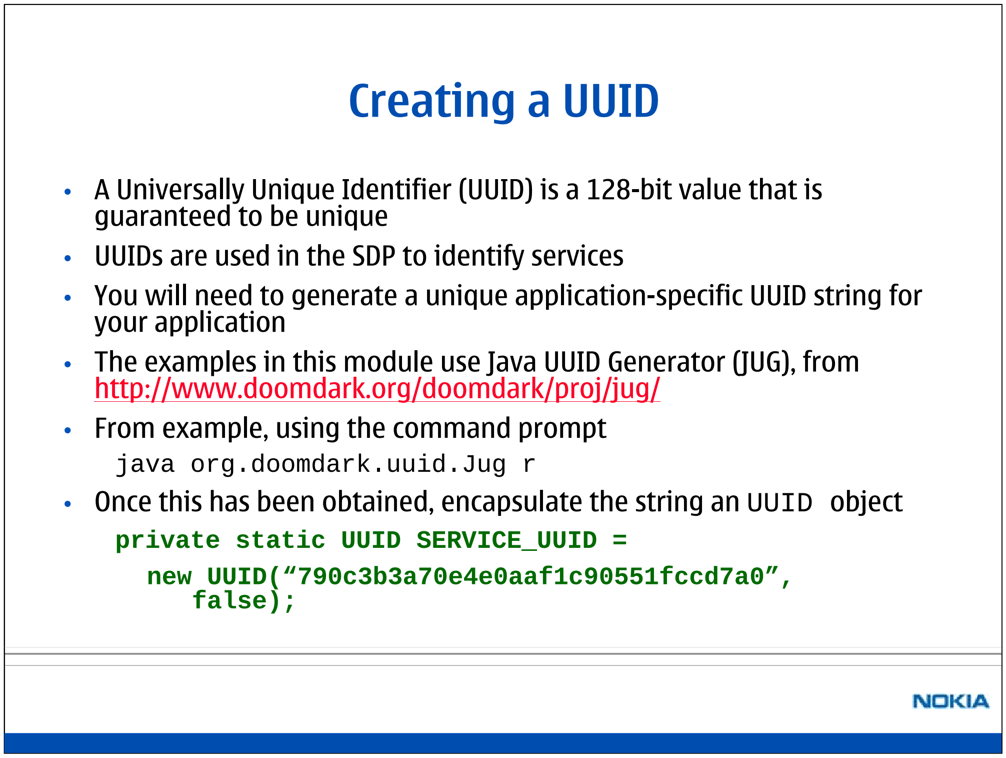
With respect to inquiry, a Bluetooth device is in either non-discoverable mode or in a discoverable mode. The two discoverable modes defined here are called limited discoverable mode and general discoverable mode.

Non-discoverable Mode: When a Bluetooth device is in non-discoverable mode, it will never enter the INQUIRY_RESPONSE state. Device is 'non-discoverable' or in 'non-discoverable mode'.

Limited discoverable Mode: Devices that need to be discoverable only for a limited period of time, during temporary conditions or for a specific event, should use the limited discoverable mode. The purpose is to respond to a device that makes a limited inquiry (inquiry using LIAC). Device is 'discoverable' or in 'discoverable mode'.

General discoverable Mode: devices that need to be discoverable continuously or for no specific condition should use the general discoverable mode. The purpose is to respond to a device that makes a general inquiry (inquiry using the GIAC). Device is 'discoverable' or in 'discoverable mode'.

In order to set the discoverable mode, JSR-82 defines the setDiscoverable() method in the LocalDevice class. The argument of the setDiscoverable() method is the discoverable mode that the local device should use for responding to inquiries.



Creating a UUID

- A Universally Unique Identifier (UUID) is a 128-bit value that is guaranteed to be unique
- UUIDs are used in the SDP to identify services
- You will need to generate a unique application-specific UUID string for your application
- The examples in this module use Java UUID Generator (JUG), from http://www.doomdark.org/doomdark/proj/jug/
- From example, using the command prompt

```
java org.doomdark.uuid.Jug r
```

- Once this has been obtained, encapsulate the string an `UUID` object

```
private static UUID SERVICE_UUID =
  new UUID(“790c3b3a70e4e0aaf1c90551fccd7a0”,
     false);
```

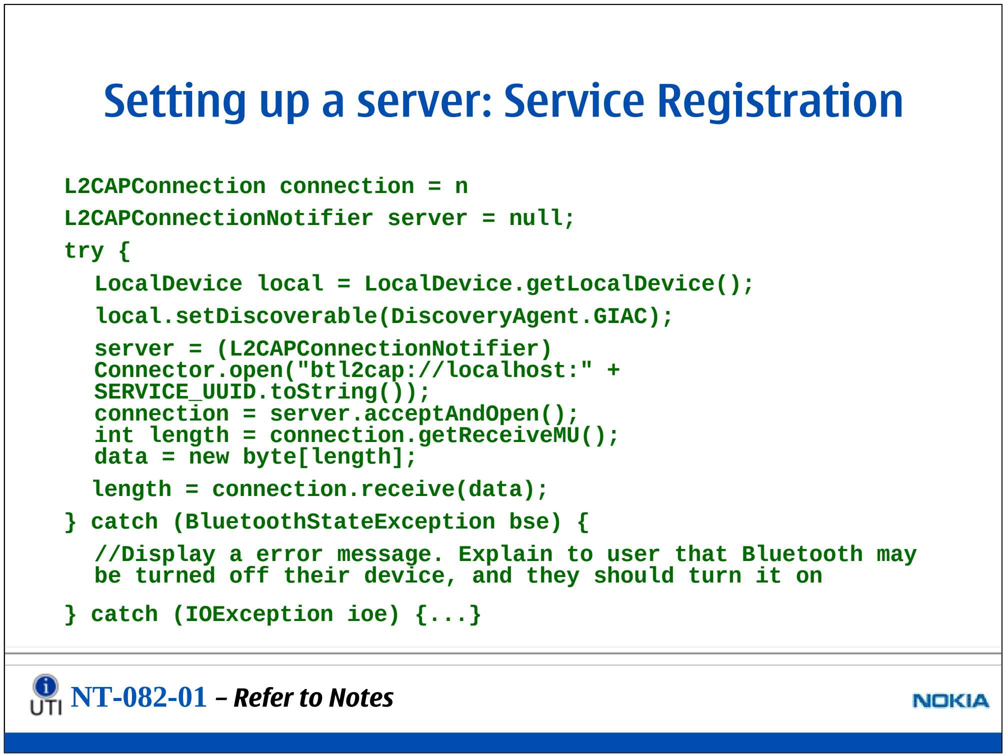
A Universally Unique Identifier (UUID) is a 128-bit value that is guaranteed to be unique across all space and time. UUIDs are used in the Bluetooth SDP to identify services. Some UUIDs are defined by the Bluetooth specification for specific protocols or uses.

You will need to generate unique application-specific UUIDs for your application(s).

A variety of different ways of generating UUIDs are possible. The application-specific UUIDs used in the example MIDlets presented in this module were generated using the “JUG” utility from http://www.doomdark.org/doomdark/proj/jug/

The class javax.Bluetooth.UUID is used to encapsulate the generated string within an object, for example

```
private static UUID SERVICE_UUID = new UUID(“790c3b3a-70e8-4e0a-af1c-
90551fccd7a0”, false);
```



Setting up a server: Service Registration

```
L2CAPConnection connection = n
L2CAPConnectionNotifier server = null;
try {
   LocalDevice local = LocalDevice.getLocalDevice();
   local.setDiscoverable(DiscoveryAgent.GIAC);
   server = (L2CAPConnectionNotifier)
   Connector.open("btl2cap://localhost:" +
   SERVICE_UUID.toString());
   connection = server.acceptAndOpen();
   int length = connection.getReceiveMU();
   data = new byte[length];
   length = connection.receive(data);
} catch (BluetoothStateException bse) {
   //Display a error message. Explain to user that Bluetooth may
   be turned off their device, and they should turn it on
} catch (IOException ioe) {...}
```

UTI **NT-082-01** – ***Refer to Notes***

The Bluetooth specification supports authentication of a remote device using a 128-bit key that is generated from a PIN code shared by both devices. If the PIN code on both devices doesn't match, the authentication fails. The ability to require authentication of Bluetooth devices is controlled primarily through optional parameters to connection URLs.

Servers that are exporting some service use these URLs to identify a connection endpoint and clients use them to specify the service they wish to connect to.

So, a server might include the following code, for example:

```
String serversConnURL =
" btl2cap://localhost:" + SERVICE_UUID.toString() + ";authenticate=true";
try {
   notifier = (L2CAPConnectionNotifier)Connector.open(serversConnURL);
   conn = (L2CAPConnection)notifier.acceptAndOpen();
}
catch (IOException e) {
   /* handle any IOexceptions */
}
```

The "authenticate=true" parameter in this code fragment tells the server to require any connecting devices to authenticate themselves prior to connection establishment. The server then goes on to wait for clients to connect to this service, by calling acceptAndOpen(). A client connects to a server by calling the method Connector.open() with a URL as a parameter -- this URL may also contain the optional authenticate parameter.

An application can request that communication over a connection is encrypted using the same technique as for authentication. That is, both the server and the client may specify "encrypt=true" as a parameter to the connection URL. **Note** that encryption depends on authentication -- a connection can only be encrypted if it is a connection to an authenticated device.

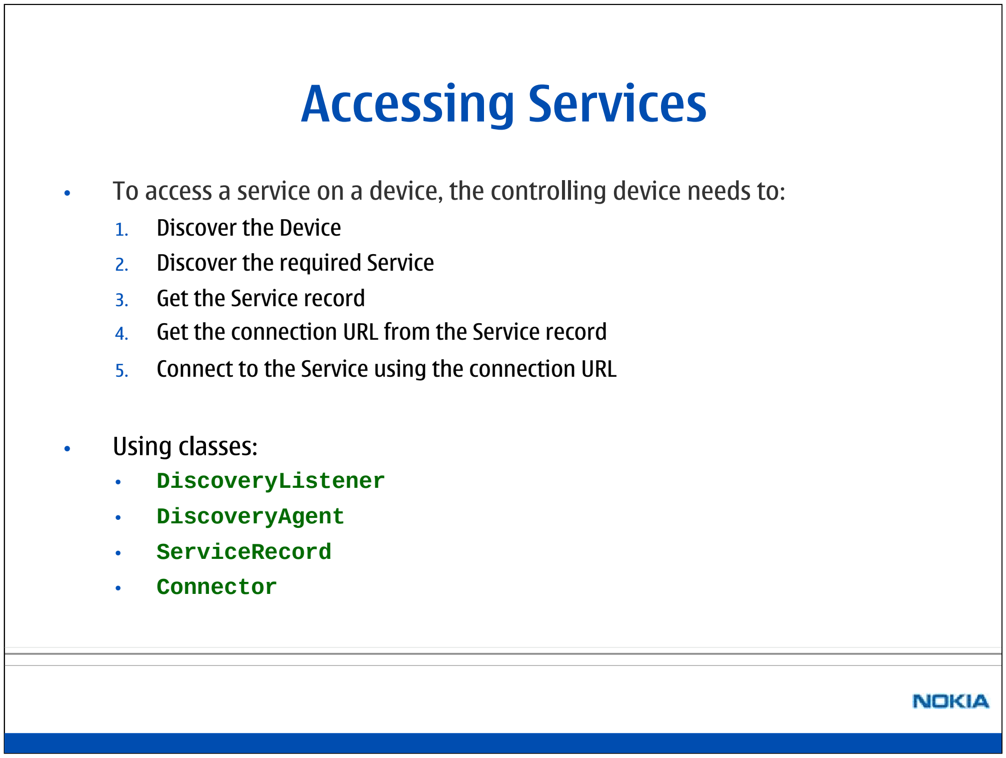
Authorisation is also controlled using the same technique; "authorize=true" as the relevant parameter. This parameter specifies that a remote device may only access a server's service if that device is authorised to access the service. The Bluetooth Control Centre (BCC) manages the list of trusted devices that are authorised to access a service. Authentication is a prerequisite of authorisation -- only authenticated devices will be authorised.

In addition to the URL parameter technique of controlling the security of a Bluetooth connection, the API supports a method-based alternative through methods of the RemoteDevice class, namely authenticate(), encrypt() and authorize().

The code in the slide uses the L2CAP communications layer. The only difference for RFCOMM or OBEX communications over Bluetooth wireless technology would be in the argument to the server's Connector.open() call. Instead of a connection string starting with "btl2cap" for the Logical Link Controller and Adaptation Protocol, the connections strings would begin with "btspp" or "btgoep." (Note: GOEP stands for generic object exchange profile).

The L2CAP API is the right choice for an application if the application implements a Bluetooth profile that uses the L2CAP protocol and that profile does not use RFCOMM or OBEX. Also L2CAP should be used if the application implements a new custom protocol that is packet oriented.

The code in the slide refers to a universally unique identifier (UUID). A UUID is a 128-bit sequence. The Bluetooth specification assigns meanings to many UUIDs. Other UUIDs are given user-defined meanings. SERVICE_UUID identifies the type of service being offered; that is, the service class



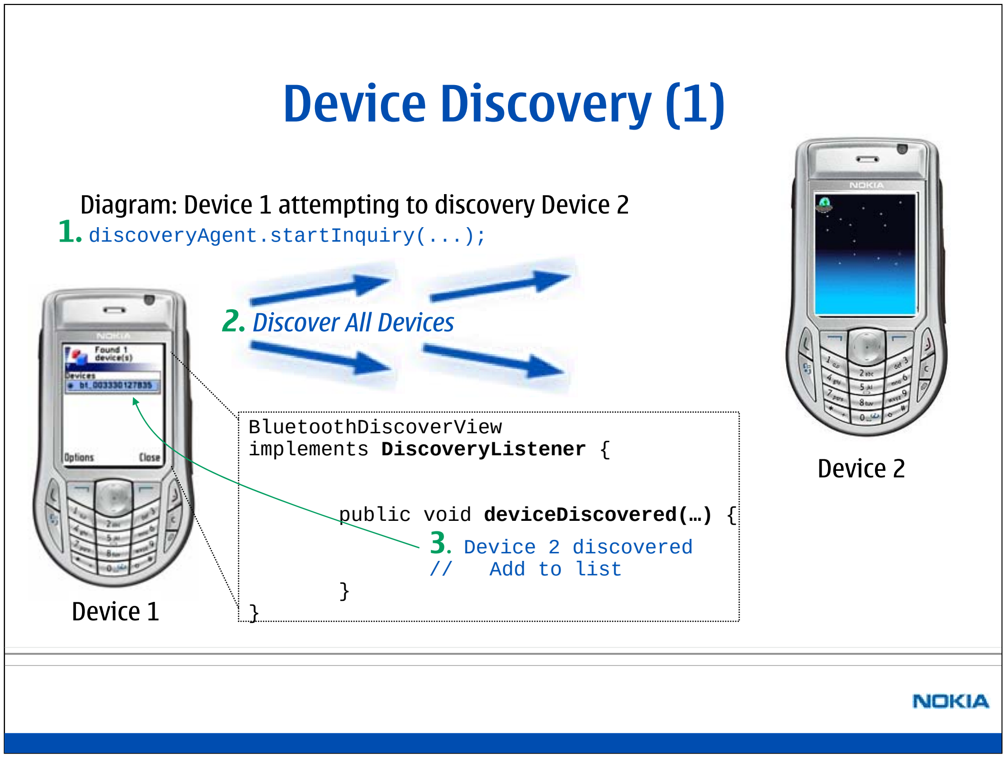
Accessing Services

- To access a service on a device, the controlling device needs to:
  1. Discover the Device
  2. Discover the required Service
  3. Get the Service record
  4. Get the connection URL from the Service record
  5. Connect to the Service using the connection URL

- Using classes:
  - `DiscoveryListener`
  - `DiscoveryAgent`
  - `ServiceRecord`
  - `Connector`

Accessing services involves using discovery; there are two aspects to discovery. First, we need to be able to discover devices that are in range of us, and then to discover the services supported by those devices. Bluetooth supports this functionality, as Device Inquiry and the Service Discovery Protocol (SDP) are core components of the Bluetooth specification.

The device that wishes to connect can analyze the services to find out whether it can use them. Once the required service has been found, a connection can be made.



Device Discovery (1)

Diagram: Device 1 attempting to discovery Device 2

1. `discoveryAgent.startInquiry(...);`

2. *Discover All Devices*

```
BluetoothDiscoverView
implements DiscoveryListener {

          public void deviceDiscovered(…) {
                  3. Device 2 discovered
                  //   Add to list
          }
}
```

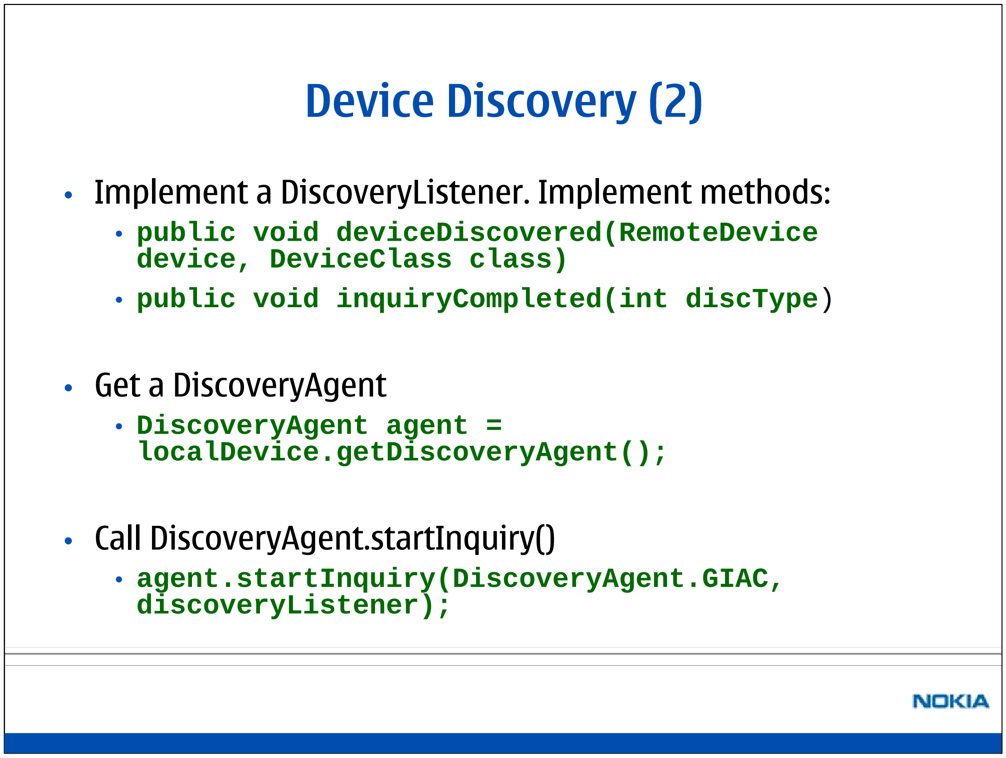
The DiscoveryAgent class is the main class for all discovery operations in JSR-82. The DiscoveryAgent has startInquiry() and retrieveDevices() methods. The startInquiry() method is used to start an inquiry.

The startInquiry() method takes two arguments, the type of inquiry to perform (limited or general) and a DiscoveryListener. An application must create a class that implements the DiscoveryListener interface. This class will define the deviceDiscovered() and inquiryCompleted() methods, which are used within device discovery.

If the device doesn't wish to wait for devices to be discovered, it can use the retrieveDevices() method to retrieve an existing list. The retrieveDevices() method does not actually perform an inquiry. Depending on the argument passed to it, the retrieveDevices() method either returns devices found during a previous inquiry or the set of devices that the local device commonly connects to. The retrieveDevices() method provides an application with a hint about at what devices may be in the area, but it does not guarantee that any of the devices can be connected to.

Here are three code snippets to demonstrate the various approaches to device discovery:

```
... // retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// place the device in inquiry mode
boolean complete = agent.startInquiry();
...
...
// retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// return an array of pre-known devices
RemoteDevice[] devices = agent.retrieveDevices(DiscoveryAgent.PREKNOWN);
...
...
// retrieve the discovery agent DiscoveryAgent agent =
local.getDiscoveryAgent();
// return an array of devices found in a previous inquiry
RemoteDevice[] devices = agent.retrieveDevices(DiscoveryAgent.CACHED);
```



Device Discovery (2)

- Implement a DiscoveryListener. Implement methods:
  - `public void deviceDiscovered(RemoteDevice device, DeviceClass class)`
  - `public void inquiryCompleted(int discType)`
- Get a DiscoveryAgent
  - `DiscoveryAgent agent = localDevice.getDiscoveryAgent();`
- Call DiscoveryAgent.startInquiry()
  - `agent.startInquiry(DiscoveryAgent.GIAC, discoveryListener);`

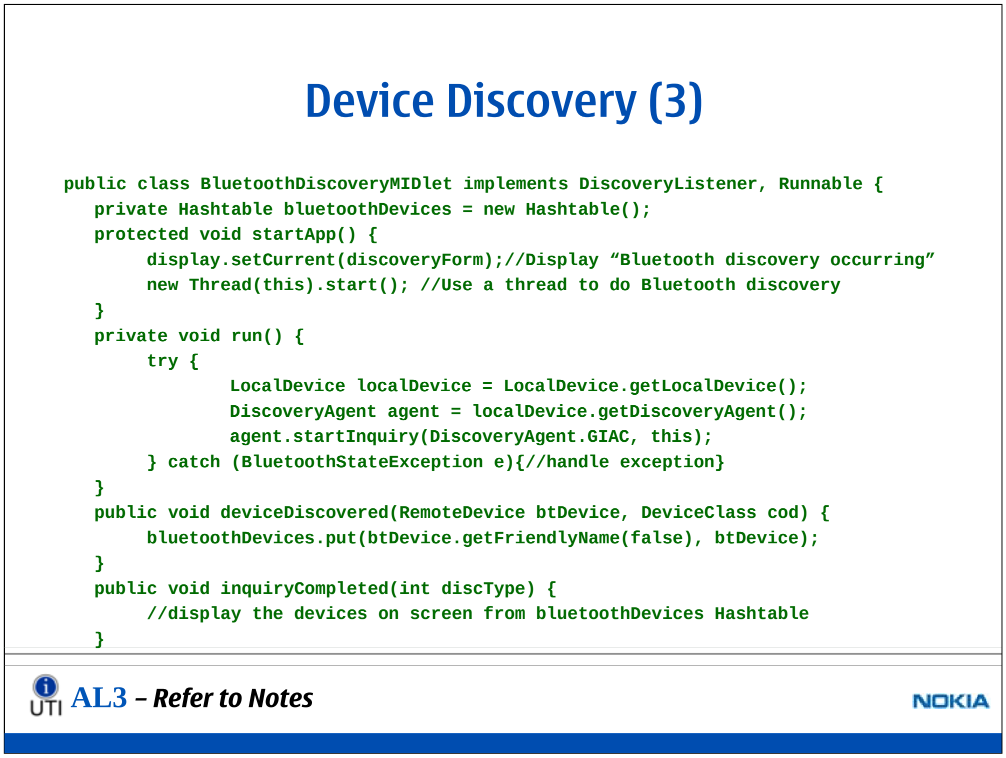
The method shown in the slide places the device into inquiry mode by calling the DiscoveryAgent.startInquiry() method. This method takes a parameter to specify the access code of the devices it should search for. This value could be DiscoveryAgent.GIAC or DiscoveryAgent.LIAC

To take advantage of this mode, the application must specify an event listener that will respond to inquiry-related events, i.e. DiscoveryListener.

The startInquiry() method is a non-blocking call. This means that the call to startInquiry() will return before the inquiry is actually completed. When the inquiry ends, the inquiryCompleted() method is called on the DiscoveryListener.

The deviceDiscovered() method is called by the JSR-82 implementation each time a remote Bluetooth device is found. The arguments passed are the RemoteDevice object that represents the remote Bluetooth device that was found along with its DeviceClass.

The RemoteDevice class provides methods that allow the application to gather additional information on the remote device like the device's Bluetooth address or user-friendly name. The DeviceClass provides additional information on the type of device and the types of services that are available on the device. This information can be used to determine if a service search should be performed on the remote device.



Device Discovery (3)

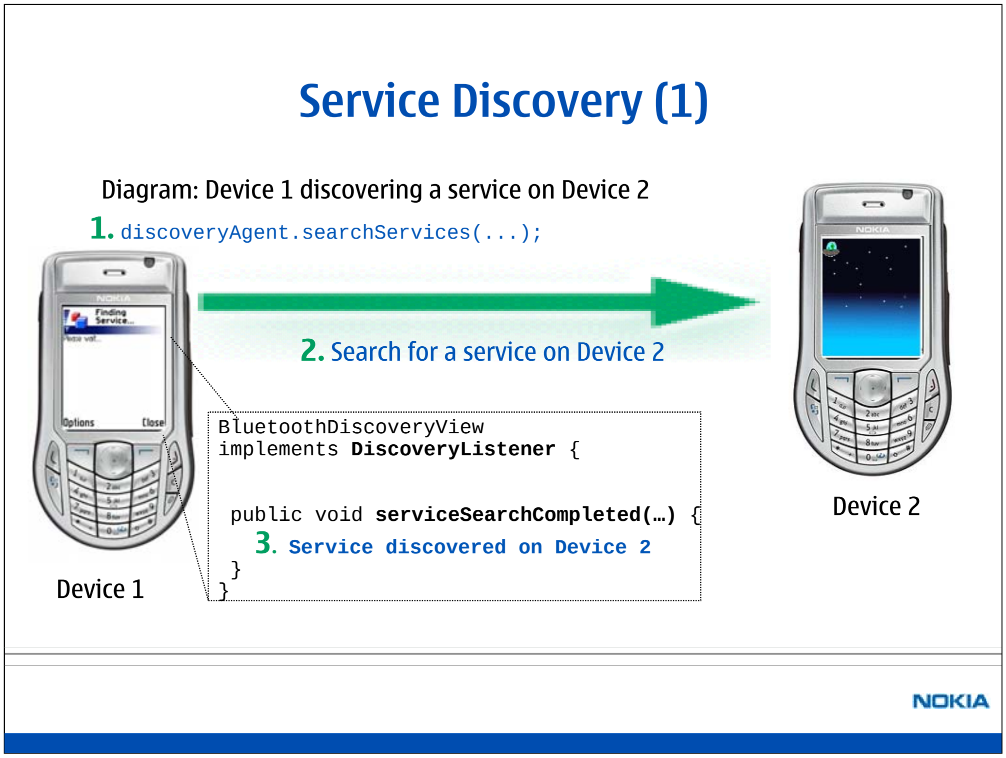
```
public class BluetoothDiscoveryMIDlet implements DiscoveryListener, Runnable {
   private Hashtable bluetoothDevices = new Hashtable();
   protected void startApp() {
        display.setCurrent(discoveryForm);//Display “Bluetooth discovery occurring”
        new Thread(this).start(); //Use a thread to do Bluetooth discovery
   }
   private void run() {
        try {
                LocalDevice localDevice = LocalDevice.getLocalDevice();
                DiscoveryAgent agent = localDevice.getDiscoveryAgent();
                agent.startInquiry(DiscoveryAgent.GIAC, this);
        } catch (BluetoothStateException e){//handle exception}
   }
   public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
        bluetoothDevices.put(btDevice.getFriendlyName(false), btDevice);
   }
   public void inquiryCompleted(int discType) {
        //display the devices on screen from bluetoothDevices Hashtable
   }
```

UTI **AL3** – ***Refer to Notes***

This slide shows a MIDlet performing Device Discovery within the MIDlets startApp method. To ensure the MIDlet does not take a long time to actually start up, a screen is first displayed to inform the user that device discovery in occurring, and then the methods to perform device discovery are contained within a separate thread. It can be seen that the startApp method creates a new thread and then starts it. The code to perform device discovery is contained within the run method.

In this slide, once the deviceDiscovered method is called, the device that is discovered is added to a hash table, indexed by devices friendly name. Once the inquiry is completed, the devices can be displayed on screen by taking each element from the hashtable. The following code shows how this could be done

```
private List deviceList = new List("Select Device", List.EXCLUSIVE);
...
private void copyDeviceTableToList() {
    // Remove old entries
       for (int i = deviceList.size(); i > 0; i--) {
      deviceList.delete(i - 1);
    }
    // Append keys from table
    for (Enumeration e = bluetoothDevices.keys(); e.hasMoreElements(); ) {
      try {
        String name = (String) e.nextElement();
        deviceList.append(name, null);
      }
      catch (Exception exception) {
      }
    }
  }
```



Service Discovery (1)

Diagram: Device 1 discovering a service on Device 2

1. `discoveryAgent.searchServices(...);`

2. Search for a service on Device 2

```
BluetoothDiscoveryView
implements DiscoveryListener {

 public void serviceSearchCompleted(…) {
   3. Service discovered on Device 2
 }
}
```

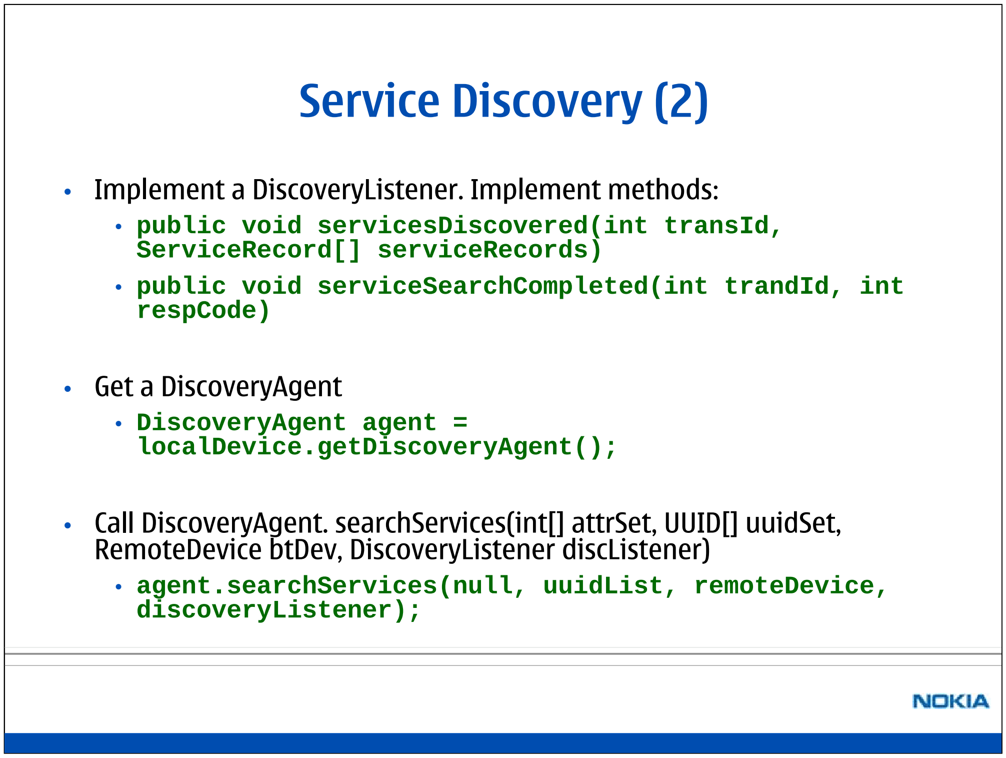
Once the local device has discovered at least one remote device, it can begin to search for available services on the devices. Because service discovery is much like device discovery, DiscoveryAgent also provides methods to discover services on a Bluetooth server device, and to initiate service-discovery transactions.

The DiscoveryAgent has the method searchServices(), this is called to use the service discovery protocol to find service records on a device.

The searchServices() method takes four arguments. The arguments are: the attribute value that should be retrieved, the set of UUIDs that are being searched for, the remote Bluetooth device to search for services on and a DiscoveryListener.

An application must create a class that implements the DiscoveryListener interface. This class will define the servicesDiscovered() and serviceSearchCompleted() methods, which are used within service discovery.

Note: the API provides mechanisms to search for services on remote devices, but not for services on the local device.



Service Discovery (2)

- Implement a DiscoveryListener. Implement methods:
  - `public void servicesDiscovered(int transId, ServiceRecord[] serviceRecords)`
  - `public void serviceSearchCompleted(int trandId, int respCode)`
- Get a DiscoveryAgent
  - `DiscoveryAgent agent = localDevice.getDiscoveryAgent();`
- Call DiscoveryAgent. searchServices(int[] attrSet, UUID[] uuidSet, RemoteDevice btDev, DiscoveryListener discListener)
  - `agent.searchServices(null, uuidList, remoteDevice, discoveryListener);`

NOKIA

The method shown in the slide searches for services on a remote Bluetooth device by calling the DiscoveryAgent.searchServices(int[] attrSet, UUID[] uuidSet, RemoteDevice btDev, DiscoveryListener discListener) method.

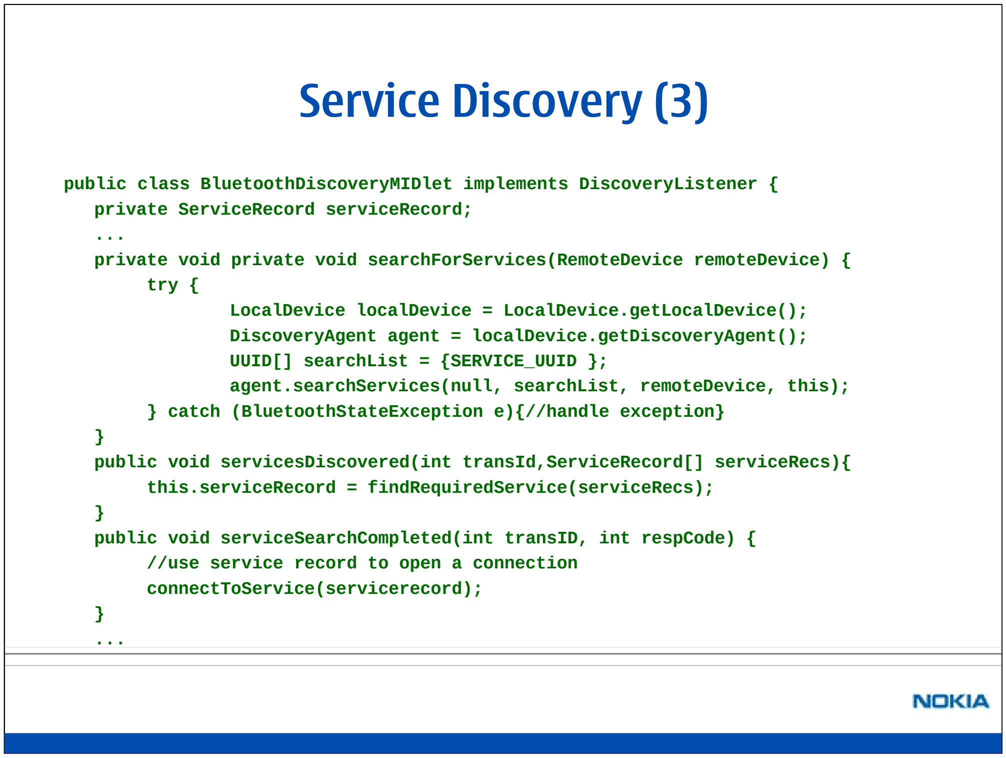
This method uses the service discovery protocol to find service records on a device that contain the UUIDs specified in the second argument, the UUID array.

The searchServices() method takes an argument that specifies the list of service attributes that should be retrieved from every service record that contains UUID.

As service records that contain UUID are returned from the remote device, the JSR-82 implementation sends them to the client's DiscoveryListener via the servicesDiscovered() method.
Client applications typically define the servicesDiscovered() method to check the service attributes of each service record to determine if this service record describes a service that this client can interact with.

Once an appropriate service record is found, the client has the option to terminate the service search by calling the cancelServiceSearch() method.

When the service search is complete, the JSR-82 implementation sends the serviceSearchCompleted() method to the client's DiscoveryListener. An argument to this method indicates how the service search ended; e.g., all matching service records have been retrieved, or the client terminated the search.



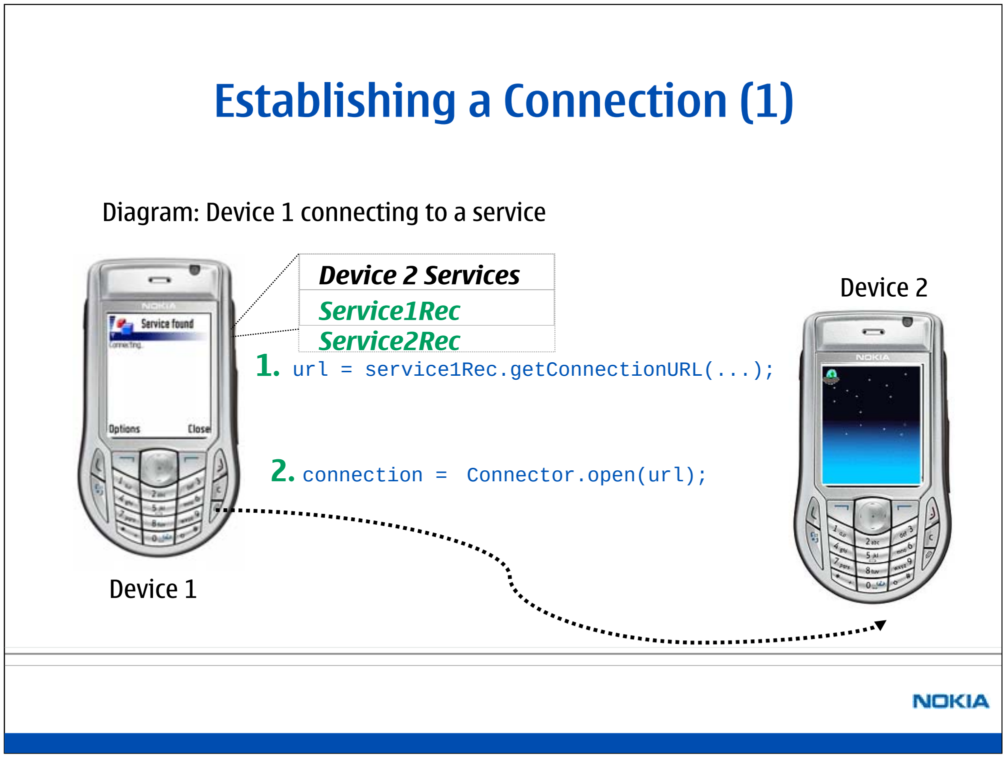
Service Discovery (3)

```
public class BluetoothDiscoveryMIDlet implements DiscoveryListener {
   private ServiceRecord serviceRecord;
   ...
   private void private void searchForServices(RemoteDevice remoteDevice) {
        try {
                LocalDevice localDevice = LocalDevice.getLocalDevice();
                DiscoveryAgent agent = localDevice.getDiscoveryAgent();
                UUID[] searchList = {SERVICE_UUID };
                agent.searchServices(null, searchList, remoteDevice, this);
        } catch (BluetoothStateException e){//handle exception}
   }
   public void servicesDiscovered(int transId,ServiceRecord[] serviceRecs){
        this.serviceRecord = findRequiredService(serviceRecs);
   }
   public void serviceSearchCompleted(int transID, int respCode) {
        //use service record to open a connection
        connectToService(servicerecord);
   }
   ...
```

In this slide, once the servicesDiscovered method is called, the ServiceRecord array passed to this method is filtered and the required ServiceRecord is stored in a class variable. The search is passed null to the attributes list and only passed one UUID in this example.

When the serviceSearchCompleted() method is called, we can use a ServiceRecord to connect to the required service.

At this point the client will typically move from a service-discovery phase into service-use phase. The following slides will describe how to connect to a service, then go on to use it.



Establishing a Connection (1)

Diagram: Device 1 connecting to a service

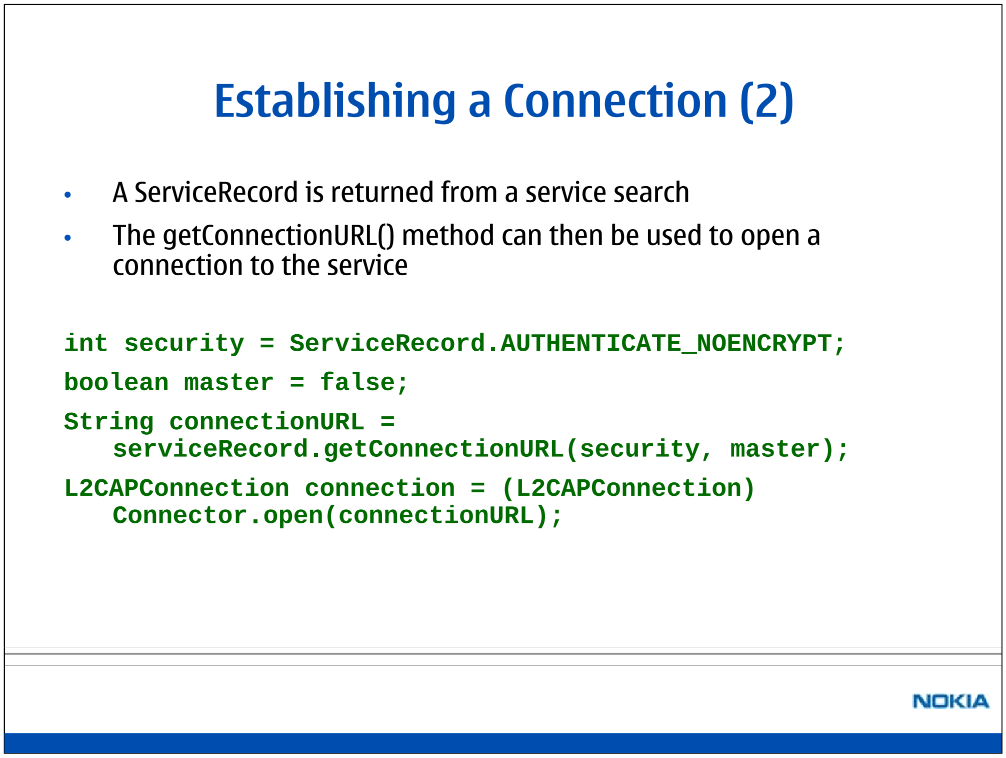
1. `url = service1Rec.getConnectionURL(...);`

2. `connection =  Connector.open(url);`

Once the client has discovered the services the remote device has, each service record is checked to determine if this service record describes a service that this client can interact with.

Once an appropriate service record is found, the client can use the method getConnectionURL() to determine the connection string that should be used as an argument to Connector.open() to connect to this service.

For a local device to use a service on a remote device, the two devices must share a common communications protocol. So that applications can access a wide variety of Bluetooth services, the Java APIs for Bluetooth provide mechanisms that allow connections to any service that uses RFCOMM, L2CAP, or OBEX as its protocol. If a service uses another protocol (such as TCP/IP) layered above one of these protocols, the application can access the service, but only if it implements the additional protocol in the application, using the CLDC Generic Connection Framework.



Establishing a Connection (2)

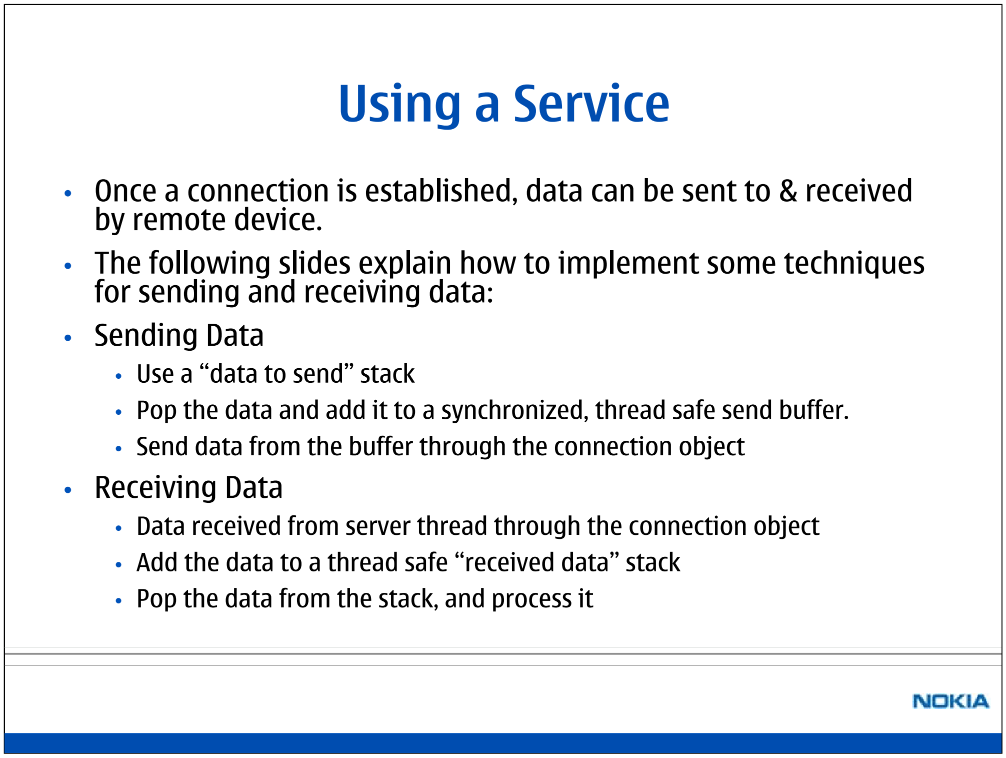
- A ServiceRecord is returned from a service search
- The getConnectionURL() method can then be used to open a connection to the service

```
int security = ServiceRecord.AUTHENTICATE_NOENCRYPT;
boolean master = false;
String connectionURL =
   serviceRecord.getConnectionURL(security, master);
L2CAPConnection connection = (L2CAPConnection)
   Connector.open(connectionURL);
```

The getConnectionURL(int requiredSecurity, boolean mustBeMaster) method takes two arguments. The first argument determines whether authentication or encryption is required for the connection. The second argument indicates whether this device will be the master in connections to this service.

The values that the first argument can take are NOAUTHENTICATE_NOENCRYPT, AUTHENTICATE_NOENCRYPT and AUTHENTICATE_ENCRYPT. These determine whether the connection should verifying the identity of a remote device and apply encryption to all data transfers in both directions.

The code in the slide uses the L2CAP communications layer. The only difference for RFCOMM or OBEX communications over Bluetooth wireless technology would be the Connection class being returned. Instead of an L2CAPConnection for Logical Link Controller and Adaptation Protocol, the returned object would be StreamConnection or ClientSession.



Using a Service

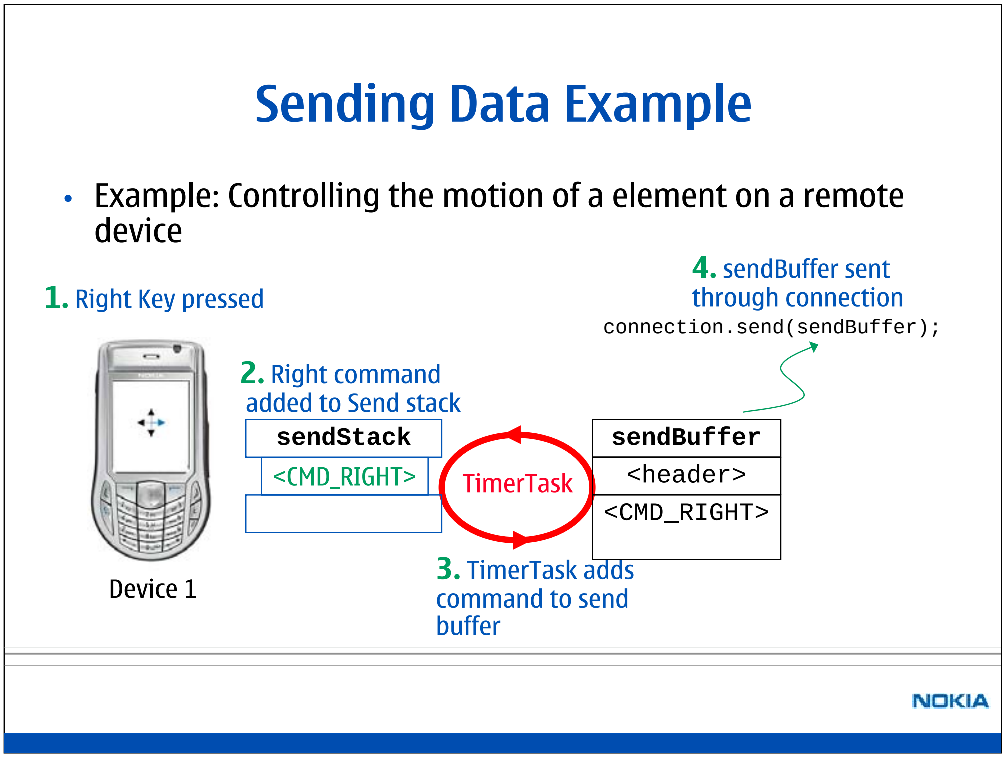
- Once a connection is established, data can be sent to & received by remote device.
- The following slides explain how to implement some techniques for sending and receiving data:
- Sending Data
  - Use a “data to send” stack
  - Pop the data and add it to a synchronized, thread safe send buffer.
  - Send data from the buffer through the connection object
- Receiving Data
  - Data received from server thread through the connection object
  - Add the data to a thread safe “received data” stack
  - Pop the data from the stack, and process it

NOKIA

Once a service has been discovered on the device, and a connection has been made, the service can be used by sending data through the connection to the remote device. The remote device must then receive this data, and process it accordingly. We will now look at some techniques that can be used to send data to reduce the chance of data being lost and receive data in a thread safe environment. An example is provided to illustrate the sending of commands to a remote device. The remote device will then process the received command and display the result.

The use of stacks is suggested to ensure that commands are sent to the remote device in the order they were issued and commands are processed in the order they were received. A “First In First Out” stack is implemented in the example, in that, if COMMAND1 is added to the stack, then COMMAND2, the first command will be processes, then the second.

It is also important to ensure that buffers to send and process data are thread-safe. By using thread-safe routines, the risk that one thread will interfere and modify data elements of another thread is eliminated by avoiding potential data race situations with coordinated access to shared data.



Sending Data Example

- Example: Controlling the motion of a element on a remote device

1. Right Key pressed

2. Right command added to Send stack

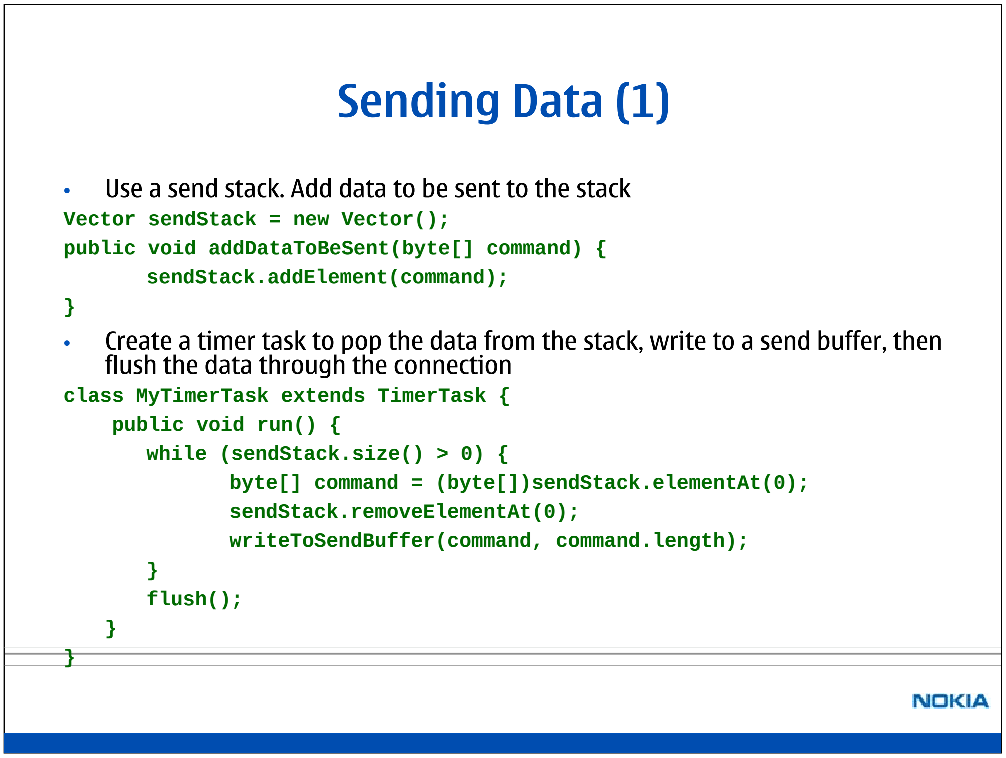
3. TimerTask adds command to send buffer

4. sendBuffer sent through connection

```
connection.send(sendBuffer);
```

The slide shows an example of a device sending a command to another device. In this case, Device 1 has requested a move to the right by issuing the command CMD_RIGHT.

When the user presses the right arrow key, the command CMD_RIGHT is added to the stack of data to send. A TimerTask is constantly checking the send stack for data. The TimerTask pops any data from the top of the stack and adds this to a thread-safe send buffer. Headers are then added to the send buffer, and this is sent through the connection using connection.send(sendBuffer);



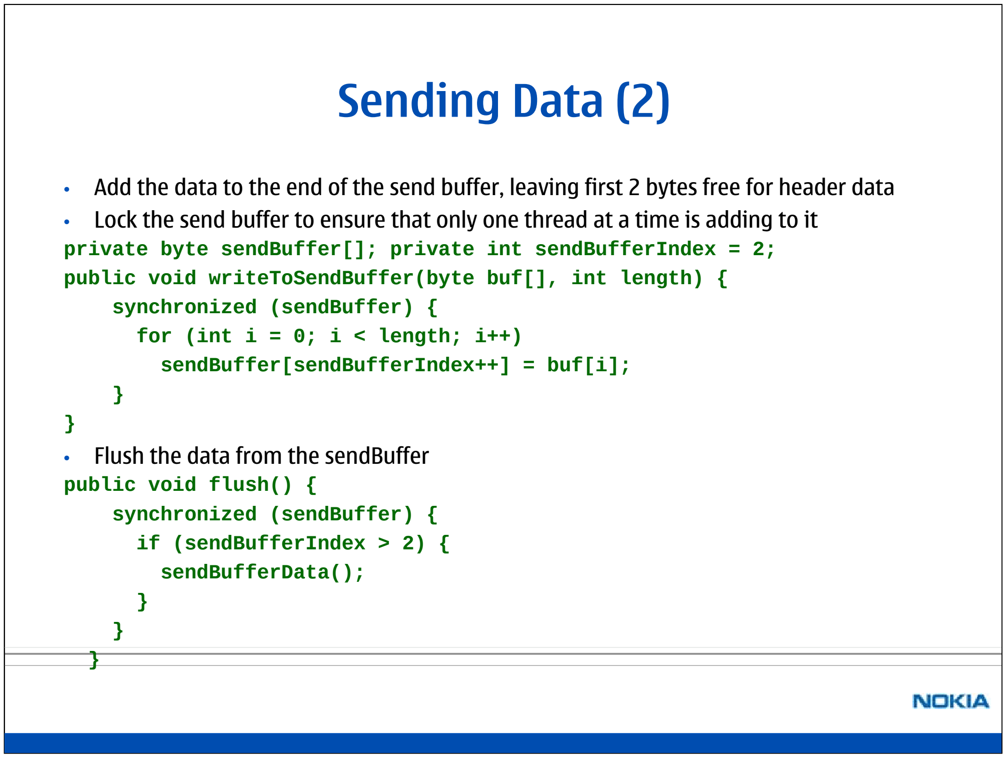
Sending Data (1)

- Use a send stack. Add data to be sent to the stack

```
Vector sendStack = new Vector();
public void addDataToBeSent(byte[] command) {
       sendStack.addElement(command);
}
```

- Create a timer task to pop the data from the stack, write to a send buffer, then flush the data through the connection

```
class MyTimerTask extends TimerTask {
    public void run() {
       while (sendStack.size() > 0) {
              byte[] command = (byte[])sendStack.elementAt(0);
              sendStack.removeElementAt(0);
              writeToSendBuffer(command, command.length);
       }
       flush();
    }
}
```



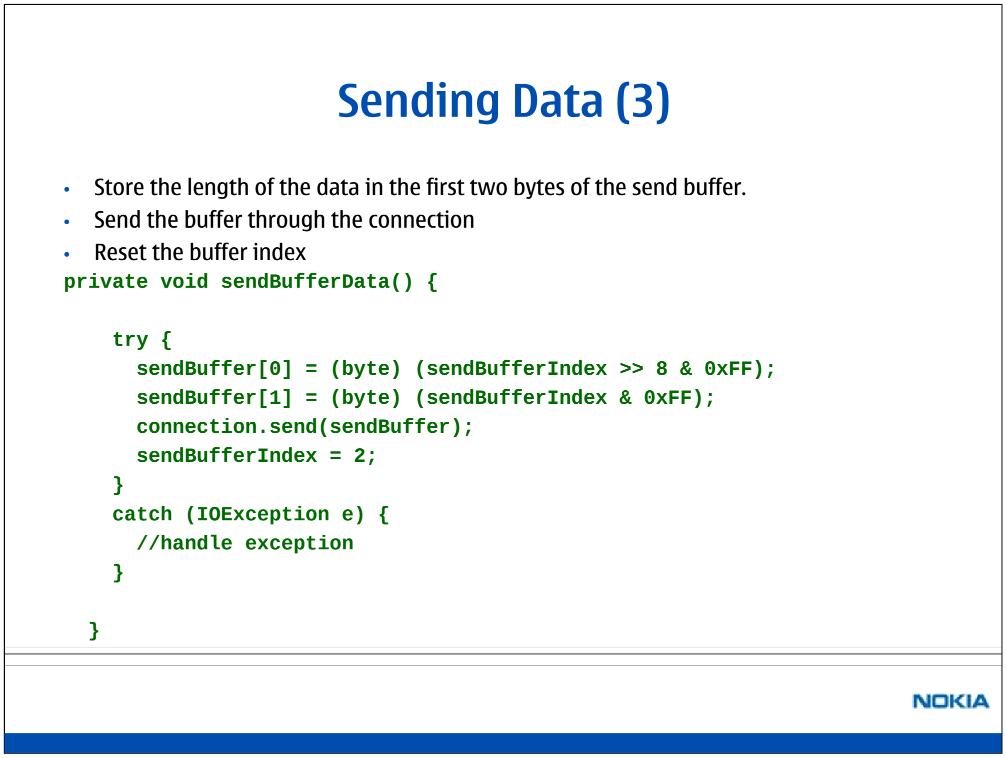
Sending Data (2)

- Add the data to the end of the send buffer, leaving first 2 bytes free for header data
- Lock the send buffer to ensure that only one thread at a time is adding to it

```
private byte sendBuffer[]; private int sendBufferIndex = 2;
public void writeToSendBuffer(byte buf[], int length) {
    synchronized (sendBuffer) {
      for (int i = 0; i < length; i++)
        sendBuffer[sendBufferIndex++] = buf[i];
    }
}
```

- Flush the data from the sendBuffer

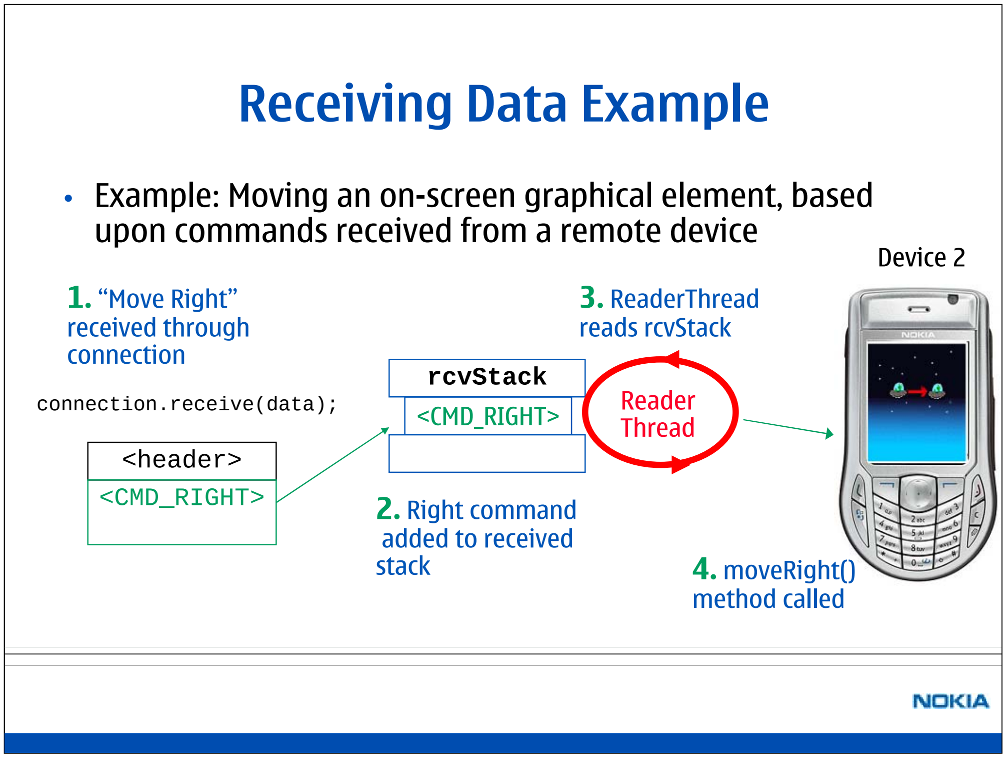
```
public void flush() {
    synchronized (sendBuffer) {
      if (sendBufferIndex > 2) {
        sendBufferData();
      }
    }
  }
```



Sending Data (3)

- Store the length of the data in the first two bytes of the send buffer.
- Send the buffer through the connection
- Reset the buffer index

```
private void sendBufferData() {

    try {
      sendBuffer[0] = (byte) (sendBufferIndex >> 8 & 0xFF);
      sendBuffer[1] = (byte) (sendBufferIndex & 0xFF);
      connection.send(sendBuffer);
      sendBufferIndex = 2;
    }
    catch (IOException e) {
      //handle exception
    }

  }
```



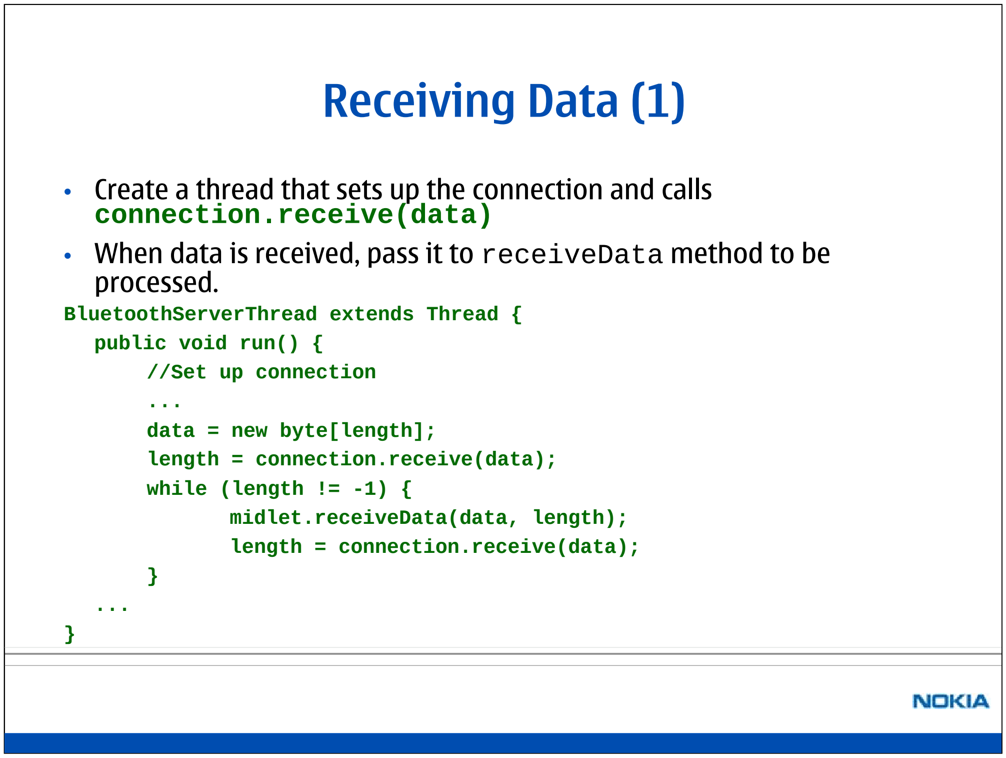
Receiving Data Example

- Example: Moving an on-screen graphical element, based upon commands received from a remote device

Device 2
1. "Move Right" received through connection
connection.receive(data);
<header>
<CMD_RIGHT>
rcvStack
<CMD_RIGHT>
2. Right command added to received stack
3. ReaderThread reads rcvStack
Reader Thread
4. moveRight() method called
NOKIA

The slide shows an example of a device receiving a command from another device. In this case, Device 2 receives the command CMD_RIGHT and displays the result on screen.

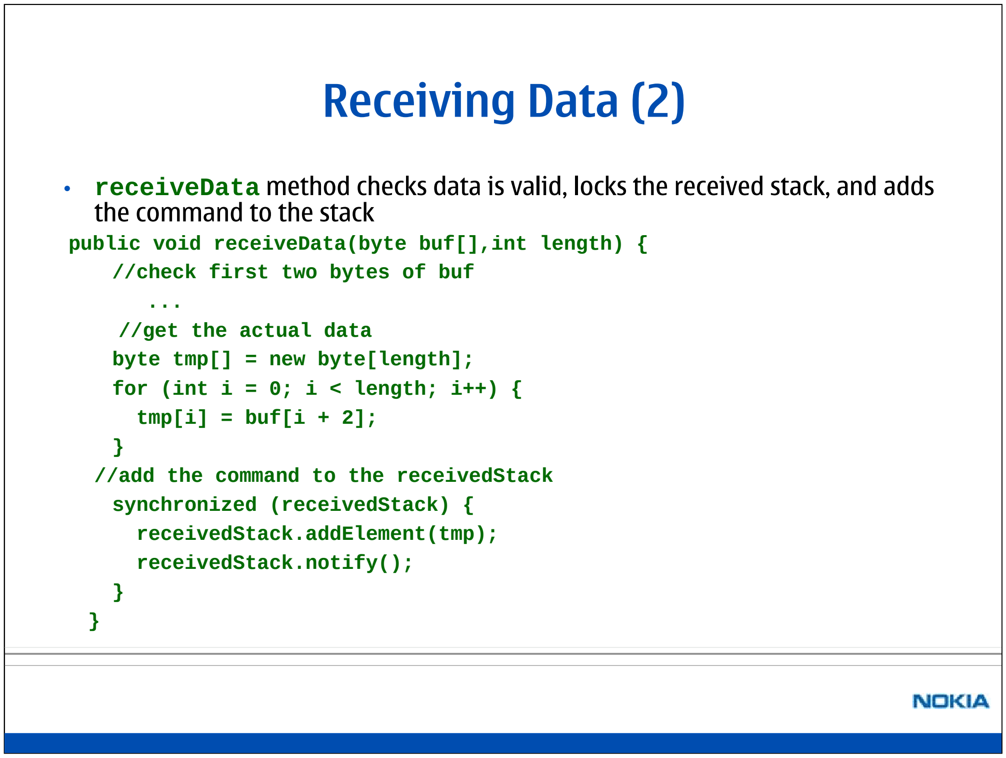
A server thread is created to establish the initial connection, and constantly check for received data by calling connection.receive(data). When data is received through the connection, this is passed onto a method that can verify the data has been received correctly by checking the header. Once this has been confirmed, the received command can be added to a thread-safe received data stack. A ReaderThread is constantly checking the received stack for data. The ReaderThread pops any data from the top of the stack, and passes it on to be processed. In this case, a CMD_RIGHT command was received, so the method moveRight() has been called on the canvas.



Receiving Data (1)

- Create a thread that sets up the connection and calls **`connection.receive(data)`**
- When data is received, pass it to `receiveData` method to be processed.

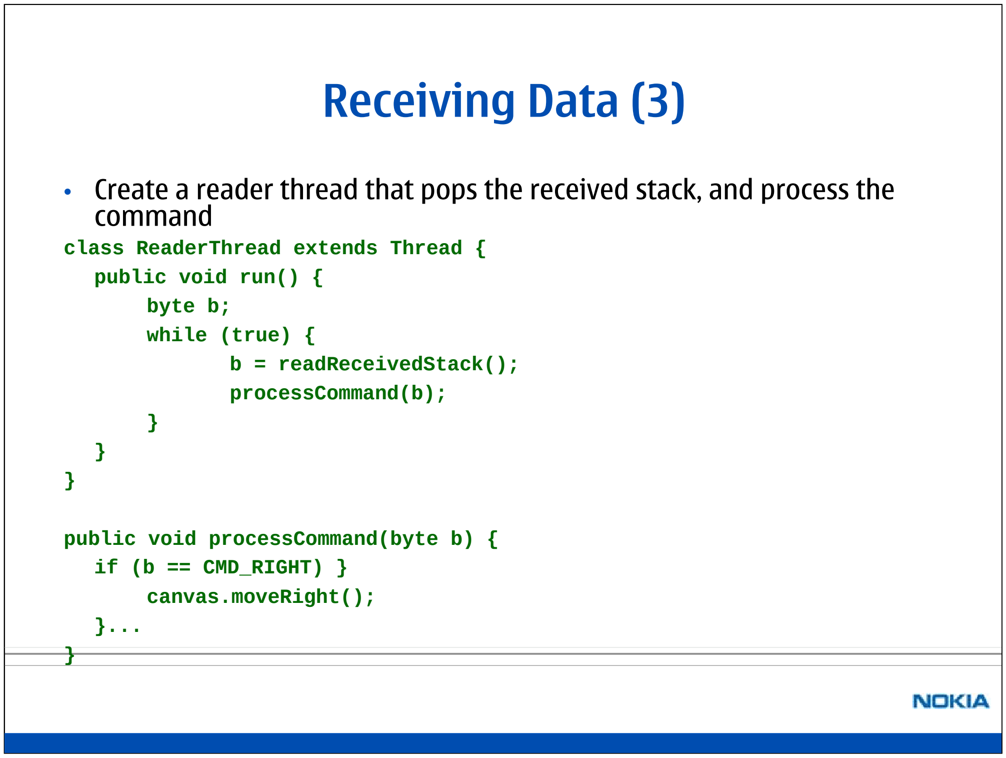
```
BluetoothServerThread extends Thread {
   public void run() {
        //Set up connection
        ...
        data = new byte[length];
        length = connection.receive(data);
        while (length != -1) {
               midlet.receiveData(data, length);
               length = connection.receive(data);
        }
   ...
}
```



Receiving Data (2)

- `receiveData` method checks data is valid, locks the received stack, and adds the command to the stack

```
public void receiveData(byte buf[],int length) {
    //check first two bytes of buf
       ...
    //get the actual data
    byte tmp[] = new byte[length];
    for (int i = 0; i < length; i++) {
      tmp[i] = buf[i + 2];
    }
  //add the command to the receivedStack
    synchronized (receivedStack) {
      receivedStack.addElement(tmp);
      receivedStack.notify();
    }
  }
```



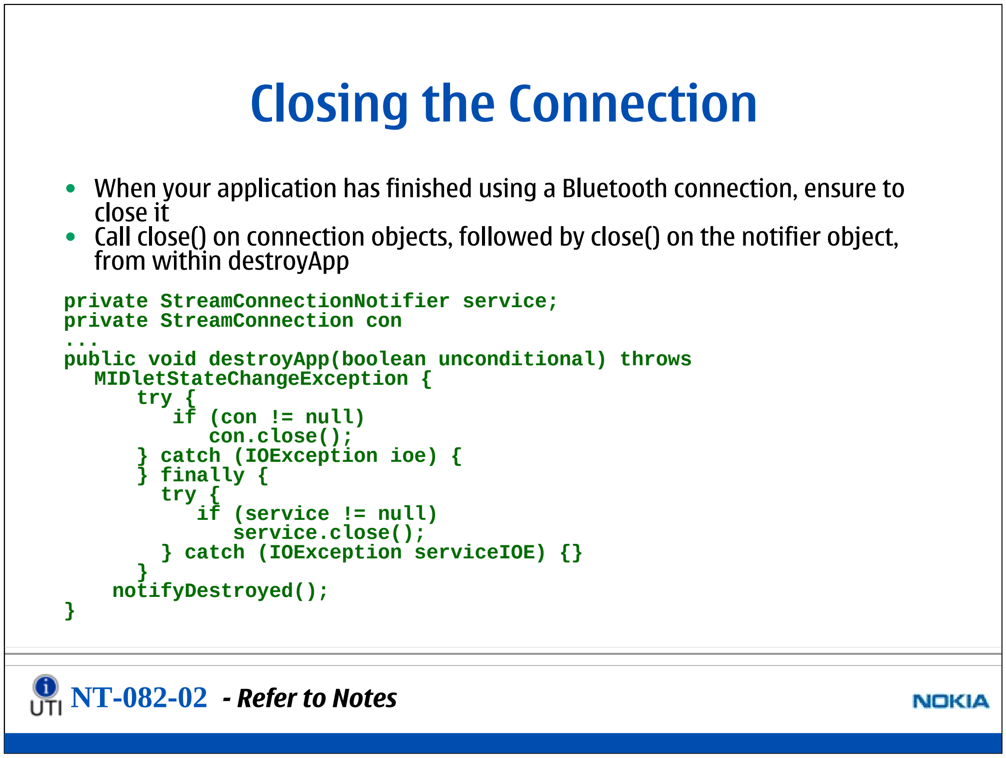
Receiving Data (3)

- Create a reader thread that pops the received stack, and process the command

```
class ReaderThread extends Thread {
  public void run() {
      byte b;
      while (true) {
            b = readReceivedStack();
            processCommand(b);
      }
  }
}

public void processCommand(byte b) {
  if (b == CMD_RIGHT) }
      canvas.moveRight();
  }...
}
```

The code in the slide checks the received stack and the processes the received data. The method below pops data off the receivedStack and returns the data be processed.

```
private byte readReceivedStack();
{
    byte rc;
    while (true) {
      if (receivedBuffer != null) {
        rc = receivedBuffer[receivedBufferIndex++];
        if (receivedBufferIndex >= receivedBuffer.length) {
          receivedBuffer = null;
        }
        return rc;
      }
      synchronized (receivedStack) {
        // Process drawing messages received from peer
        while (receivedStack.size() == 0) {
          try {
            receivedStack.wait();
          }
          catch (InterruptedException e) {
          }
        }
        receivedBuffer = (byte[]) receivedStack.elementAt(0);
        receivedStack.removeAllElements();
        receivedBufferIndex = 0;
      }
    }
  }
```



Closing the Connection

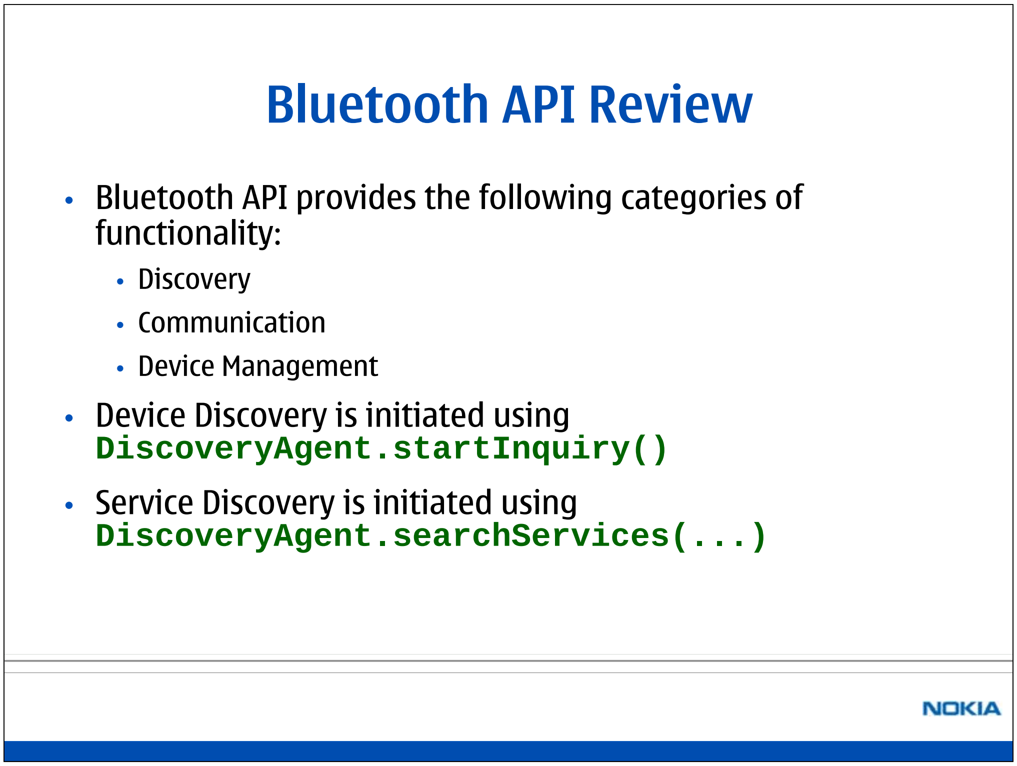
- When your application has finished using a Bluetooth connection, ensure to close it
- Call close() on connection objects, followed by close() on the notifier object, from within destroyApp

```
private StreamConnectionNotifier service;
private StreamConnection con
...
public void destroyApp(boolean unconditional) throws
   MIDletStateChangeException {
      try {
         if (con != null)
            con.close();
      } catch (IOException ioe) {
      } finally {
        try {
           if (service != null)
              service.close();
        } catch (IOException serviceIOE) {}
      }
    notifyDestroyed();
}
```

UTI NT-082-02 - ***Refer to Notes***

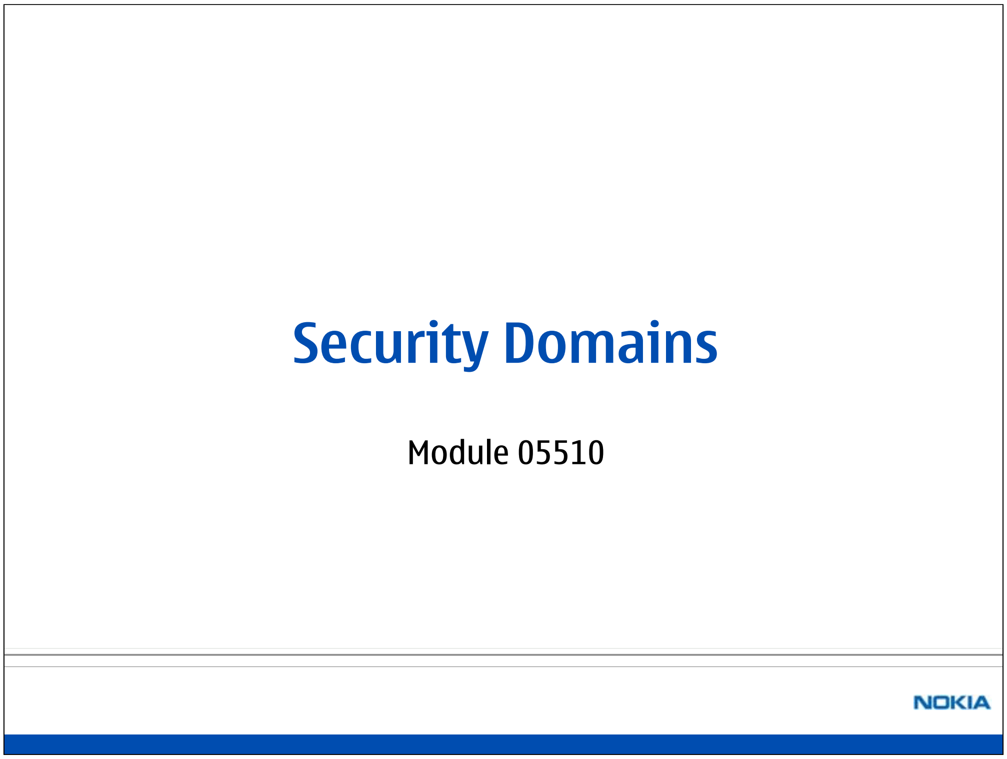
NOKIA

To ensure a Bluetooth connection is closed when the application closes, you must call the close() method on any connection objects you have been using, followed by close() on any associated notifier objects from within the destroyApp method.



Bluetooth API Review

- Bluetooth API provides the following categories of functionality:
  - Discovery
  - Communication
  - Device Management
- Device Discovery is initiated using `DiscoveryAgent.startInquiry()`
- Service Discovery is initiated using `DiscoveryAgent.searchServices(...)`



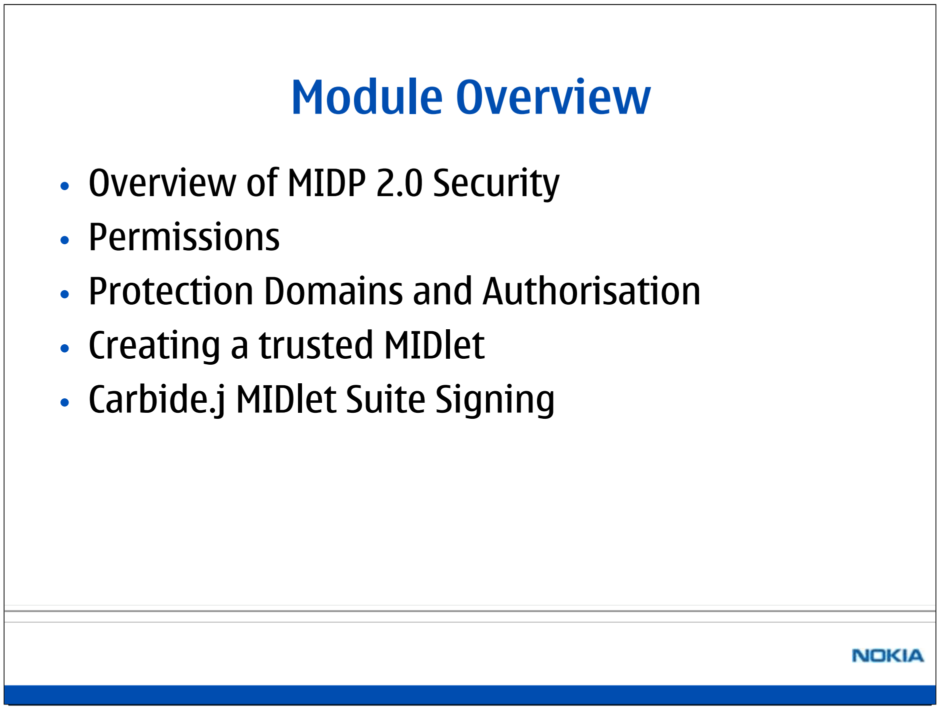
Security Domains

Module 05510

Running a downloaded MIDlet, although safer than binary code (due to it running in a virtual machine), could result in security risks if the source of the code is unknown. For example, the MIDlet could try to collect information about the device and its user and send it to a server by making unauthorized network connections.

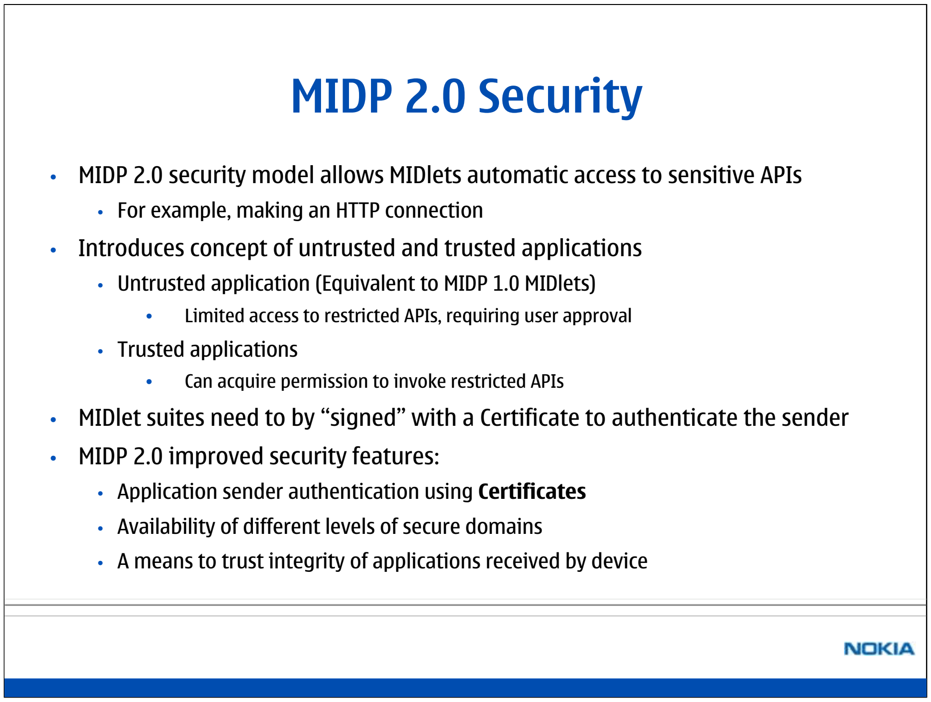
Protection domains can be configured to make it safer for users to run downloaded MIDlets. The MIDP 2.0 specification suggests a protection domain based on cryptographic signatures and certificates, where the software developer creates a signing-key pair and acquires a certificate from a recognized certificate authority. The developer generates a signature of the MIDlet suite JAR, then places that signature and the related certificate in the application's descriptor file.

This module will describe new concepts introduced in MIDP 2.0 relating to security, and explain how implement these concepts using Carbide.j.



Module Overview

- Overview of MIDP 2.0 Security
- Permissions
- Protection Domains and Authorisation
- Creating a trusted MIDlet
- Carbide.j MIDlet Suite Signing

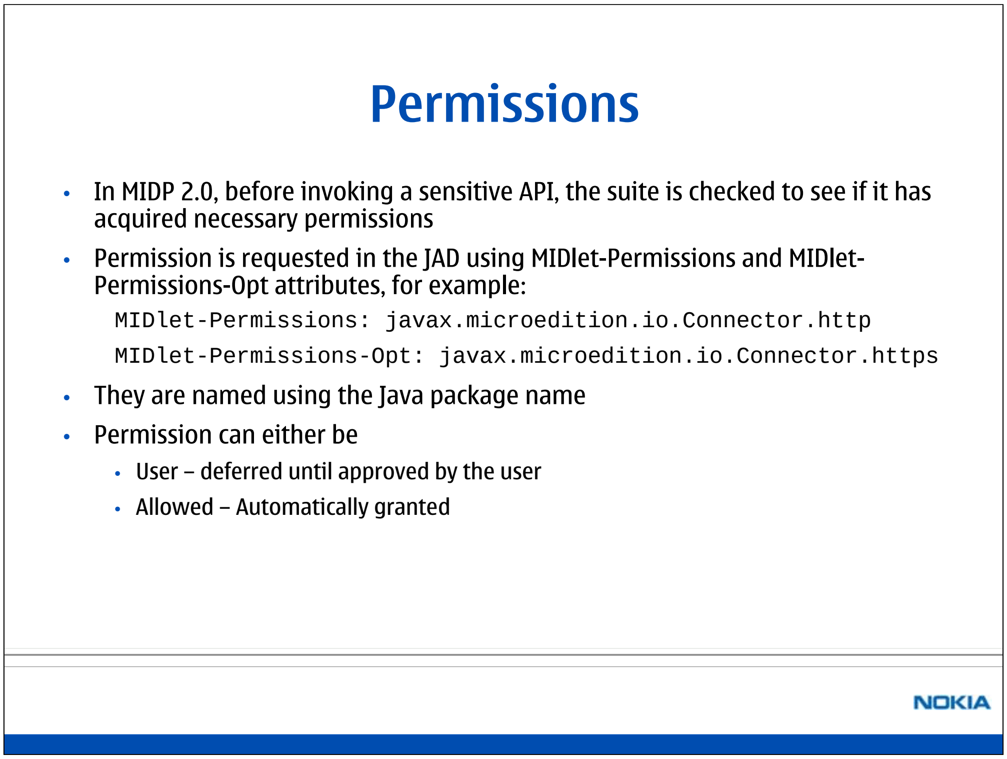


MIDP 2.0 Security

- MIDP 2.0 security model allows MIDlets automatic access to sensitive APIs
  - For example, making an HTTP connection
- Introduces concept of untrusted and trusted applications
  - Untrusted application (Equivalent to MIDP 1.0 MIDlets)
    - Limited access to restricted APIs, requiring user approval
  - Trusted applications
    - Can acquire permission to invoke restricted APIs
- MIDlet suites need to by “signed” with a Certificate to authenticate the sender
- MIDP 2.0 improved security features:
  - Application sender authentication using **Certificates**
  - Availability of different levels of secure domains
  - A means to trust integrity of applications received by device

MIDP 1.0 specified that all MIDlets ran using a sandbox security model. This meant that a MIDlet could only use libraries that were included in the same MIDlet suite or APIs that were part of the MIDP specification (e.g. record store, user interface components). It did not allow any access to device-specific functionality.

MIDP 2.0 changed this by introducing the idea of splitting applications into two categories, untrusted and trusted applications. The security model was enhanced to allow MIDlets access to APIs that are considered sensitive, such as those that make an HTTP connection. Untrusted applications are the same as those specified in MIDP 1.0, i.e. only use APIs that are part of the specification and in the suite. They have limited access to restricted APIs, requiring user approval depending on the security of the device. A trusted MIDlet, however, can use APIs outside the scope of the MIDlet suite, e.g. access to device specific configuration settings. They can acquire permissions automatically depending on the security policy.



Permissions

- In MIDP 2.0, before invoking a sensitive API, the suite is checked to see if it has acquired necessary permissions
- Permission is requested in the JAD using MIDlet-Permissions and MIDlet-Permissions-Opt attributes, for example:

```
MIDlet-Permissions: javax.microedition.io.Connector.http
MIDlet-Permissions-Opt: javax.microedition.io.Connector.https
```

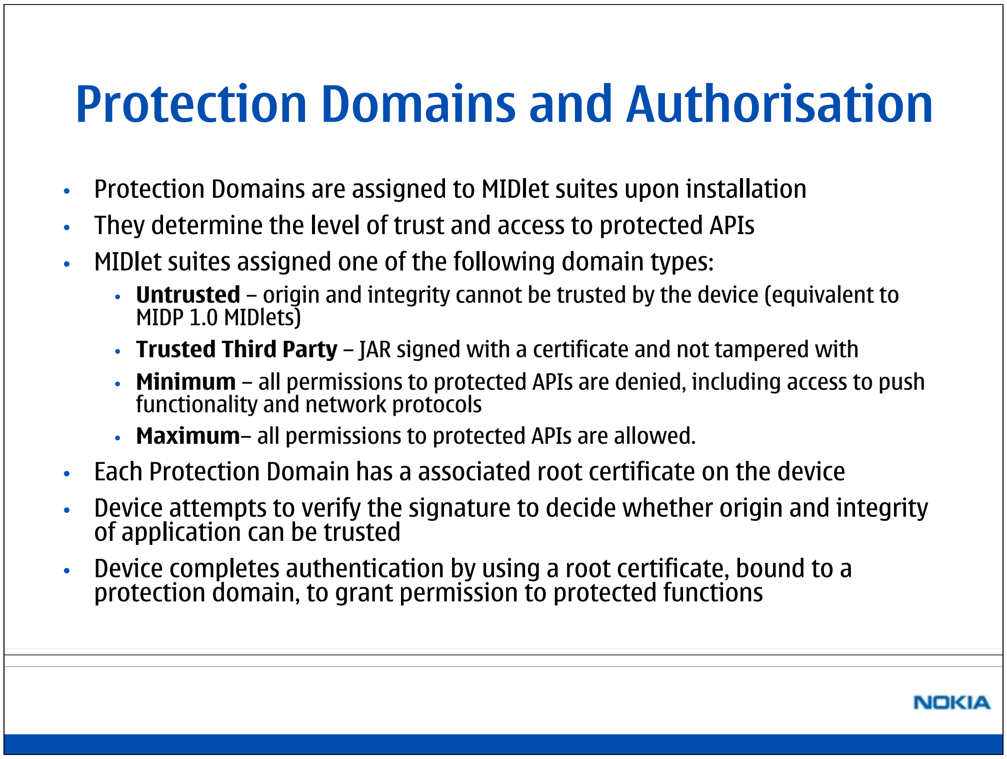
- They are named using the Java package name
- Permission can either be
  - User – deferred until approved by the user
  - Allowed – Automatically granted

Permissions are used to protect APIs that are sensitive and require authorization. The MIDP 2.0 implementation has to check whether a MIDlet suite has acquired the necessary permission before invoking the API. Permissions have names starting with the package name in the same way as Java Platform, Standard Edition (Java SE) permissions. For instance, the permission to make an HTTP connection is called javax.microedition.io.Connector.http. Permissions are documented along the class or package documentation of the protected.

Permissions can be either automatically granted or deferred until user approval. They are called allowed and user permissions respectively.

User permissions may require an explicit approval by the user. The user can either deny the permission or allow it. There are three interaction modes in which user permissions can be granted: blanket, session, and oneshot. When the blanket interaction mode is used, the MIDlet suite acquires the permission as long as the suite is installed, unless explicitly revoked by the user. The session mode requests the user authorization the first time the API is invoked and its validity is guaranteed while any of the MIDlets in the same suite are running. Finally, oneshot permissions request user approval every time the API is invoked. The protection domain determines which of the modes are available for each user permission as well as the default mode.

A MIDlet suite has to request permissions declaratively using the MIDlet-Permissions and MIDlet-Permissions-Opt attributes, either in the application descriptor or in the manifest file. MIDlet-Permissions contains permissions that are critical for the suite's functionality and MIDlet-Permissions-Opt indicates desired permissions, which are not so fundamental for the core functionality.

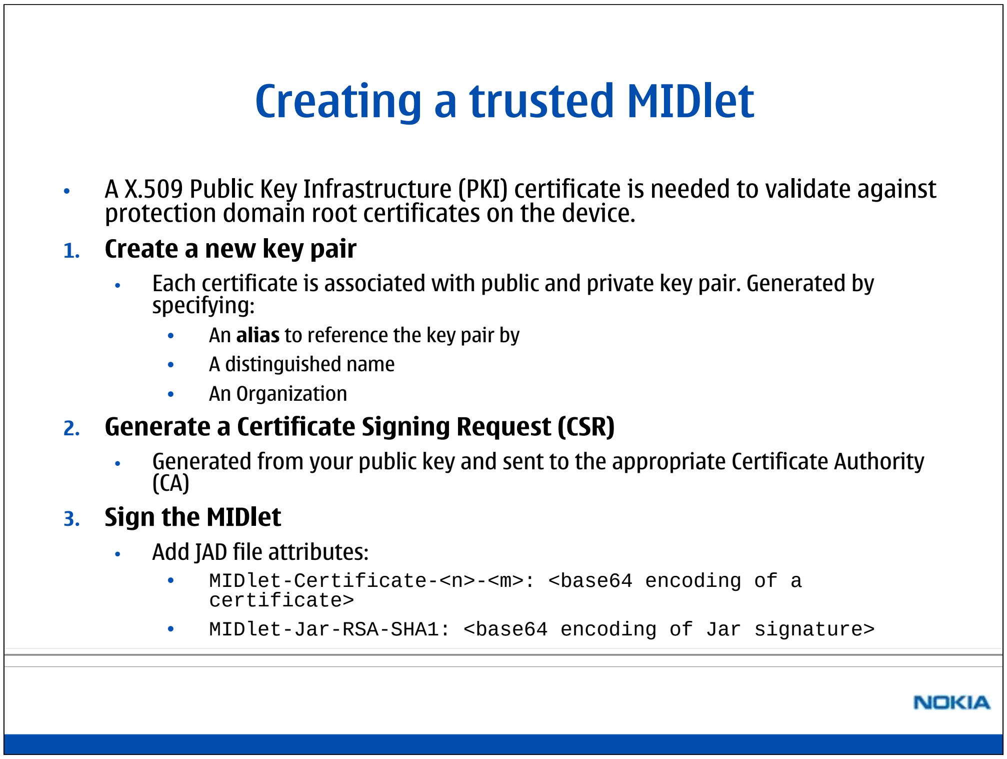


Protection Domains and Authorisation

- Protection Domains are assigned to MIDlet suites upon installation
- They determine the level of trust and access to protected APIs
- MIDlet suites assigned one of the following domain types:
  - **Untrusted** – origin and integrity cannot be trusted by the device (equivalent to MIDP 1.0 MIDlets)
  - **Trusted Third Party** – JAR signed with a certificate and not tampered with
  - **Minimum** – all permissions to protected APIs are denied, including access to push functionality and network protocols
  - **Maximum**– all permissions to protected APIs are allowed.
- Each Protection Domain has a associated root certificate on the device
- Device attempts to verify the signature to decide whether origin and integrity of application can be trusted
- Device completes authentication by using a root certificate, bound to a protection domain, to grant permission to protected functions

Prior to transmission, the MIDlet server signs an MIDlet JAR file using its digital certificate. Upon receipt, the device verifies the signature and decides whether the origin and integrity of the application can be trusted. If the digital signature cannot be verified, the runtime exits with an error. If the signature can be verified, the device uses the digital certificate to determine the permission domain for that entity. Once the verification process has been successfully completed, the application code is delivered to the client.

Note that each permission domain contains a set of rules to access specific APIs. For example, an application from a lesser known source might not be allowed to read/write local storage devices or make arbitrary network connections. Assigning a security domain to the certificate designates a level of trust the certification holder has to access protected APIs and the level of access to those APIs.



Creating a trusted MIDlet

- A X.509 Public Key Infrastructure (PKI) certificate is needed to validate against protection domain root certificates on the device.

1. **Create a new key pair**
   - Each certificate is associated with public and private key pair. Generated by specifying:
     - An **alias** to reference the key pair by
     - A distinguished name
     - An Organization
2. **Generate a Certificate Signing Request (CSR)**
   - Generated from your public key and sent to the appropriate Certificate Authority (CA)
3. **Sign the MIDlet**
   - Add JAD file attributes:
     - `MIDlet-Certificate-<n>-<m>: <base64 encoding of a certificate>`
     - `MIDlet-Jar-RSA-SHA1: <base64 encoding of Jar signature>`

NOKIA

The permissions that may be given to a MIDlet suite are defined in a protection domain. The domain owner has to decide who has access to the protected APIs and how to identify a trusted MIDlet suite.

A X.509 Public Key Infrastructure (PKI) certificate can be used as identification. To use PKI, a PKI certificate has to be generated for the MIDlet suite. This is the process known as signing.

The signer of the MIDlet suite may be the developer or some entity that is responsible for distributing, supporting, and perhaps billing for its use. The signer will need to have a public key certificate that can be validated to one of the protection domain root certificates on the device. The public key is used to verify the signature on the MIDlet suite. It is provided as a RSA X.509 certificate included in the application descriptor.

With each certificate, there is a key pair (public and private key) associated with it. Tools such as Nokia Developer's Suite assist generation of these. An alias, distinguished name and organization are used to generate the key pair. These are then stored in a keystore, referenced by the alias name.

The signer will need to be aware of the authorization policy for the target device and contact the appropriate certificate authority (CA). For example, the signer may need to send its distinguished name (DN) and public key, which are packaged into a Certificate Signing Request (CSR) file, to CA. The CA creates a RSA X.509 (version 3) certificate and returns it to the signer. Once a signature and the certification are obtained, these are added to the application description JAD file as attributes. If multiple CA's are used then all the signer certificates in the application descriptor must contain the same public key.

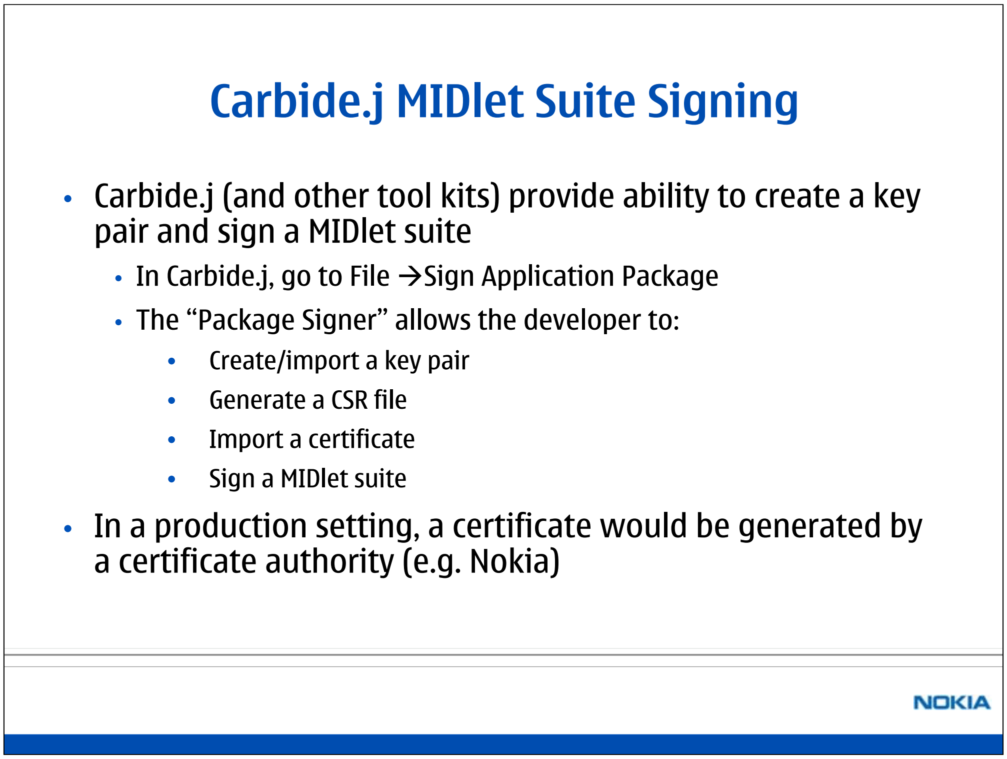
The certificate path includes the signer certificate and any necessary certificates. The certificate is encoded (using base64 but without line breaks) and inserted into the application descriptor as:

MIDlet-Certificate-<n>-<m>: <base64 encoding of a certificate>

<n>:= a number equal to 1 for first certification path in the descriptor or 1 greater than the previous number for additional certification paths. This defines the sequence in which the certificates are tested to see if the corresponding root certificate is on the device.

<m>:= a number equal to 1 for the signer's certificate in a certification path or 1 greater than the previous number for any subsequent intermediate certificates.

The signature of the JAR is created with the signer's private key according to an encoding method of Public Key Cryptography Standards (PKCS). The signature is base64 encoded, formatted as a single MIDlet-Jar-RSA-SHA1 attribute without line breaks and inserted in the application descriptor.



Carbide.j MIDlet Suite Signing

- Carbide.j (and other tool kits) provide ability to create a key pair and sign a MIDlet suite
  - In Carbide.j, go to File →Sign Application Package
  - The "Package Signer" allows the developer to:
    - Create/import a key pair
    - Generate a CSR file
    - Import a certificate
    - Sign a MIDlet suite
- In a production setting, a certificate would be generated by a certificate authority (e.g. Nokia)

The Package Signer tool is started from the main menu.

If integrated to an IDE, select:

Tools -> Carbide.j -> Sign Application Package...

If running stand alone, choose

File -> Sign Application Package

or press Sign Application Package button in the main window of the stand alone application.

Creating a new key pair

To create a new key pair from the Carbide.j Package Signer tool, click the New Key Pair button. This will then open a dialog box where you can specify an alias, distinguished name and organization. When you click Create, the tool generates a public and a private key that are references by the alias that you specified. A self signed public key certificate is also creates and the information for this stored in the keystore.

Generating a Certificate Signing Request (CSR)

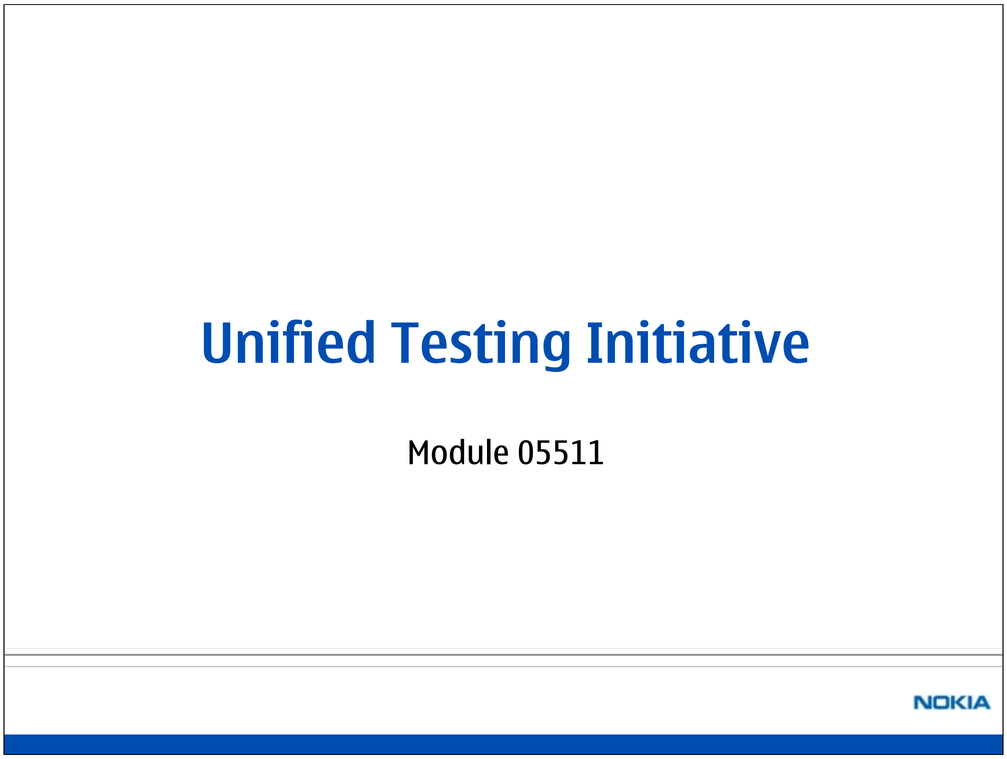
A CSR file needs to be sent to the Certificate Authority (CA) to obtain a certificate. You can do this from the tools by selecting an alias for signing from the list, then clicking the Generate CSR button.

Importing the public key certificate

The CA will send you a signed RSA X.509v3 certificate for your public key after you have sent the generated CSR. To import the certificate, select the corresponding alias, and click the Import Certificate button.

Signing

To actually sign an application, select an alias for the key pair you wish to use and press the Sign button. This will bring up a file chooser dialog, from which you must select the MIDlet Application Package's JAD file. The private key and the public key certificate corresponding to

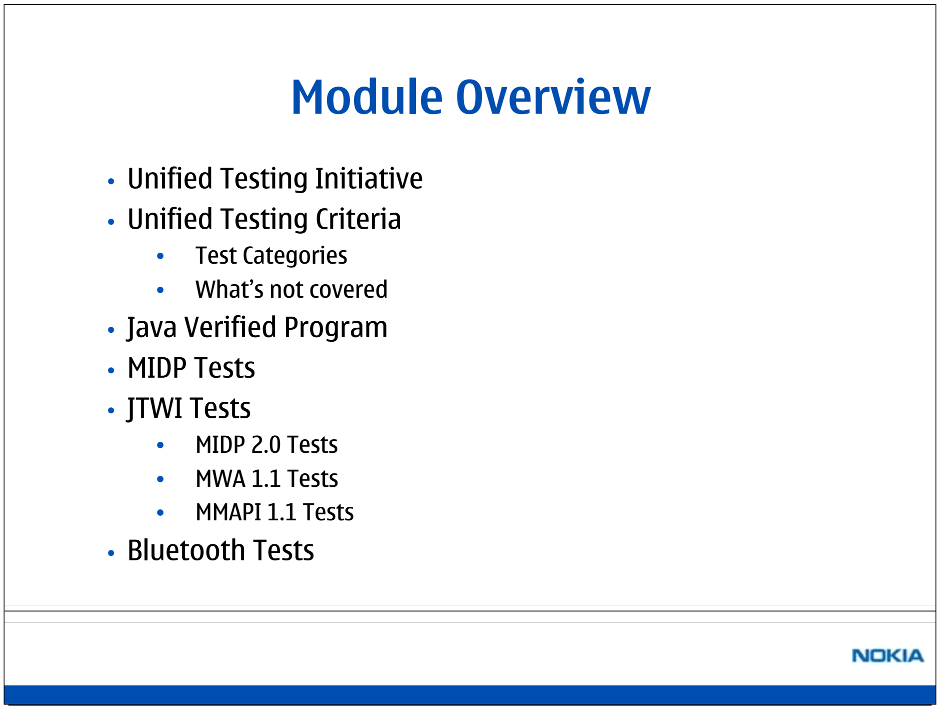


Unified Testing Initiative

Module 05511

The Unified Testing Initiative (UTI) is a joint effort between Sun Microsystems and handset manufacturers Motorola, Nokia, Siemens, and Sony Ericsson. The UTI has developed definite testing requirements related to the operating characteristics of a Java ME based applications. These testing requirements are commonly referred as the Unified Testing Criteria (UTC). The Java Verified program, born out of the UTI, is an industry-recognized common testing program for Java ME based applications.

This module will first explain the main concepts of the UTI including the benefits, and how the criteria are organised. It then goes on to explain the testing process of the Java Verified Program. Finally, the main part of this module goes through each criteria set out by the UTI and present explanations of why the test has been included and gives recommendations on how your application can pass the criteria.



Module Overview

- Unified Testing Initiative
- Unified Testing Criteria
  - Test Categories
  - What’s not covered
- Java Verified Program
- MIDP Tests
- JTWI Tests
  - MIDP 2.0 Tests
  - MWA 1.1 Tests
  - MMAPI 1.1 Tests
- Bluetooth Tests

- ***Unified Testing Initiative. Explains the main concepts of the UTI, explaining the benefits of having such an initiative and the companies that are involved***

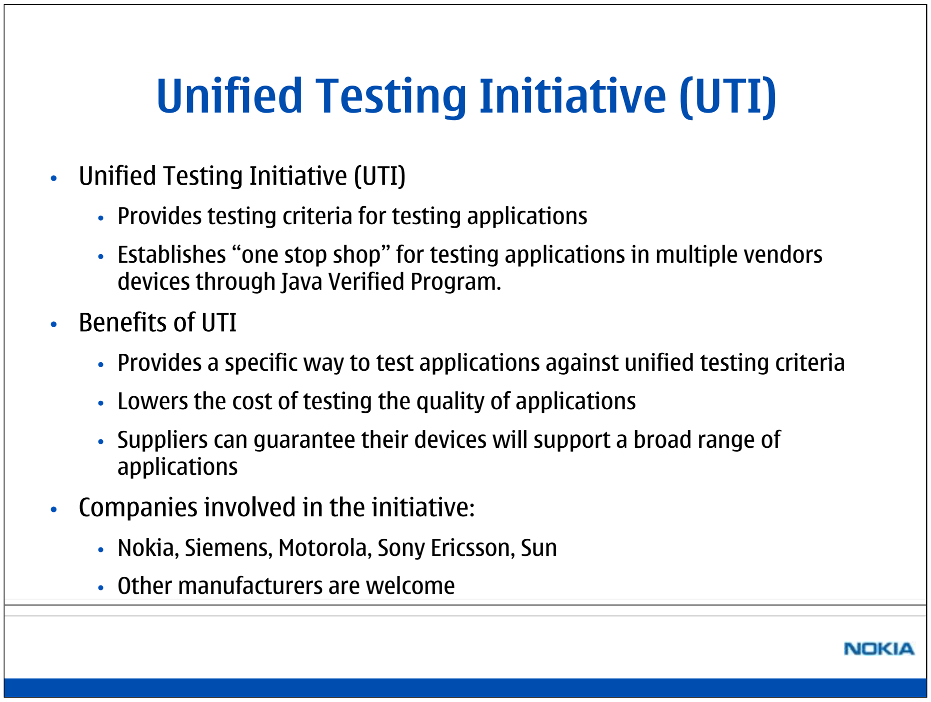
- ***Unified Testing Criteria. Gives details of what the criteria is, how it organized and how it is subdivided in test categories. Information about what is not actually tested by the criteria is also included in this section***

- ***Java Verified Program. Explains what the program is, gives details of the authorized testers, shows what the testing process is and explains what pre-testing is actually done.***

- ***MIDP Tests. This section goes through each criterion that covers tests for MIDP 1.0 and MIDP 2.0. It details why each criteria has been included, and gives recommendations on how they can be passed***

- ***JTWI Tests. This section goes through each criterion that covers tests for MIDP 2.0, WMA 1.1 and MMAPI 1.1. It details why each criteria has been included, and gives recommendations on how they can be passed***

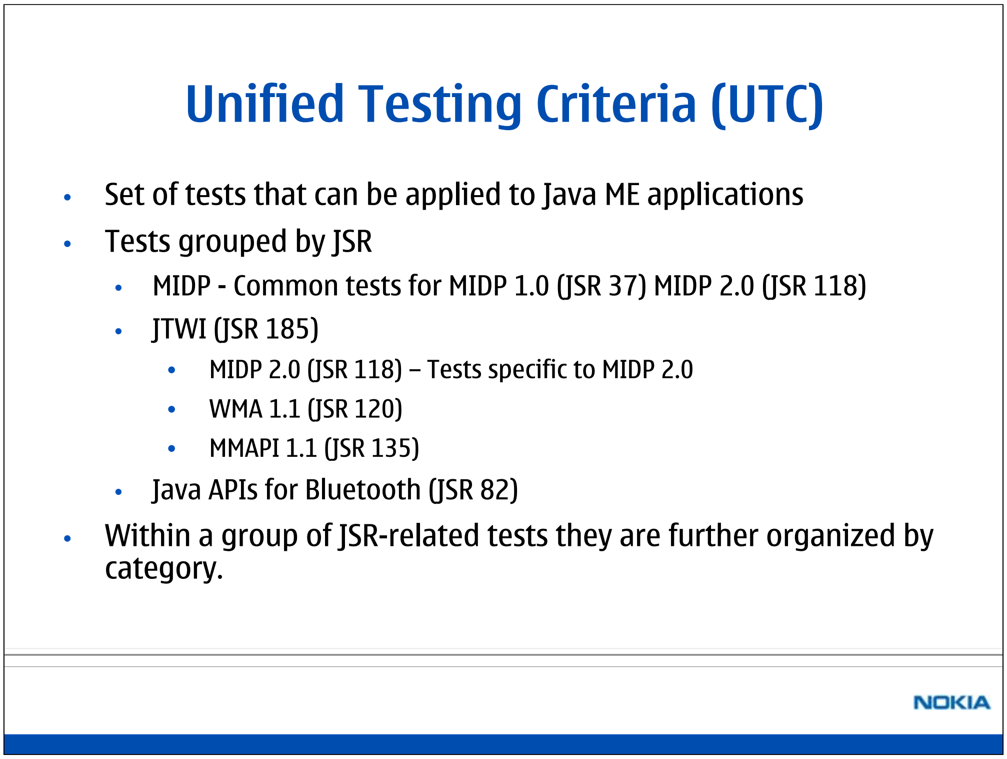
- ***Bluetooth Tests. This section goes through each criterion that covers tests for the Bluetooth API. It details why each criteria has been included, and gives recommendations on how they can be passed***



Unified Testing Initiative (UTI)

- Unified Testing Initiative (UTI)
  - Provides testing criteria for testing applications
  - Establishes “one stop shop” for testing applications in multiple vendors devices through Java Verified Program.
- Benefits of UTI
  - Provides a specific way to test applications against unified testing criteria
  - Lowers the cost of testing the quality of applications
  - Suppliers can guarantee their devices will support a broad range of applications
- Companies involved in the initiative:
  - Nokia, Siemens, Motorola, Sony Ericsson, Sun
  - Other manufacturers are welcome

The primary focus of the Unified Testing Initiative (UTI) is on the development and organisation of testing process for mobile application. The member companies of Nokia, Siemens, Motorola, Sony Ericsson, Sun aimed to establish a process based around Unified Testing Criteria (UTC).

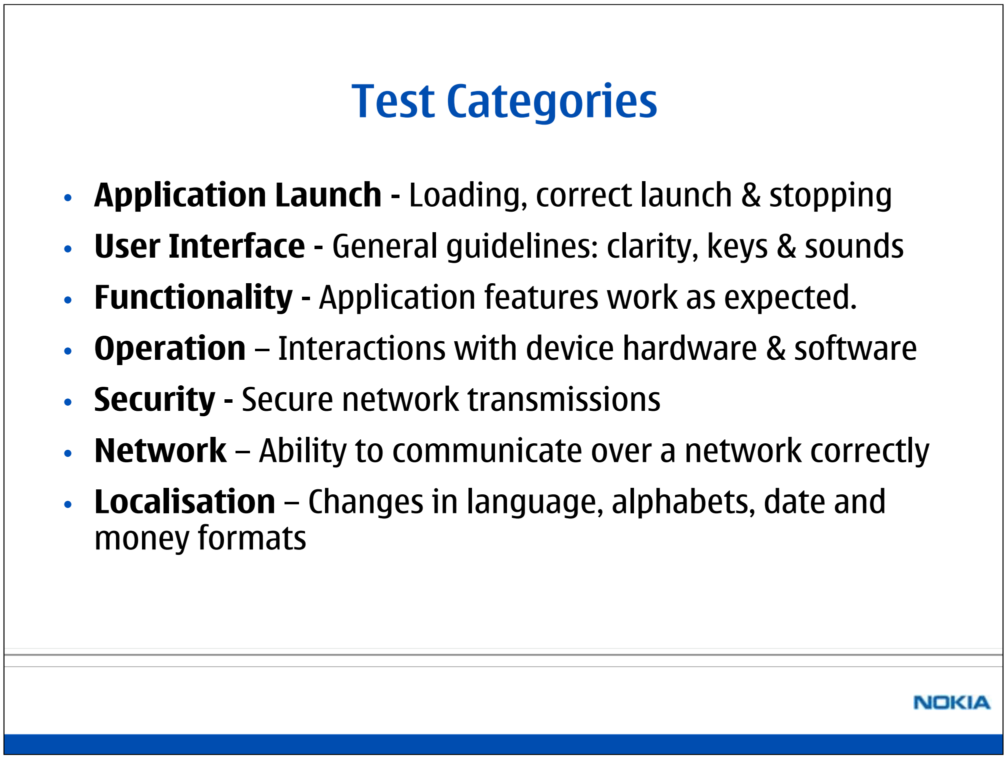


Unified Testing Criteria (UTC)

- Set of tests that can be applied to Java ME applications
- Tests grouped by JSR
  - MIDP - Common tests for MIDP 1.0 (JSR 37) MIDP 2.0 (JSR 118)
  - JTWI (JSR 185)
    - MIDP 2.0 (JSR 118) – Tests specific to MIDP 2.0
    - WMA 1.1 (JSR 120)
    - MMAPI 1.1 (JSR 135)
  - Java APIs for Bluetooth (JSR 82)
- Within a group of JSR-related tests they are further organized by category.

The member companies met and reviewed the tests from the respective partners. A common set of tests was decided upon, known as Unified Testing Criteria (UTC). This set was deemed broad enough to provide a good set of tests but not so specific to favour one device over the other. By restricting the tests to applications written in the Java programming language it was possible to ignore device-specific features. The member companies will also evolve the UTC on a regular basis to meet future requirements as determined by market needs. The UTC centres on the requirements for a quality application. Its aim is less to ensure totally bug-free applications than to assure users of a certain level of application quality.

The UTC covers all types of MIDP 1.0 and MIDP 2.0 applications and additional Java Specification Requests (such as JSR 120, JSR 135, and JSR 82). As well as being grouped by JSR, the tests are also further organised into test categories.

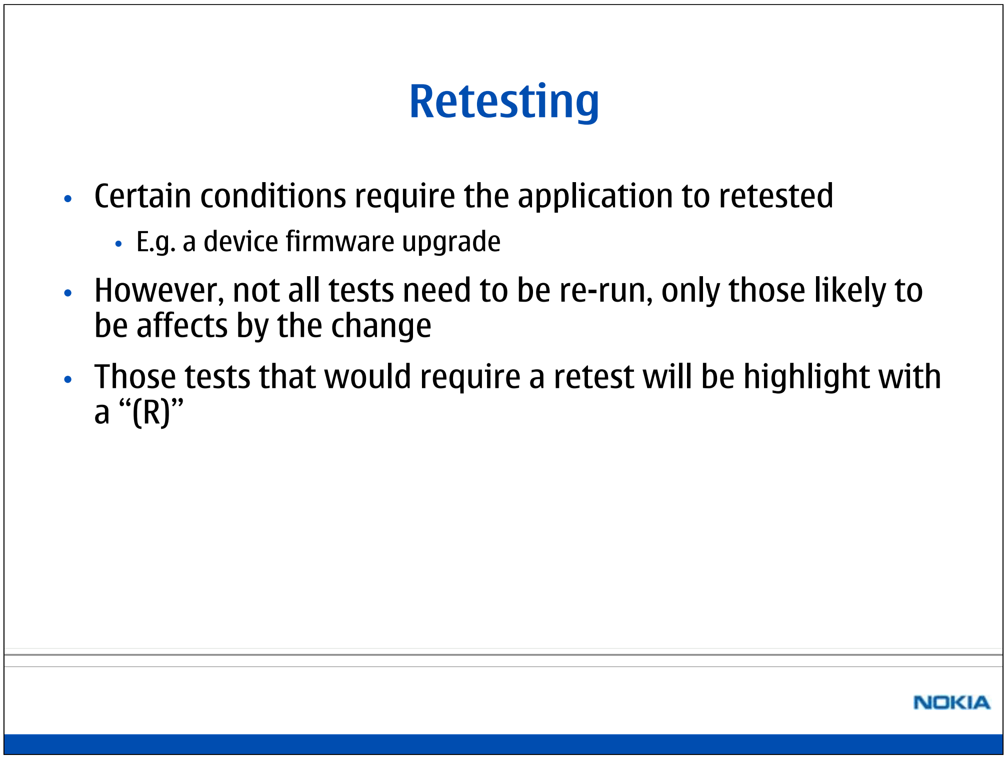


Test Categories

- **Application Launch -** Loading, correct launch & stopping
- **User Interface -** General guidelines: clarity, keys & sounds
- **Functionality -** Application features work as expected.
- **Operation** – Interactions with device hardware & software
- **Security -** Secure network transmissions
- **Network** – Ability to communicate over a network correctly
- **Localisation** – Changes in language, alphabets, date and money formats

As well as being grouped by JSR, the test criteria are also further organised into test categories

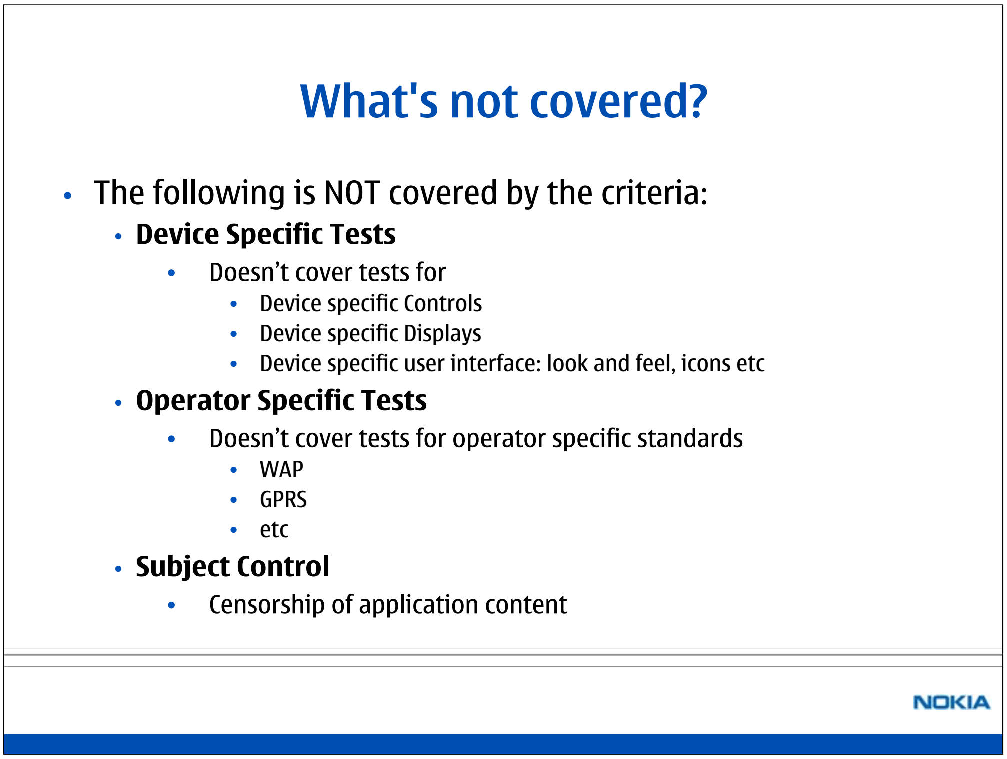
- *Application Launch – This checks that the application is able to start up and shut down correctly. The tests check that the application doesn't take too long to start up, all resources are released when the application is closed and the performance of the device is not affected by the application*
- *User Interface – These tests make sure the application is not confusing to the user, is displayed correctly on the devices screen, is intuitive and provides enough assistance to the user for them to use the application correctly.*
- *Functionality – The category of tests is included to ensure the application functions as expected, in terms of being able to exit the application, and not consuming power unnecessarily.*
- *Operation – These tests are included to ensure the application can interact correctly with device hardware and software.*
- *Security – This checks that the application ensures information such as passwords, credit card details and other sensitive data are sent securely to prevent potential eavesdroppers from intercepting and using it.*
- *Network – These tests check that the application has the ability to deal with unreliable network connections, delays on the network connection, and errors whilst communicate over a network.*
- *Localisation – Checks that changes in language, alphabets, date and money formats are dealt with in the application*



Retesting

- Certain conditions require the application to retested
  - E.g. a device firmware upgrade
- However, not all tests need to be re-run, only those likely to be affects by the change
- Those tests that would require a retest will be highlight with a “(R)”

There are certain conditions that require that an application be tested more than once. For example, if the version of the firmware in the device were upgraded to add functionality and the application needed to run on both versions then a retest would be required. Not all tests in this document need to be applied to an application that is getting retested - only those that would be most likely affected by a change. In the rest of this document, those tests that would be part of a retest are marked with a "(R)"



What's not covered?

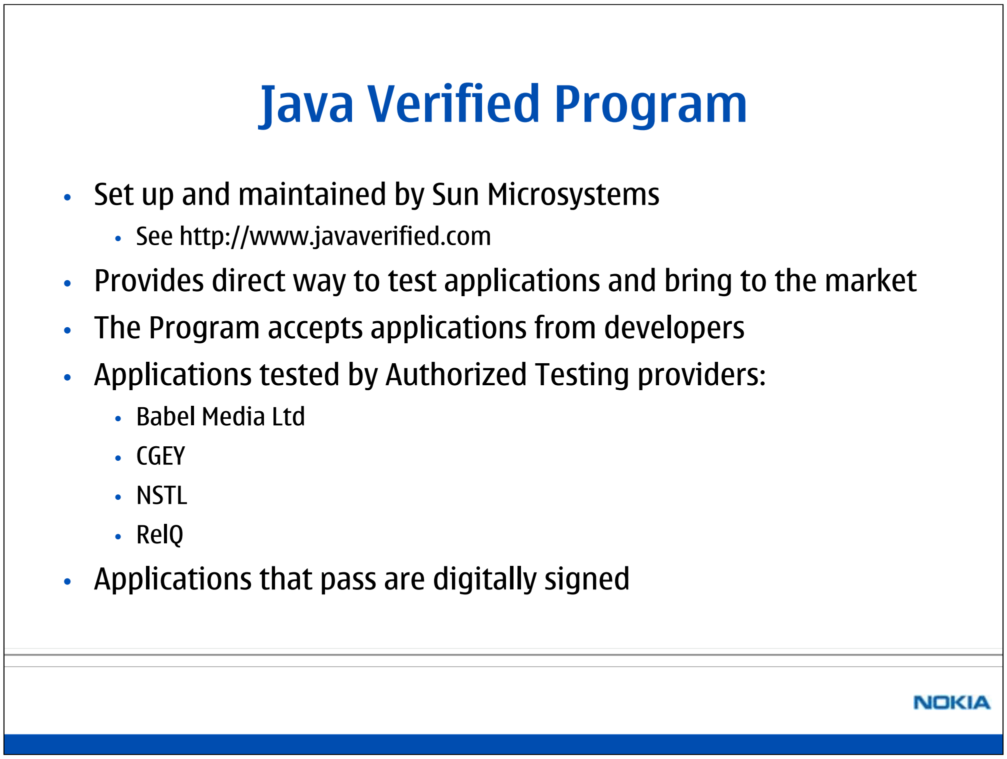
- The following is NOT covered by the criteria:
  - **Device Specific Tests**
    - Doesn’t cover tests for
      - Device specific Controls
      - Device specific Displays
      - Device specific user interface: look and feel, icons etc
  - **Operator Specific Tests**
    - Doesn’t cover tests for operator specific standards
      - WAP
      - GPRS
      - etc
  - **Subject Control**
    - Censorship of application content

NOKIA

The Unified Testing Criteria does not cover the following tests:

- ***Operator-specific tests***
- ***Network communication protocols: Wireless Application Protocol (WAP), General Packet Radio Service (GPRS)***
- ***Last-mile tests (Location-based application programming interfaces or APIs, or specific download servers)***
- ***Device-specific tests***
- ***User interface: Look and feel, icons***
- ***Content censorship***

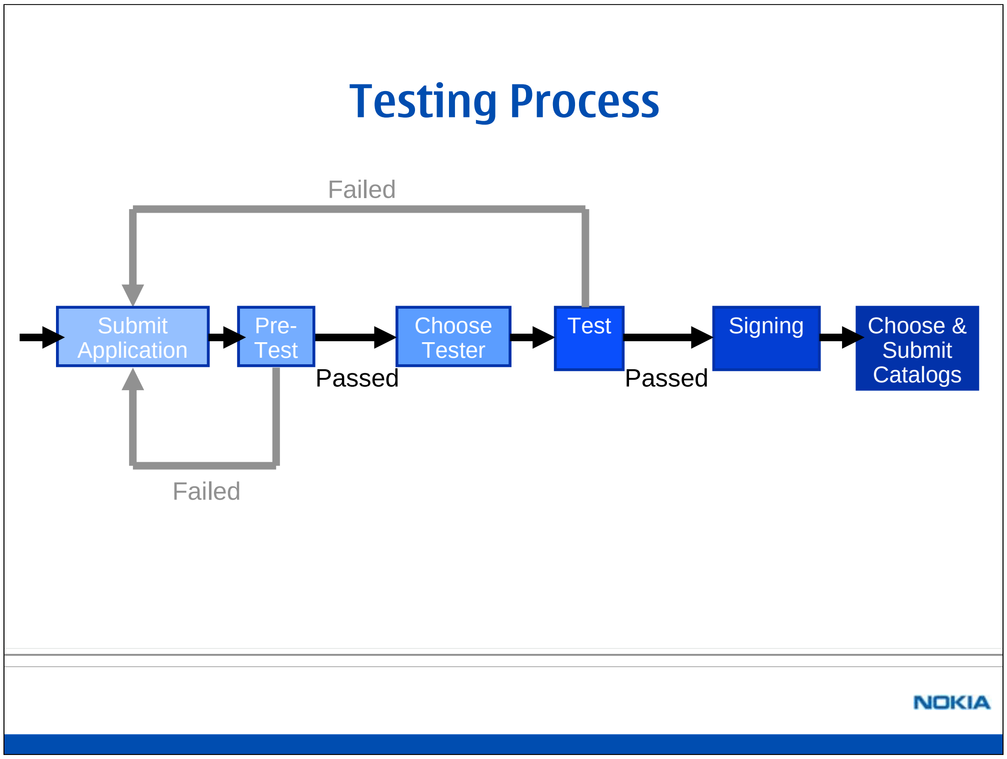
The points mentioned above are test cases that are left up to the responsible party (network operator, manufacturer, or developer).



Java Verified Program

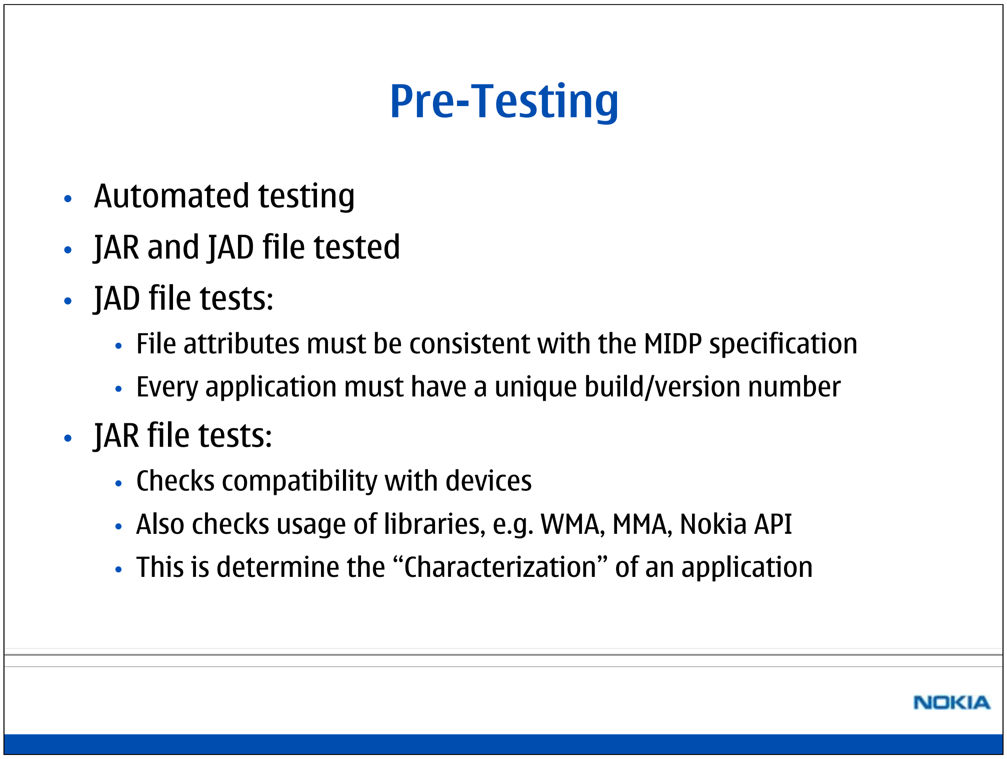
- Set up and maintained by Sun Microsystems
  - See http://www.javaverified.com
- Provides direct way to test applications and bring to the market
- The Program accepts applications from developers
- Applications tested by Authorized Testing providers:
  - Babel Media Ltd
  - CGEY
  - NSTL
  - RelQ
- Applications that pass are digitally signed

The Java Verified program is the first to support the Unified Testing Criteria. The program provides developers with the most direct way to have their applications tested and brought to market. It uses approved selected test providers to test Java ME applications on its behalf, based on the Unified Testing Criteria. An application that successfully passes through this program can be placed into the Member Companies on-line catalogs for subsequent marketing and promotion, and in the commercial catalogs of participating operators.



Testing Process

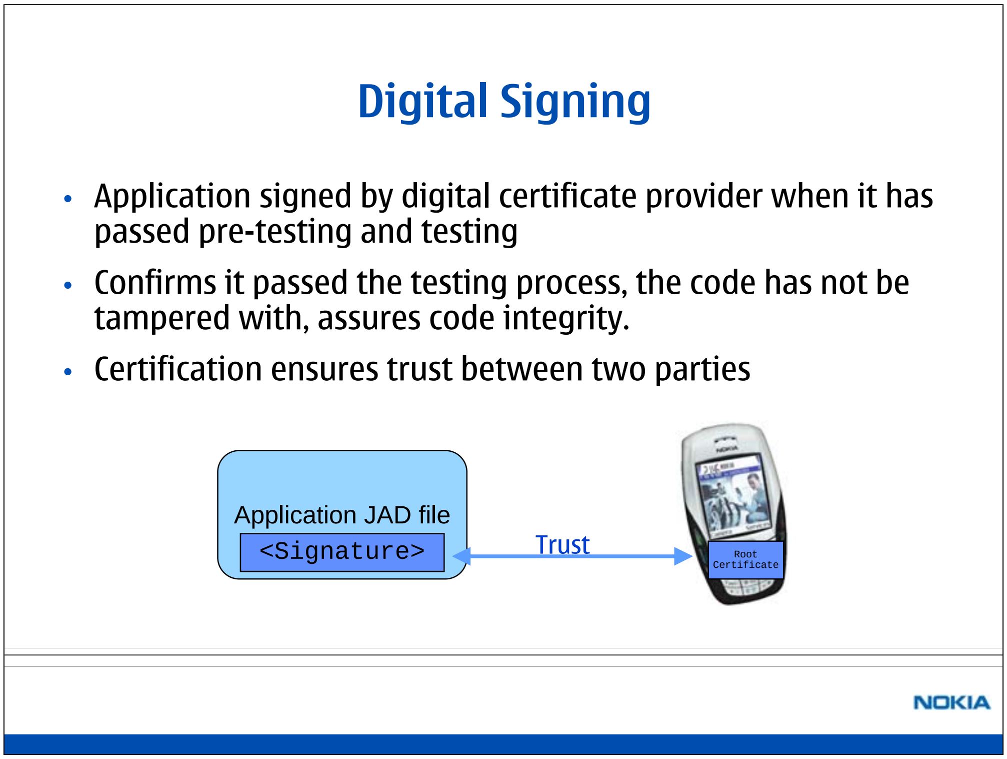
Once Developers have registered with the Java Verified Program and have logged on, the application is submitted for testing. Pretesting is carried out first. Pretesting consists of tests that are executed automatically by examining the JAR and JAD. If the application does not pass pretesting then the Developer is informed and, if the Developer wants to resubmit the application it must be fixed first. After the application passes pretesting Developers can go to a page where they will be able to decide on which Testing Provider that they want to test their application. The Provider contacts the Developer, establishes a business connection with the Developer and looks at the application. The Testing Provider applies a sequence of manual tests that are based on the UTI Testing Criteria. The developer is informed if there are any problems. If testing is to carry on, the application needs to be corrected and resubmitted. Once all tests have been passed, the application is certified – digitally signed so that anyone can verify that the application has passed testing successfully. Finally, the Developers can optionally have their application submitted to various catalogs maintained by members of the Program.



Pre-Testing

- Automated testing
- JAR and JAD file tested
- JAD file tests:
  - File attributes must be consistent with the MIDP specification
  - Every application must have a unique build/version number
- JAR file tests:
  - Checks compatibility with devices
  - Also checks usage of libraries, e.g. WMA, MMA, Nokia API
  - This is determine the “Characterization” of an application

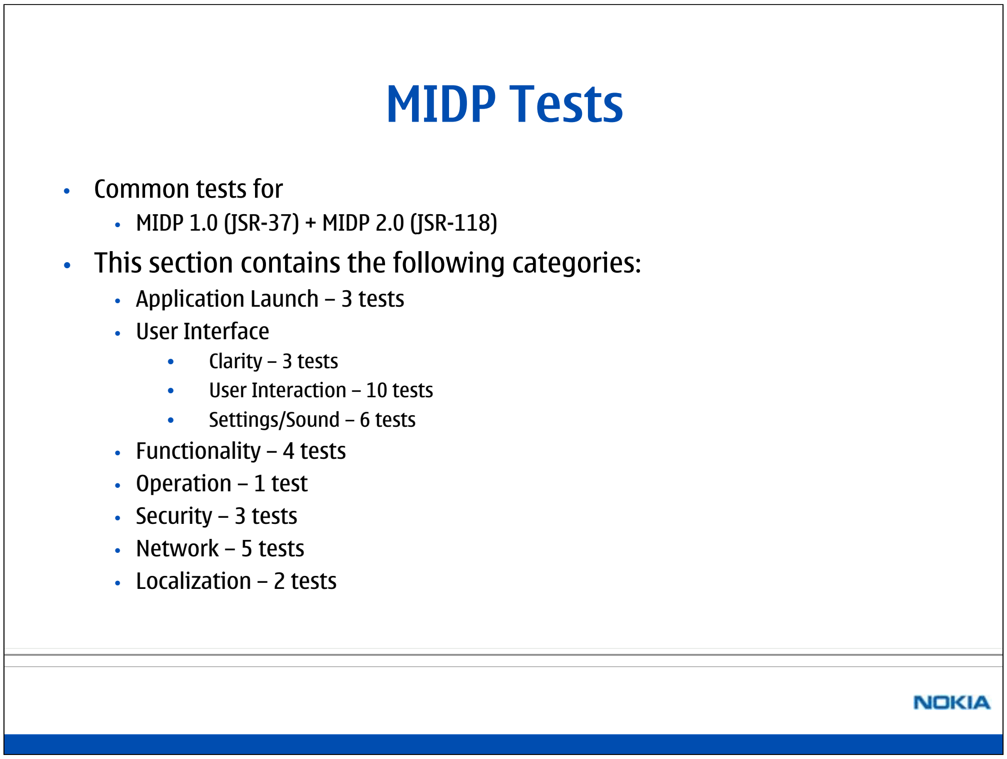
An application submitted for testing is first run through a set of automatic tests that can determine the correctness of certain packaging attributes. These tests produce a testing report and a profile of the application. The profile describes relevant attributes of the application, such as the set of standard and nonstandard classes that are used, size of images, ability to run on specific platforms, and so on. Employees at the test providers then subject the application to a set of manual tests. The test results are made available to the developer.



Digital Signing

- Application signed by digital certificate provider when it has passed pre-testing and testing
- Confirms it passed the testing process, the code has not be tampered with, assures code integrity.
- Certification ensures trust between two parties

Tested MIDP applications are digitally signed with the UTI signature to verify that an application meets the UTI criteria. Additionally, the Java Powered logo is issued for use by the content owner to show that the application has passed the Unified Testing Criteria. The Java Powered logo represents a marketing advantage for developers.

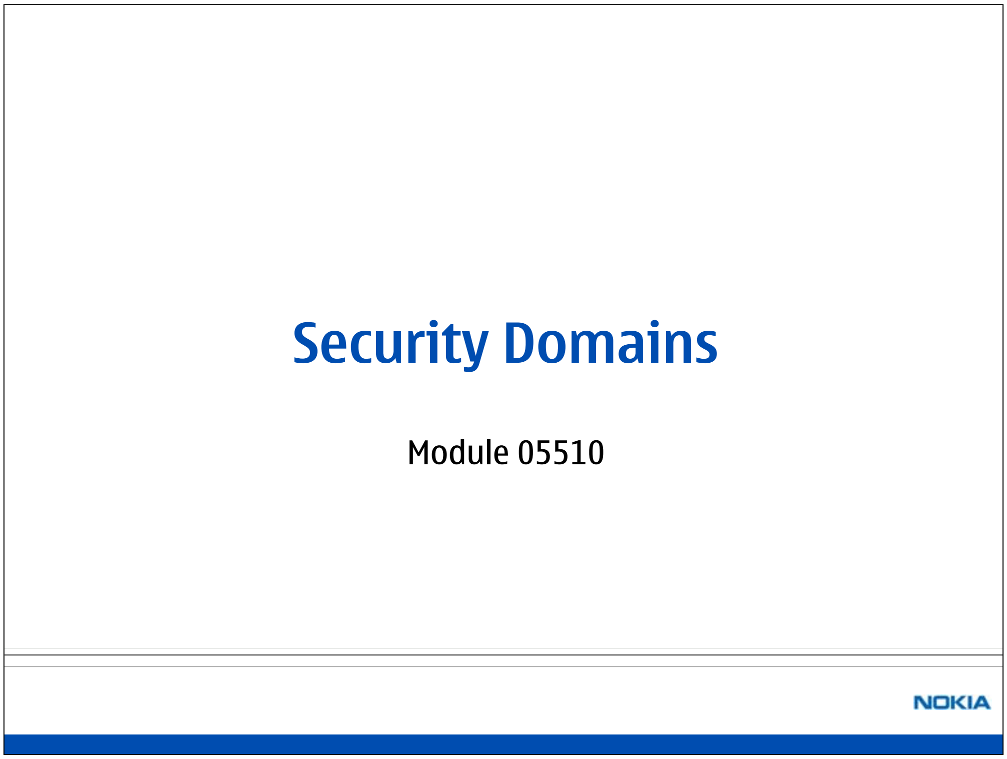


MIDP Tests

- Common tests for
  - MIDP 1.0 (JSR-37) + MIDP 2.0 (JSR-118)
- This section contains the following categories:
  - Application Launch – 3 tests
  - User Interface
    - Clarity – 3 tests
    - User Interaction – 10 tests
    - Settings/Sound – 6 tests
  - Functionality – 4 tests
  - Operation – 1 test
  - Security – 3 tests
  - Network – 5 tests
  - Localization – 2 tests

NOKIA

The tests in this section are for applications that use either version 1.0 (JSR 37) or 2.0 (JSR 118) of the Mobile Information Device Profile (MIDP) applications. That is, they represent a common set of tests that can be applied to applications that use either version of the MIDP. The tests are organized by test category.



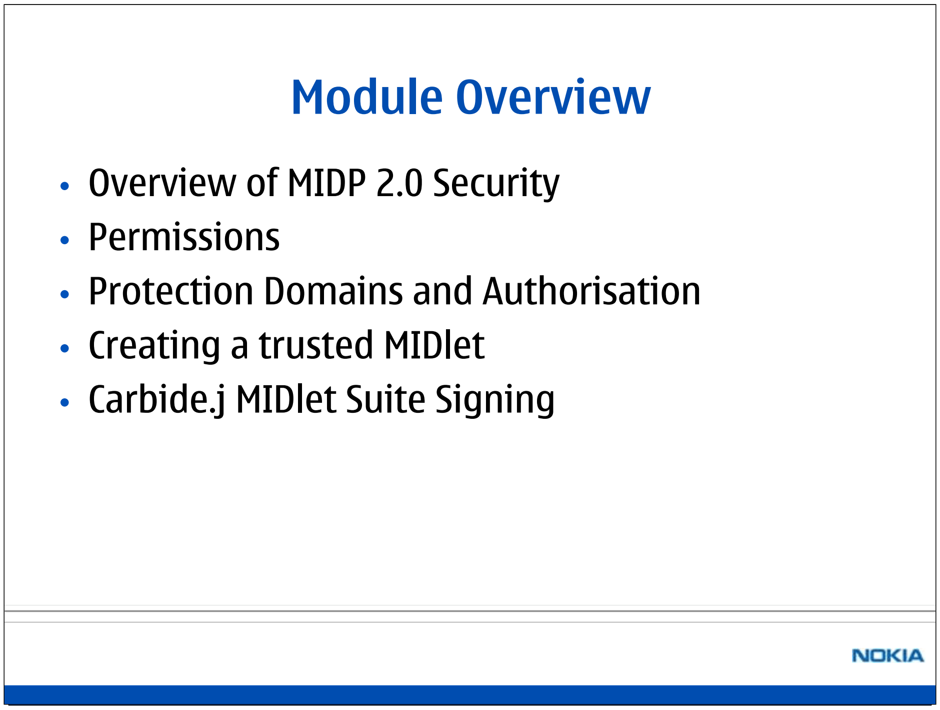
Security Domains

Module 05510

Running a downloaded MIDlet, although safer than binary code (due to it running in a virtual machine), could result in security risks if the source of the code is unknown. For example, the MIDlet could try to collect information about the device and its user and send it to a server by making unauthorized network connections.

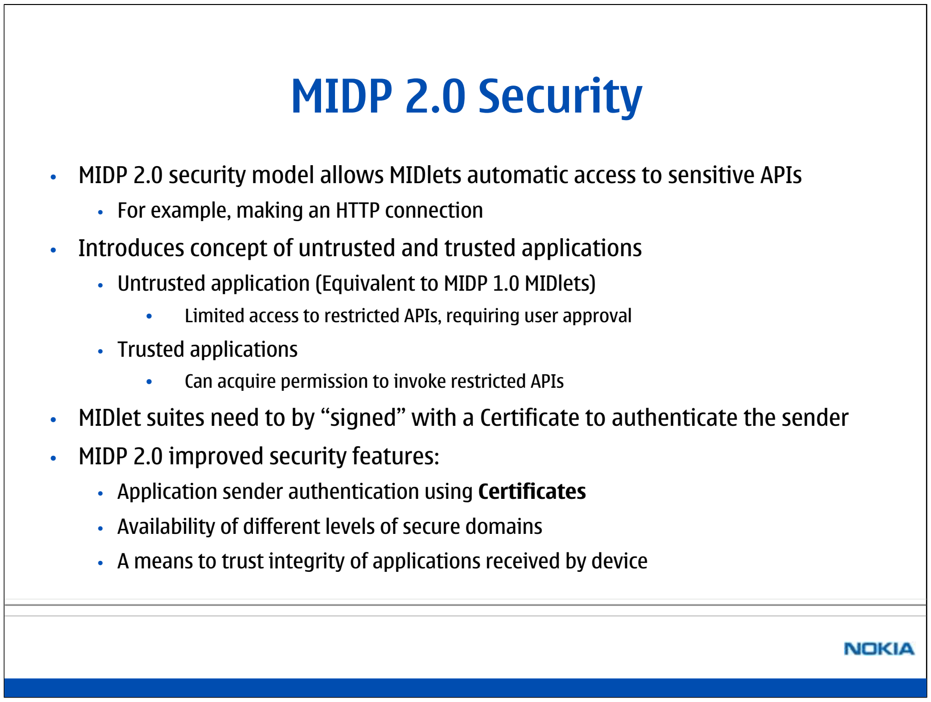
Protection domains can be configured to make it safer for users to run downloaded MIDlets. The MIDP 2.0 specification suggests a protection domain based on cryptographic signatures and certificates, where the software developer creates a signing-key pair and acquires a certificate from a recognized certificate authority. The developer generates a signature of the MIDlet suite JAR, then places that signature and the related certificate in the application's descriptor file.

This module will describe new concepts introduced in MIDP 2.0 relating to security, and explain how implement these concepts using Carbide.j.



Module Overview

- Overview of MIDP 2.0 Security
- Permissions
- Protection Domains and Authorisation
- Creating a trusted MIDlet
- Carbide.j MIDlet Suite Signing

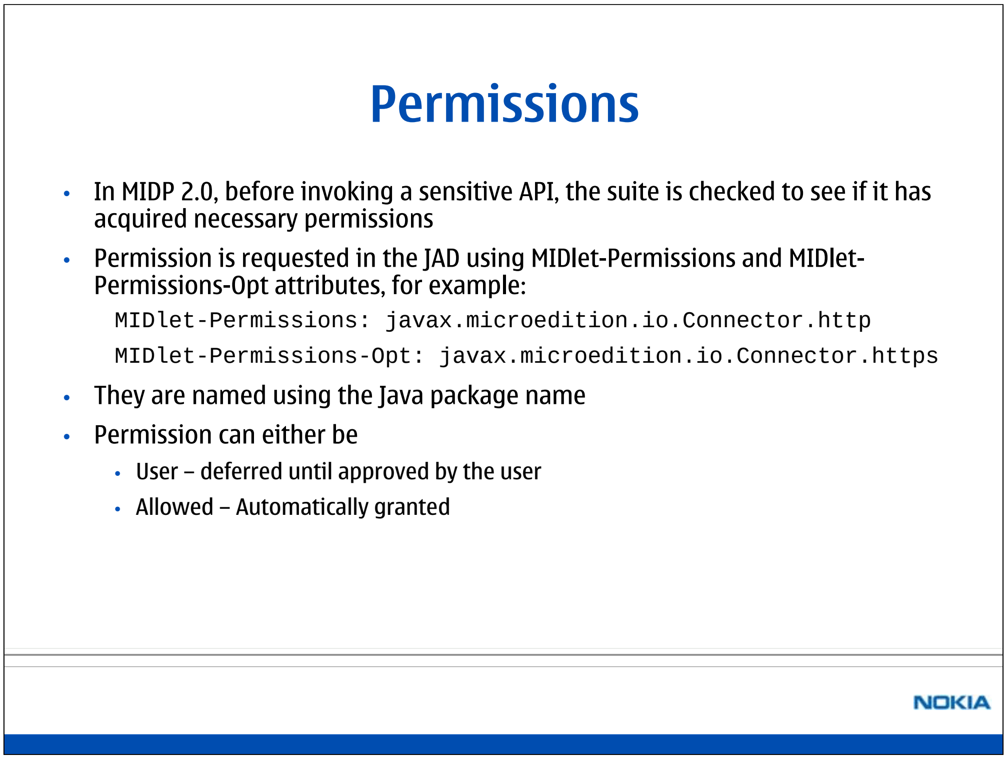


MIDP 2.0 Security

- MIDP 2.0 security model allows MIDlets automatic access to sensitive APIs
  - For example, making an HTTP connection
- Introduces concept of untrusted and trusted applications
  - Untrusted application (Equivalent to MIDP 1.0 MIDlets)
    - Limited access to restricted APIs, requiring user approval
  - Trusted applications
    - Can acquire permission to invoke restricted APIs
- MIDlet suites need to by “signed” with a Certificate to authenticate the sender
- MIDP 2.0 improved security features:
  - Application sender authentication using **Certificates**
  - Availability of different levels of secure domains
  - A means to trust integrity of applications received by device

MIDP 1.0 specified that all MIDlets ran using a sandbox security model. This meant that a MIDlet could only use libraries that were included in the same MIDlet suite or APIs that were part of the MIDP specification (e.g. record store, user interface components). It did not allow any access to device-specific functionality.

MIDP 2.0 changed this by introducing the idea of splitting applications into two categories, untrusted and trusted applications. The security model was enhanced to allow MIDlets access to APIs that are considered sensitive, such as those that make an HTTP connection. Untrusted applications are the same as those specified in MIDP 1.0, i.e. only use APIs that are part of the specification and in the suite. They have limited access to restricted APIs, requiring user approval depending on the security of the device. A trusted MIDlet, however, can use APIs outside the scope of the MIDlet suite, e.g. access to device specific configuration settings. They can acquire permissions automatically depending on the security policy.



Permissions

- In MIDP 2.0, before invoking a sensitive API, the suite is checked to see if it has acquired necessary permissions
- Permission is requested in the JAD using MIDlet-Permissions and MIDlet-Permissions-Opt attributes, for example:

```
MIDlet-Permissions: javax.microedition.io.Connector.http
MIDlet-Permissions-Opt: javax.microedition.io.Connector.https
```

- They are named using the Java package name
- Permission can either be
  - User – deferred until approved by the user
  - Allowed – Automatically granted

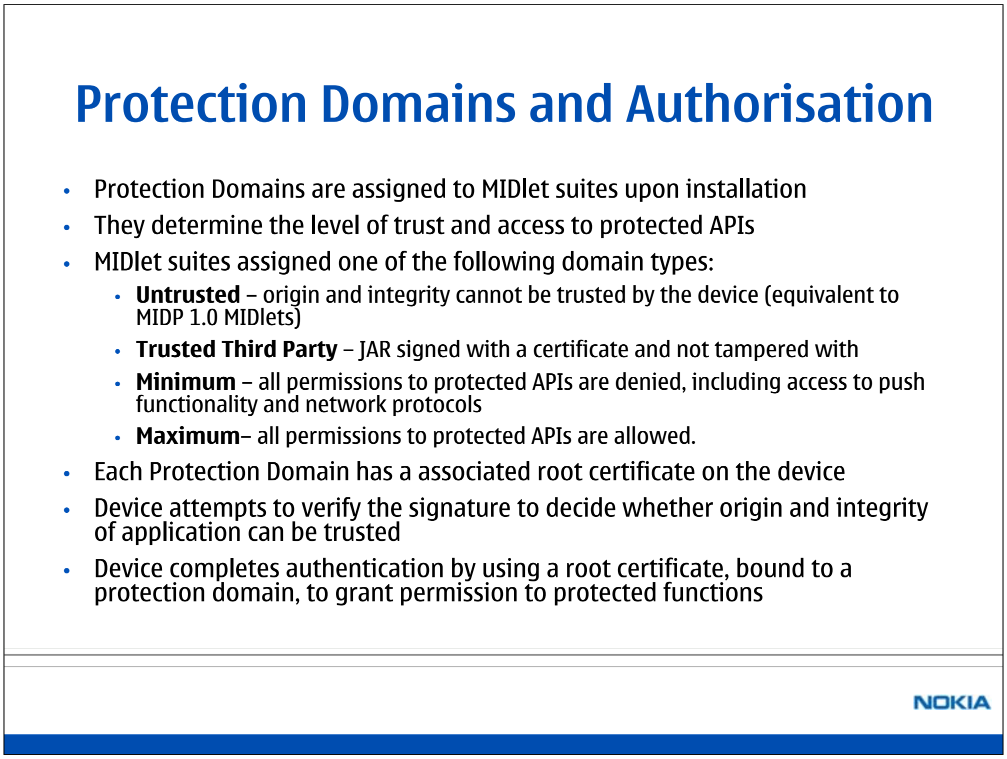
NOKIA

Permissions are used to protect APIs that are sensitive and require authorization. The MIDP 2.0 implementation has to check whether a MIDlet suite has acquired the necessary permission before invoking the API. Permissions have names starting with the package name in the same way as Java Platform, Standard Edition (Java SE) permissions. For instance, the permission to make an HTTP connection is called javax.microedition.io.Connector.http. Permissions are documented along the class or package documentation of the protected.

Permissions can be either automatically granted or deferred until user approval. They are called allowed and user permissions respectively.

User permissions may require an explicit approval by the user. The user can either deny the permission or allow it. There are three interaction modes in which user permissions can be granted: blanket, session, and oneshot. When the blanket interaction mode is used, the MIDlet suite acquires the permission as long as the suite is installed, unless explicitly revoked by the user. The session mode requests the user authorization the first time the API is invoked and its validity is guaranteed while any of the MIDlets in the same suite are running. Finally, oneshot permissions request user approval every time the API is invoked. The protection domain determines which of the modes are available for each user permission as well as the default mode.

A MIDlet suite has to request permissions declaratively using the MIDlet-Permissions and MIDlet-Permissions-Opt attributes, either in the application descriptor or in the manifest file. MIDlet-Permissions contains permissions that are critical for the suite's functionality and MIDlet-Permissions-Opt indicates desired permissions, which are not so fundamental for the core functionality.

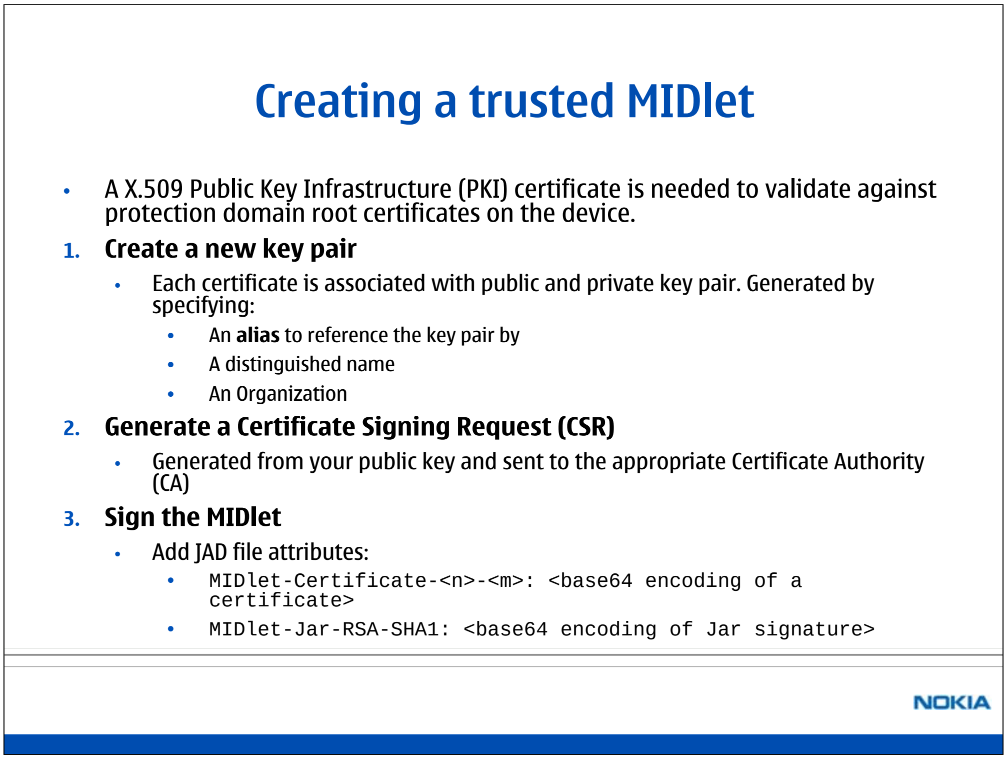


Protection Domains and Authorisation

- Protection Domains are assigned to MIDlet suites upon installation
- They determine the level of trust and access to protected APIs
- MIDlet suites assigned one of the following domain types:
  - **Untrusted** – origin and integrity cannot be trusted by the device (equivalent to MIDP 1.0 MIDlets)
  - **Trusted Third Party** – JAR signed with a certificate and not tampered with
  - **Minimum** – all permissions to protected APIs are denied, including access to push functionality and network protocols
  - **Maximum**– all permissions to protected APIs are allowed.
- Each Protection Domain has a associated root certificate on the device
- Device attempts to verify the signature to decide whether origin and integrity of application can be trusted
- Device completes authentication by using a root certificate, bound to a protection domain, to grant permission to protected functions

Prior to transmission, the MIDlet server signs an MIDlet JAR file using its digital certificate. Upon receipt, the device verifies the signature and decides whether the origin and integrity of the application can be trusted. If the digital signature cannot be verified, the runtime exits with an error. If the signature can be verified, the device uses the digital certificate to determine the permission domain for that entity. Once the verification process has been successfully completed, the application code is delivered to the client.

Note that each permission domain contains a set of rules to access specific APIs. For example, an application from a lesser known source might not be allowed to read/write local storage devices or make arbitrary network connections. Assigning a security domain to the certificate designates a level of trust the certification holder has to access protected APIs and the level of access to those APIs.



Creating a trusted MIDlet

- A X.509 Public Key Infrastructure (PKI) certificate is needed to validate against protection domain root certificates on the device.

1. **Create a new key pair**
   - Each certificate is associated with public and private key pair. Generated by specifying:
     - An **alias** to reference the key pair by
     - A distinguished name
     - An Organization
2. **Generate a Certificate Signing Request (CSR)**
   - Generated from your public key and sent to the appropriate Certificate Authority (CA)
3. **Sign the MIDlet**
   - Add JAD file attributes:
     - `MIDlet-Certificate-<n>-<m>: <base64 encoding of a certificate>`
     - `MIDlet-Jar-RSA-SHA1: <base64 encoding of Jar signature>`

The permissions that may be given to a MIDlet suite are defined in a protection domain. The domain owner has to decide who has access to the protected APIs and how to identify a trusted MIDlet suite.

A X.509 Public Key Infrastructure (PKI) certificate can be used as identification. To use PKI, a PKI certificate has to be generated for the MIDlet suite. This is the process known as signing.

The signer of the MIDlet suite may be the developer or some entity that is responsible for distributing, supporting, and perhaps billing for its use. The signer will need to have a public key certificate that can be validated to one of the protection domain root certificates on the device. The public key is used to verify the signature on the MIDlet suite. It is provided as a RSA X.509 certificate included in the application descriptor.

With each certificate, there is a key pair (public and private key) associated with it. Tools such as Nokia Developer's Suite assist generation of these. An alias, distinguished name and organization are used to generate the key pair. These are then stored in a keystore, referenced by the alias name.

The signer will need to be aware of the authorization policy for the target device and contact the appropriate certificate authority (CA). For example, the signer may need to send its distinguished name (DN) and public key, which are packaged into a Certificate Signing Request (CSR) file, to CA. The CA creates a RSA X.509 (version 3) certificate and returns it to the signer. Once a signature and the certification are obtained, these are added to the application description JAD file as attributes. If multiple CA's are used then all the signer certificates in the application descriptor must contain the same public key.

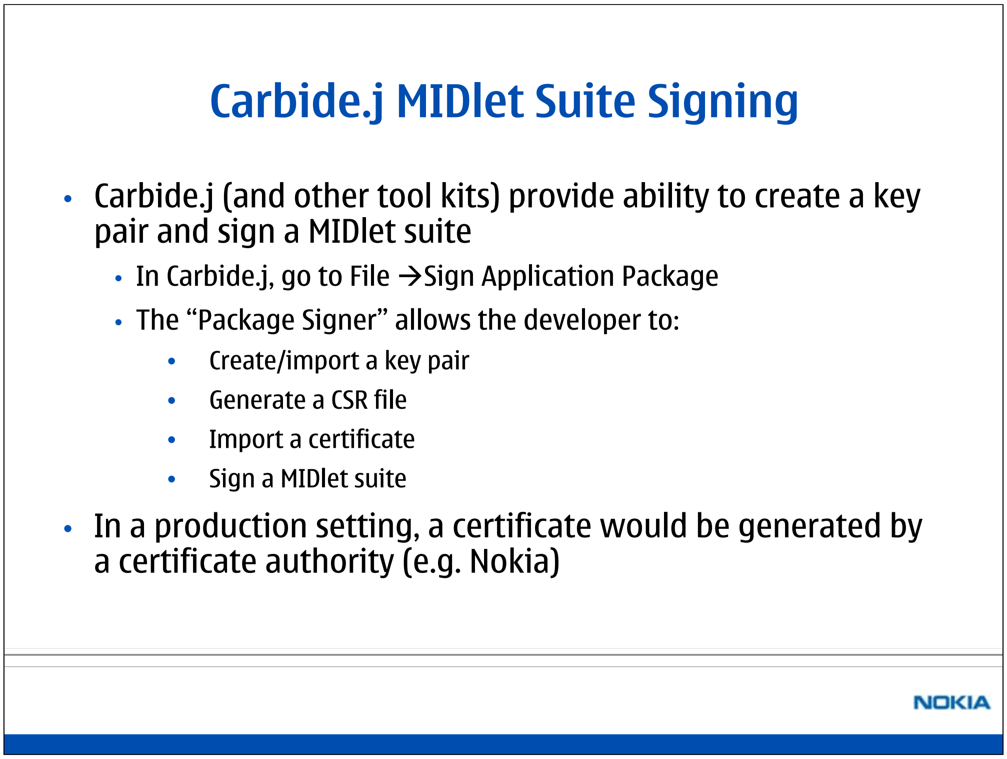
The certificate path includes the signer certificate and any necessary certificates. The certificate is encoded (using base64 but without line breaks) and inserted into the application descriptor as:

MIDlet-Certificate-<n>-<m>: <base64 encoding of a certificate>

<n>:= a number equal to 1 for first certification path in the descriptor or 1 greater than the previous number for additional certification paths. This defines the sequence in which the certificates are tested to see if the corresponding root certificate is on the device.

<m>:= a number equal to 1 for the signer's certificate in a certification path or 1 greater than the previous number for any subsequent intermediate certificates.

The signature of the JAR is created with the signer's private key according to an encoding method of Public Key Cryptography Standards (PKCS). The signature is base64 encoded, formatted as a single MIDlet-Jar-RSA-SHA1 attribute without line breaks and inserted in the application descriptor.



Carbide.j MIDlet Suite Signing

- Carbide.j (and other tool kits) provide ability to create a key pair and sign a MIDlet suite
  - In Carbide.j, go to File →Sign Application Package
  - The "Package Signer" allows the developer to:
    - Create/import a key pair
    - Generate a CSR file
    - Import a certificate
    - Sign a MIDlet suite
- In a production setting, a certificate would be generated by a certificate authority (e.g. Nokia)

The Package Signer tool is started from the main menu.

If integrated to an IDE, select:

Tools -> Carbide.j -> Sign Application Package...

If running stand alone, choose

File -> Sign Application Package

or press Sign Application Package button in the main window of the stand alone application.

Creating a new key pair

To create a new key pair from the Carbide.j Package Signer tool, click the New Key Pair button. This will then open a dialog box where you can specify an alias, distinguished name and organization. When you click Create, the tool generates a public and a private key that are references by the alias that you specified. A self signed public key certificate is also creates and the information for this stored in the keystore.

Generating a Certificate Signing Request (CSR)

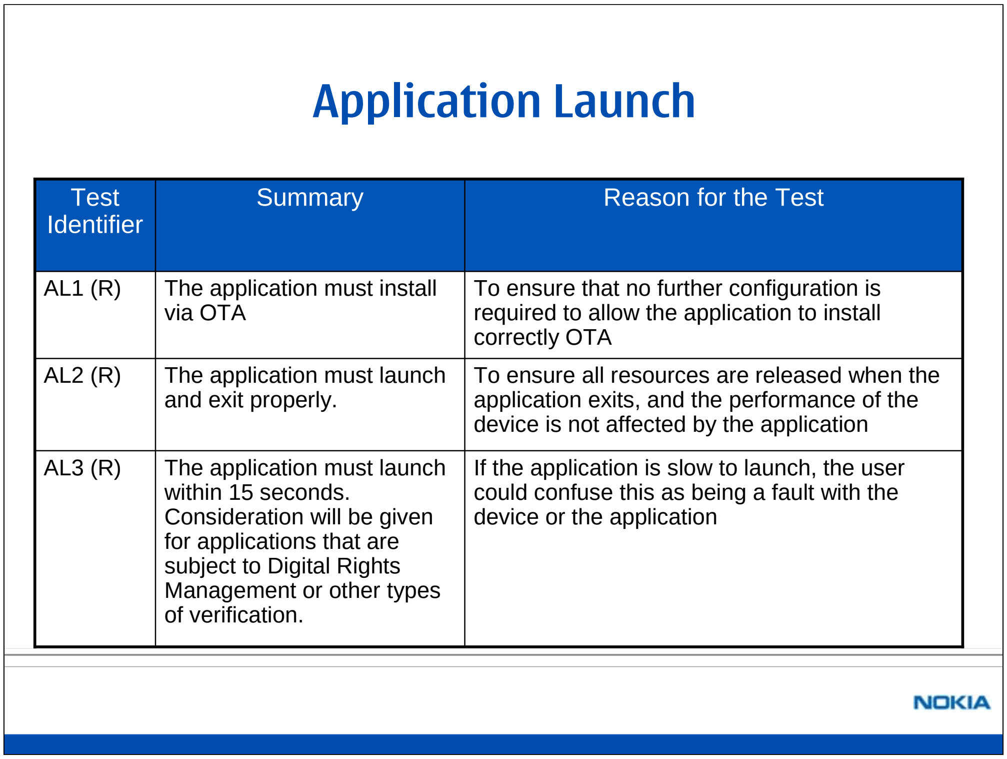
A CSR file needs to be sent to the Certificate Authority (CA) to obtain a certificate. You can do this from the tools by selecting an alias for signing from the list, then clicking the Generate CSR button.

Importing the public key certificate

The CA will send you a signed RSA X.509v3 certificate for your public key after you have sent the generated CSR. To import the certificate, select the corresponding alias, and click the Import Certificate button.

Signing

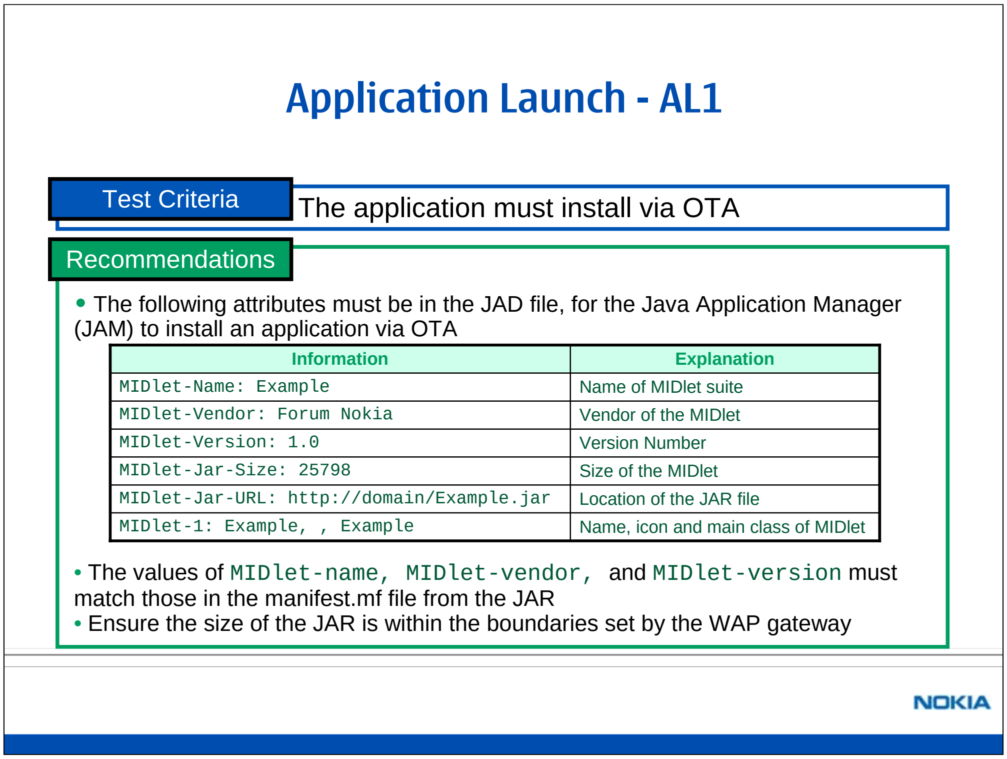
To actually sign an application, select an alias for the key pair you wish to use and press the Sign button. This will bring up a file chooser dialog, from which you must select the MIDlet Application Package's JAD file. The private key and the public key certificate corresponding to



Application Launch

Test Identifier	Summary	Reason for the Test
AL1 (R)	The application must install via OTA	To ensure that no further configuration is required to allow the application to install correctly OTA
AL2 (R)	The application must launch and exit properly.	To ensure all resources are released when the application exits, and the performance of the device is not affected by the application
AL3 (R)	The application must launch within 15 seconds. Consideration will be given for applications that are subject to Digital Rights Management or other types of verification.	If the application is slow to launch, the user could confuse this as being a fault with the device or the application

This category checks that the application is able to start up and shut down correctly. The tests check that the application doesn't take too long to start up, all resources are released when the application is closed and the performance of the device is not affected by the application.



Application Launch - AL1

Test Criteria The application must install via OTA

Recommendations

- The following attributes must be in the JAD file, for the Java Application Manager (JAM) to install an application via OTA

Information	Explanation
`MIDlet-Name: Example`	Name of MIDlet suite
`MIDlet-Vendor: Forum Nokia`	Vendor of the MIDlet
`MIDlet-Version: 1.0`	Version Number
`MIDlet-Jar-Size: 25798`	Size of the MIDlet
`MIDlet-Jar-URL: http://domain/Example.jar`	Location of the JAR file
`MIDlet-1: Example, , Example`	Name, icon and main class of MIDlet

- The values of `MIDlet-name, MIDlet-vendor,` and `MIDlet-version` must match those in the manifest.mf file from the JAR
- Ensure the size of the JAR is within the boundaries set by the WAP gateway

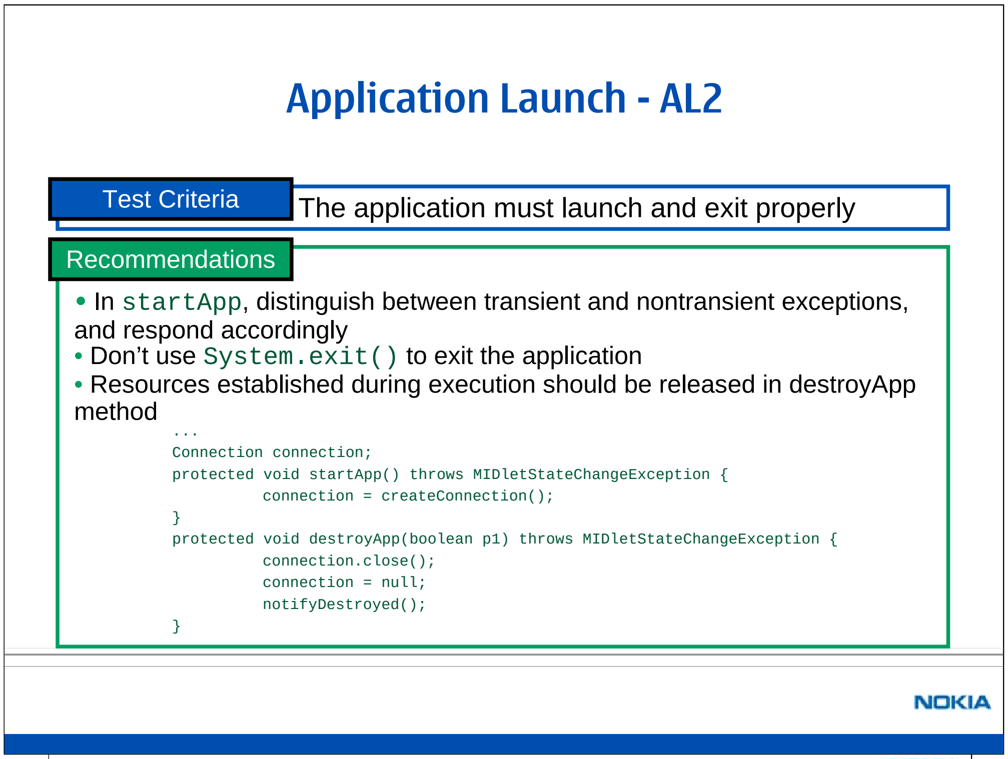
Over-the-Air (OTA) provisioning works by first sending a WAP request from the device for a JAD (Java Application Description) file. The Web server responds by sending the JAD file to the device through the WAP gateway. The Java Application Manager (JAM) uses attributes contained the JAD file to confirm the application can be run on the device, and then uses the URL attribute to download the JAR file from the Web server. The application is then installed on the device, from the JAR file.

To ensure that an application can be installed via OTA, it must include specific attributes in its JAD and manifest file:

- *MIDlet-Name: The name of the MIDlet suite. The value of this attribute must match the value of the same attribute contained within the manifest file of the downloaded JAR.*
- *MIDlet-Vendor: The vendor of the MIDlet suite. The value of this attribute must match the value of the same attribute contained within the manifest file of the downloaded JAR.*
- *MIDlet-Version: This attribute is used to check if there is a newer version of the application available. If an older version of the MIDlet is already installed, notification about updating can then be provided. The value of the attribute must match the value of the same attribute contained within the manifest file of the downloaded JAR.*
- *MIDlet-Jar-Size: Since the JAM has to check the amount of free space and memory available on a device to ensure an application can be installed, the attribute MIDlet-Jar-Size must be included in the JAD file. However, the actual memory required may be larger or smaller depending on the device, and so this value is only actually used as a hint to the JAM. This value is also used to check that the received JAR file size matches the size specified in the JAD.*
- *MIDlet-Jar-URL: To be able to locate the JAR file for the MIDlet suite, the JAM uses the MIDlet-Jar-URL attribute.*
- *MIDlet-1: Name, icon and main class of MIDlet*

The JAR files manifest must contain information about MicroEdition-Profile, MicroEdition-Configuration, and MIDlet-<n>. In MIDlet-<n> attribute, the <n> is the number of a particular MIDlet in the MIDlet suite. There must be a separate MIDlet-<n> attribute present for each MIDlet in the MIDlet suite.

One other requirement, to allow an application to install via OTA, is that of maximum file size. Some WAP gateways have settings for the maximum file size that is allowed to pass through. The size of the JAR file, therefore, needs to be within the boundaries set by the individual WAP gateway configuration. The default value of the WAP gateway configuration should either be altered to allow the JAR file to be downloaded, or the size of the JAR file decreased to fit in with the current settings. To reduce the size of the MIDlet suite, the developer could use an obfuscator tool, such as ProGuard, to remove unused methods and classes and rename methods and variables. Another strategy for reducing the size of your MIDlet suite JAR is to minimize the size of its resource



Application Launch - AL2

Test Criteria: The application must launch and exit properly

Recommendations

- In `startApp`, distinguish between transient and nontransient exceptions, and respond accordingly
- Don’t use `System.exit()` to exit the application
- Resources established during execution should be released in destroyApp method

```
...
Connection connection;
protected void startApp() throws MIDletStateChangeException {
        connection = createConnection();
}
protected void destroyApp(boolean p1) throws MIDletStateChangeException {
        connection.close();
        connection = null;
        notifyDestroyed();
}
```

This test, from a coding point a view, involves making sure that the startApp, pauseApp, and destroyApp methods all act accordingly to guarantee the MIDlet launches and exits correctly, without any detrimental effects to the device. This requires the MIDlet methods to handle errors robustly, free up unused resources and ensure the MIDlet is exited cleanly.

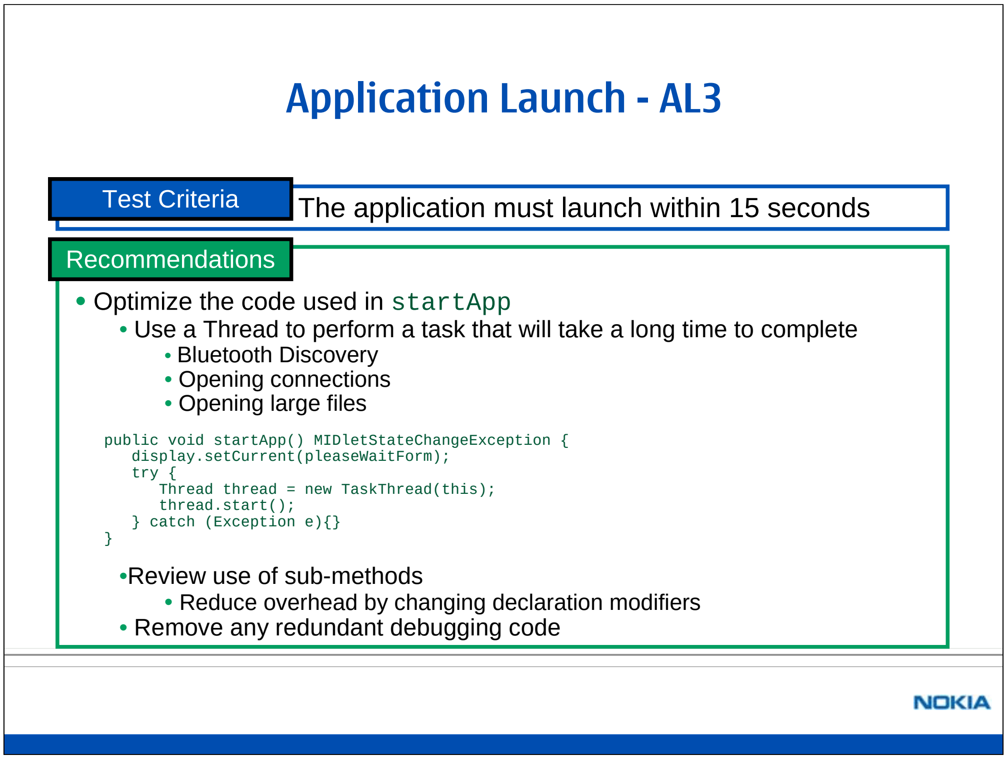
A MIDlet may not launch correctly if a problem occurs in the startApp method. The startApp method can “fail” in two ways: transient and non-transient. A transient failure is one that might have to do with a certain point in time (such as a lack of resources, network connectivity, and so forth). Since this type of error may not be fatal, a MIDlet can tell the system that it would like to try and initialize later by throwing a javax.microedition.midlet.MidletStateChangeException. A non-transient failure is one that is caused by any other unexpected and uncaught runtime error. In this case, if the runtime error is not caught by the startApp method, it is propagated back to the system that will destroy the MIDlet immediately and call the MIDlet’s destroyApp method.

A robust startApp method should distinguish between transient and nontransient exceptions and respond accordingly, as illustrated in the following code fragment:

```
void startApp() throws MIDletStateChangeException {
   HttpConnection c = null;
   int status = -1;
   try {
      c = (HttpConnection)Connector.open(...);
      status = c.getResponseCode();
      ...
   } catch (IOException ioe) {
      // Transient failure: could not connect to the network
      throw new MIDletStateChangeException("Network not available");
   } catch (Exception e) {
destroyApp(true);
      notifyDestroyed();
   }
}
```

The pauseApp method is called by the system to ask a MIDlet to “pause”. Its should work in conjunction with the startApp method to release as many temporary resources as possible and become passive so that there is more memory and/or resources available to other MIDlets or native applications. The use of the pauseApp method is illustrated in the following code fragment:

```
boolean firstTime= false;
int[] runTimeArray = null;
   ...
   void startApp() {
      runTimeArray = new int[200];
      if (firstTime) {
...
         firstTime = false;
      } else {
```



Application Launch - AL3

Test Criteria | The application must launch within 15 seconds

Recommendations

- Optimize the code used in `startApp`
  - Use a Thread to perform a task that will take a long time to complete
    - Bluetooth Discovery
    - Opening connections
    - Opening large files

```
public void startApp() MIDletStateChangeException {
   display.setCurrent(pleaseWaitForm);
   try {
      Thread thread = new TaskThread(this);
      thread.start();
   } catch (Exception e){}
}
```

- •Review use of sub-methods
  - Reduce overhead by changing declaration modifiers
- Remove any redundant debugging code

There are many factors that could affect how long a MIDlet takes to actually start up.

In-device verification may affect the launch of a MIDlet. The KVM class file verifier carries this out on the MIDlet byte code. The verifier analyzes the byte code to make sure the instructions are valid and allocates enough memory for storing the types of all local variables used in MIDlet.

Certification checking on device may also affect the launch of a MIDlet.

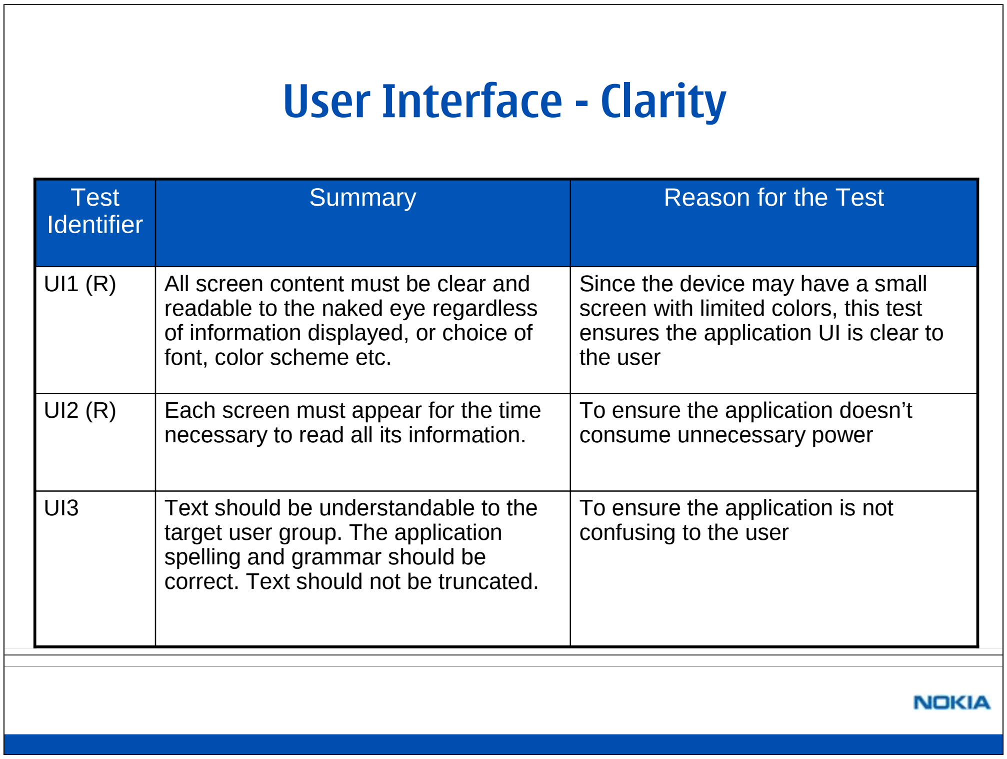
Code that is executed in the constructor of MIDlet will contribute to the start up time.

The startApp method code will also affect how long it takes for the MIDlet to launch completely.

Since it is specified, in this particular test, that consideration will be given for applications that are subject to Digital Rights Management or other types of verification, we don't need to worry about the in-device verification and certificate-checking factors when considering the time it takes for the MIDlet to launch. Our focus should be the code that executes in the constructor and startApp method of the MIDlet. Since these are the first methods to be executed when a MIDlet is launched, they are a major factor that will affect the start up time of the MIDlet.

Like every Java class, a MIDlet can have a constructor. In the MIDP application model, the system calls the public no-argument constructor of a MIDlet exactly once to instantiate the MIDlet. The functionality that needs to be defined in the constructor depends on how the MIDlet is written, but in general, any operations that must be performed only once when the application is launched should be placed in the constructor. These operations could be creating instances of the Command objects, or getting a Display object and setting it to an instance variable.

At some point after construction, the Application Management Software (AMS) activates the MIDlet and invokes its startApp() method. The code within this method should be analyzed carefully to ensure the start up time of the MIDlet is kept to a minimum.

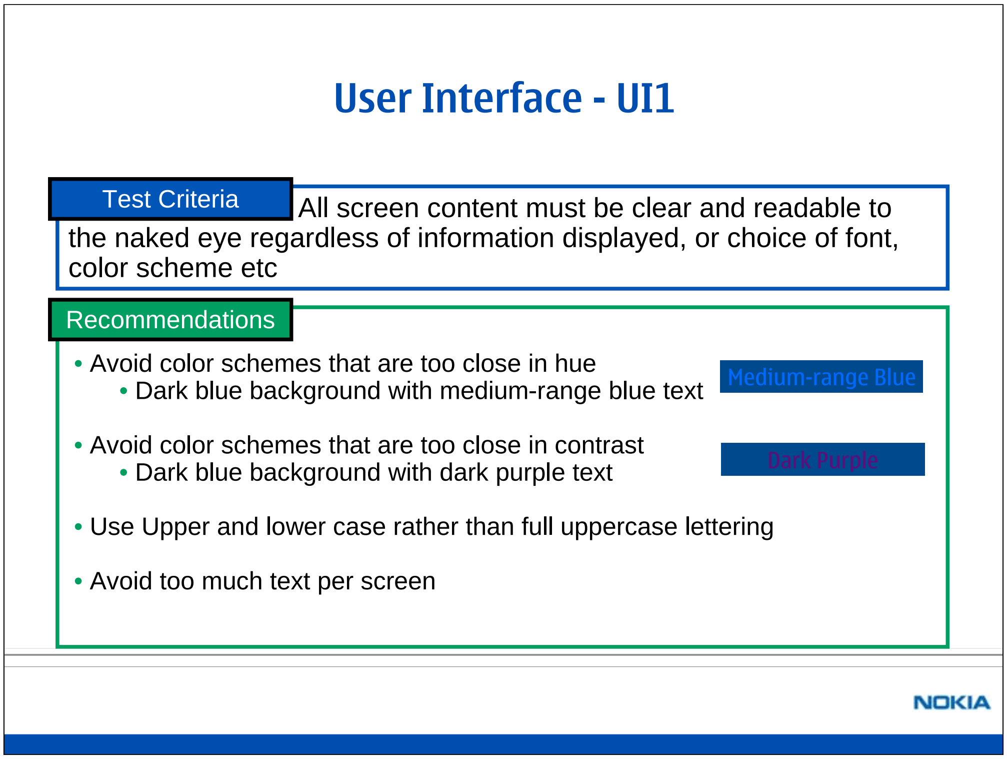


User Interface - Clarity

Test Identifier	Summary	Reason for the Test
UI1 (R)	All screen content must be clear and readable to the naked eye regardless of information displayed, or choice of font, color scheme etc.	Since the device may have a small screen with limited colors, this test ensures the application UI is clear to the user
UI2 (R)	Each screen must appear for the time necessary to read all its information.	To ensure the application doesn’t consume unnecessary power
UI3	Text should be understandable to the target user group. The application spelling and grammar should be correct. Text should not be truncated.	To ensure the application is not confusing to the user

NOKIA

The category ensures the application should aim for visual and conceptual clarity. The user interface should be clear to the user, and not be visually confusing. This is especially important, as the screen size is relatively small. The application should not clutter the screen with controls, unnecessarily. Dialogs and messages should be clear and to the point conveying the appropriate information in as simple a method as possible. Messages should not just indicate what the user has done wrong, but also what the user needs to do to correct the problem, if the screen real estate permits.



User Interface - UI1

Test Criteria

All screen content must be clear and readable to the naked eye regardless of information displayed, or choice of font, color scheme etc

Recommendations

- Avoid color schemes that are too close in hue
  - Dark blue background with medium-range blue text

Medium-range Blue

- Avoid color schemes that are too close in contrast
  - Dark blue background with dark purple text

Dark Purple

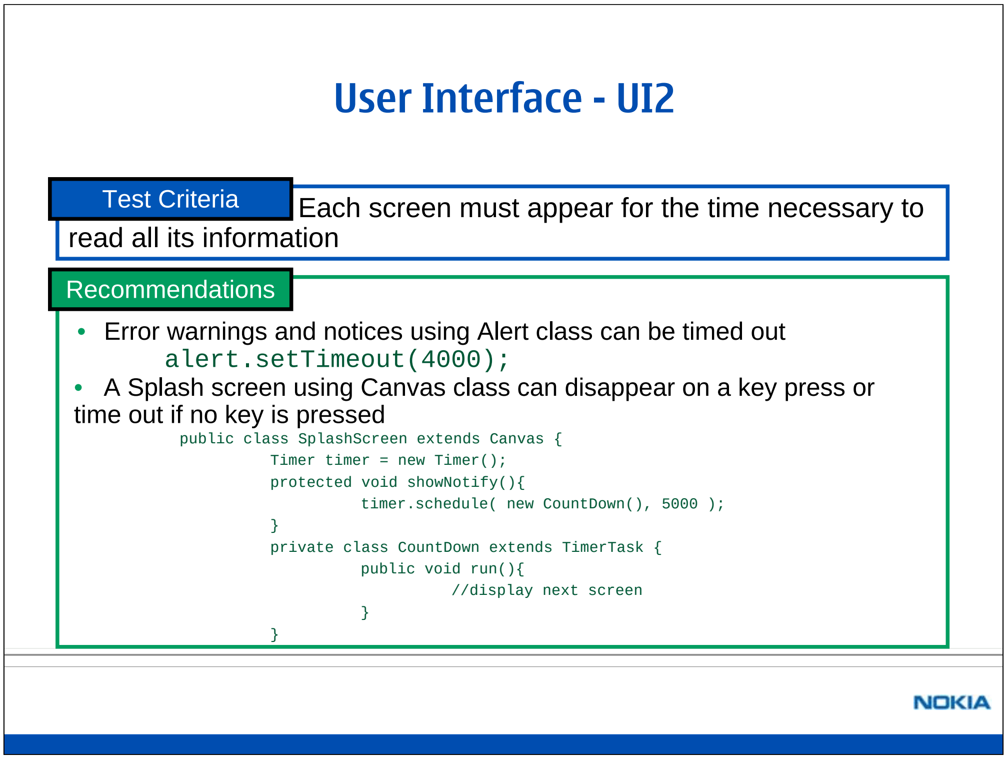
- Use Upper and lower case rather than full uppercase lettering
- Avoid too much text per screen

NOKIA

The test verifies that all screen content is readable. All text must be readable to the naked eye (i.e. 20x20 vision) regardless of the chosen font or color scheme. Care must be taken to avoid visually confusing selection, scrolling, and text highlighting. All elements must provide appropriate contrast for readability.

When choosing your color scheme for your application, avoid colors that are close in hue, i.e. the shade of the color. In the slide, there is an example of a screen that uses two colors that are close in hue, dark blue using g.setColor(0, 66, 134) and medium range blue using g.setColor(0, 80, 154). It can be seen that since these colors are so close in shade, it is difficult to see the text from the background.

You should also avoid a color scheme that includes colors that are close in contrast, i.e. the brightness/darkness of the color. In the slide, these is an example of a screen that uses two colors that are close in contrast, dark blue using g.setColor(0, 66, 134) and dark purple using g.setColor(66, 24, 123). It can be seen that since these colors are both relative dark colors, it is difficult to see the test from the background.



User Interface - UI2

Test Criteria Each screen must appear for the time necessary to read all its information

Recommendations

- Error warnings and notices using Alert class can be timed out
  `alert.setTimeout(4000);`
- A Splash screen using Canvas class can disappear on a key press or time out if no key is pressed

```
public class SplashScreen extends Canvas {
        Timer timer = new Timer();
        protected void showNotify(){
                timer.schedule( new CountDown(), 5000 );
        }
        private class CountDown extends TimerTask {
                public void run(){
                        //display next screen
                }
        }
```

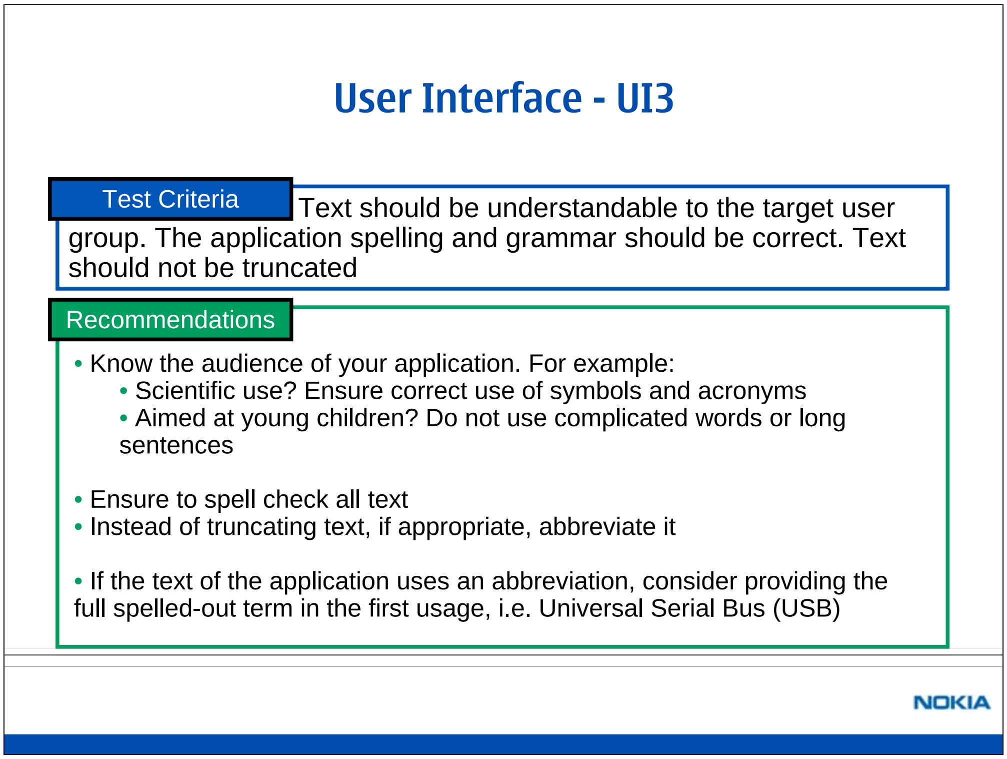
An alert is basically a message dialog, that is, a way of presenting a single string to the user (along with an optional image or sound). The developer, depending on the nature of the usage, should decide Timeouts for these alerts that display splash screens, warnings, errors, alarms, notices, or confirmations. They also can be used for splash screens. The alert then either times out automatically (the timeout value is programmable) or remains on screen until the user explicitly dismisses it.

```
import javax.microedition.lcdui.*;
Image icon = ...; // code omitted
Alert msg = new Alert( "Error!" );
msg.setImage( icon );
msg.setString( "No connection was possible." );
msg.setTimeout( 5000 ); // in milliseconds
msg.setType( AlertType.ERROR );
```

The timeout value defaults to a system-specific value, which you can determine at runtime by calling the Alert.getDefaultTimeout method. If you specify a timeout value, it should be the number of milliseconds to display the alert, or the special value Alert.FOREVER for a modal alert, which allows the user to decide when the alert should disappear. It is suggested that the Alerts should be displayed on screen for no more than 5 seconds, or should disappear when the user presses a softkey.

If your application does include a splash screen, it should also disappear after a set period of time. The splash screen should be displayed for around than 4 seconds. For example, if you are using an Alert for the splash screen, you should call the method setTimeOut on the Alert object. If you are using a Canvas to display a splash screen, you can dismiss the splash screen with any key or automatically after a set period of time.

To dismiss the splash screen after a period of timer, you should use a Timer and a TimerTask. You can schedule a task to run after 5 seconds of displaying the Canvas that changes the display to the next required screen.



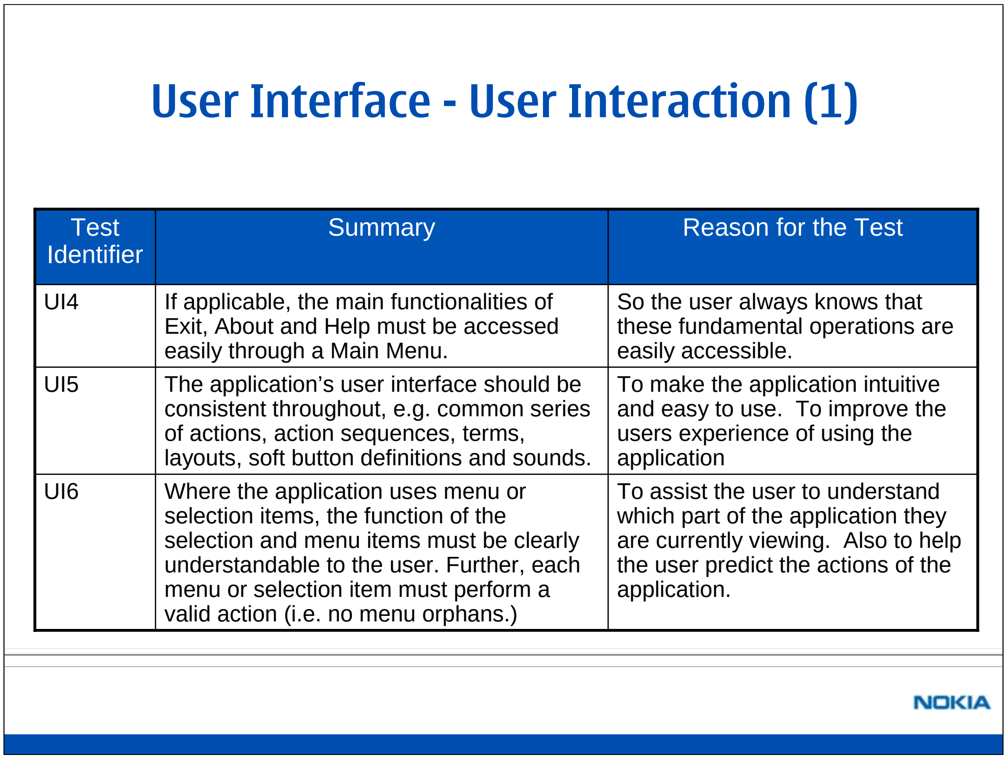
User Interface - UI3

Test Criteria Text should be understandable to the target user group. The application spelling and grammar should be correct. Text should not be truncated

Recommendations

- Know the audience of your application. For example:
    - Scientific use? Ensure correct use of symbols and acronyms
    - Aimed at young children? Do not use complicated words or long sentences
- Ensure to spell check all text
- Instead of truncating text, if appropriate, abbreviate it
- If the text of the application uses an abbreviation, consider providing the full spelled-out term in the first usage, i.e. Universal Serial Bus (USB)

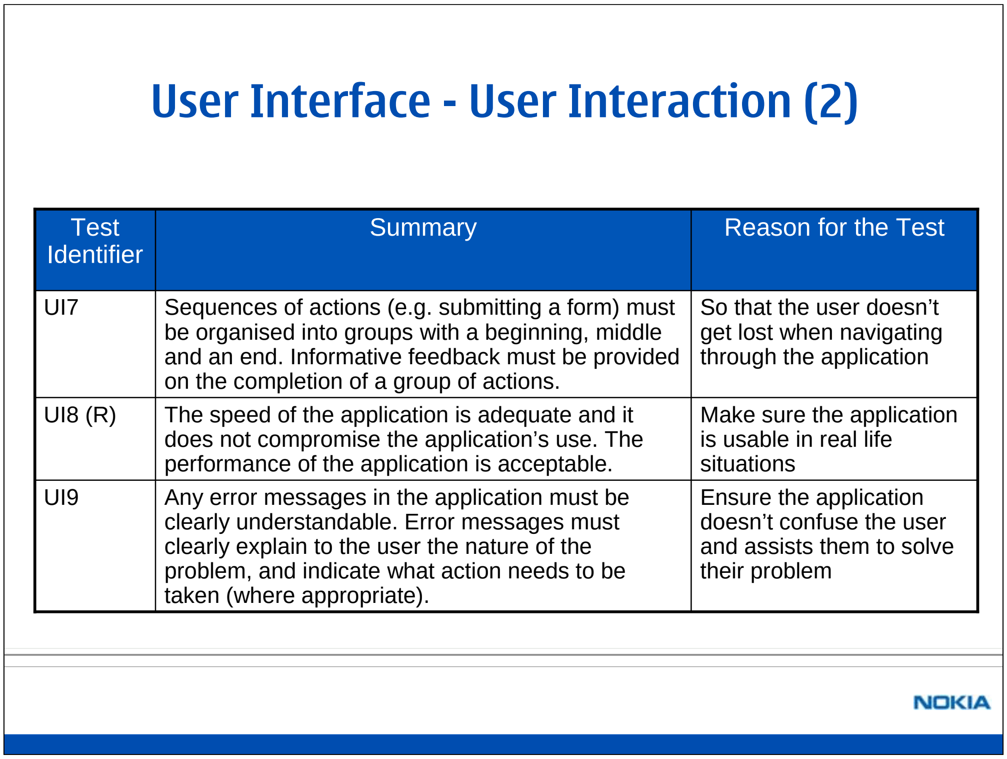
The test verifies that all text and/or numeric messages in dialog boxes and in the main application, including soft button indicators and field labels, are clear, and understandable to the target user.



User Interface - User Interaction (1)

Test Identifier	Summary	Reason for the Test
UI4	If applicable, the main functionalities of Exit, About and Help must be accessed easily through a Main Menu.	So the user always knows that these fundamental operations are easily accessible.
UI5	The application's user interface should be consistent throughout, e.g. common series of actions, action sequences, terms, layouts, soft button definitions and sounds.	To make the application intuitive and easy to use. To improve the users experience of using the application
UI6	Where the application uses menu or selection items, the function of the selection and menu items must be clearly understandable to the user. Further, each menu or selection item must perform a valid action (i.e. no menu orphans.)	To assist the user to understand which part of the application they are currently viewing. Also to help the user predict the actions of the application.

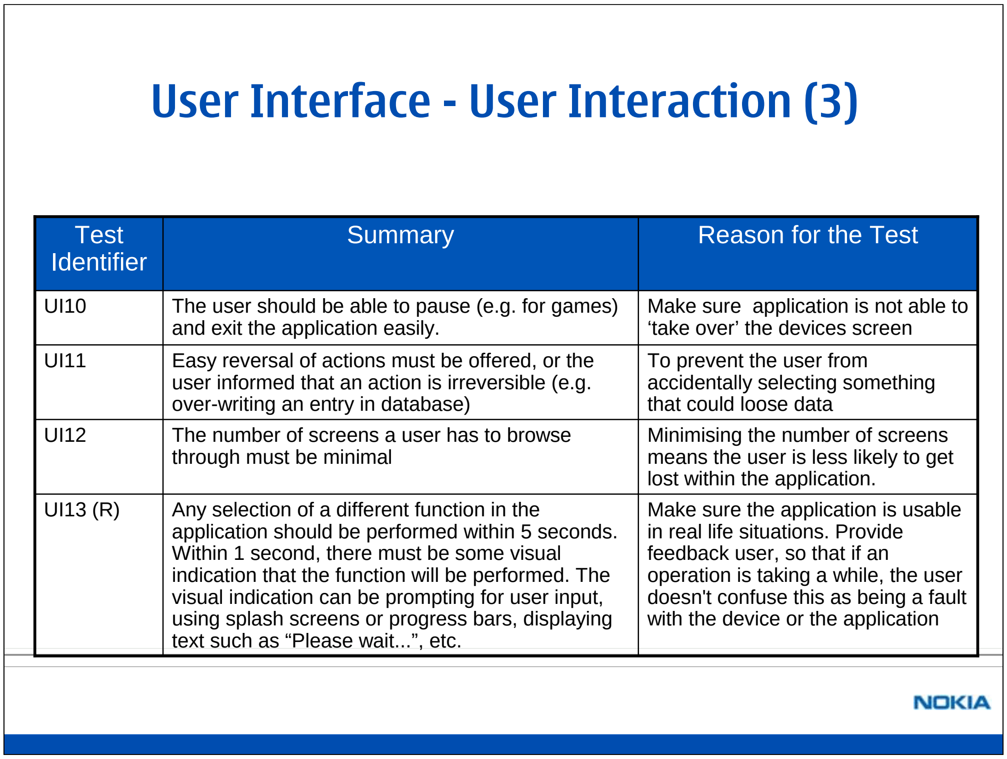
This category is to make sure the user does not get confused when using the application. It suggests trying to make the application intuitive and easy to use. Visual feedback should be given to the user when possible to show which parts of the application they are currently viewing, and how they can navigate between the different sections of the applications. Also covered in this category is ensuring that when a problem has occurred in the application, the user should be properly informed with a clear error message, detailing the problem and a possible solution.



User Interface - User Interaction (2)

Test Identifier	Summary	Reason for the Test
UI7	Sequences of actions (e.g. submitting a form) must be organised into groups with a beginning, middle and an end. Informative feedback must be provided on the completion of a group of actions.	So that the user doesn't get lost when navigating through the application
UI8 (R)	The speed of the application is adequate and it does not compromise the application's use. The performance of the application is acceptable.	Make sure the application is usable in real life situations
UI9	Any error messages in the application must be clearly understandable. Error messages must clearly explain to the user the nature of the problem, and indicate what action needs to be taken (where appropriate).	Ensure the application doesn't confuse the user and assists them to solve their problem

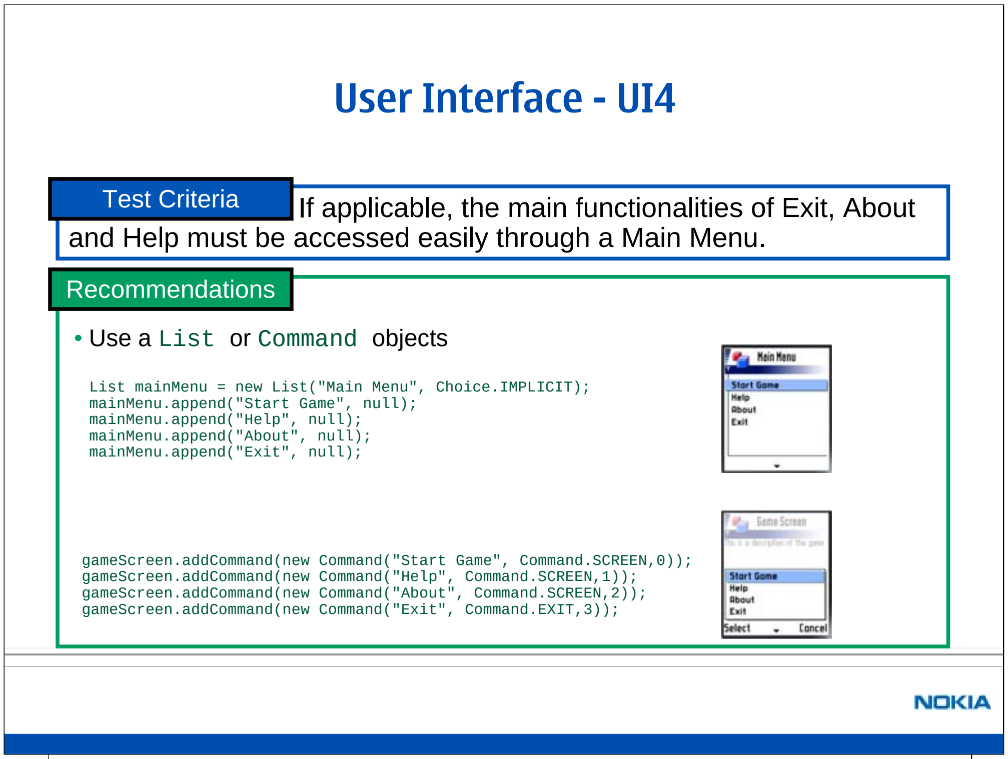
The tests make sure the application is easy to use and can be used practically, i.e. it is fast enough to be used in real life situations.



User Interface - User Interaction (3)

Test Identifier	Summary	Reason for the Test
UI10	The user should be able to pause (e.g. for games) and exit the application easily.	Make sure application is not able to 'take over' the devices screen
UI11	Easy reversal of actions must be offered, or the user informed that an action is irreversible (e.g. over-writing an entry in database)	To prevent the user from accidentally selecting something that could loose data
UI12	The number of screens a user has to browse through must be minimal	Minimising the number of screens means the user is less likely to get lost within the application.
UI13 (R)	Any selection of a different function in the application should be performed within 5 seconds. Within 1 second, there must be some visual indication that the function will be performed. The visual indication can be prompting for user input, using splash screens or progress bars, displaying text such as "Please wait...", etc.	Make sure the application is usable in real life situations. Provide feedback user, so that if an operation is taking a while, the user doesn't confuse this as being a fault with the device or the application

The tests make sure the user is properly informed at every stage of application. Feedback in terms of wait screens and confirmation boxes should be displayed when appropriate.



User Interface - UI4

Test Criteria If applicable, the main functionalities of Exit, About and Help must be accessed easily through a Main Menu.

Recommendations

- Use a `List` or `Command` objects

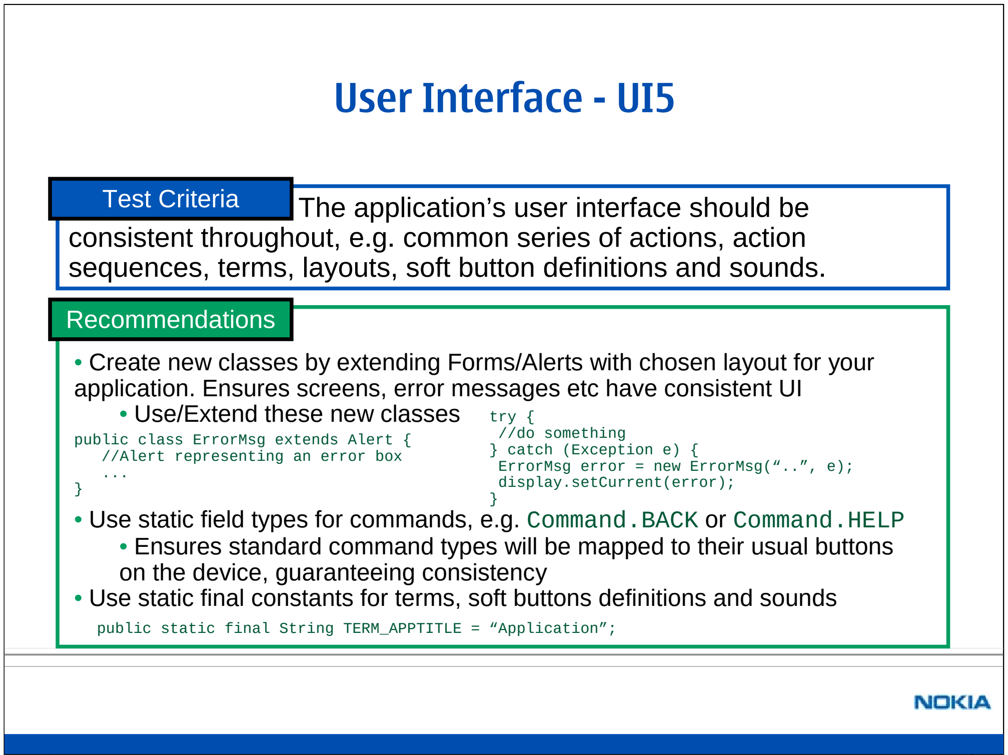
```
List mainMenu = new List("Main Menu", Choice.IMPLICIT);
mainMenu.append("Start Game", null);
mainMenu.append("Help", null);
mainMenu.append("About", null);
mainMenu.append("Exit", null);
```

```
gameScreen.addCommand(new Command("Start Game", Command.SCREEN,0));
gameScreen.addCommand(new Command("Help", Command.SCREEN,1));
gameScreen.addCommand(new Command("About", Command.SCREEN,2));
gameScreen.addCommand(new Command("Exit", Command.EXIT,3));
```

To pass this particular test you need to ensure that your application can easily access the fundamental functions Exit, About and Help. The example in slide shows how this can be done using a List or a series of Command objects.

If you implement a main menu using a List, you must ensure that all the screens that can be accessed from it, can navigate back to the main menu. The code below shows an example of how a MIDlet could be structured using a List for a main menu.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
public class MainMenuMIDlet extends MIDlet implements CommandListener {
  private Display display;
  private List mainMenu;
  private static final String START_GAME = "Start Game";
  private static final String HELP = "Help";
  private static final String ABOUT = "About";
  private static final String EXIT = "Exit";
  private static final String MENU = "Menu";
  private Command mainMenuCommand = new Command(MENU, Command.SCREEN, 0);
  protected void startApp() throws MIDletStateChangeException {
    this.display = Display.getDisplay(this);
    mainMenu = new List("Main Menu", Choice.IMPLICIT);
    mainMenu.append(START_GAME, null);
    mainMenu.append(HELP, null);
    mainMenu.append(ABOUT, null);
    mainMenu.append(EXIT, null);
    mainMenu.setCommandListener(this);
    display.setCurrent(mainMenu);
  }
  public void commandAction(Command command, Displayable displayable) {
    int selected = mainMenu.getSelectedIndex();
    String selectedString = mainMenu.getString(selected);
    if (command == mainMenuCommand) {
      display.setCurrent(mainMenu);
    } else {
      if (selectedString.equals(START_GAME)) {
        GameScreen game = new GameScreen();
        game.addCommand(mainMenuCommand);
        game.setCommandListener(this);
        display.setCurrent(game);
      } else if (selectedString.equals(HELP)) {
          HelpScreen help = new HelpScreen();
        help.addCommand(mainMenuCommand);
        help.setCommandListener(this);
```



User Interface - UI5

Test Criteria

The application's user interface should be consistent throughout, e.g. common series of actions, action sequences, terms, layouts, soft button definitions and sounds.

Recommendations

- Create new classes by extending Forms/Alerts with chosen layout for your application. Ensures screens, error messages etc have consistent UI
  - Use/Extend these new classes

```
public class ErrorMsg extends Alert {
   //Alert representing an error box
   ...
}
```

```
try {
 //do something
} catch (Exception e) {
 ErrorMsg error = new ErrorMsg(“..”, e);
 display.setCurrent(error);
}
```

- Use static field types for commands, e.g. `Command.BACK` or `Command.HELP`
  - Ensures standard command types will be mapped to their usual buttons on the device, guaranteeing consistency
- Use static final constants for terms, soft buttons definitions and sounds

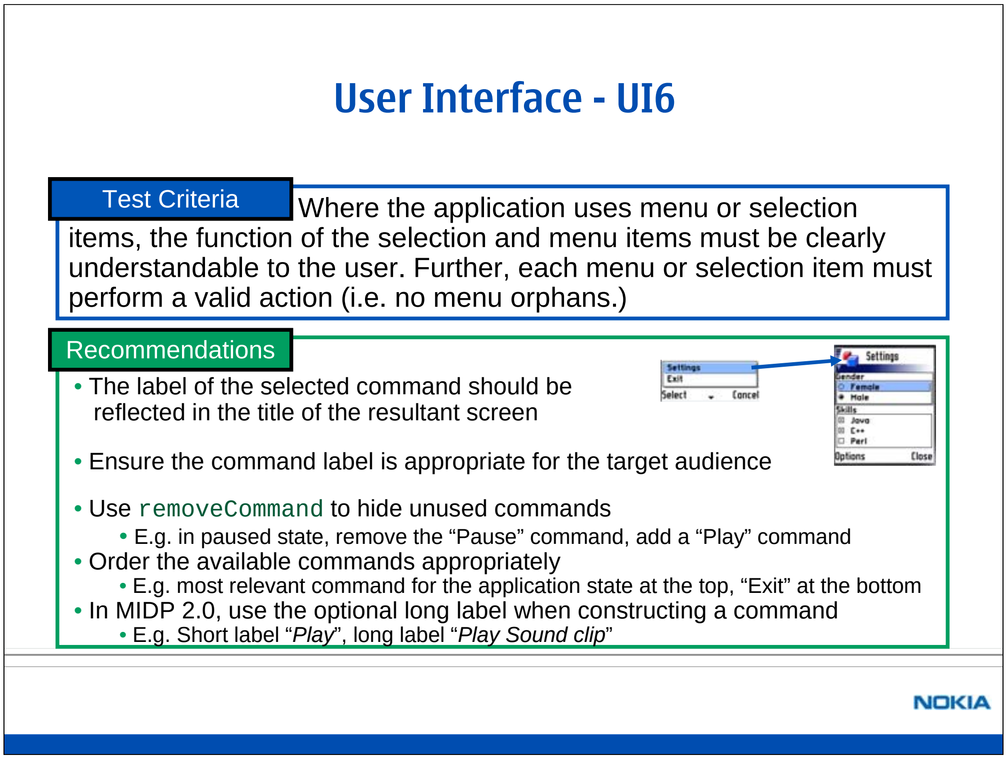
```
public static final String TERM_APPTITLE = “Application”;
```

The test is to ensure the application is intuitive and easy to use. To make sure soft button definitions and any terms used throughout your application are consistent, it is a good idea to use static final constants. For example, if the application uses Lists on different screens, but with common items, the name of the common items can be set as a static final string

```
private static final String START_GAME = "Start Game";
private static final String HELP = "Help";
private static final String ABOUT = "About";
private static final String EXIT = "Exit";
private static final String MENU = "Menu";
protected void startApp() throws MIDletStateChangeException {
    this.display = Display.getDisplay(this);
    List list1 = new List (MENU + “1”, Choice.IMPLICIT);
    list1.append(START_GAME, null);
    list1.append(HELP, null);
    list1.append(ABOUT, null);
    list1.append(EXIT, null);
    List list2 = new List (MENU + “2”, Choice.IMPLICIT);
    list2.append(ABOUT, null);
    list2.append(EXIT, null);
   ...
  }
```

To ensure consistency within screens in your application, you could create a new “base screen” class that includes the items you require on each screen, e.g. a logo, a description, certain command buttons etc. Then when creating a new screen for you application, to ensure it is consistent with the other screens, you just need to extend your “base screen” class. For example, if we had a class BaseScreen that creates a form with a logo, a ticker and an exit button

```
public class BaseScreen
    extends Form {
  private static final String EXIT = "Exit";
  private Command exitCommand = new Command(EXIT, Command.EXIT, 99);
  private Ticker descTicker;
  private ImageItem logo;
  public BaseScreen(String s, String desc) {
    super(s);
    Image logoImage = null;
    try {
      logoImage = Image.createImage("/RES/logo.png");
    }
    catch (IOException ex) {
    }
    logo = new ImageItem("", logoImage,
                    ImageItem.LAYOUT_CENTER | ImageItem.LAYOUT_NEWLINE_AFTER,
                    "");
    descTicker = new Ticker(desc);
    append(logo);
```



User Interface - UI6

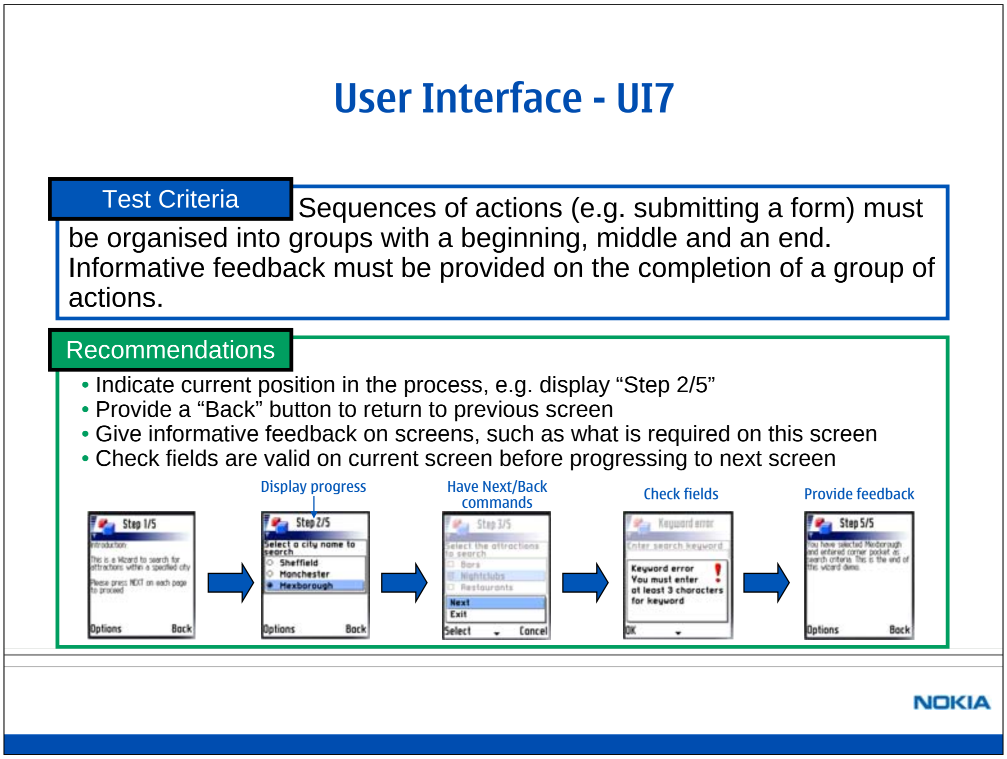
Test Criteria

Where the application uses menu or selection items, the function of the selection and menu items must be clearly understandable to the user. Further, each menu or selection item must perform a valid action (i.e. no menu orphans.)

Recommendations

- The label of the selected command should be reflected in the title of the resultant screen
- Ensure the command label is appropriate for the target audience
- Use `removeCommand` to hide unused commands
  - E.g. in paused state, remove the “Pause” command, add a “Play” command
- Order the available commands appropriately
  - E.g. most relevant command for the application state at the top, “Exit” at the bottom
- In MIDP 2.0, use the optional long label when constructing a command
  - E.g. Short label “*Play*”, long label “*Play Sound clip*”

Each command includes a label string. The label string is what the application requests to be shown to the user to represent this Command. For example, this string may appear next to a soft button on the device or as an element in a menu. For command types other than SCREEN, this label may be overridden by a system-specific label that is more appropriate for this command on this device. The contents of the label string are otherwise not interpreted by the implementation.



User Interface - UI7

Test Criteria Sequences of actions (e.g. submitting a form) must be organised into groups with a beginning, middle and an end. Informative feedback must be provided on the completion of a group of actions.

Recommendations

- Indicate current position in the process, e.g. display “Step 2/5”
- Provide a “Back” button to return to previous screen
- Give informative feedback on screens, such as what is required on this screen
- Check fields are valid on current screen before progressing to next screen

This is a Wizard to search for attractions within a specified city

Please press NEXT on each page to proceed

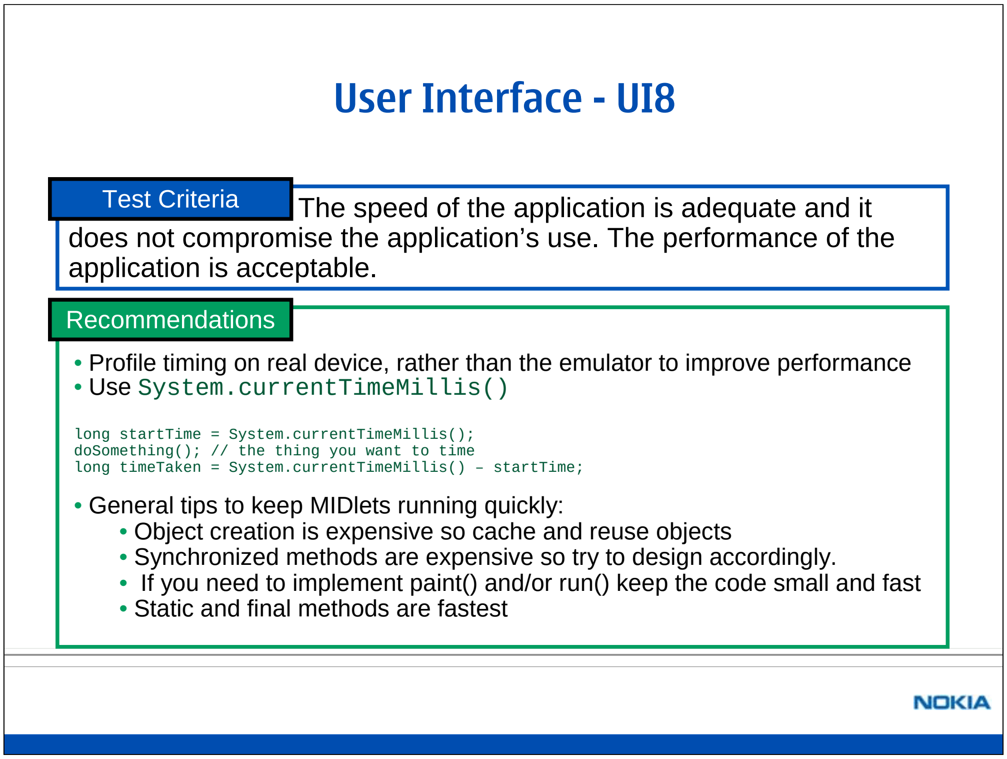
Select a city name to search

Select the attractions to search

Keyword error
You must enter at least 3 characters for keyword

You have selected Mexborough and entered corner pocket as search criteria. This is the end of this wizard demo.

This test is describing the elements that should be included on screens when creating a “Wizard” type application. In a wizard application, users are prompted through a series of screens. Each screen asks one or two questions. Two actions are allowed in wizard applications: Next and Back actions. Users press the Next button to proceed to the next screen. Users press the Back button to return to the previous screen and alter a previous answer.



User Interface - UI8

Test Criteria

The speed of the application is adequate and it does not compromise the application’s use. The performance of the application is acceptable.

Recommendations

- Profile timing on real device, rather than the emulator to improve performance
- Use `System.currentTimeMillis()`

```
long startTime = System.currentTimeMillis();
doSomething(); // the thing you want to time
long timeTaken = System.currentTimeMillis() – startTime;
```

- General tips to keep MIDlets running quickly:
  - Object creation is expensive so cache and reuse objects
  - Synchronized methods are expensive so try to design accordingly.
  - If you need to implement paint() and/or run() keep the code small and fast
  - Static and final methods are fastest

NOKIA

To optimize the performance of your MIDlet, you might want to do some profiling. One thing to remember is that any profile tool that runs on the emulator can only determine measurement for the emulator and not the device itself. So profiling on the emulator isn't really that useful other than to give you an indication of function call coverage. Real profiling should be done on a real device.

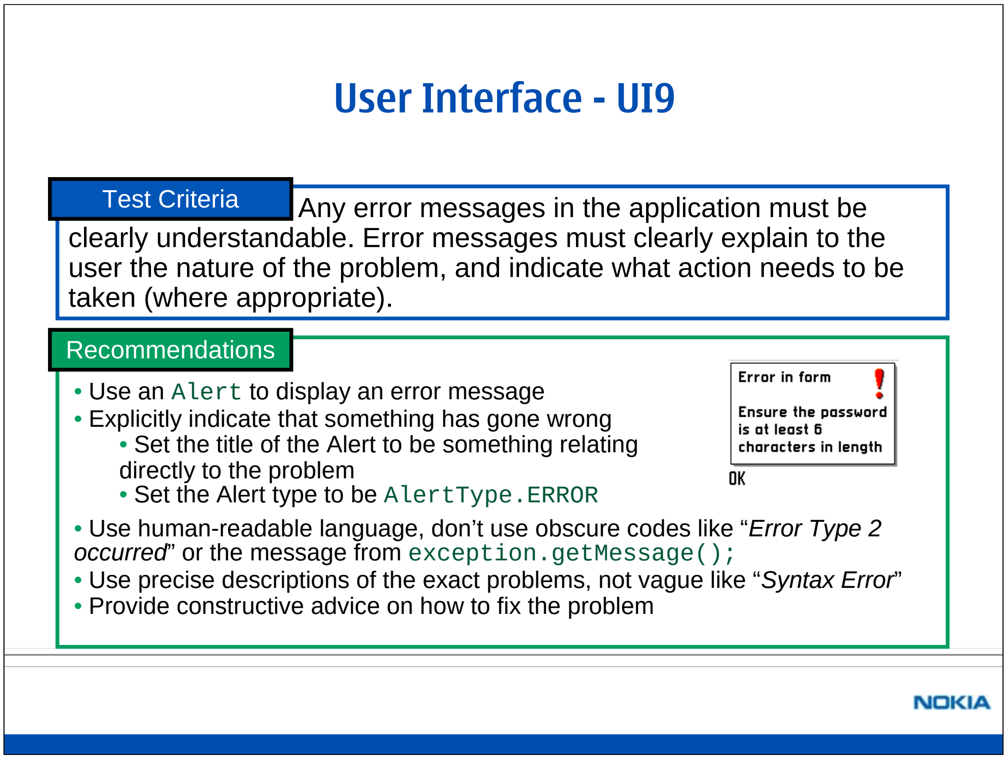
In the case of performance profiling, this can be accomplished by using the System.currentTimeMillis() call to find out how fast parts of your program are running. This will give you the time in milliseconds. To avoid varying results due to garbage collection during your test, you may want to call System.gc() before starting the test. To display the test measurements, use a special MIDlet screen or overwrite the results on the normal screen display. Make sure to check the resolution of the phone's system clock. Its returned values may not have millisecond precision, so note if the returned value is, for example, always a multiple of ten, and make sure your test takes long enough that this isn't a problem.

If you remember a few key points about performance your MIDlets will run quickly from the start:

- As a rule of thumb about 50% of the execution time is going to be spent in the paint() and run() methods if you implement those methods. So if you can make your MIDlet single threaded and avoid user drawing you'll be better off.
- Cache values like crazy when you can.
- The KVM is pretty fast at internal stuff like crunching numbers but slow at making native system calls -- e.g. all the low-level graphics calls dispatch to native calls.
- Pre-calculate values
- Structure your logic so you don't force repaints too often
- Only repaint what's needed
- drawString is unreasonably slow use drawChar with a character array or draw to an offscreen Image
- Avoid writing overly elegant OO code

Some lower-level optimization techniques may also help if you really need the extra performance however be aware that you won't get as much improvements as compared to using better design and smarter algorithms. There are many techniques on low-level optimizations that have been covered countless times and for your curiosity here are a couple of those tips that apply to Java ME:

- Use the StringBuffer class for concatenation if you're concatenating a lot of Strings.
- Declare methods static and final where applicable as they are fastest.
- Avoid synchronized methods as they are slowest
- Unroll loops (don't worry about code size increasing for unrolled loops as the JAR file compresses these well because it's repetitive code)



User Interface - UI9

Test Criteria Any error messages in the application must be clearly understandable. Error messages must clearly explain to the user the nature of the problem, and indicate what action needs to be taken (where appropriate).

Recommendations

- Use an `Alert` to display an error message
- Explicitly indicate that something has gone wrong
  - Set the title of the Alert to be something relating directly to the problem
  - Set the Alert type to be `AlertType.ERROR`
- Use human-readable language, don't use obscure codes like "*Error Type 2 occurred*" or the message from `exception.getMessage();`
- Use precise descriptions of the exact problems, not vague like "*Syntax Error*"
- Provide constructive advice on how to fix the problem

Error in form

Ensure the password is at least 6 characters in length

OK

Good error message should include:

- ***Explicit indication that something has gone wrong. The very worst error messages are those that don't exist. When users make mistakes and get no feedback, they're completely lost.***

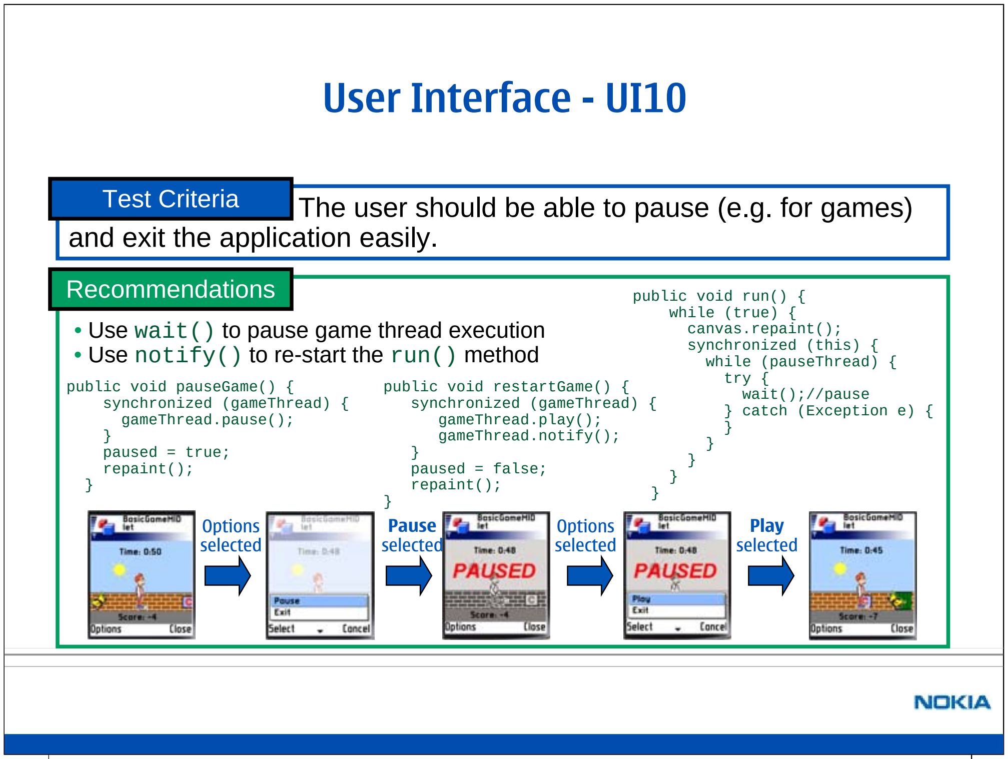
- ***Human-readable language, instead of obscure codes or abbreviations such as "an error of type 2 has occurred."***

- ***Polite phrasing that doesn't blame users or imply that they are either stupid or doing something wrong, as in "illegal command."***

- ***Precise descriptions of exact problems, rather than vague generalities such as "syntax error."***

- ***Constructive advice on how to fix the problem. For example, instead of saying "out of stock," your error message should either tell users when the product will be available or provide a way for users to ask to be notified when the product is restocked.***

To display an error message, in a MIDP application, use an Alert. This class provides a specific type, to be used for error message boxes; this is the AlertType.ERROR type. It is recommended to set the time out for the Alert to be Alert.FOREVER. This way, an OK button is provided the user to dismiss the error box, once they have had time to read and understand the error message.



User Interface - UI10

Test Criteria The user should be able to pause (e.g. for games) and exit the application easily.

Recommendations

- Use `wait()` to pause game thread execution
- Use `notify()` to re-start the `run()` method

```
public void pauseGame() {
    synchronized (gameThread) {
      gameThread.pause();
    }
    paused = true;
    repaint();
  }
```

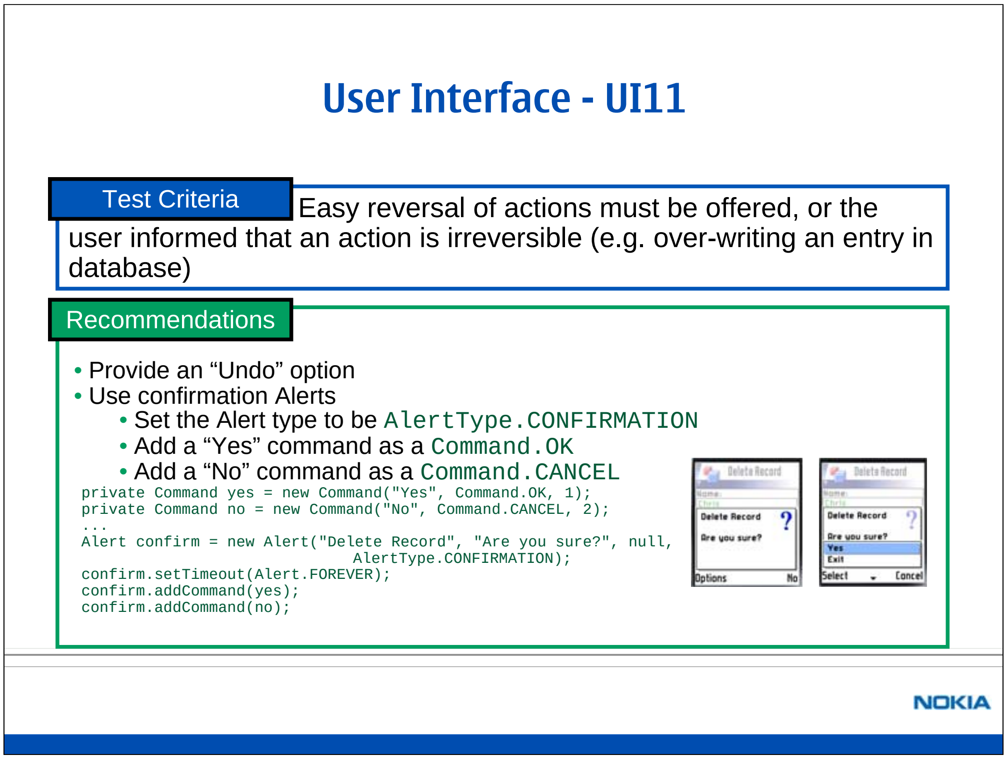
```
public void restartGame() {
    synchronized (gameThread) {
       gameThread.play();
       gameThread.notify();
    }
    paused = false;
    repaint();
}
```

```
public void run() {
    while (true) {
      canvas.repaint();
      synchronized (this) {
        while (pauseThread) {
          try {
            wait();//pause
          } catch (Exception e) {
          }
        }
      }
    }
  }
```

To be able pause a game, the thread that updates the canvas should be paused. This can be done by setting a variable that the game thread checks occasionally. When the game thread detects that the variable is set, it calls Object.wait(). The paused thread can then be woken up by calling its Object.notify() method.

The following code shows this technique. It can be seen that when the pause method is called, a variable pauseThread is set to true. A check is made in the run method. When this variable is true, the run method is calls wait().

```
public class GameThread
    extends Thread {
  private boolean pauseThread = false;
  private GameCanvas canvas;
  public GameThread(GameCanvas canvas) {
    this.canvas = canvas;
  }
  public void pause() {
    this.pauseThread = true;
  }
  public void play() {
    this.pauseThread = false;
  }
  public void run() {
    while (true) {
      canvas.repaint();
      // Check if should wait
      synchronized (this) {
        while (pauseThread) {
          try {
            wait();
          }
          catch (Exception e) {
          }
        }
      }
    }
  }
```



User Interface - UI11

Test Criteria Easy reversal of actions must be offered, or the user informed that an action is irreversible (e.g. over-writing an entry in database)

Recommendations

- Provide an “Undo” option
- Use confirmation Alerts
  - Set the Alert type to be `AlertType.CONFIRMATION`
  - Add a “Yes” command as a `Command.OK`
  - Add a “No” command as a `Command.CANCEL`

```
private Command yes = new Command("Yes", Command.OK, 1);
private Command no = new Command("No", Command.CANCEL, 2);
...
Alert confirm = new Alert("Delete Record", "Are you sure?", null,
                                     AlertType.CONFIRMATION);
confirm.setTimeout(Alert.FOREVER);
confirm.addCommand(yes);
confirm.addCommand(no);
```

When including functions that may result in an action that may be irreversible, such a removing a file, it might be a good idea to present a confirmation screen to the user before the action is executed. This could explain what action will be taken, and give the user the opportunity to not accept the execution of this action. A confirmation box could be implemented using an Alert class, using AlertType.CONFIRMATION as the type. For example

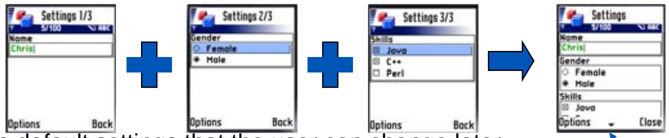
```
private Command yes = new Command("Yes", Command.OK, 1);
private Command no = new Command("No", Command.CANCEL, 2);
Alert confirm = new Alert("Delete Record", "Are you sure?", null,
             AlertType.CONFIRMATION);
confirm.setTimeout(Alert.FOREVER);
confirm.addCommand(yes);
confirm.addCommand(no);
```

User Interface - UI12

Test Criteria The number of screens a user has to browse through must be minimal

Recommendations

- Try to combine information on screens with limited content
  - Don’t put too much information on a screen.Try to limited excessive scrolling of the screen



- Have default settings that the user can change later
  - E.g. a game can be started immediately without specifying settings
- Provide shortcuts to main areas
  - E.g. pressing # during game play pauses the game and takes the user to the settings

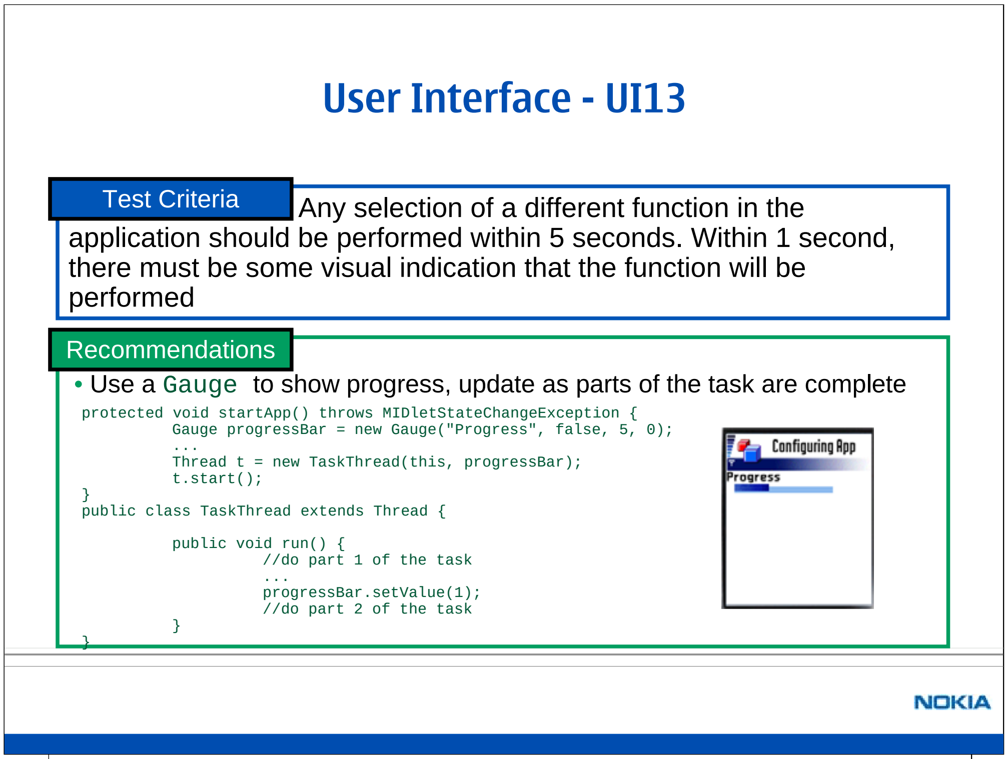
NOKIA

Minimizing the number of screens means the user is less likely to get lost within the application. There are a number of ways in which you can keep the number of screen the user has to page through to a minimum.

One way is to try to combine the information on multiple screens with limited content. For example if you have 3 screens with only 1 item on each, try to combine these screens into 1. Try not to put too much information on a screen, however, as this could result in a very long screen that requires excessive scrolling. Excessive scrolling could cause the application to become slow and cumbersome to use.

Another idea to reduce the amount of screens the user has to page through is to have default settings when the user first starts the application. For example, if the application is a game, the game can be started immediately and then the user can choose to change the settings later.

You could always implement shortcuts in your application, which take the user directly to the main areas of the application, without having to trawl through menus and other screens. For example, you could implement a shortcut in a game that takes the user directly to the settings options.



User Interface - UI13

Test Criteria Any selection of a different function in the application should be performed within 5 seconds. Within 1 second, there must be some visual indication that the function will be performed

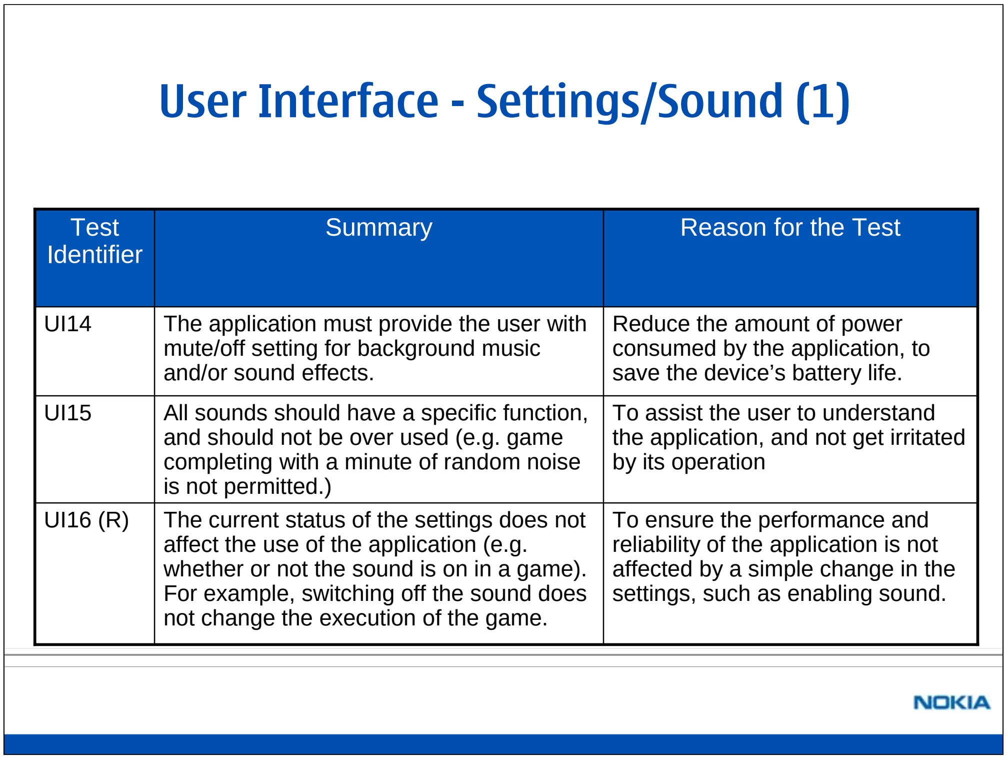
Recommendations

- Use a `Gauge` to show progress, update as parts of the task are complete

```
protected void startApp() throws MIDletStateChangeException {
          Gauge progressBar = new Gauge("Progress", false, 5, 0);
          ...
          Thread t = new TaskThread(this, progressBar);
          t.start();
}
public class TaskThread extends Thread {

          public void run() {
                    //do part 1 of the task
                    ...
                    progressBar.setValue(1);
                    //do part 2 of the task
          }
}
```

If you know that one part of your application may take a while to execute, you should present the user with some visual indication that shows how much of the execution is complete, and how much there is to do. MIDP provides the Gauge class, which can be used especially for this purpose. The code below shows how to use the Gauge class within a separate thread of execution. It can be seen that the MIDlet starts the thread that contains the code that will take a while to execute. The Gauge class is also passed to the thread, so that it can update it, as it does parts of the task.

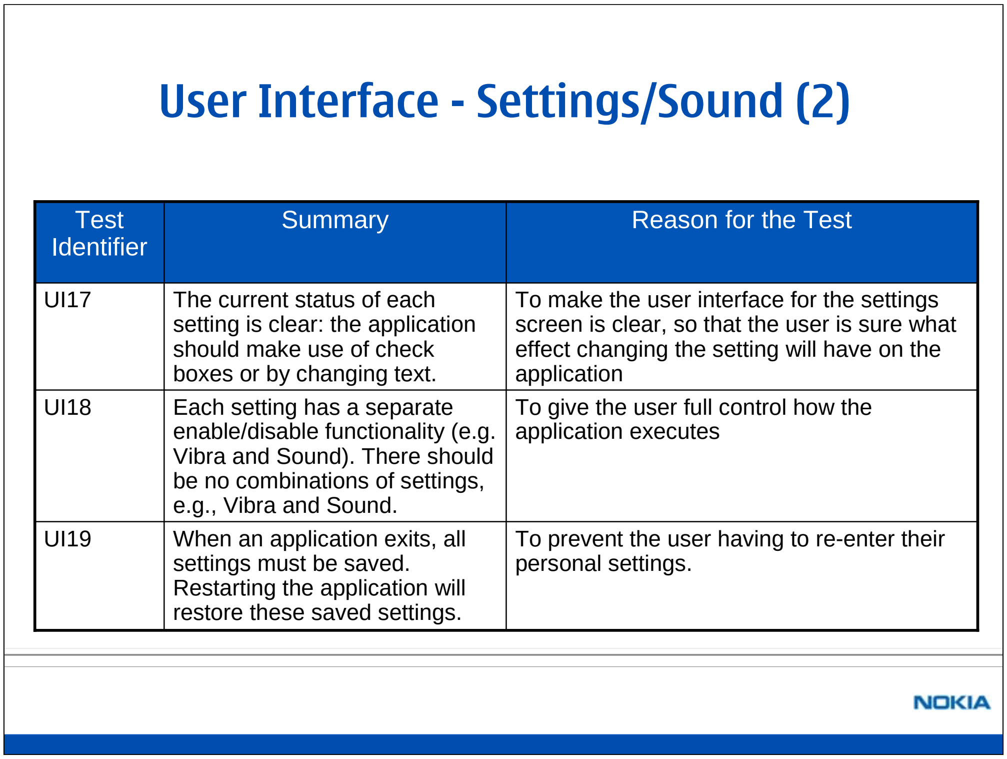
```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ConfigureAppMIDlet
    extends MIDlet {
  Display d;
  public ConfigureAppMIDlet() {
    d = Display.getDisplay(this);
  }
  protected void startApp() throws javax.microedition.midlet.
      MIDletStateChangeException {
    Gauge progressbar = new Gauge("Progress", false, 20, 0);
    Form form = new Form("Configuring App");
    form.append(progressbar);
    d.setCurrent(form);
    Thread t = new MyThread(this, progressbar);
    t.start();
  }
  public void setComplete() {
    d.setCurrent(new Form("Complete"));
  }
  public class MyThread extends Thread {
    private Gauge progressbar;
    private ConfigureAppMIDlet midlet;
    public MyThread(ConfigureAppMIDlet m, Gauge g) {
      progressbar = g;
      midlet = m;
    }
    public void run() {
      for (int i = 0; i < 21; i++) {
        try {Thread.sleep(500);} catch (Exception e){}
        progressbar.setValue(i);
      }
      midlet.setComplete();
    }
  }
  protected void pauseApp() {
    /**@todo Implement this javax.microedition.midlet.MIDlet abstract method*/
  }
  protected void destroyApp(boolean parm1) throws javax.microedition.midlet.
```



User Interface - Settings/Sound (1)

Test Identifier	Summary	Reason for the Test
UI14	The application must provide the user with mute/off setting for background music and/or sound effects.	Reduce the amount of power consumed by the application, to save the device’s battery life.
UI15	All sounds should have a specific function, and should not be over used (e.g. game completing with a minute of random noise is not permitted.)	To assist the user to understand the application, and not get irritated by its operation
UI16 (R)	The current status of the settings does not affect the use of the application (e.g. whether or not the sound is on in a game). For example, switching off the sound does not change the execution of the game.	To ensure the performance and reliability of the application is not affected by a simple change in the settings, such as enabling sound.

The category is to make sure the performance and reliability of the application is not affected by a simple change in the settings, such as enabling sound. It also covers making sure the application does not consumer too much power by having the option turning of the sound and that sound is used to assist the user to understand the application.

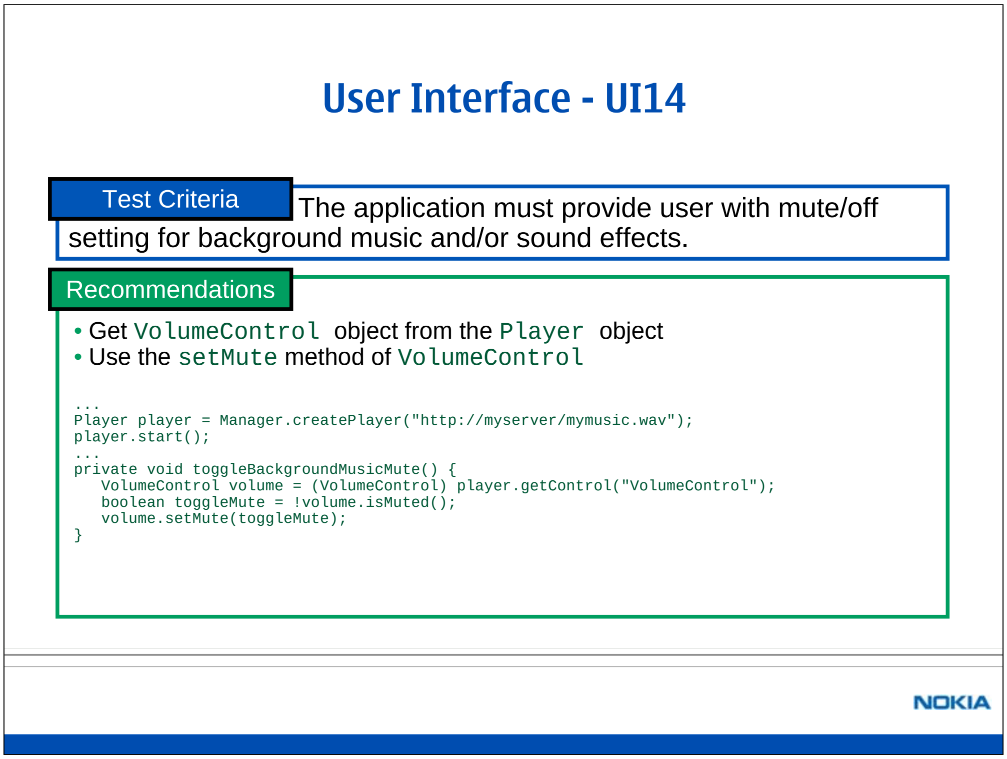


User Interface - Settings/Sound (2)

Test Identifier	Summary	Reason for the Test
UI17	The current status of each setting is clear: the application should make use of check boxes or by changing text.	To make the user interface for the settings screen is clear, so that the user is sure what effect changing the setting will have on the application
UI18	Each setting has a separate enable/disable functionality (e.g. Vibra and Sound). There should be no combinations of settings, e.g., Vibra and Sound.	To give the user full control how the application executes
UI19	When an application exits, all settings must be saved. Restarting the application will restore these saved settings.	To prevent the user having to re-enter their personal settings.

NOKIA

This category also makes sure that the user has full control over how the application executes and the interface for the settings screen is clear, so that the user is sure what effect changing the setting will have on the application.



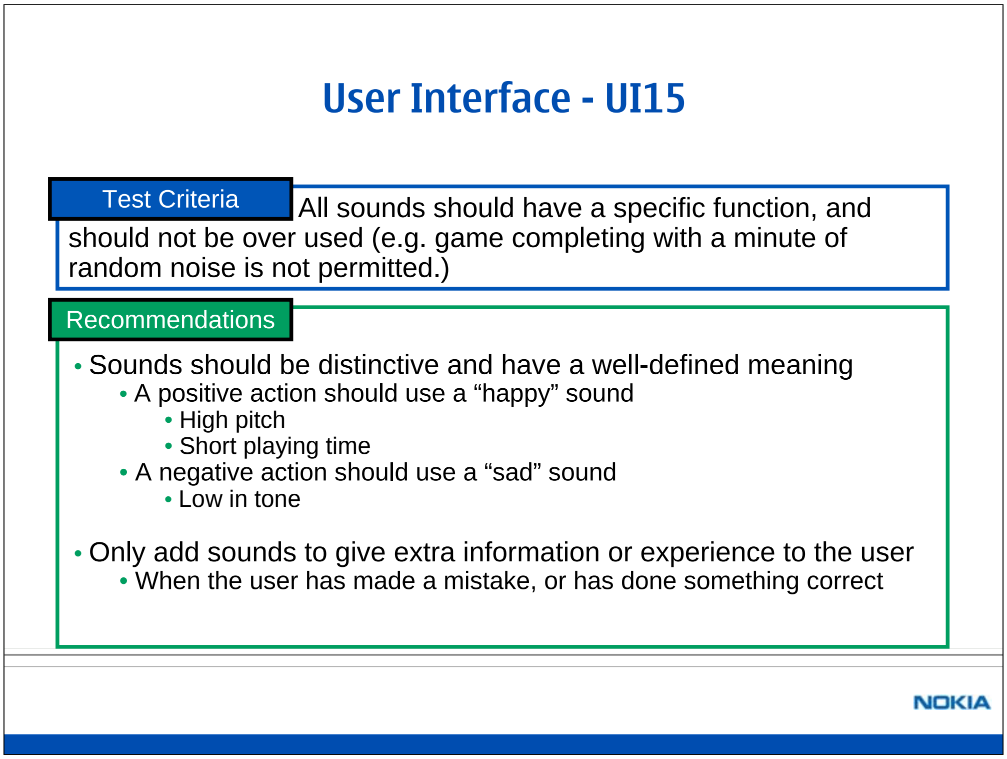
User Interface - UI14

Test Criteria The application must provide user with mute/off setting for background music and/or sound effects.

Recommendations

- Get `VolumeControl` object from the `Player` object
- Use the `setMute` method of `VolumeControl`

```
...
Player player = Manager.createPlayer("http://myserver/mymusic.wav");
player.start();
...
private void toggleBackgroundMusicMute() {
   VolumeControl volume = (VolumeControl) player.getControl("VolumeControl");
   boolean toggleMute = !volume.isMuted();
   volume.setMute(toggleMute);
}
```



User Interface - UI15

Test Criteria All sounds should have a specific function, and should not be over used (e.g. game completing with a minute of random noise is not permitted.)

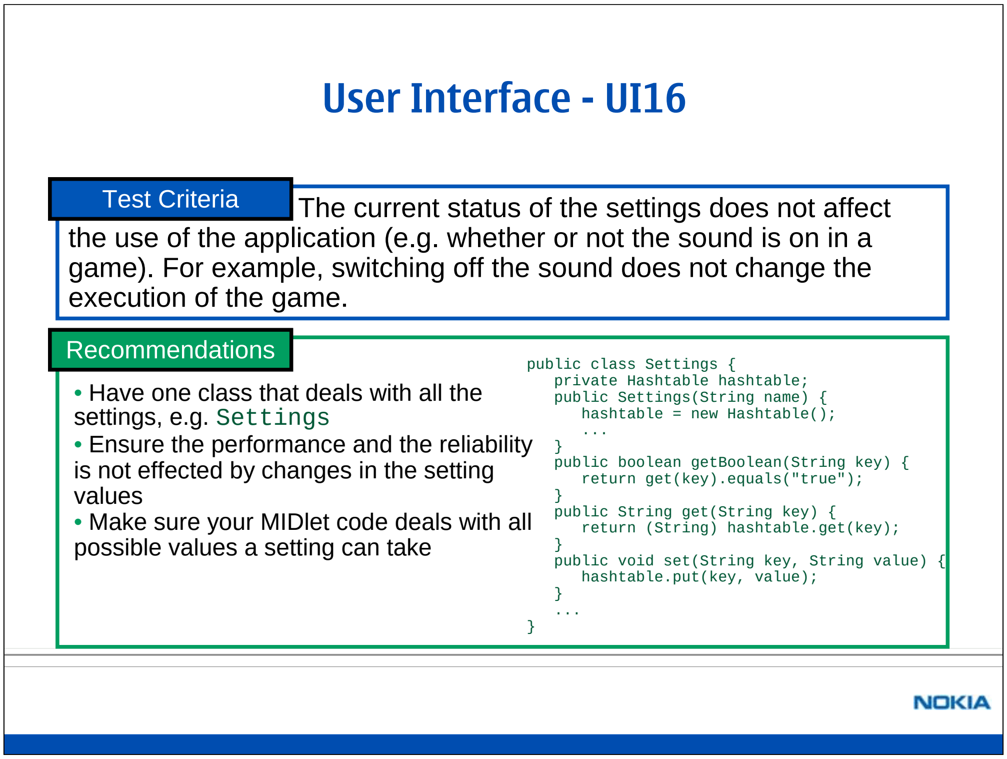
Recommendations

- Sounds should be distinctive and have a well-defined meaning
  - A positive action should use a “happy” sound
    - High pitch
    - Short playing time
  - A negative action should use a “sad” sound
    - Low in tone
- Only add sounds to give extra information or experience to the user
  - When the user has made a mistake, or has done something correct

The reason for this test is to ensure that sounds are used to assist the user to understand the application, and prevent the user from getting irritated by its operation.

To be able to do this you must ensure that sounds are distinctive and have a well-defined meaning. For example, positive actions should use “happy” sounds, i.e. high pitch with a short playing time, while negative actions should use “sad” sounds, i.e. low in tone.

Also be restrictive in your use of sounds in your application. Only add sounds to give extra information or experience to the user, for example, when the user has made a mistake, or has done something correct



User Interface - UI16

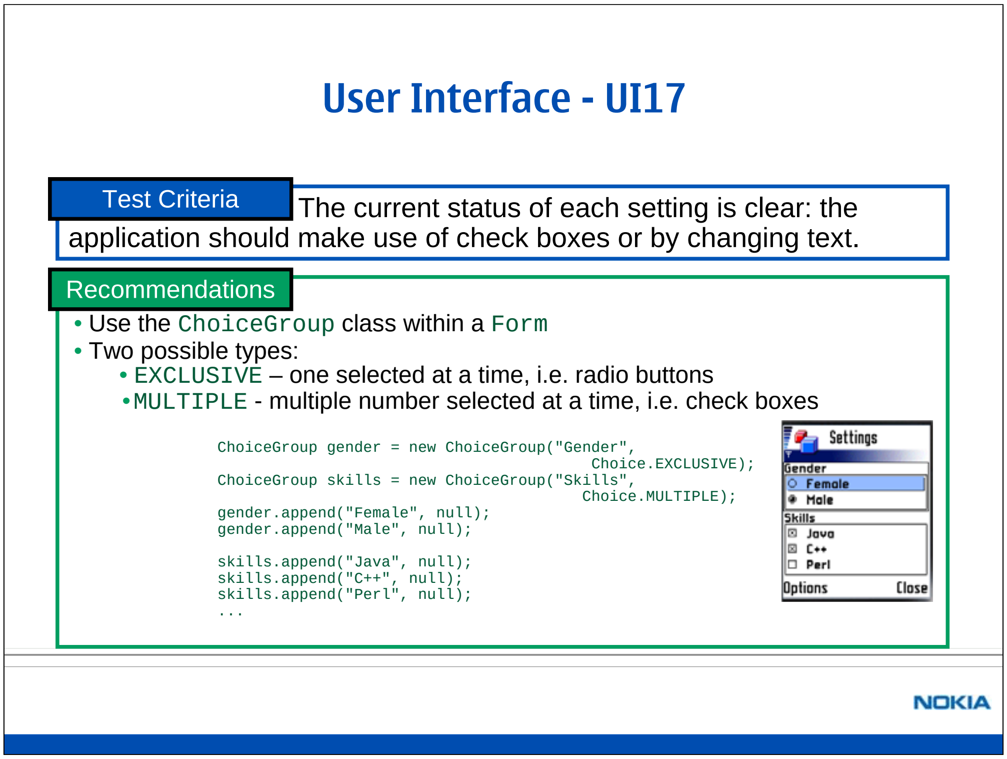
Test Criteria

The current status of the settings does not affect the use of the application (e.g. whether or not the sound is on in a game). For example, switching off the sound does not change the execution of the game.

Recommendations

- Have one class that deals with all the settings, e.g. `Settings`
- Ensure the performance and the reliability is not effected by changes in the setting values
- Make sure your MIDlet code deals with all possible values a setting can take

```
public class Settings {
   private Hashtable hashtable;
   public Settings(String name) {
      hashtable = new Hashtable();
      ...
   }
   public boolean getBoolean(String key) {
      return get(key).equals("true");
   }
   public String get(String key) {
      return (String) hashtable.get(key);
   }
   public void set(String key, String value) {
      hashtable.put(key, value);
   }
   ...
}
```



User Interface - UI17

Test Criteria The current status of each setting is clear: the application should make use of check boxes or by changing text.

Recommendations

- Use the `ChoiceGroup` class within a `Form`
- Two possible types:
  - `EXCLUSIVE` – one selected at a time, i.e. radio buttons
  - `MULTIPLE` - multiple number selected at a time, i.e. check boxes

```
ChoiceGroup gender = new ChoiceGroup("Gender",
                                         Choice.EXCLUSIVE);
ChoiceGroup skills = new ChoiceGroup("Skills",
                                        Choice.MULTIPLE);
gender.append("Female", null);
gender.append("Male", null);

skills.append("Java", null);
skills.append("C++", null);
skills.append("Perl", null);
...
```

Settings

Gender

○ Female

● Male

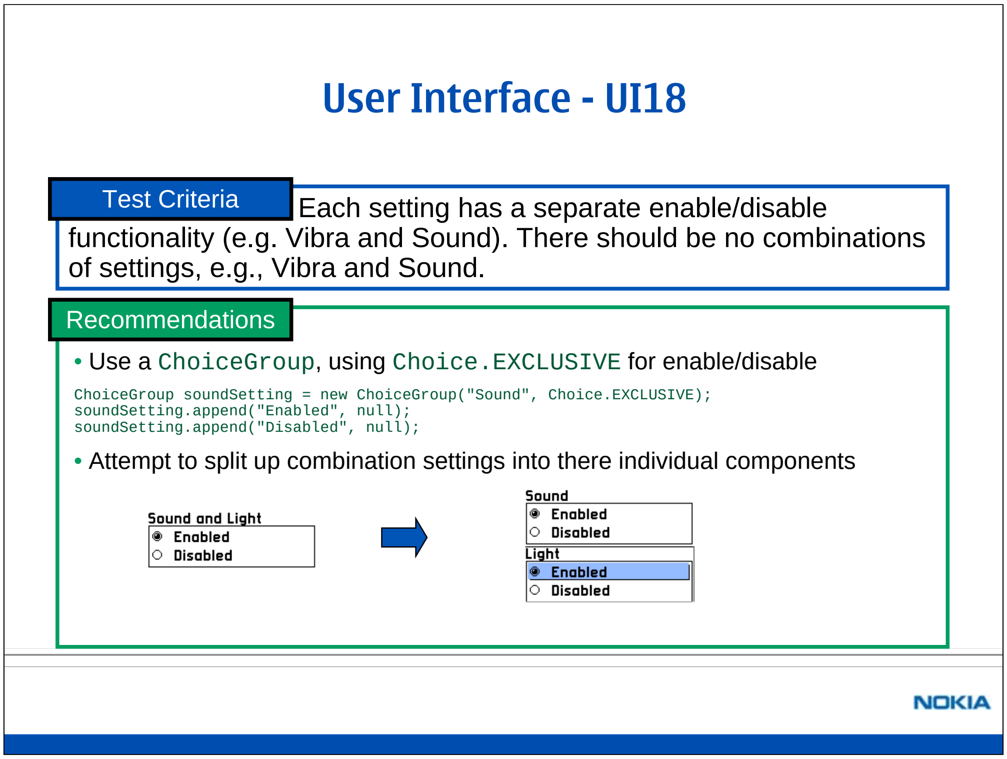
Skills

☒ Java

☒ C++

☐ Perl

This test is to make sure the user interface for the settings screen is clear, so that the user is sure what effect changing the setting will have on the application. One of the classes that can assist your creation of a settings interface is the ChoiceGroup class. A ChoiceGroup is a group of selectable elements intended to be placed within a Form. The group may be created with a mode that requires a single choice to be made or that allows multiple choices. The implementation is responsible for providing the graphical representation of these modes and must provide visually different graphics for different modes. For example, it might use "radio buttons" for the single choice mode and "check boxes" for the multiple choice mode.



User Interface - UI18

Test Criteria Each setting has a separate enable/disable functionality (e.g. Vibra and Sound). There should be no combinations of settings, e.g., Vibra and Sound.

Recommendations

- Use a `ChoiceGroup`, using `Choice.EXCLUSIVE` for enable/disable

```
ChoiceGroup soundSetting = new ChoiceGroup("Sound", Choice.EXCLUSIVE);
soundSetting.append("Enabled", null);
soundSetting.append("Disabled", null);
```

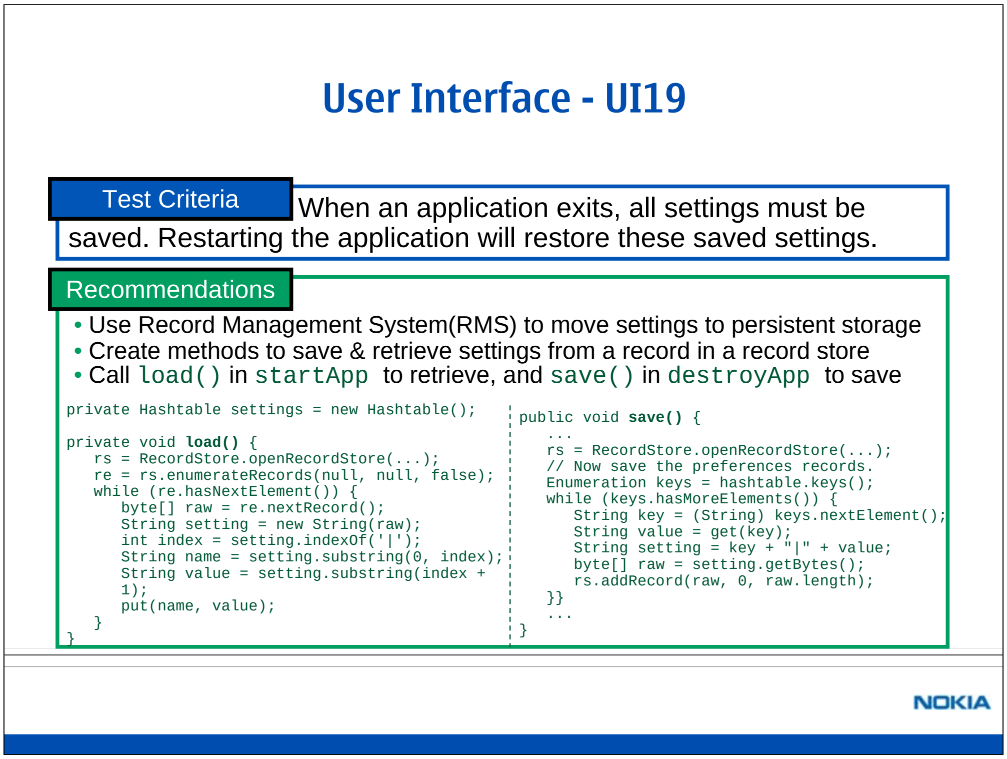
- Attempt to split up combination settings into there individual components

Sound and Light
● Enabled
○ Disabled

Sound
● Enabled
○ Disabled

Light
● Enabled
○ Disabled

This test is to make sure your application gives the user full control how it executes by allowing them to change individual settings. Try to avoid combining settings (for example "Sound and Light") and just provide the user with separate individual settings for them to change. This way, the user will be fully aware of how the application will react when they change a setting, and they can alter the way the application executes to suit their personal preference.



User Interface - UI19

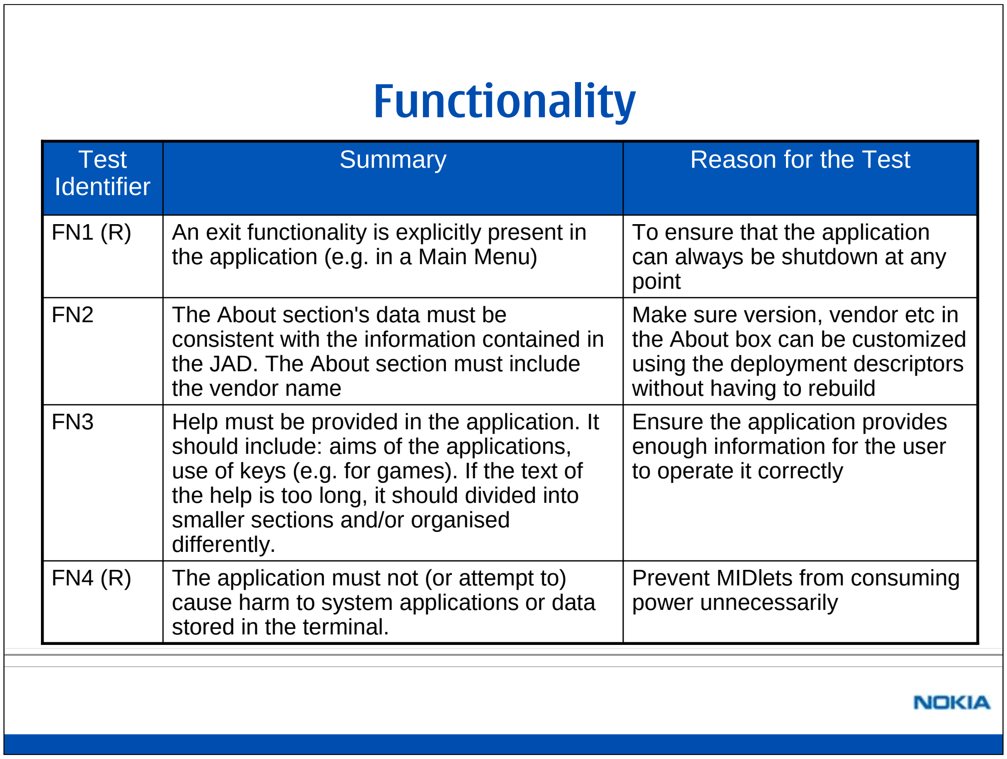
Test Criteria When an application exits, all settings must be saved. Restarting the application will restore these saved settings.

Recommendations

- Use Record Management System(RMS) to move settings to persistent storage
- Create methods to save & retrieve settings from a record in a record store
- Call `load()` in `startApp` to retrieve, and `save()` in `destroyApp` to save

```
private Hashtable settings = new Hashtable();

private void load() {
   rs = RecordStore.openRecordStore(...);
   re = rs.enumerateRecords(null, null, false);
   while (re.hasNextElement()) {
      byte[] raw = re.nextRecord();
      String setting = new String(raw);
      int index = setting.indexOf('|');
      String name = setting.substring(0, index);
      String value = setting.substring(index +
      1);
      put(name, value);
   }
}
```

```
public void save() {
   ...
   rs = RecordStore.openRecordStore(...);
   // Now save the preferences records.
   Enumeration keys = hashtable.keys();
   while (keys.hasMoreElements()) {
      String key = (String) keys.nextElement();
      String value = get(key);
      String setting = key + "|" + value;
      byte[] raw = setting.getBytes();
      rs.addRecord(raw, 0, raw.length);
   }}
   ...
}
```

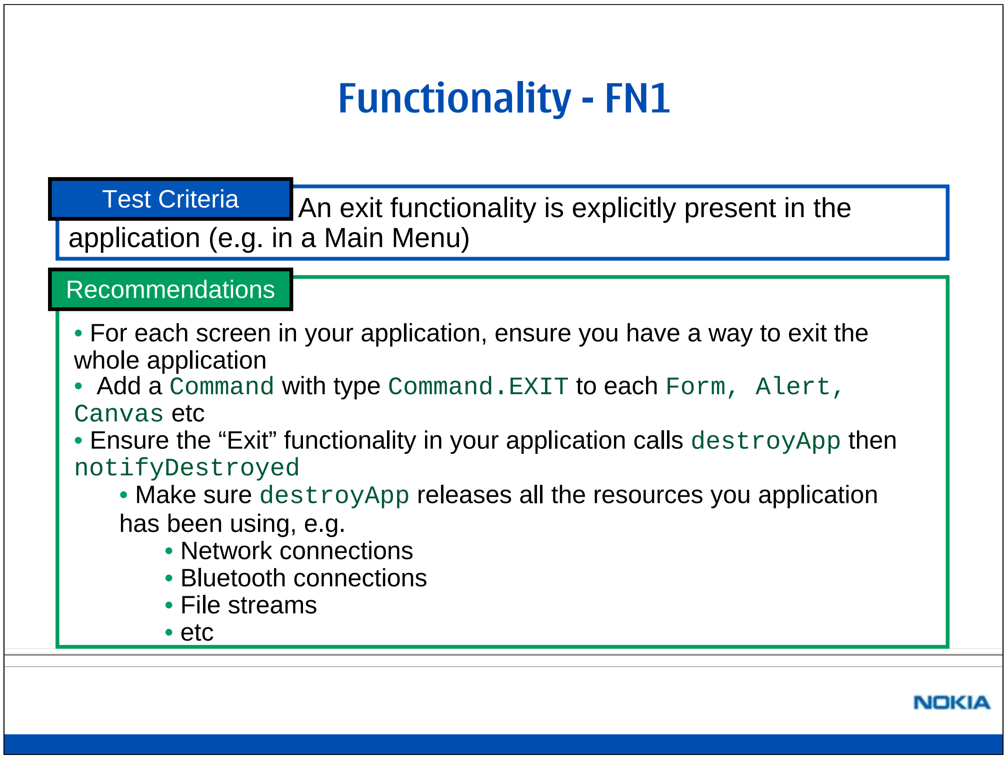


Functionality

Test Identifier	Summary	Reason for the Test
FN1 (R)	An exit functionality is explicitly present in the application (e.g. in a Main Menu)	To ensure that the application can always be shutdown at any point
FN2	The About section's data must be consistent with the information contained in the JAD. The About section must include the vendor name	Make sure version, vendor etc in the About box can be customized using the deployment descriptors without having to rebuild
FN3	Help must be provided in the application. It should include: aims of the applications, use of keys (e.g. for games). If the text of the help is too long, it should divided into smaller sections and/or organised differently.	Ensure the application provides enough information for the user to operate it correctly
FN4 (R)	The application must not (or attempt to) cause harm to system applications or data stored in the terminal.	Prevent MIDlets from consuming power unnecessarily

NOKIA

The emphasis of this category is to ensure the application functions as expected, in terms of being able to exit the application, and not consuming power unnecessarily. The application should provide enough information for the user to operate it correctly and it must make sure the version, vendor etc in the About box can be customized using the deployment descriptors without having to rebuild.



Functionality - FN1

Test Criteria An exit functionality is explicitly present in the application (e.g. in a Main Menu)

Recommendations

- For each screen in your application, ensure you have a way to exit the whole application
- Add a `Command` with type `Command.EXIT` to each `Form, Alert, Canvas` etc
- Ensure the “Exit” functionality in your application calls `destroyApp` then `notifyDestroyed`
  - Make sure `destroyApp` releases all the resources you application has been using, e.g.
    - Network connections
    - Bluetooth connections
    - File streams
    - etc

NOKIA

This test is to ensure that the application can always be shutdown at any point. To do this, for each screen in your application, you must ensure you have a way to exit the whole application. This can be done by adding a Command with type Command.EXIT to each Form, Alert, Canvas etc. The “Exit” functionality in your application must call destroyApp then notifyDestroyed, making sure the destroyApp method releases all the resources you application has been using.

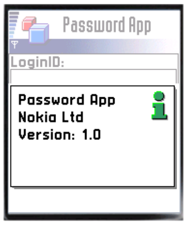
Functionality - FN2

Test Criteria The About section's data must be consistent with the information contained in the JAD. The About section must include the vendor name

Recommendations

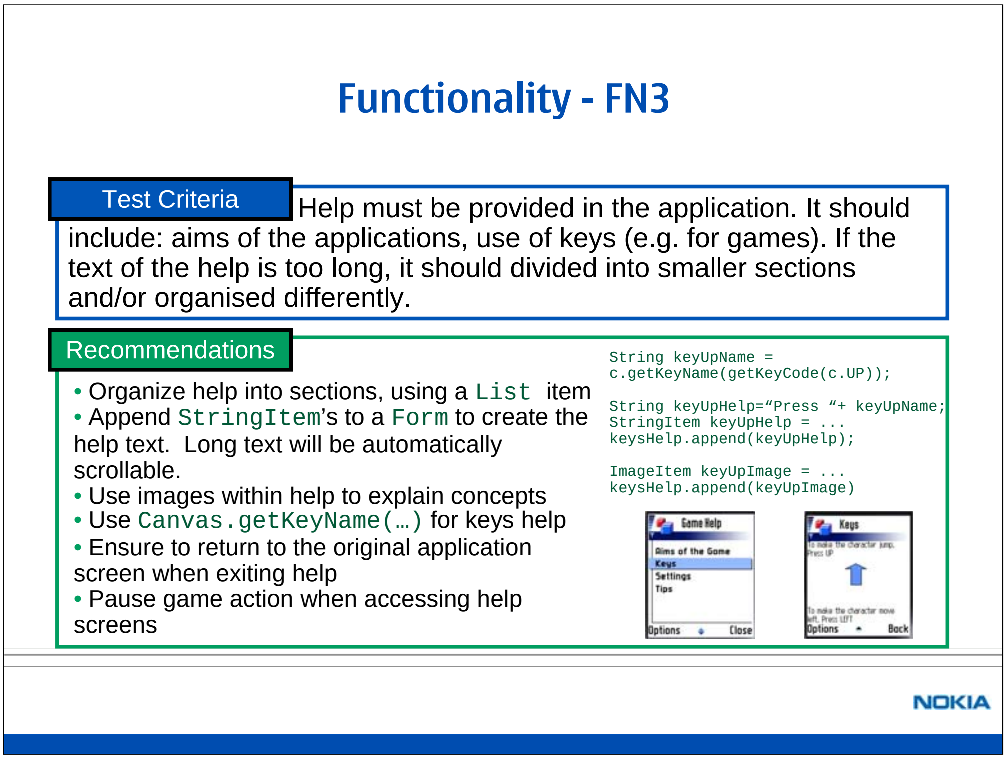
- Use the `getAppProperty` method of MIDlet
- Retrieves named properties from JAD file and the JAR’s manifest

```
String name = getAppProperty("MIDlet-Name");
String vendor = getAppProperty("MIDlet-Vendor");
String version = getAppProperty("MIDlet-Version");

Alert aboutBox = new Alert(name);
aboutBox.setTimeout(10000);

String aboutString = vendor + "\nVersion: " + version;
aboutBox.setString(aboutString);
```



Password App
Nokia Ltd
Version: 1.0

This test checks that the details reported in the JAD file are consistent with the information reported within the application, for example the information shown in the application’s About section. The best way to ensure this is to request the getAppProperty() method of the MIDlet class when this kind of information is needed. The getAppProperty() method retrieves the requested information from the JAD file or the JAR package’s manifest file. For example, getAppProperty(“MIDlet-Version”) returns the version of the MIDlet.



Functionality - FN3

Test Criteria Help must be provided in the application. It should include: aims of the applications, use of keys (e.g. for games). If the text of the help is too long, it should divided into smaller sections and/or organised differently.

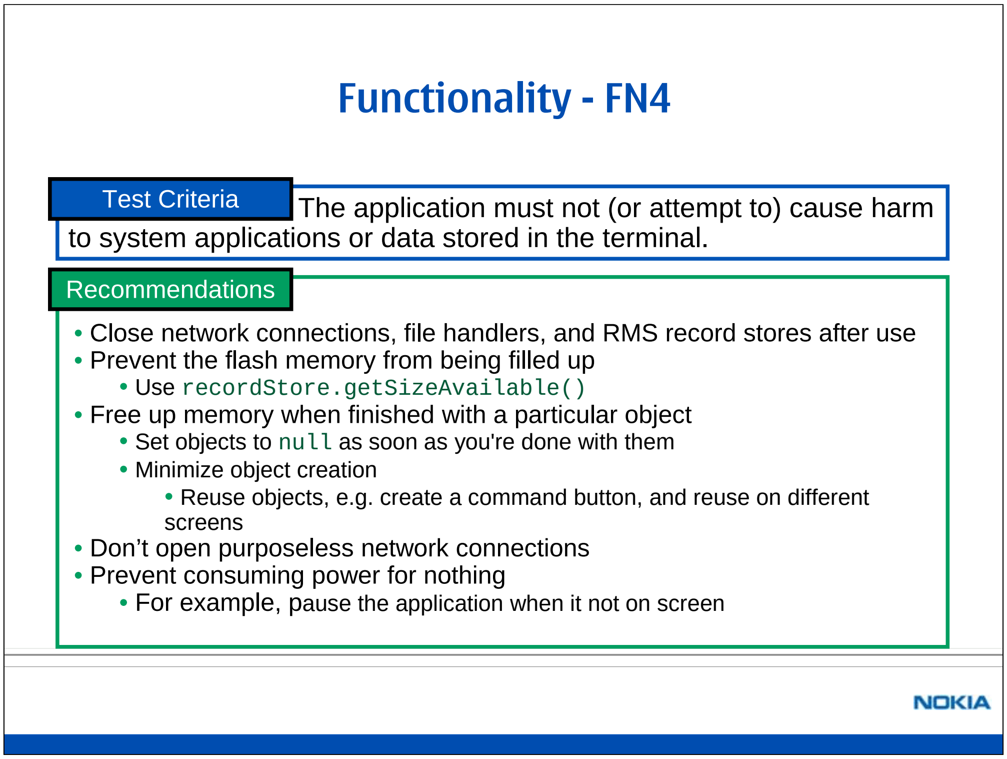
Recommendations

- Organize help into sections, using a `List` item
- Append `StringItem`'s to a `Form` to create the help text. Long text will be automatically scrollable.
- Use images within help to explain concepts
- Use `Canvas.getKeyName(…)` for keys help
- Ensure to return to the original application screen when exiting help
- Pause game action when accessing help screens

```
String keyUpName =
c.getKeyName(getKeyCode(c.UP));

String keyUpHelp=“Press “+ keyUpName;
StringItem keyUpHelp = ...
keysHelp.append(keyUpHelp);

ImageItem keyUpImage = ...
keysHelp.append(keyUpImage)
```

This test ensures your application provides enough information for the user to operate it correctly. To fully explain how the application works, you should include a help section. The code below shows a simple Form that represents the help screen for a game. To display the name of the keys, it can be seen that this code uses the canvas.getKeyName. This ensures the help is consistent with the current key settings

```
Form keysHelp = new Form("Keys");

String aimsText = "To control the character on screen use the direction
buttons on your handset.";

aimsText += "\nTo make the character jump, Press " +
canvas.getKeyName(getKeyCode(c.UP));

aims.append(new StringItem("", aimsText));

Image upImage = null;

try {

upImage = Image.createImage("/res/up.png");

} catch(Exception e){}

aims.append(new ImageItem("", upImage, ImageItem.LAYOUT_CENTER,"<UP>"));

aims.append(new StringItem("", "\n\nTo make the character move left, Press
LEFT"));
```



Functionality - FN4

Test Criteria The application must not (or attempt to) cause harm to system applications or data stored in the terminal.

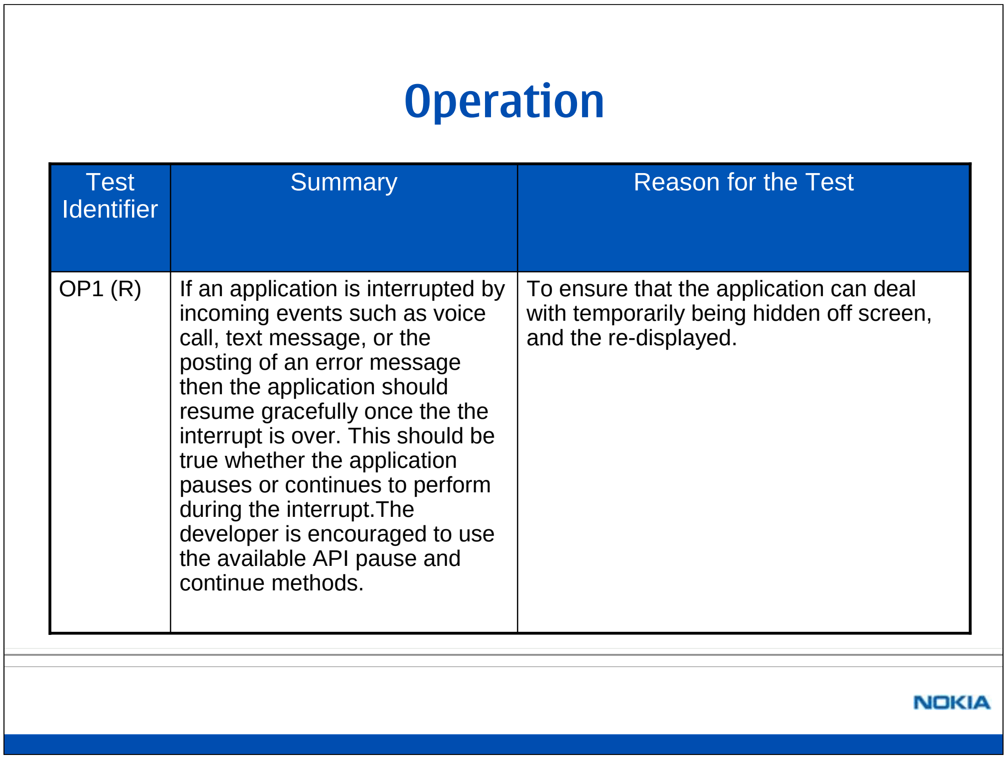
Recommendations

- Close network connections, file handlers, and RMS record stores after use
- Prevent the flash memory from being filled up
  - Use `recordStore.getSizeAvailable()`
- Free up memory when finished with a particular object
  - Set objects to `null` as soon as you're done with them
  - Minimize object creation
    - Reuse objects, e.g. create a command button, and reuse on different screens
- Don’t open purposeless network connections
- Prevent consuming power for nothing
  - For example, pause the application when it not on screen

NOKIA

One of the main points this test is making is that your application should avoid consuming power unnecessarily. This can be avoided by freeing up memory when your finished with a particular object by setting objects to null as soon as you're done with them, minimizing object creation and reusing objects. You should also ensure that your application is paused when it is not on screen to save power. You should be restrictive when opening network connections, and be sure to close them after use.

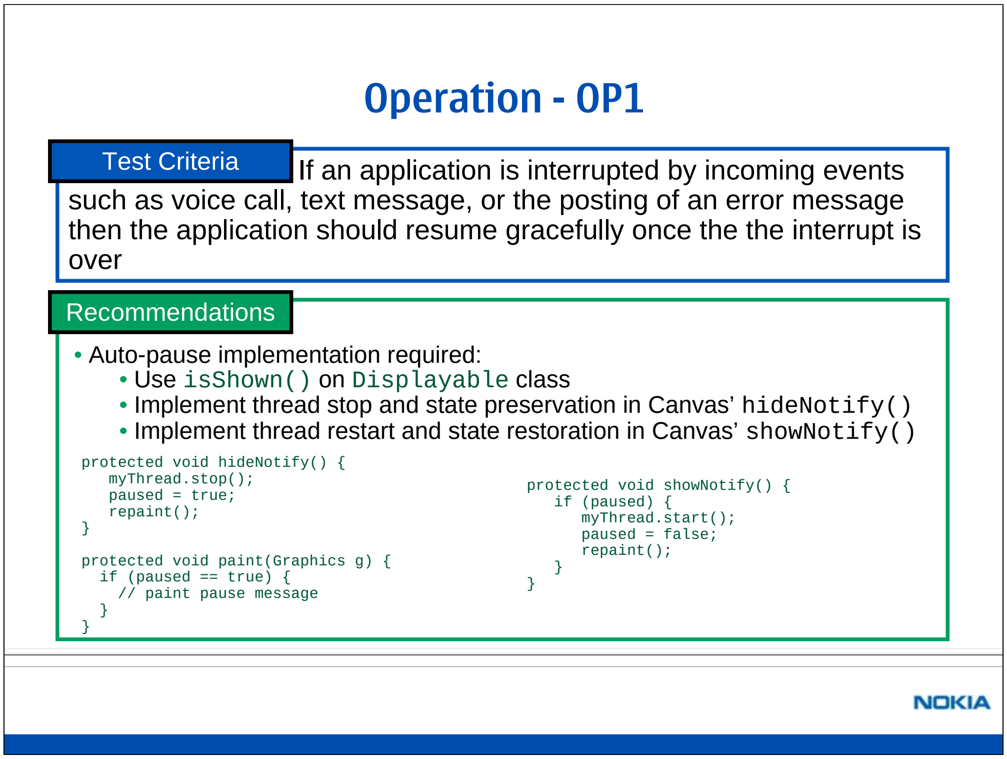
The other type of memory that you should be concerned with is persistent storage, for record stores, the databases accessed through the javax.microedition.rms.RecordStore class. The MIDP specification calls for only 8 kB of persistent storage. More space is usually available, but the amount varies by device. Your key concern here is to be sure your MIDlet behaves well if it runs out of space for its record stores. You create record stores with the RecordStore class's openRecordStore() method and add records with its addRecord() method. Your application should catch and gracefully handle any RecordStoreFullException thrown by either. RecordStore's getSizeAvailable() method indicates how many additional bytes are available for a particular record store. Note that this may or may not be the total number of bytes available for all record stores, but it gives some indication of how much persistent storage is available for a single record store instance. The size() method returns the current size of a record store in bytes.



Operation

Test Identifier	Summary	Reason for the Test
OP1 (R)	If an application is interrupted by incoming events such as voice call, text message, or the posting of an error message then the application should resume gracefully once the the interrupt is over. This should be true whether the application pauses or continues to perform during the interrupt.The developer is encouraged to use the available API pause and continue methods.	To ensure that the application can deal with temporarily being hidden off screen, and the re-displayed.

The emphasis of this category is to ensure your application can interact correctly with device hardware and software that is it running on. There is only one test in this category. It investigates whether your application can deal with temporarily being hidden off screen, and the re-displayed. This could happen, for example, if there was an incoming SMS, incoming call or notification of a low battery.



Operation - OP1

Test Criteria

If an application is interrupted by incoming events such as voice call, text message, or the posting of an error message then the application should resume gracefully once the the interrupt is over

Recommendations

- Auto-pause implementation required:
  - Use `isShown()` on `Displayable` class
  - Implement thread stop and state preservation in Canvas’ `hideNotify()`
  - Implement thread restart and state restoration in Canvas’ `showNotify()`

```
protected void hideNotify() {
   myThread.stop();
   paused = true;
   repaint();
}

protected void paint(Graphics g) {
  if (paused == true) {
    // paint pause message
  }
}
```

```
protected void showNotify() {
   if (paused) {
      myThread.start();
      paused = false;
      repaint();
   }
}
```

This test ensures that your application can deal with other operations of the mobile phone intervening into execution, such as an incoming SMS, incoming call or notification of a low battery.

For incoming SMS in Nokia models, the alert tone and vibration is triggered. There is no other intervening effects, and so your application should be able to carry on with execution as normal when this happens.

When an incoming call occurs, in Nokia models, the alert tone and vibration is triggered and a note for the incoming call is shown on screen. An auto-pause implementation is required, therefore, for this occurrence.

Developers must create a method that pauses an active session when one of the following events occurs during the MIDlet interaction.

- *A system notification is shown on the screen (for example, an incoming call, battery full or low).*
- *The user presses the red dial key, the power key, or the application key of the device.*
- *The main screen of the application is hidden by a system menu or another application is set to run on the foreground.*

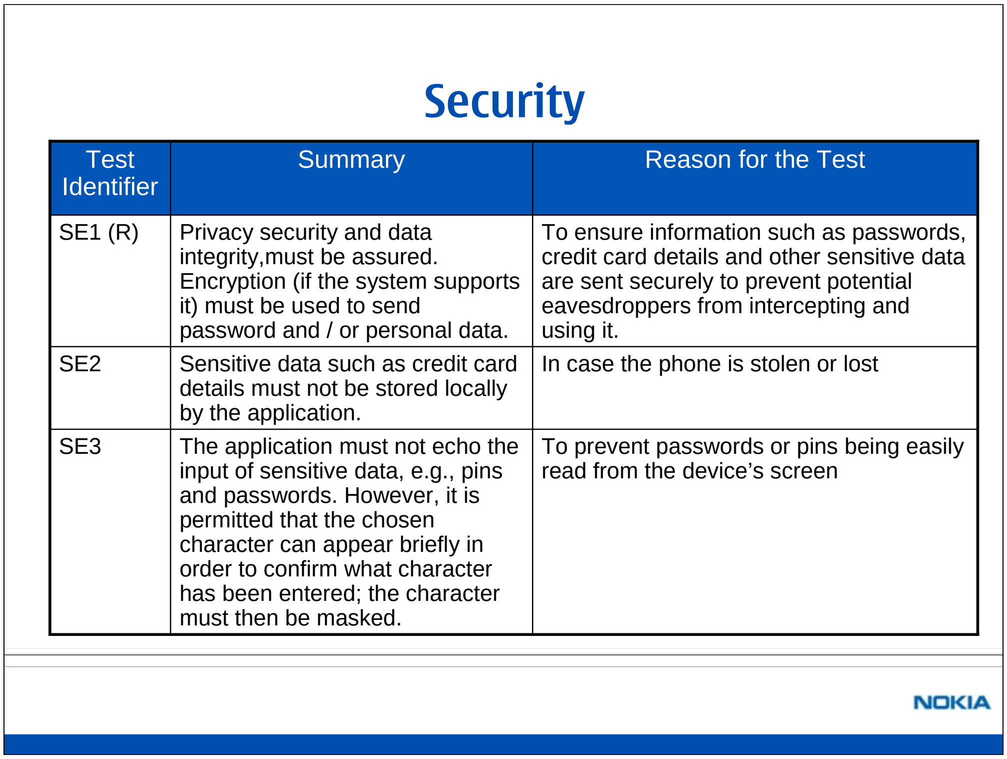
In practice, the MIDlet must always be paused when it is hidden. This is especially important in game applications, because the player will probably lose the game if it is not immediately paused when the game is hidden.

MIDlets that have been created for the Series 60 Developer Platform can be paused with the isShown() method of the Displayable class or hideNotify() method of the Canvas or CustomItem classes.

All User Interface (UI) components are inherited from the Displayable class. Thus the isShown() method can be used to test whether a UI component is visible or not. Repetitive requesting of the isShown() method can be used to get information if the UI component is hidden for some reason. The isShown() method can be used for high-level UI components like Form or List, but in case of low-level UI components that are inherited from Canvas or CustomItem classes it is better to use their proprietary hideNotify() and showNotify() methods.

The hideNotify() method is called right after the Canvas object has left the display. Create an auto-pause mechanism inside the hideNotify() method to stop threads, cancel timers, save important values, and so on.

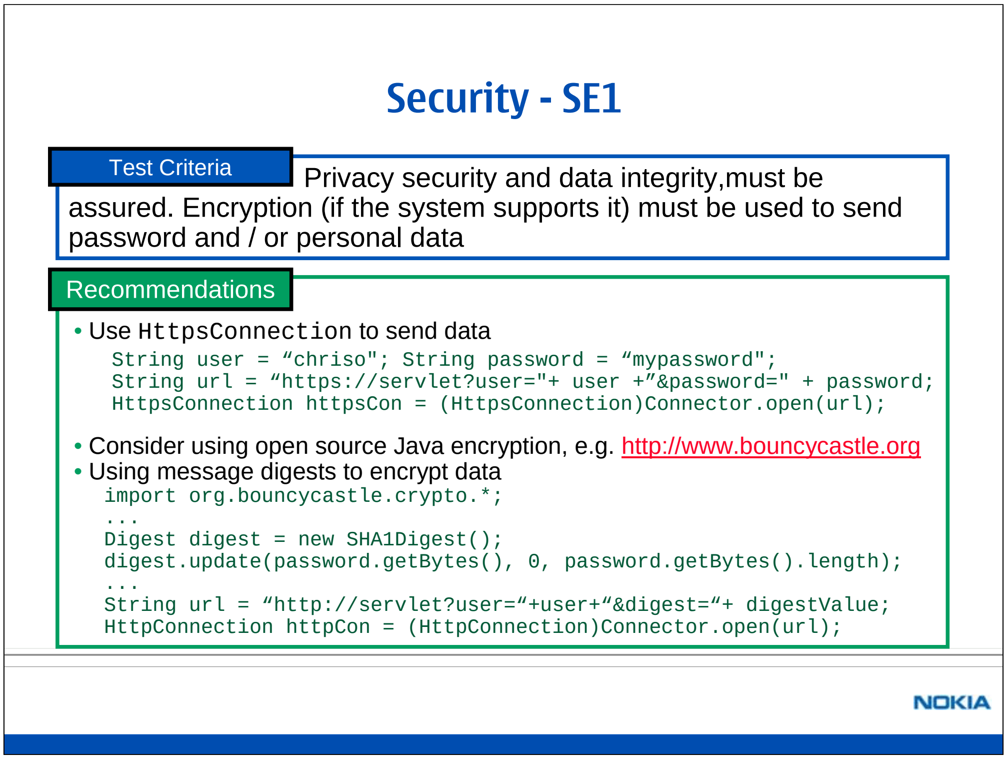
It is possible to create an auto-continue mechanism with the showNotify() method. The showNotify() method is called just before the Canvas is getting back on the display. In this method it is useful to, for example, restore important values, timers or restart threads so the application execution can continue



Security

Test Identifier	Summary	Reason for the Test
SE1 (R)	Privacy security and data integrity,must be assured. Encryption (if the system supports it) must be used to send password and / or personal data.	To ensure information such as passwords, credit card details and other sensitive data are sent securely to prevent potential eavesdroppers from intercepting and using it.
SE2	Sensitive data such as credit card details must not be stored locally by the application.	In case the phone is stolen or lost
SE3	The application must not echo the input of sensitive data, e.g., pins and passwords. However, it is permitted that the chosen character can appear briefly in order to confirm what character has been entered; the character must then be masked.	To prevent passwords or pins being easily read from the device’s screen

The emphasis of this category is to guarantee information such as passwords, credit card details and other sensitive data are sent securely to prevent potential eavesdroppers from intercepting and using it. Since a mobile phone is carried everywhere by its owner, there is always a risk of losing the phone. These tests, therefore, ensure that your application does not store sensitive data into the mobile device without encryption and it prevents passwords or pins being easily read from the device’s screen.



Security - SE1

Test Criteria Privacy security and data integrity,must be assured. Encryption (if the system supports it) must be used to send password and / or personal data

Recommendations

- Use `HttpsConnection` to send data

```
String user = “chriso"; String password = “mypassword";
String url = “https://servlet?user="+ user +”&password=" + password;
HttpsConnection httpsCon = (HttpsConnection)Connector.open(url);
```

- Consider using open source Java encryption, e.g. http://www.bouncycastle.org
- Using message digests to encrypt data

```
import org.bouncycastle.crypto.*;
...
Digest digest = new SHA1Digest();
digest.update(password.getBytes(), 0, password.getBytes().length);
...
String url = “http://servlet?user=“+user+“&digest=“+ digestValue;
HttpConnection httpCon = (HttpConnection)Connector.open(url);
```

NOKIA

When dealing with security, you may have to deal with some of the following in your applications:

- *Integrity. This is making sure the data being sent is not getting changed or corrupted in any way.*
- *Authentication. This is the process of proving identity.*
- *Confidentiality. This is being able to send sensitive data over the network without other people being able to see the information.*
- *Repudiatability. This is auditing so that the identification of who-ever conducted a transaction can be positively established. This can be implemented on the server side by logging any operations to save or retrieve sensitive data*

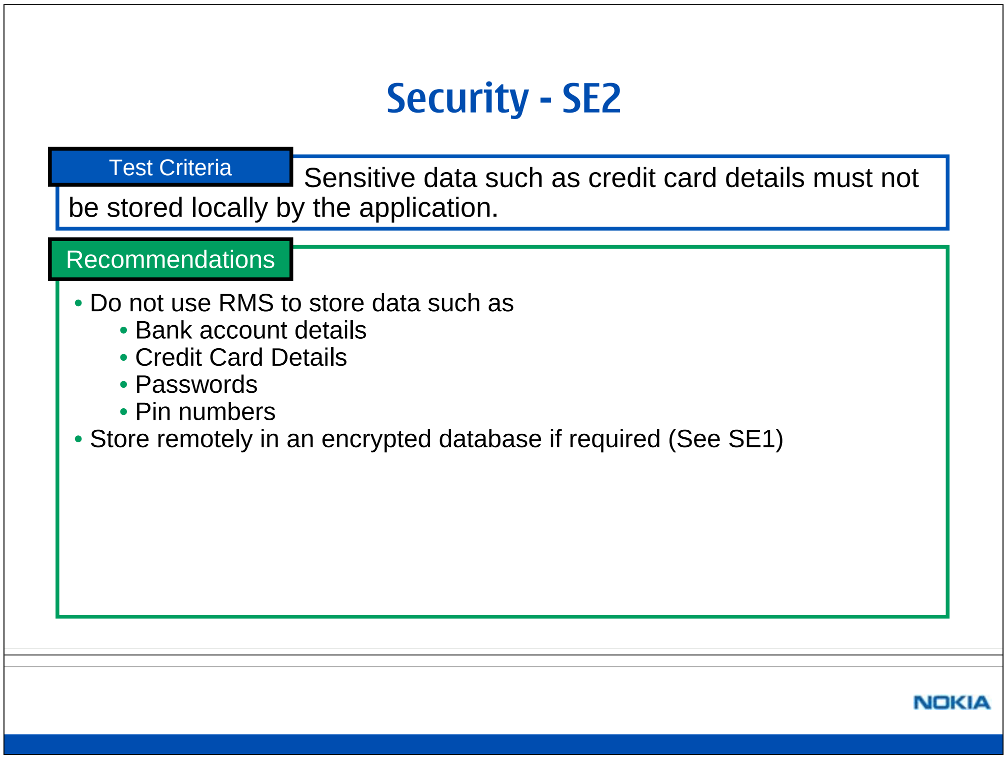
Cryptography provides some solutions for each of these needs:

- *Message digests. A message digest combines large pieces of data into a small piece of data. You might, for example, run an entire file through a message digest to end up with a 160-bit digest value. If you change even 1 bit of the file and run it through the message digest again, you’ll get an entirely different digest value. A message digest value is sometimes called a digital fingerprint.*
- *Digital signatures. A digital signature is like a message digest except it is produced by a particular person, the signer. The signer must have a private key that is used to create the signature. A corresponding public key can be used by anyone to verify that the signature came from the signer. The private key and public key together are called a key pair. Keys are really just data—think of an array of bytes.Certificates are really just an extension of digital signatures. A certificate is a document, signed by some authority like the U.S. Postal Service, that proves your identity. It’s like a driver’s license, except it’s based on digital signatures.*
- *Ciphers. Ciphers can either encrypt data or decrypt it. An encrypting cipher accepts your data, called plaintext, and produces an unreadable mess, called ciphertext. A decrypting cipher takes ciphertext and converts it back to plaintext. Ciphers use keys; if you encrypt the same plaintext with two different keys, you’ll get two different sets of ciphertext. A symmetric cipher uses the same key for encryption and decryption. An asymmetric cipher operates with a key pair—one key is used for encrypting, while the matching key is used for decrypting. Ciphers operate in different modes that determine how plaintext is encrypted into ciphertext. This, in turn, affects the use and security of the cipher.*

When an application requires a password, for example, this data often needs to be sent to a remote server for authentication. To ensure is it sent securely, your applications should use encryption. At the moment, there is no standard MIDP API to deal with encryption; however, there are still ways to encrypt your personal data to send over the network.

If you are using MIDP 2.0, your application can use the HttpsConnection class that provides a secure connection. For example, a simple password authentication scheme could be created by adding the user name and password to an HTTPS URL as follows:

```
String user = "chriso";
String password = "mypassword";
```



Security - SE2

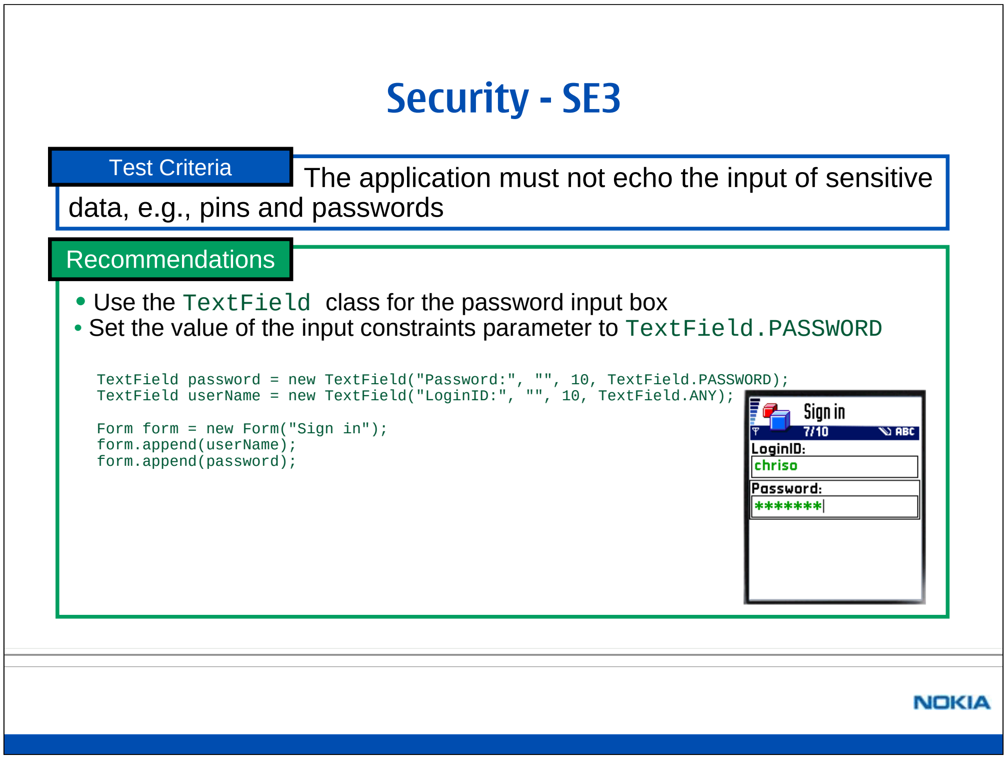
Test Criteria

Sensitive data such as credit card details must not be stored locally by the application.

Recommendations

- Do not use RMS to store data such as
  - Bank account details
  - Credit Card Details
  - Passwords
  - Pin numbers
- Store remotely in an encrypted database if required (See SE1)

If the device that the application is running on is ever stolen or lost, this test makes sure that any sensitive data is not actually stored on the device that can be used illegally or maliciously. Data such as Bank account details, Credit Card Details, Passwords, Pin numbers should be stored remotely in an encrypted database and sent via a secure connection.



Security - SE3

Test Criteria The application must not echo the input of sensitive data, e.g., pins and passwords

Recommendations

- Use the `TextField` class for the password input box
- Set the value of the input constraints parameter to `TextField.PASSWORD`

```
TextField password = new TextField("Password:", "", 10, TextField.PASSWORD);
TextField userName = new TextField("LoginID:", "", 10, TextField.ANY);

Form form = new Form("Sign in");
form.append(userName);
form.append(password);
```

Sign in

7/10 ABC

LoginID:

chriso

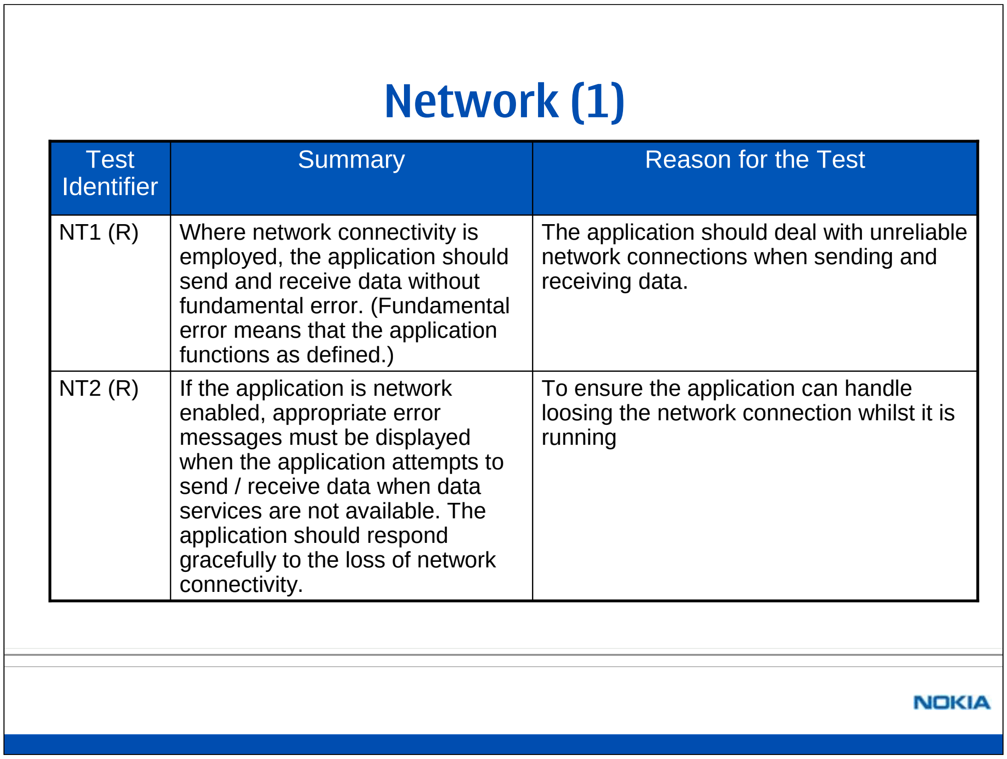
Password:

****

NOKIA

To be able to pass this test, you need to make sure that in your application, the input box that any pins or passwords are entered uses the TextField.PASSWORD input constraint.

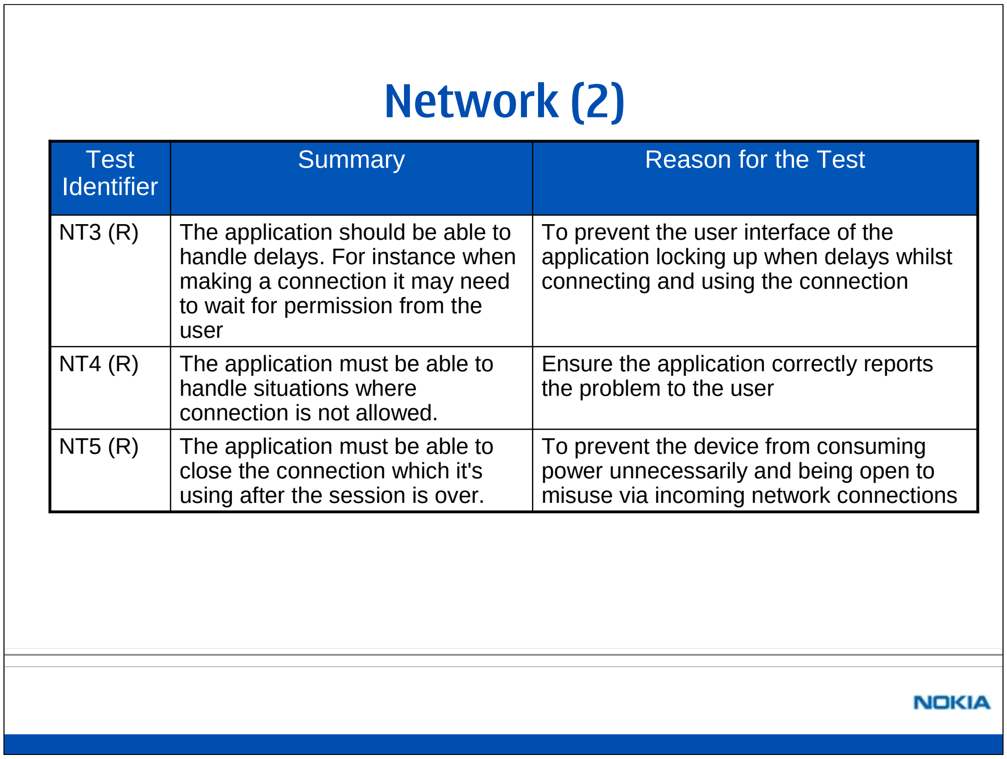
Setting the input constraint to the TextField.PASSWORD value indicates that this is private information and should be obscured when displayed on screen. The way in which the text within this box is obscured is dependant on the device the application is running on. In the slide example, the each character of the password is masked with a "" character.*



Network (1)

Test Identifier	Summary	Reason for the Test
NT1 (R)	Where network connectivity is employed, the application should send and receive data without fundamental error. (Fundamental error means that the application functions as defined.)	The application should deal with unreliable network connections when sending and receiving data.
NT2 (R)	If the application is network enabled, appropriate error messages must be displayed when the application attempts to send / receive data when data services are not available. The application should respond gracefully to the loss of network connectivity.	To ensure the application can handle loosing the network connection whilst it is running

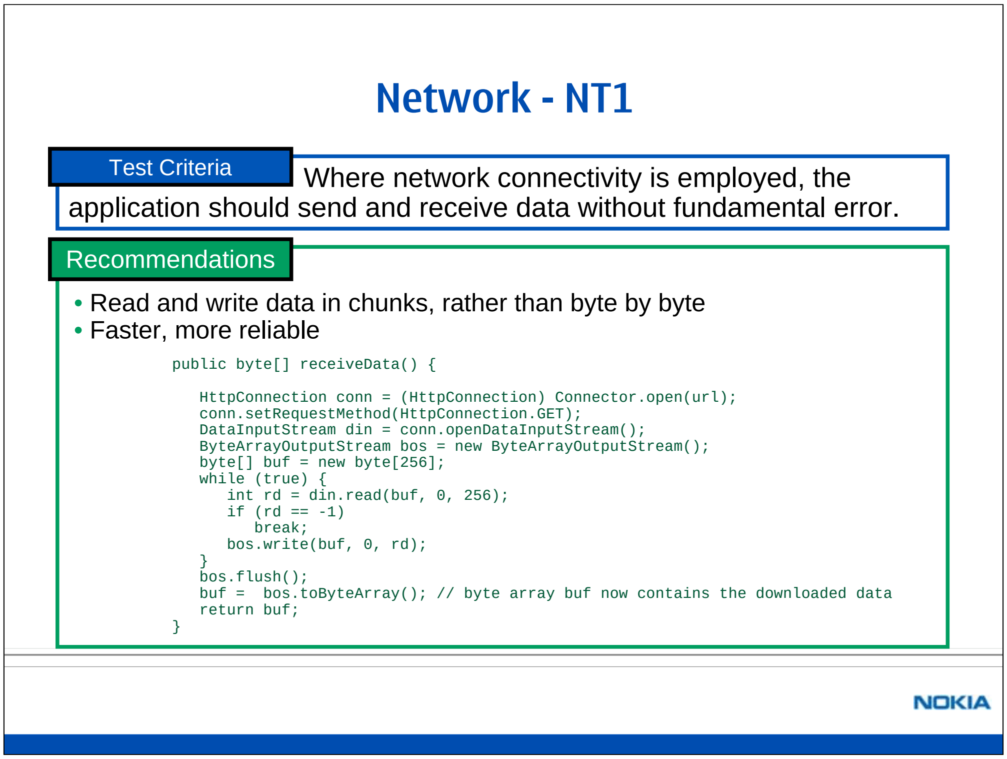
The emphasis of this category is to be certain that your application can deal with unreliable network connections when sending and receiving data and can handle loosing the network connection whilst it is running.



Network (2)

Test Identifier	Summary	Reason for the Test
NT3 (R)	The application should be able to handle delays. For instance when making a connection it may need to wait for permission from the user	To prevent the user interface of the application locking up when delays whilst connecting and using the connection
NT4 (R)	The application must be able to handle situations where connection is not allowed.	Ensure the application correctly reports the problem to the user
NT5 (R)	The application must be able to close the connection which it's using after the session is over.	To prevent the device from consuming power unnecessarily and being open to misuse via incoming network connections

This category also covers preventing the user interface locking up when delays whilst connecting and using the connection, reporting network problems correctly to the user, and prevents the device from consuming power by leaving network connections open unnecessarily.



Network - NT1

Test Criteria

Where network connectivity is employed, the application should send and receive data without fundamental error.

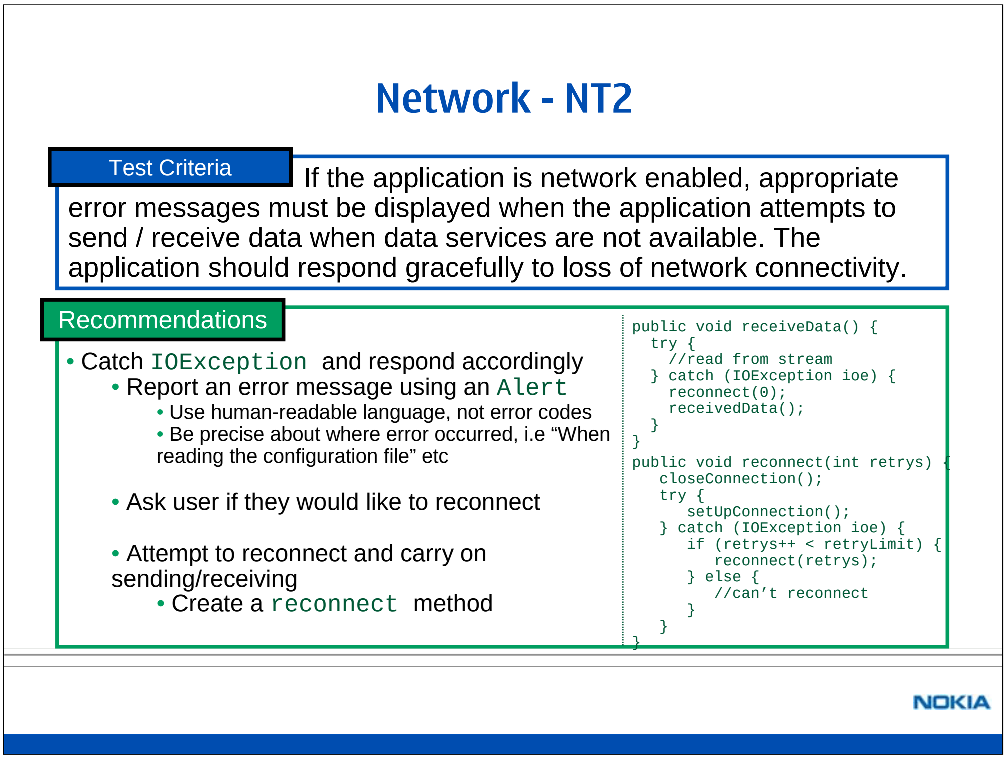
Recommendations

- Read and write data in chunks, rather than byte by byte
- Faster, more reliable

```
public byte[] receiveData() {

   HttpConnection conn = (HttpConnection) Connector.open(url);
   conn.setRequestMethod(HttpConnection.GET);
   DataInputStream din = conn.openDataInputStream();
   ByteArrayOutputStream bos = new ByteArrayOutputStream();
   byte[] buf = new byte[256];
   while (true) {
      int rd = din.read(buf, 0, 256);
      if (rd == -1)
         break;
      bos.write(buf, 0, rd);
   }
   bos.flush();
   buf =  bos.toByteArray(); // byte array buf now contains the downloaded data
   return buf;
}
```

To ensure the data your application is sending and receiving over the network is done quickly and in a reliable way, the data should be handled in chunks, rather than byte by byte. The code below shows how to send an array of bytes over a HttpConnection by using a DataOutputStream and its method write(byte[] b, int offset, int length)

```
public String sendData(byte[] data) throw IOException {
       HttpConnection hc = (HttpConnection)Connector.open(url,
Connector.WRITE);
       hc.setRequestMethod(HttpConnection.POST);
       DataOutputStream dos = hc.openDataOutputStream();
       dos.write(data, 0, data.length);
       dos.flush();
       dos.close();
}
```



Network - NT2

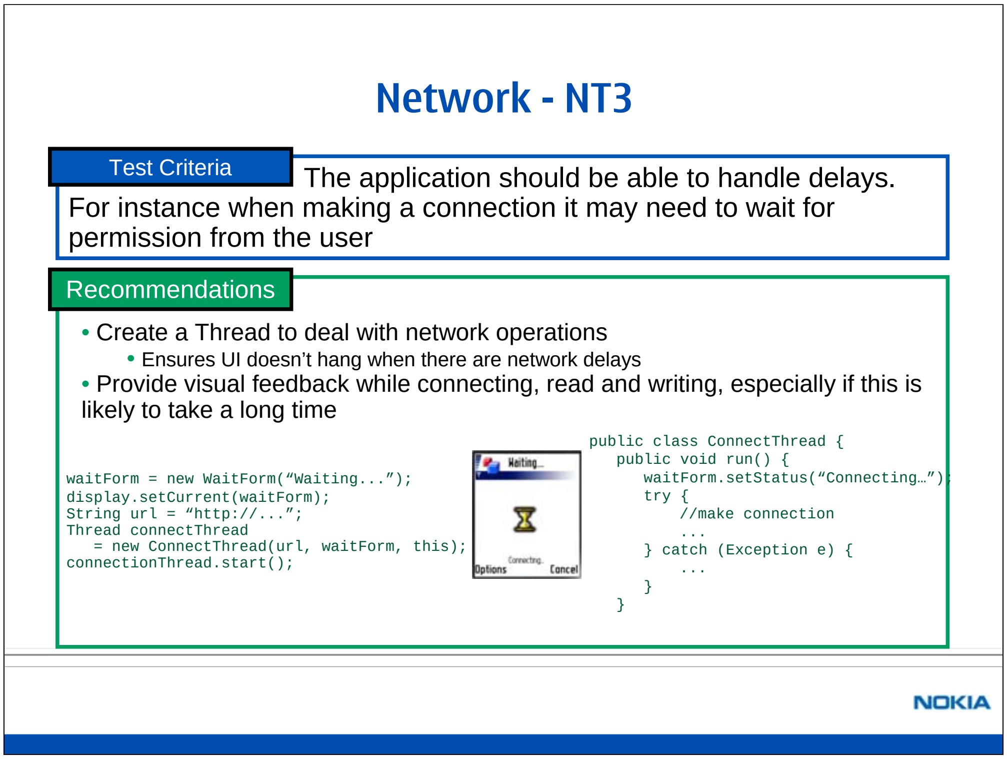
Test Criteria If the application is network enabled, appropriate error messages must be displayed when the application attempts to send / receive data when data services are not available. The application should respond gracefully to loss of network connectivity.

Recommendations

- Catch `IOException` and respond accordingly
  - Report an error message using an `Alert`
    - Use human-readable language, not error codes
    - Be precise about where error occurred, i.e "When reading the configuration file" etc
  - Ask user if they would like to reconnect
  - Attempt to reconnect and carry on sending/receiving
    - Create a `reconnect` method

```
public void receiveData() {
  try {
    //read from stream
  } catch (IOException ioe) {
    reconnect(0);
    receivedData();
  }
}
public void reconnect(int retrys) {
   closeConnection();
   try {
      setUpConnection();
   } catch (IOException ioe) {
      if (retrys++ < retryLimit) {
         reconnect(retrys);
      } else {
         //can’t reconnect
      }
   }
}
```

NOKIA



Network - NT3

Test Criteria

The application should be able to handle delays. For instance when making a connection it may need to wait for permission from the user

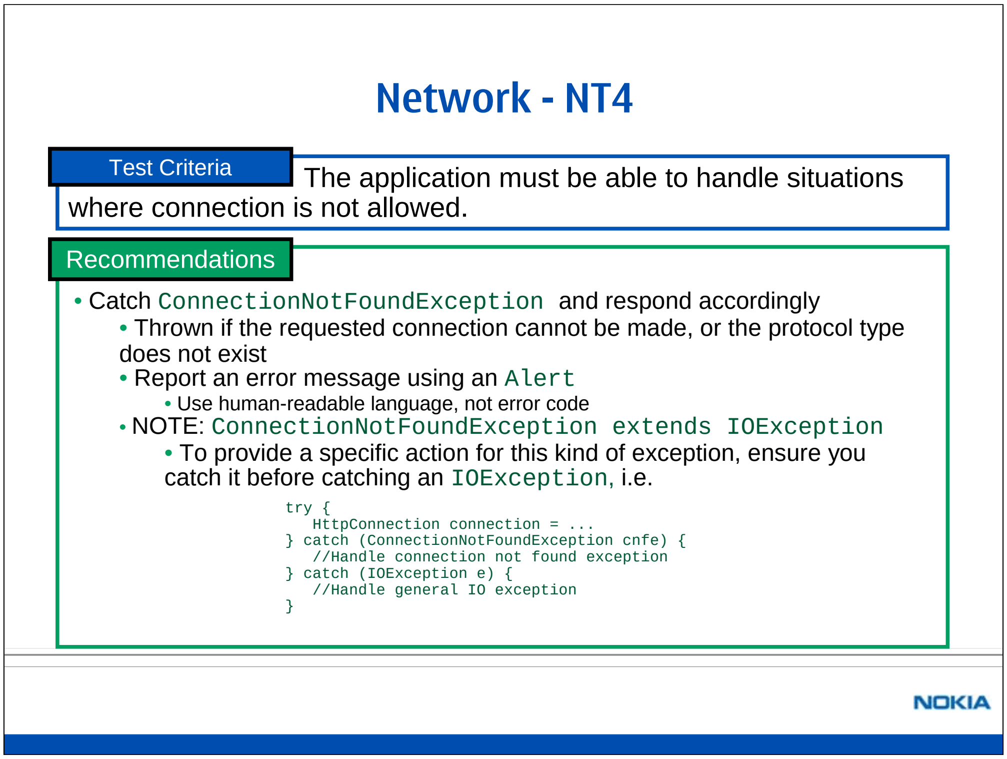
Recommendations

- Create a Thread to deal with network operations
  - Ensures UI doesn't hang when there are network delays
- Provide visual feedback while connecting, read and writing, especially if this is likely to take a long time

```
waitForm = new WaitForm(“Waiting...”);
display.setCurrent(waitForm);
String url = “http://...”;
Thread connectThread
   = new ConnectThread(url, waitForm, this);
connectionThread.start();
```

```
public class ConnectThread {
   public void run() {
      waitForm.setStatus(“Connecting…”);
      try {
          //make connection
          ...
      } catch (Exception e) {
          ...
      }
   }
```

To avoid hang-ups in the main UI thread is to put all lengthy or potentially blocking operations in separate threads. Network calls can block the user interface from redrawing or responding to user input until the call returns. In some cases having the user interface and a network connection on the same thread can cause a device to deadlock. For example, if an untrusted MIDlet attempts the make an HTTP connection, the device may need to prompt the user for permission to use this service. The connection blocks until this user permission is granted. However, if the device needs to use the same thread to prompt the user for this permission, the prompt cannot be displayed. This is because the network is blocking, waiting for the response from the user prompt. Thus, a deadlock can ensue leaving the application in a hung state.



Network - NT4

Test Criteria The application must be able to handle situations where connection is not allowed.

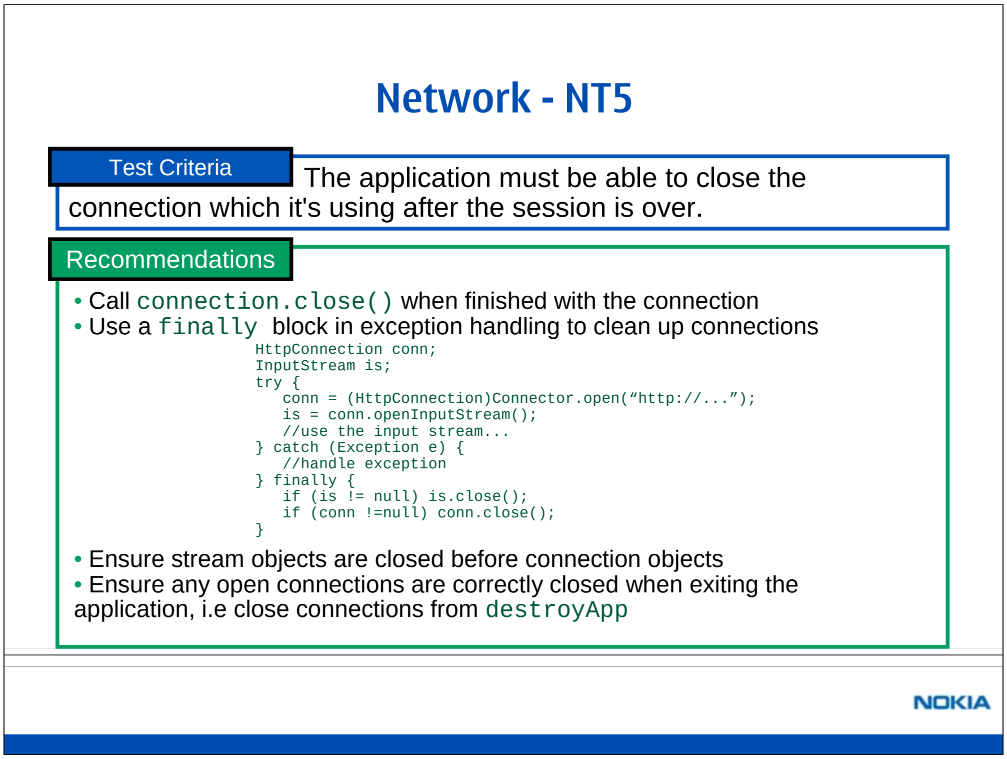
Recommendations

- Catch `ConnectionNotFoundException` and respond accordingly
  - Thrown if the requested connection cannot be made, or the protocol type does not exist
  - Report an error message using an `Alert`
    - Use human-readable language, not error code
  - NOTE: `ConnectionNotFoundException extends IOException`
    - To provide a specific action for this kind of exception, ensure you catch it before catching an `IOException`, i.e.

```
try {
   HttpConnection connection = ...
} catch (ConnectionNotFoundException cnfe) {
   //Handle connection not found exception
} catch (IOException e) {
   //Handle general IO exception
}
```

NOKIA

This test ensures the application can correctly report network problems to the user. A problem that could occur when making a connection, is that the connection is not allowed or the protocol type does not exist. In this case, the Connector.open method throws a ConnectionNotFoundException. To be able to report this particular problem to the user, this exception should be caught before the IOException class. This is due to the fact the ConnectionNotFoundException extends IOException.



Network - NT5

Test Criteria

The application must be able to close the connection which it's using after the session is over.

Recommendations

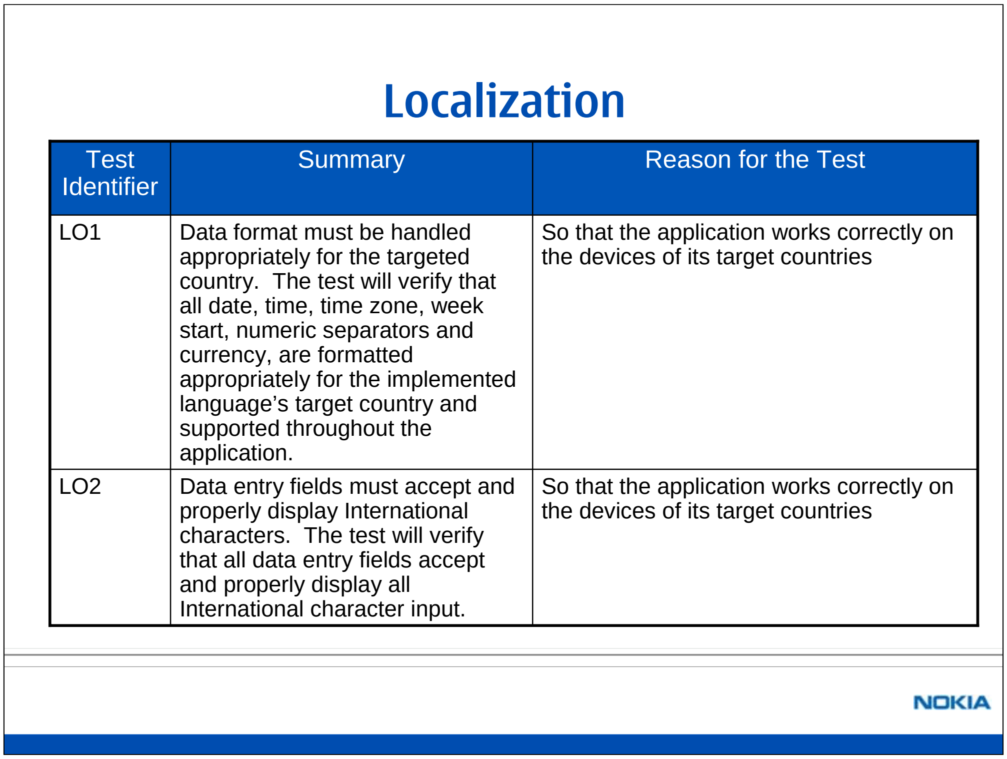
- Call `connection.close()` when finished with the connection
- Use a `finally` block in exception handling to clean up connections

```
HttpConnection conn;
InputStream is;
try {
   conn = (HttpConnection)Connector.open(“http://...”);
   is = conn.openInputStream();
   //use the input stream...
} catch (Exception e) {
   //handle exception
} finally {
   if (is != null) is.close();
   if (conn !=null) conn.close();
}
```

- Ensure stream objects are closed before connection objects
- Ensure any open connections are correctly closed when exiting the application, i.e close connections from `destroyApp`

The finally construct enables code to execute whether or not an exception occurred. Using finally is good to maintain the internal state of an object and to clean up non-memory resources.

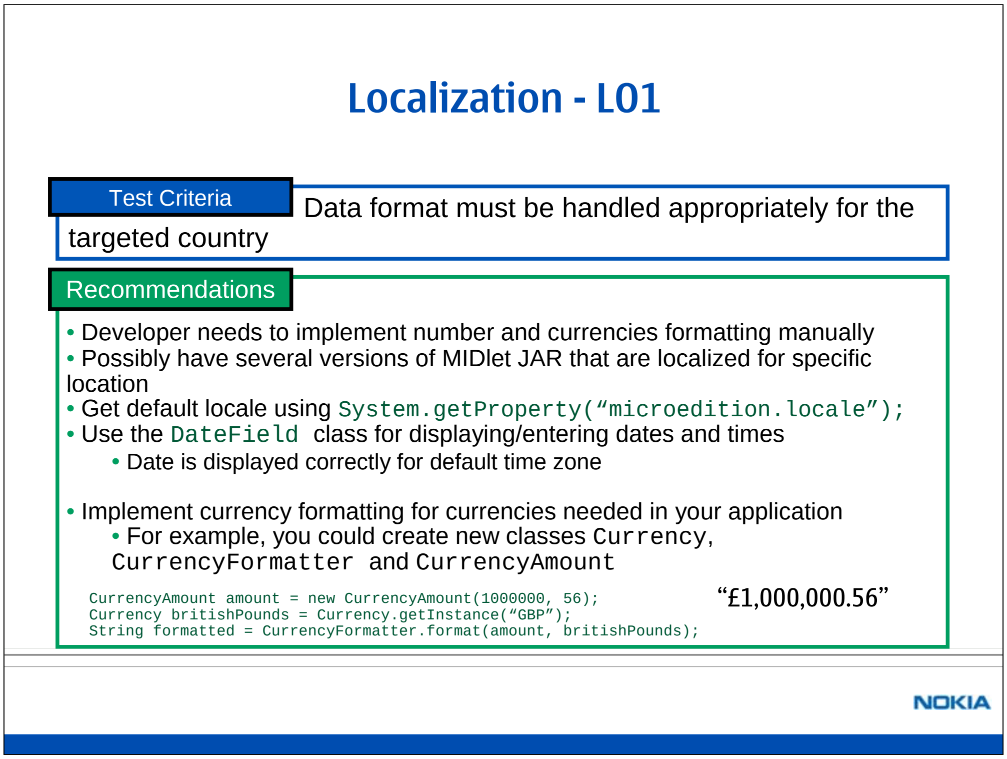
A finally block ensures the close method is executed whether or not an exception is thrown from within the try block. Therefore, the close method is guaranteed to be called before the method exits. You are then sure the socket is closed and you have not leaked a resource.



Localization

Test Identifier	Summary	Reason for the Test
LO1	Data format must be handled appropriately for the targeted country. The test will verify that all date, time, time zone, week start, numeric separators and currency, are formatted appropriately for the implemented language’s target country and supported throughout the application.	So that the application works correctly on the devices of its target countries
LO2	Data entry fields must accept and properly display International characters. The test will verify that all data entry fields accept and properly display all International character input.	So that the application works correctly on the devices of its target countries

The emphasis of this category is to check that your application deals with changes in language, alphabets, date and money formats.



Localization - L01

Test Criteria

Data format must be handled appropriately for the targeted country

Recommendations

- Developer needs to implement number and currencies formatting manually
- Possibly have several versions of MIDlet JAR that are localized for specific location
- Get default locale using `System.getProperty(“microedition.locale”);`
- Use the `DateField` class for displaying/entering dates and times
  - Date is displayed correctly for default time zone
- Implement currency formatting for currencies needed in your application
  - For example, you could create new classes `Currency`, `CurrencyFormatter` and `CurrencyAmount`

```
CurrencyAmount amount = new CurrencyAmount(1000000, 56);
Currency britishPounds = Currency.getInstance(“GBP”);
String formatted = CurrencyFormatter.format(amount, britishPounds);
```

“£1,000,000.56”

CLDC provides no support for any formatting of strings, numbers, currencies or other locale-specific operations. MIDP, however, requires the implementation to define a microedition.locale system property that returns the device's locale in language-country format (as in "en-US" or "fr-FR"). Its value can be retrieved using the method System.getProperty. The MIDP specification allows a null value to be used in the system property. However, in Nokia implementations this value is never null. The CLDC system property microedition.encoding defines the default character encoding of the device. Its value can be retrieved using the method System.getProperty.

Adding localization features to a MIDlet is valuable, but may increase the size of your MIDlet. If MIDlet size is an issue for a particular MIDP device, you might wish to generate several different compiled versions of the MIDlet. Each version could be localized for one or more locations or languages. Designing your MIDlet so that it is easily localized may be beneficial in this respect as well.

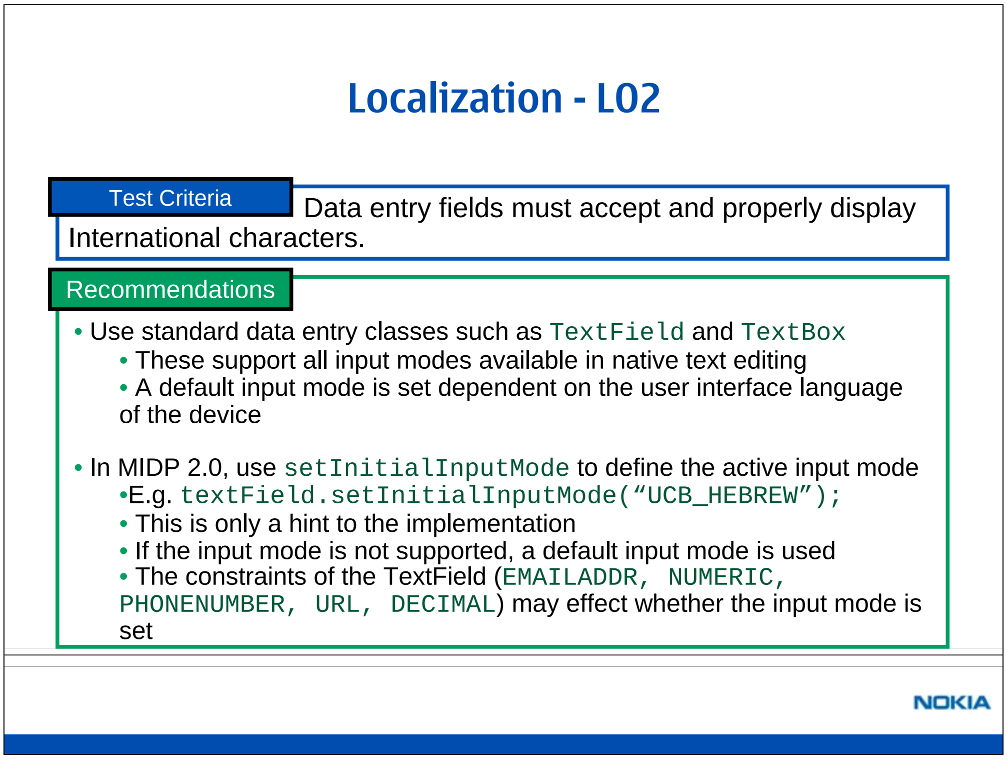
Parsing and displaying dates and times is often complicated because of formatting and locale issues. MIDP defines only subsets of the Calendar, Date and TimeZone classes, and does not include any form of DateFormat. To deal with Dates and times, use the javax.microedition.lcdui.DateField class. This is an interactive user interface component that displays a date, time, or both. It also allows you to edit the date and time. To display date and times correctly, the DateField needs to know which time zone to use. DateField provides two constructors:

```
public DateField( String label, int mode );
public DateField( String label, int mode, java.util.TimeZone zone );
```

The default time zone is used in the two-argument constructor, while in the three-argument constructor; you can specify the TimeZone that should be used. The mode of the DateField can also be set in these two constructors; this can either be DateField.DATE, DateField.TIME or DateField.DATE_TIME.

When dealing with currencies, the currency symbol, the placement of the currency symbol, the character used to separate thousands, the number of digits after the decimal, and the character to separate the decimal amount must be considered. Since there is no standard Java ME API available to format currency amounts, you must implement methods to support the required currencies within the application yourself. One suggestion is to create a CurrencyFormatter class that takes a money amount and a currency, and returns a correctly formatted string. To support this class, two classes Currency and CurrencyAmount could also be created that contain details needed to format a currency, and the amount to format respectively. So the call to the CurrencyFormatter could be:

```
CurrencyAmount amount = new CurrencyAmount(1000000, 56);
Currency britishPounds = Currency.getInstance(“GBP”);
String formatted = CurrencyFormatter.format(amount, britishPounds);
// formatted = “£1,000,000.56”;
```



Localization - L02

Test Criteria

Data entry fields must accept and properly display International characters.

Recommendations

- Use standard data entry classes such as `TextField` and `TextBox`
  - These support all input modes available in native text editing
  - A default input mode is set dependent on the user interface language of the device

- In MIDP 2.0, use `setInitialInputMode` to define the active input mode
  - E.g. `textField.setInitialInputMode(“UCB_HEBREW”);`
  - This is only a hint to the implementation
  - If the input mode is not supported, a default input mode is used
  - The constraints of the TextField (`EMAILADDR, NUMERIC, PHONENUMBER, URL, DECIMAL`) may effect whether the input mode is set

NOKIA

Both TextField and TextBox support all different input modes that are available in native text editing view. If implementation supports special input mechanisms like Chinese or Thai etc. input those are available via TextField and TextBox. Note that different input constraints limit the set of input modes user can use in a particular TextBox/TextField.

In NUMERIC TextField/TextBox the implementation supports input modes required to enter numbers. For example, in Latin device there is probably one number input mode so there is no way to switch to different modes. But in Arabic device there are different number input modes for Latin number and Arabic-Indic numbers. User can change between these input modes.

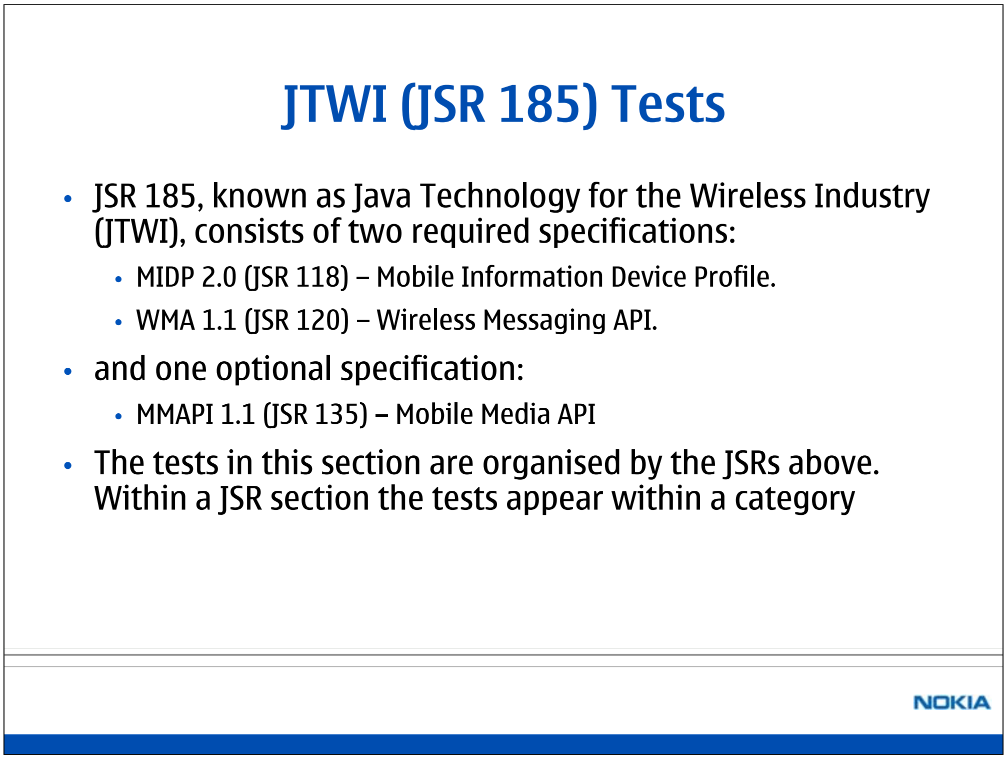
In MIDP 2.0 it possible for MIDlet to set the initial input mode. This allows the MIDlet to define which input mode is initially active when the editing session starts. In many user interface styles the input mode is normally changed with the #-key in the editor. Available input modes are dependent on the user interface language of the device.

The application can use setInitialInputMode method to define which input mode is active when the editing starts, but user still can access all modes that are available, for example, with the #-key.

Also the constraint and modifiers that has been set to the editor may limit the available input modes user or application can access. If the constraint does not allow the requested input mode then the initial input mode is ignored and default input mode is used. Similarly if the requested input mode is not available in the given user interface language of the device the input mode is ignored and default input mode is used.

All implementations must support following input modes:

```
MIDP_UPPERCASE_LATIN
MIDP_LOWERCASE_LATIN
IS_LATIN
IS_LATIN_DIGITS
UCB_BASIC_LATIN
```



JTWI (JSR 185) Tests

- JSR 185, known as Java Technology for the Wireless Industry (JTWI), consists of two required specifications:
  - MIDP 2.0 (JSR 118) – Mobile Information Device Profile.
  - WMA 1.1 (JSR 120) – Wireless Messaging API.
- and one optional specification:
  - MMAPI 1.1 (JSR 135) – Mobile Media API
- The tests in this section are organised by the JSRs above. Within a JSR section the tests appear within a category

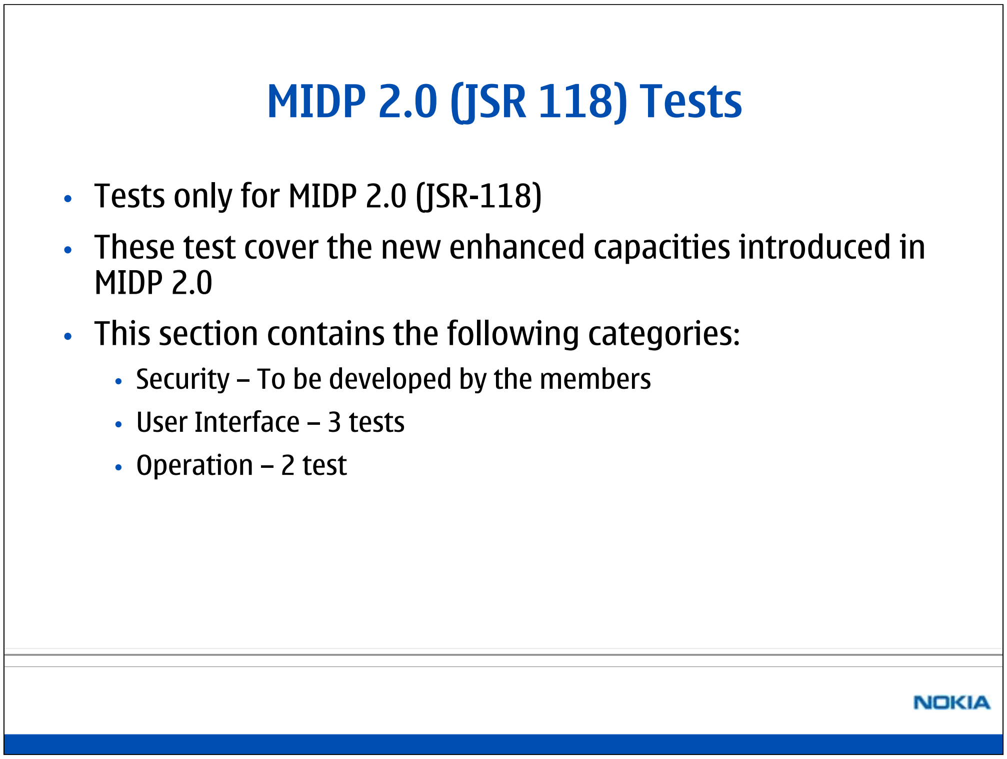
JSR 185, known as Java Technology for the Wireless Industry (JTWI), consists of two required specifications:

- ***MIDP 2.0 (JSR 118) – Mobile Information Device Profile.***
- ***WMA 1.1 (JSR 120) – Wireless Messaging API.***

and one optional specification:

- ***MMAPI 1.1 (JSR 135) – Mobile Media API***

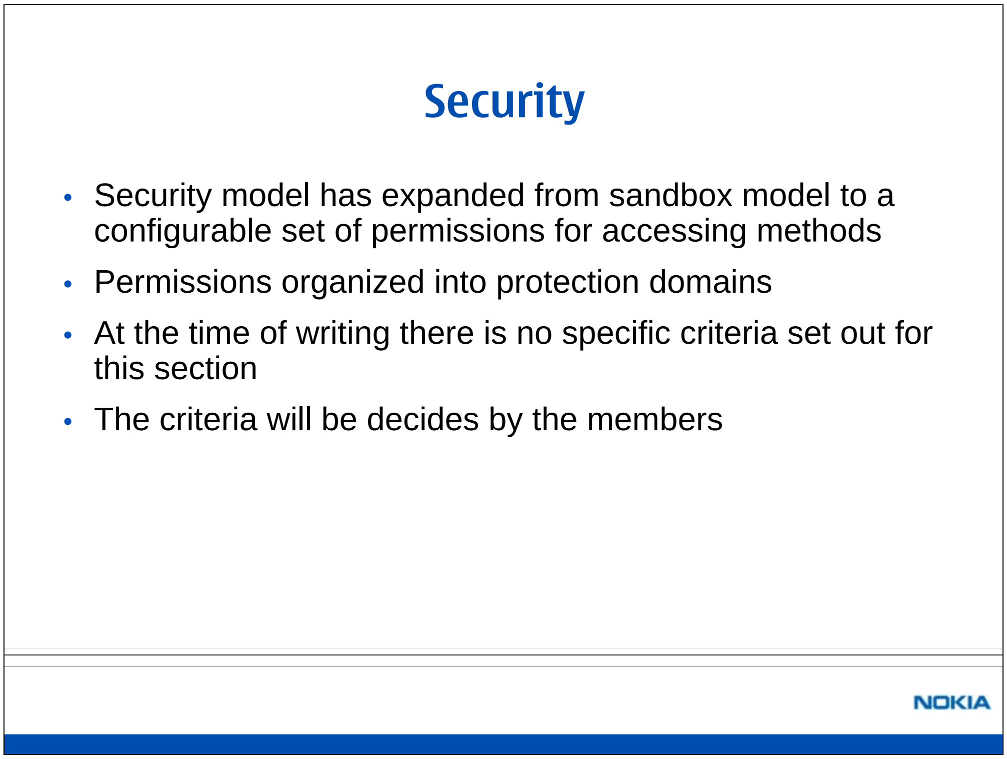
The tests in this section are organised by the JSRs above.



MIDP 2.0 (JSR 118) Tests

- Tests only for MIDP 2.0 (JSR-118)
- These test cover the new enhanced capacities introduced in MIDP 2.0
- This section contains the following categories:
  - Security – To be developed by the members
  - User Interface – 3 tests
  - Operation – 2 test

The MIDP 2.0 release, with its new and enhanced capabilities requires additional tests to be specified. This section includes brief descriptions of these capabilities organized by the UTI's current testing categories.



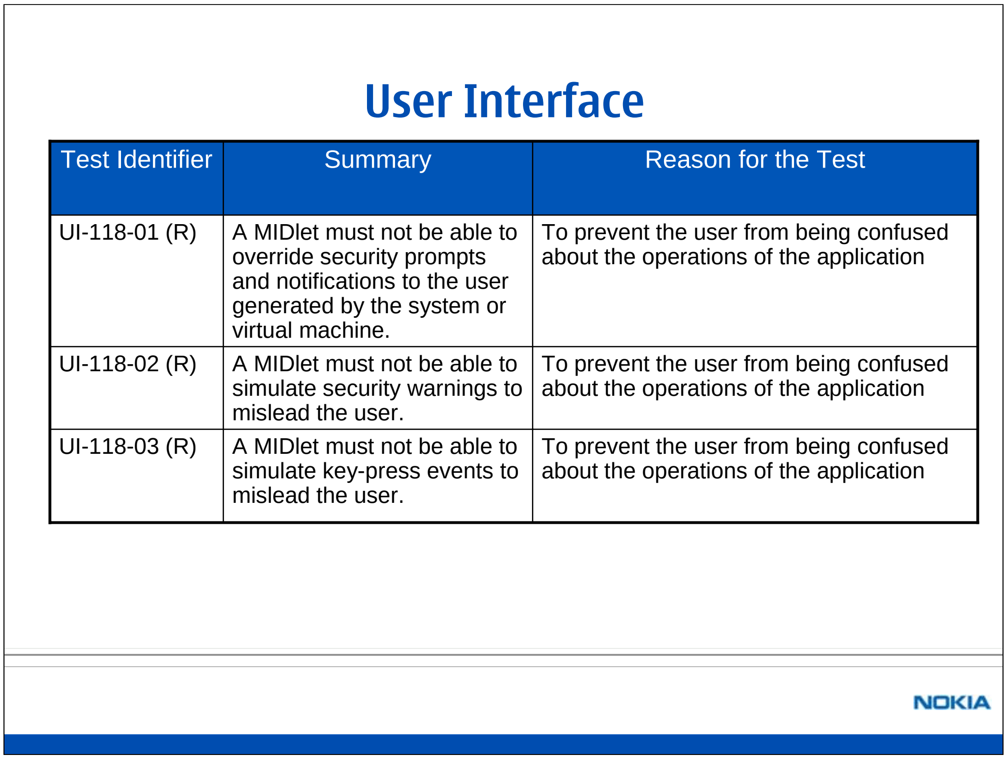
Security

- Security model has expanded from sandbox model to a configurable set of permissions for accessing methods
- Permissions organized into protection domains
- At the time of writing there is no specific criteria set out for this section
- The criteria will be decides by the members

A significant addition to the MID Profile is security. The security model has expanded from a strict sandbox model to a highly configurable set of permissions for accessing certain method calls. Permissions are organized into protection domains. Many of the tests center around the proper behavior of these domains. There are four domains:

1. ***Manufacturer***
2. ***Operator***
3. ***Trusted Third Party***
4. ***Untrusted***

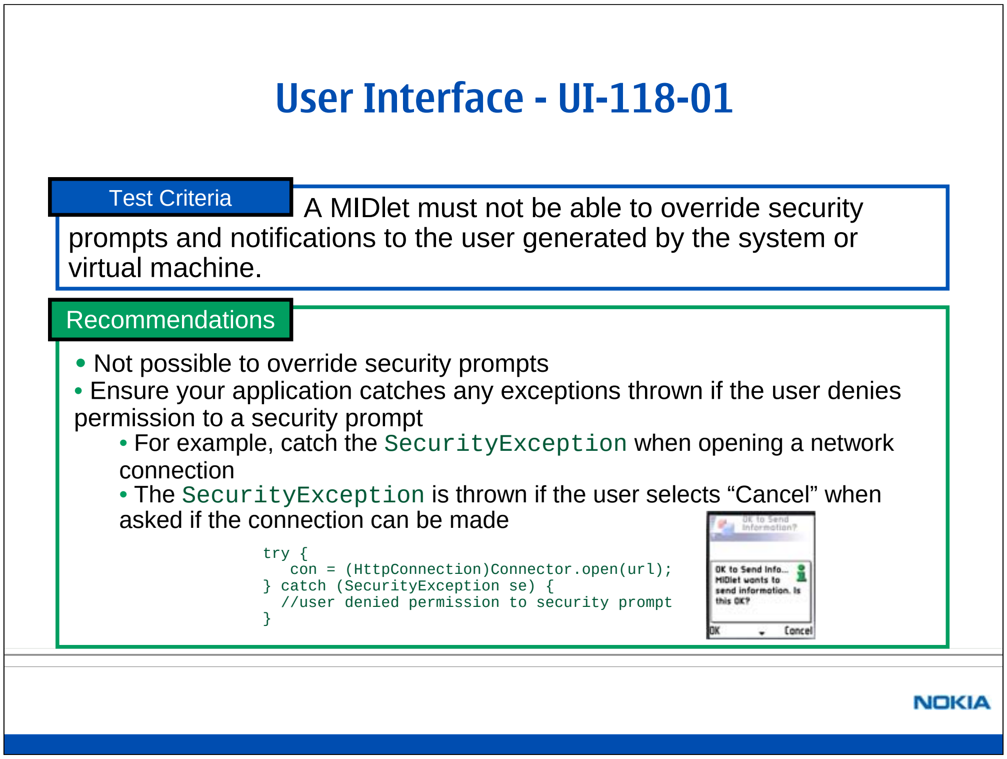
A MIDlet that has not been signed or whose signature cannot be verified and authenticated with any one of the Manufacturer, Operators or Trusted Third Party certificates in the device is considered Untrusted.



User Interface

Test Identifier	Summary	Reason for the Test
UI-118-01 (R)	A MIDlet must not be able to override security prompts and notifications to the user generated by the system or virtual machine.	To prevent the user from being confused about the operations of the application
UI-118-02 (R)	A MIDlet must not be able to simulate security warnings to mislead the user.	To prevent the user from being confused about the operations of the application
UI-118-03 (R)	A MIDlet must not be able to simulate key-press events to mislead the user.	To prevent the user from being confused about the operations of the application

The main emphasis of this category is to prevent the user from being confused about the operations of the application by not simulating security warnings and key-press events.



User Interface - UI-118-01

Test Criteria

A MIDlet must not be able to override security prompts and notifications to the user generated by the system or virtual machine.

Recommendations

- Not possible to override security prompts
- Ensure your application catches any exceptions thrown if the user denies permission to a security prompt
  - For example, catch the `SecurityException` when opening a network connection
  - The `SecurityException` is thrown if the user selects “Cancel” when asked if the connection can be made

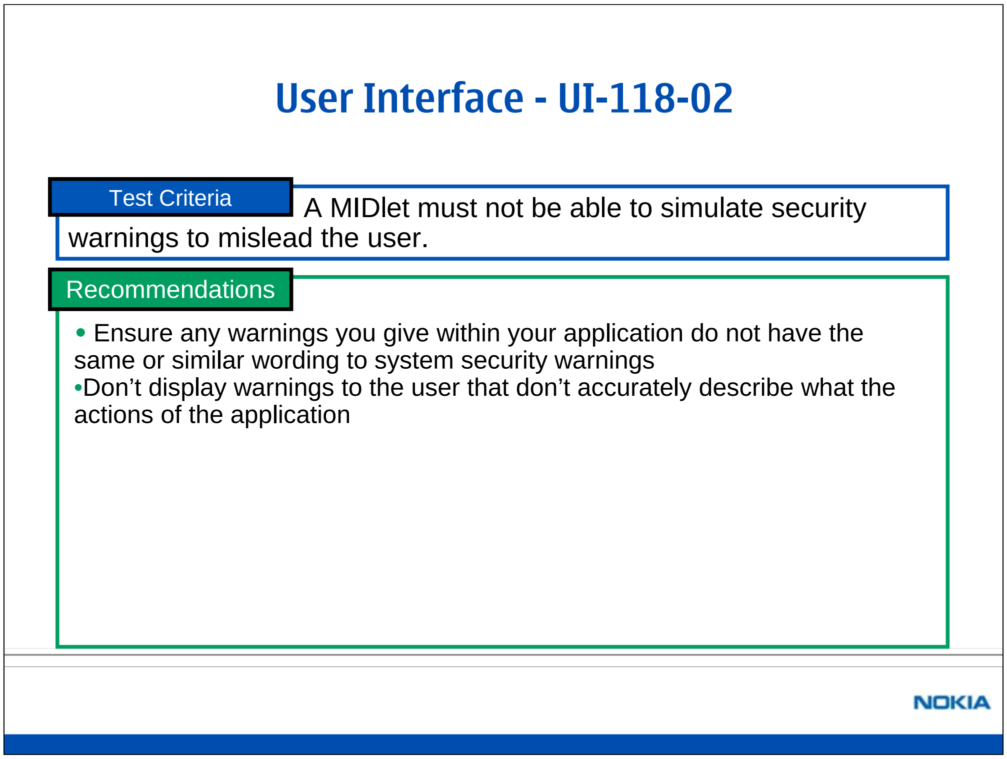
```
try {
   con = (HttpConnection)Connector.open(url);
} catch (SecurityException se) {
  //user denied permission to security prompt
}
```

OK to Send Information?

OK to Send Info...
MIDlet wants to send information. Is this OK?

If a security prompt is shown to the user, for example when opening a network connection, the user has the option to decline permission to open the connection.

If the user does decline, a SecurityException throw by the Connector.open method. This should be caught in your application, and dealt with appropriately.



User Interface - UI-118-02

Test Criteria

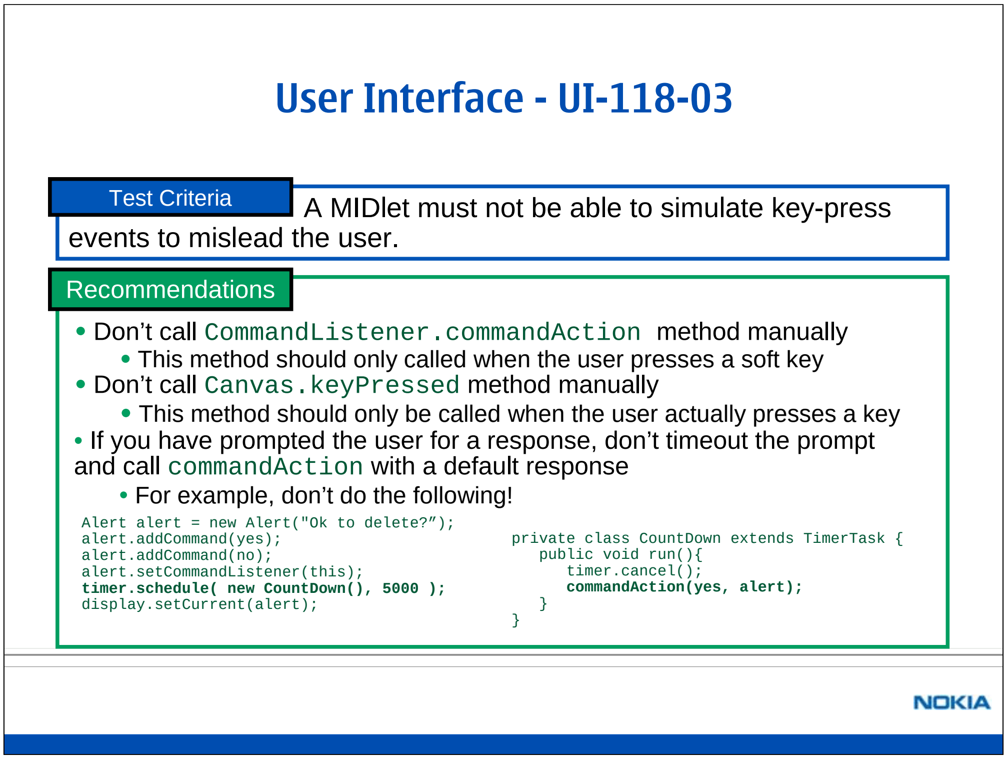
A MIDlet must not be able to simulate security warnings to mislead the user.

Recommendations

- Ensure any warnings you give within your application do not have the same or similar wording to system security warnings
- Don’t display warnings to the user that don’t accurately describe what the actions of the application

NOKIA

To ensure the user does not get confused, make sure that any warning, prompts or error messages within your application do not resemble those displayed by the system when there is a security warning



User Interface - UI-118-03

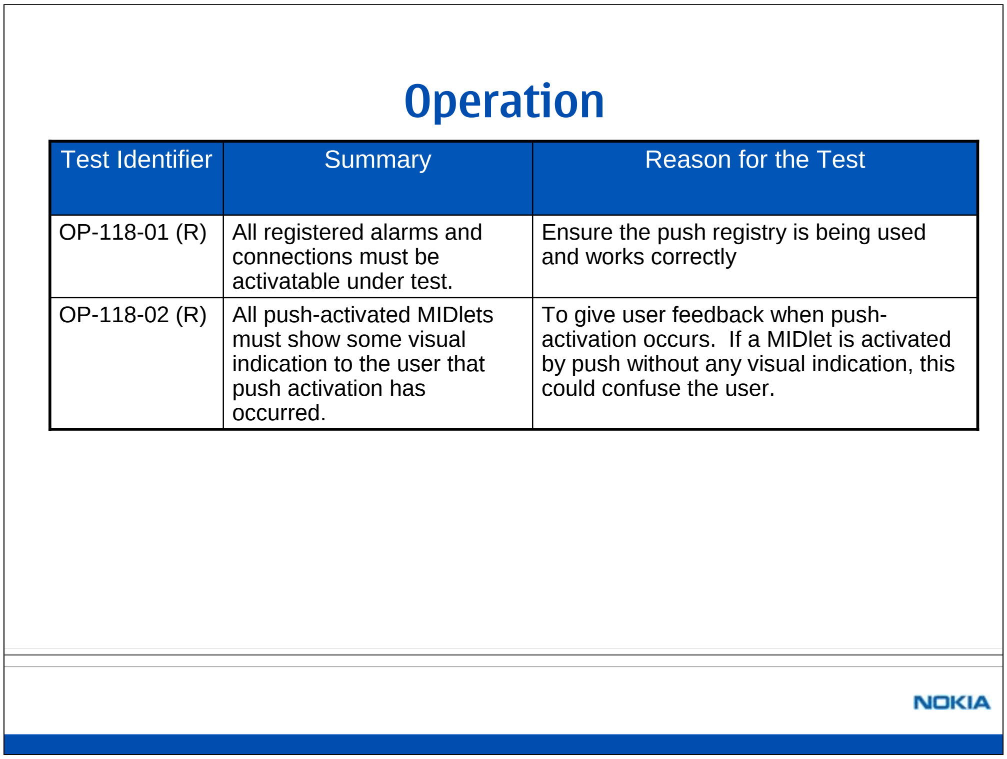
Test Criteria A MIDlet must not be able to simulate key-press events to mislead the user.

Recommendations

- Don't call `CommandListener.commandAction` method manually
  - This method should only called when the user presses a soft key
- Don't call `Canvas.keyPressed` method manually
  - This method should only be called when the user actually presses a key
- If you have prompted the user for a response, don't timeout the prompt and call `commandAction` with a default response
  - For example, don't do the following!

```
Alert alert = new Alert("Ok to delete?");
alert.addCommand(yes);
alert.addCommand(no);
alert.setCommandListener(this);
timer.schedule( new CountDown(), 5000 );
display.setCurrent(alert);
```

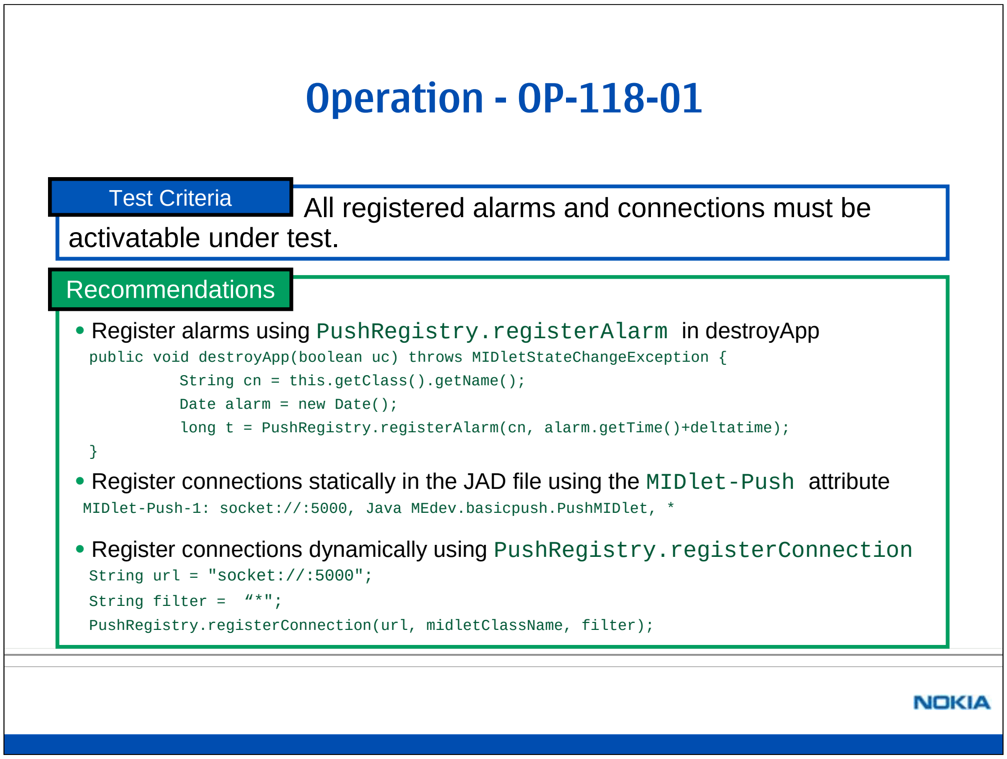
```
private class CountDown extends TimerTask {
   public void run(){
      timer.cancel();
      commandAction(yes, alert);
   }
}
```



Operation

Test Identifier	Summary	Reason for the Test
OP-118-01 (R)	All registered alarms and connections must be activatable under test.	Ensure the push registry is being used and works correctly
OP-118-02 (R)	All push-activated MIDlets must show some visual indication to the user that push activation has occurred.	To give user feedback when push-activation occurs. If a MIDlet is activated by push without any visual indication, this could confuse the user.

The emphasis of this category is to ensure the application can interact correctly with device hardware and software, such as the Push Registry. The tests make sure your application is using the push registry correctly, and gives user feedback when push-activation occurs.



Operation - OP-118-01

Test Criteria

All registered alarms and connections must be activatable under test.

Recommendations

- Register alarms using `PushRegistry.registerAlarm` in destroyApp

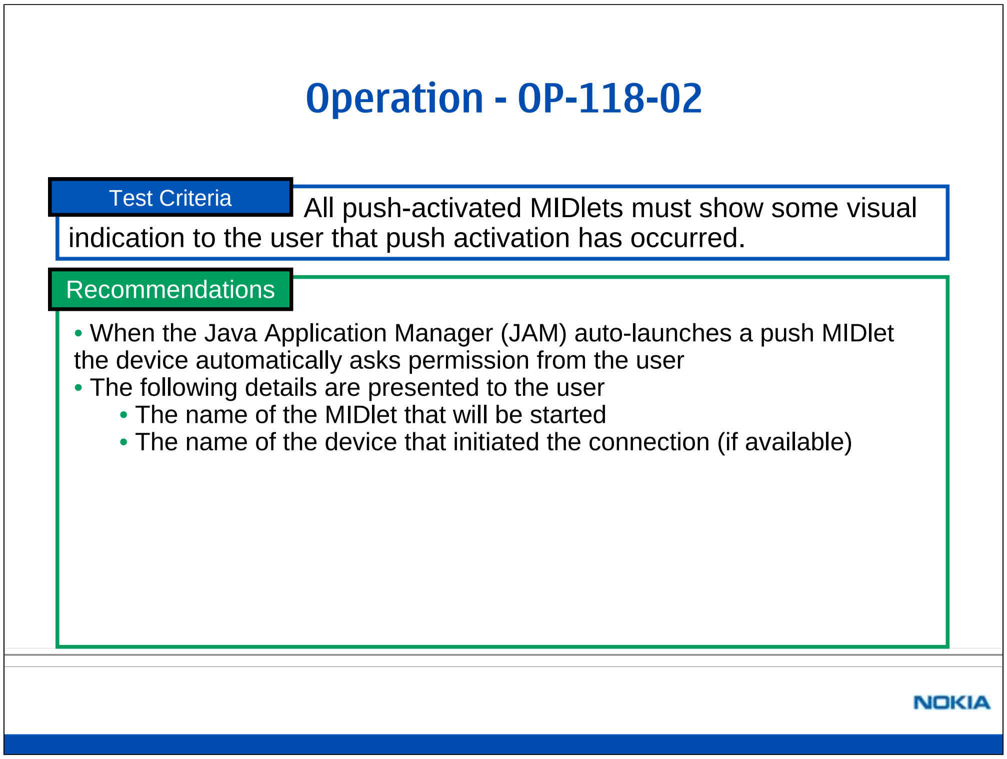
```
public void destroyApp(boolean uc) throws MIDletStateChangeException {
          String cn = this.getClass().getName();
          Date alarm = new Date();
          long t = PushRegistry.registerAlarm(cn, alarm.getTime()+deltatime);
}
```

- Register connections statically in the JAD file using the `MIDlet-Push` attribute

```
MIDlet-Push-1: socket://:5000, Java MEdev.basicpush.PushMIDlet, *
```

- Register connections dynamically using `PushRegistry.registerConnection`

```
String url = "socket://:5000";
String filter =  “*";
PushRegistry.registerConnection(url, midletClassName, filter);
```



Operation - OP-118-02

Test Criteria

All push-activated MIDlets must show some visual indication to the user that push activation has occurred.

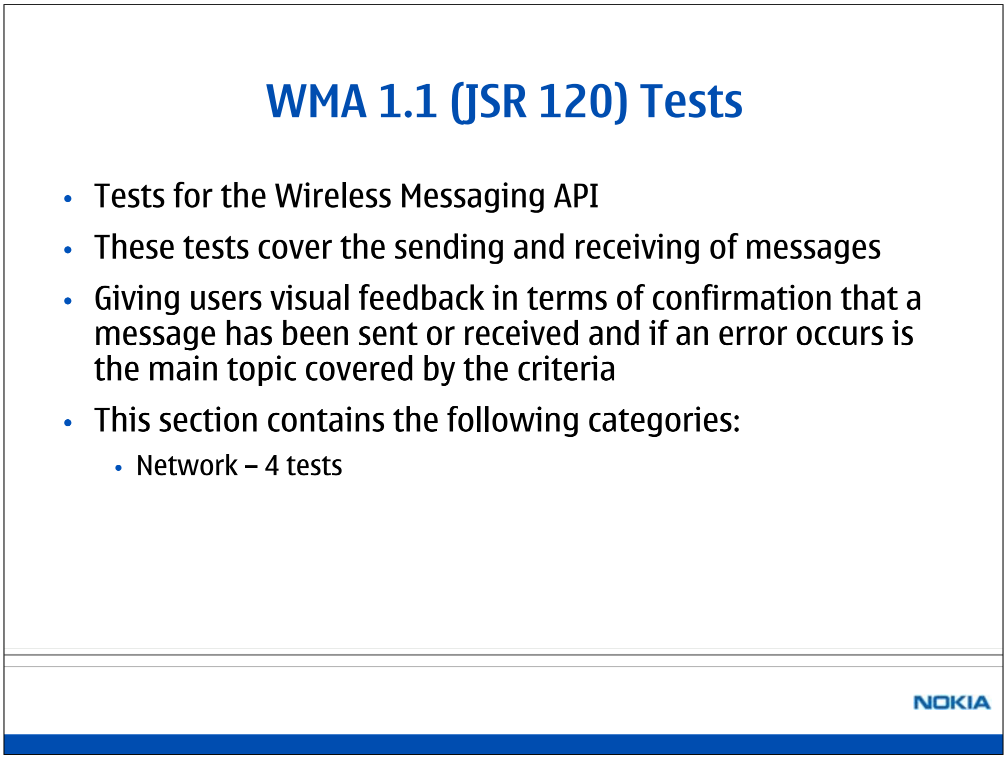
Recommendations

- When the Java Application Manager (JAM) auto-launches a push MIDlet the device automatically asks permission from the user
- The following details are presented to the user
  - The name of the MIDlet that will be started
  - The name of the device that initiated the connection (if available)

This test checks that your application gives user feedback when push-activation occurs so as to not confuse the user. When the Java Application Manager (JAM) auto-launches a push MIDlet the device automatically asks permission from the user. The following details should be presented to the user

- ***The name of the MIDlet that will be started***

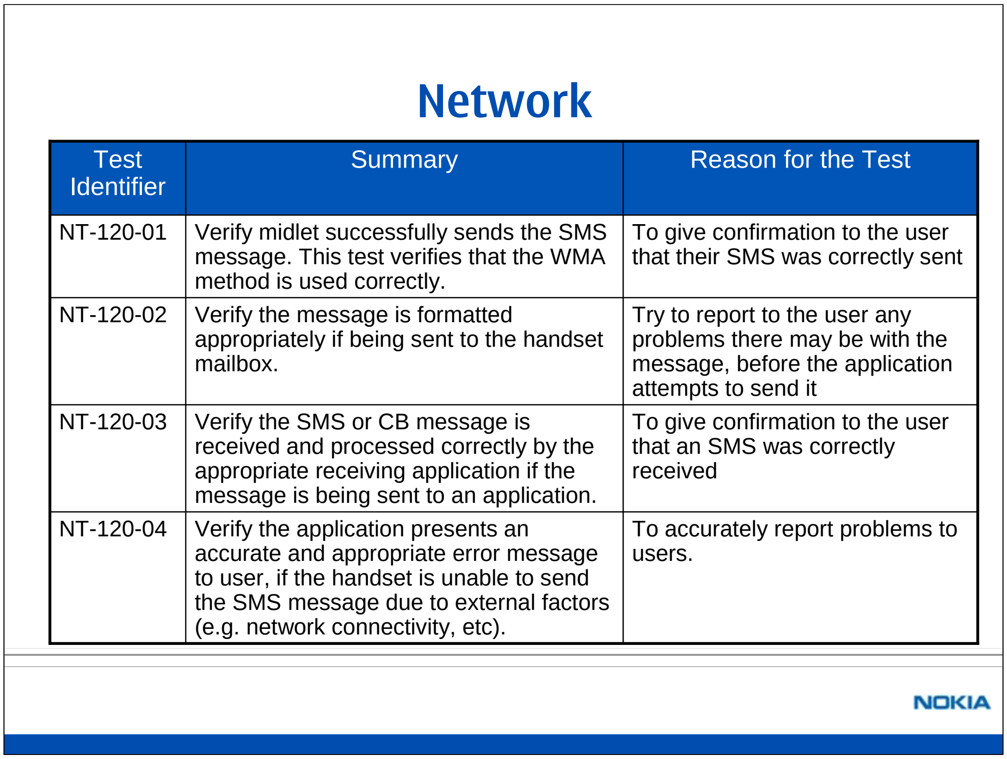
- ***The name of the device that initiated the connection (if available)***



WMA 1.1 (JSR 120) Tests

- Tests for the Wireless Messaging API
- These tests cover the sending and receiving of messages
- Giving users visual feedback in terms of confirmation that a message has been sent or received and if an error occurs is the main topic covered by the criteria
- This section contains the following categories:
  - Network – 4 tests

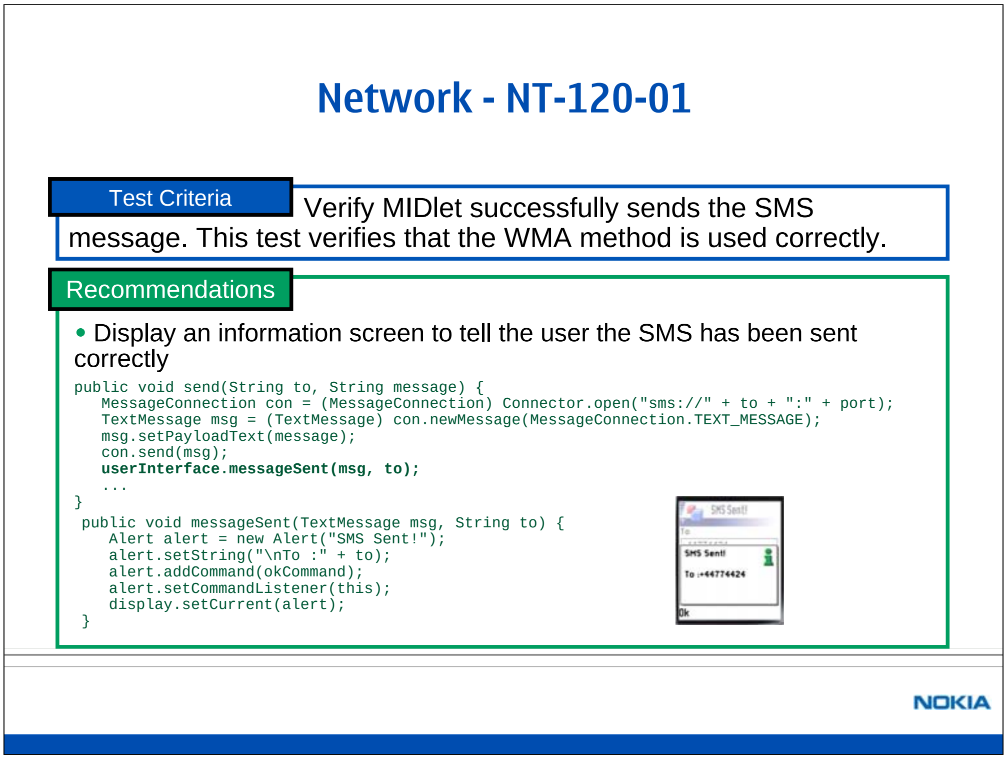
The tests in this section are for applications that use version 1.1 (JSR 120) of the Wireless Messaging API (WMA). That is, they represent a common set of tests that can be applied to applications that send and receive messages through the WMA. The tests are organized by test category.



Network

Test Identifier	Summary	Reason for the Test
NT-120-01	Verify midlet successfully sends the SMS message. This test verifies that the WMA method is used correctly.	To give confirmation to the user that their SMS was correctly sent
NT-120-02	Verify the message is formatted appropriately if being sent to the handset mailbox.	Try to report to the user any problems there may be with the message, before the application attempts to send it
NT-120-03	Verify the SMS or CB message is received and processed correctly by the appropriate receiving application if the message is being sent to an application.	To give confirmation to the user that an SMS was correctly received
NT-120-04	Verify the application presents an accurate and appropriate error message to user, if the handset is unable to send the SMS message due to external factors (e.g. network connectivity, etc).	To accurately report problems to users.

The emphasis of this category is make sure your applications provides as much user feedback as possible when sending and receiving message and if any problems occurred.



Network - NT-120-01

Test Criteria Verify MIDlet successfully sends the SMS message. This test verifies that the WMA method is used correctly.

Recommendations

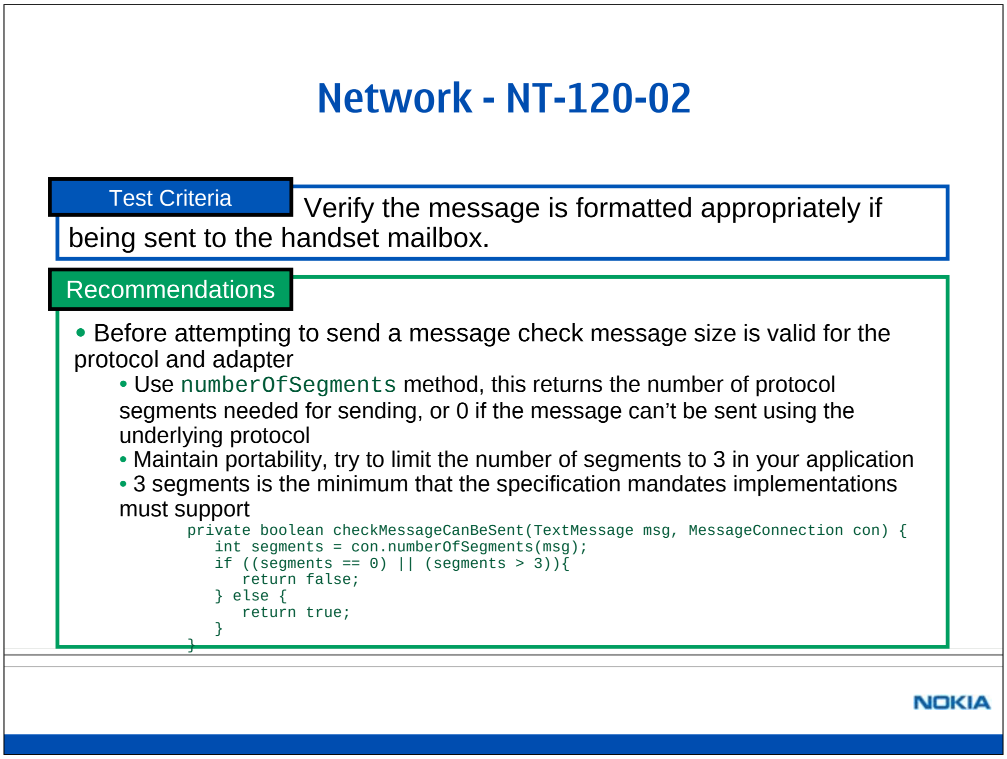
- Display an information screen to tell the user the SMS has been sent correctly

```
public void send(String to, String message) {
   MessageConnection con = (MessageConnection) Connector.open("sms://" + to + ":" + port);
   TextMessage msg = (TextMessage) con.newMessage(MessageConnection.TEXT_MESSAGE);
   msg.setPayloadText(message);
   con.send(msg);
   userInterface.messageSent(msg, to);
   ...
}
 public void messageSent(TextMessage msg, String to) {
    Alert alert = new Alert("SMS Sent!");
    alert.setString("\nTo :" + to);
    alert.addCommand(okCommand);
    alert.setCommandListener(this);
    display.setCurrent(alert);
 }
```

SMS Sent!

SMS Sent!

To :+44774424



Network - NT-120-02

Test Criteria

Verify the message is formatted appropriately if being sent to the handset mailbox.

Recommendations

- Before attempting to send a message check message size is valid for the protocol and adapter
  - Use `numberOfSegments` method, this returns the number of protocol segments needed for sending, or 0 if the message can't be sent using the underlying protocol
  - Maintain portability, try to limit the number of segments to 3 in your application
  - 3 segments is the minimum that the specification mandates implementations must support

```
private boolean checkMessageCanBeSent(TextMessage msg, MessageConnection con) {
   int segments = con.numberOfSegments(msg);
   if ((segments == 0) || (segments > 3)){
      return false;
   } else {
      return true;
   }
}
```

A message sent on the SMS network is typically limited to 160 single-byte (text) characters (or 140 binary bytes). Segmentation and reassembly (SAR) is a feature of some low-level transports that entails breaking a large message into a number of smaller sequential, ordered segments or transmission units. Some transports impose a limit on the size of a single message or on the number of segments used for it. The specification mandates that WMA-based SMS implementations must support at least three SMS-protocol segments for a single message. Some implementations may support more, but applications will be more portable if they adhere to the three-segment limit, and refrain from sending messages that are larger than 456 single-byte characters, 198 double-byte characters, or 399 binary bytes. Sending a larger message may result in an IOException.

The WMA provides a special method called MessageConnection.numberOfSegments(), which provides segmentation information for a Message before it is sent. Method numberOfSegments() returns the number of segments it will take to send a message (even if the connection is closed), or 0 if the message can't be sent.

Your application should call this method to ensure that the message can be sent. The code below calls this method, and checks the result is not above 3 or equals to 0. In then informs the user if there is a problem and they should attempt to reduce the number of characters used in their message

Network - NT-120-03

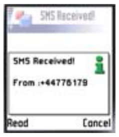
Test Criteria

Verify the SMS or CB message is received and processed correctly by the appropriate receiving application if the message is being sent to an application.

Recommendations

- When a message is successful received, indicate this to the user. For example:
    - Display a status box “Message Received” or an “envelope” icon
    - Display an `Alert` with the title “Message Received”, possible giving details of the sender and the option to actually read the message

```
public void receiveMessage() {
   Message msg = null;
   try {
      msg = smsConnection.receive();
      if ((msg != null) && (msg instanceof TextMessage)) {
          userInterface.messageReceived((TextMessage)msg);
      }
   } catch (IOException ioe){
     //handle exception
   }
}
```

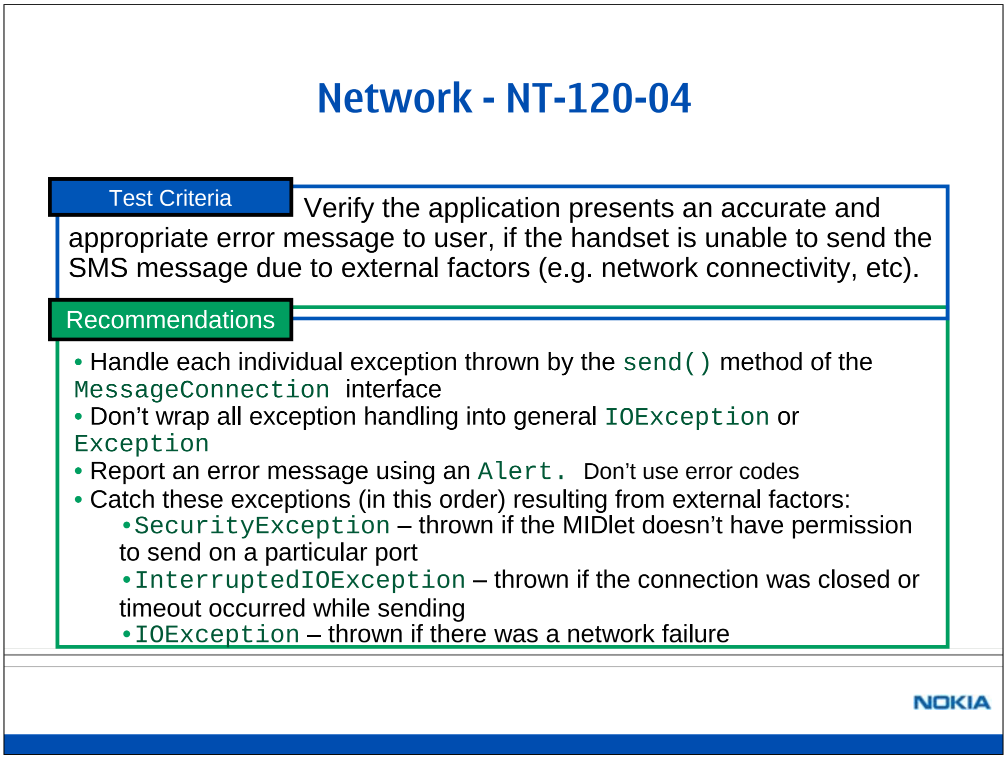


SMS Received!

SMS Received!

From :+44776179

NOKIA



Network - NT-120-04

Test Criteria Verify the application presents an accurate and appropriate error message to user, if the handset is unable to send the SMS message due to external factors (e.g. network connectivity, etc).

Recommendations

- Handle each individual exception thrown by the `send()` method of the `MessageConnection` interface
- Don't wrap all exception handling into general `IOException` or `Exception`
- Report an error message using an `Alert.` Don't use error codes
- Catch these exceptions (in this order) resulting from external factors:
  - `SecurityException` – thrown if the MIDlet doesn't have permission to send on a particular port
  - `InterruptedIOException` – thrown if the connection was closed or timeout occurred while sending
  - `IOException` – thrown if there was a network failure

When using the send() method of the MessageConnection interface, your application may encounter some exceptions that should be handled. The following exceptions are thrown by this method as result of problems with external factors: SecurityException, InterruptedIOException and IOException.

SecurityException is thrown if the application doesn't have permission to send a message at the given port

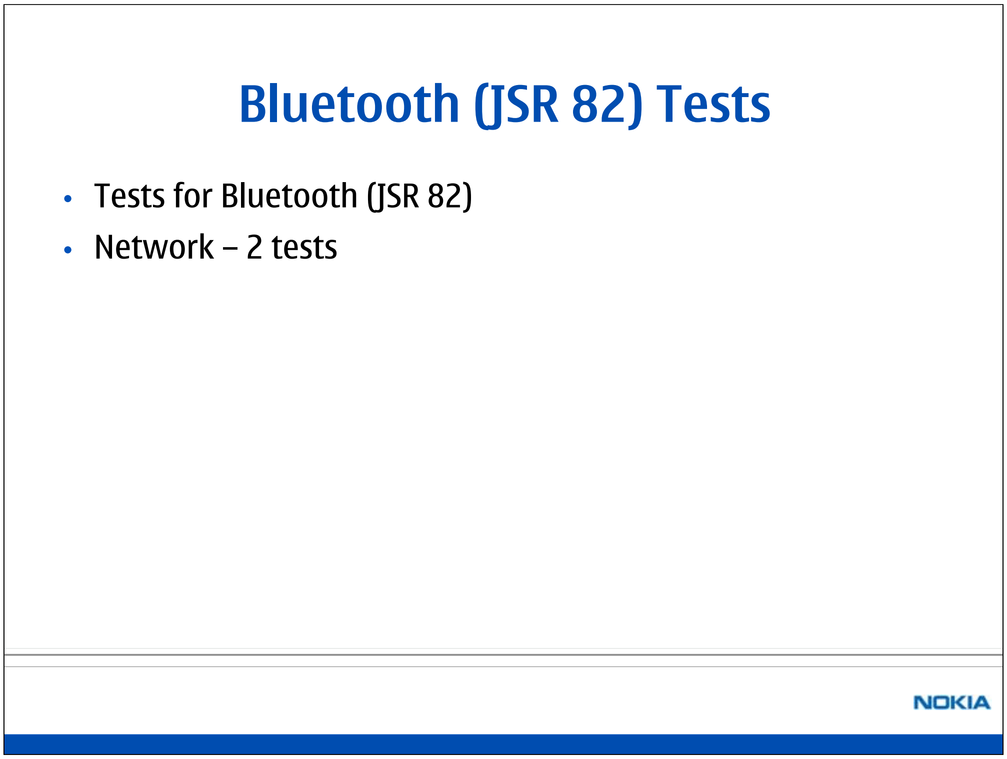
InterruptedIOException is thrown if a timeout occurred while trying to send the message, or the Connection object was closed during the send operation.

IOException is thrown if the message couldn't be sent, or there was a network failure, or the connection was closed

Each of these exceptions should be caught, and an appropriate message displayed. It is recommended that the last exception in the try catch block should be IOException. This is so that the InterruptedIOException may be dealt with appropriately and not just generalized as an IOException.

The code below shows how the sending code should be handled, the exceptions that are thrown and possible error message that can be displayed to the user:

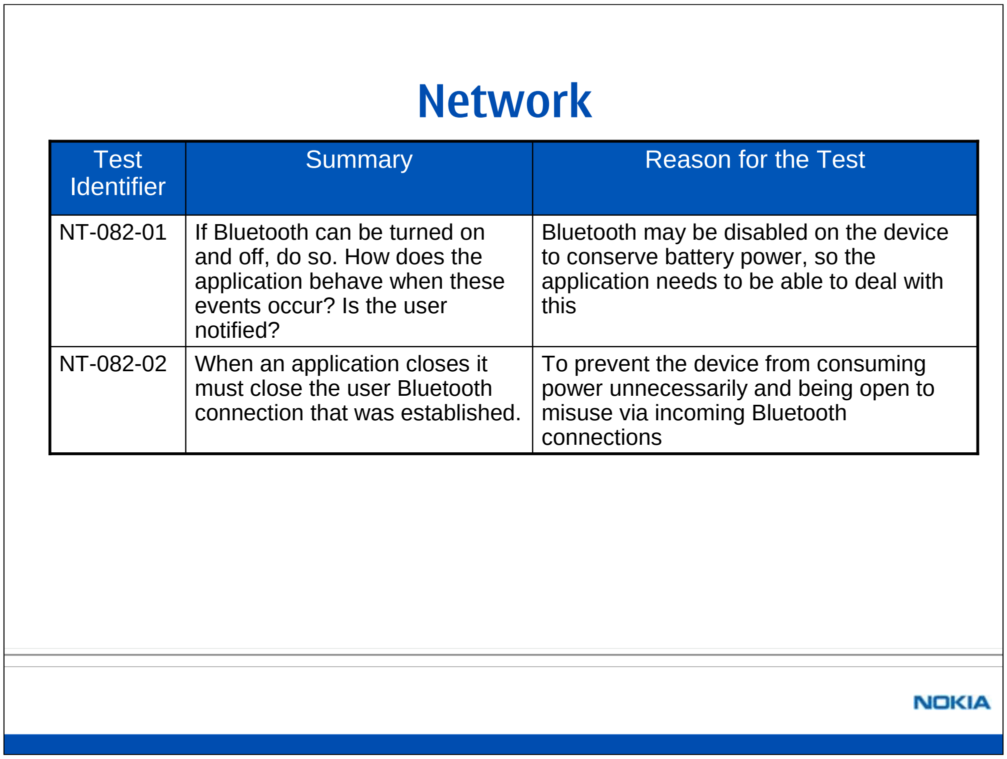
```
private MessageConnection con;
...
public void sendMessage() {
               try {
                       TextMessage sms = (TextMessage)conn.newMessage(MessageConnection.TEXT_MESSAGE);
                       sms.setAddress("sms://+" + addressField.getString());
                       sms.setPayloadText(messageField.getString());
                       con.send(sms);
               } catch (SecurityException e) {
//Explain there was a problem with security when //attempting to send the SMS
//Ask the user if they would like to try to resend or //maybe check any settings that may be available
               } catch (InterruptedIOException e) {
                       //Explain there was a problem when attempting to send
                       //the SMS.
                       //A timeout occurred
//Ask the user if they would like to try to resend, or possible attempt to reconnect
               } catch (IOException e) {
//Explain there was a network failure problem when //attempting to send the SMS.
//Ask the user try resending later
```



Bluetooth (JSR 82) Tests

- Tests for Bluetooth (JSR 82)
- Network – 2 tests

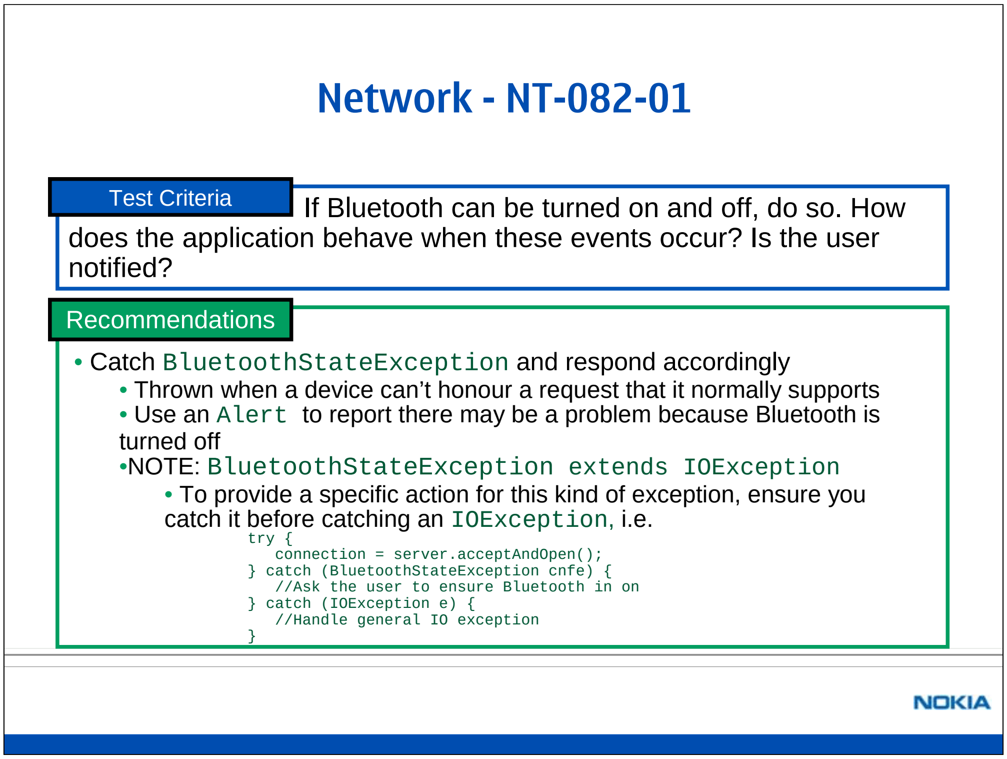
The tests in this section are for applications that use the Bluetooth API (JSR 82).



Network

Test Identifier	Summary	Reason for the Test
NT-082-01	If Bluetooth can be turned on and off, do so. How does the application behave when these events occur? Is the user notified?	Bluetooth may be disabled on the device to conserve battery power, so the application needs to be able to deal with this
NT-082-02	When an application closes it must close the user Bluetooth connection that was established.	To prevent the device from consuming power unnecessarily and being open to misuse via incoming Bluetooth connections

The emphasis of this category is to ensure that your application can deal with Bluetooth being be disabled on the device for security and battery power reasons, and to ensure Bluetooth connections are closed when the session is over.



Network - NT-082-01

Test Criteria

If Bluetooth can be turned on and off, do so. How does the application behave when these events occur? Is the user notified?

Recommendations

- Catch `BluetoothStateException` and respond accordingly
  - Thrown when a device can't honour a request that it normally supports
  - Use an `Alert` to report there may be a problem because Bluetooth is turned off
  - NOTE: `BluetoothStateException extends IOException`
    - To provide a specific action for this kind of exception, ensure you catch it before catching an `IOException`, i.e.

```
try {
   connection = server.acceptAndOpen();
} catch (BluetoothStateException cnfe) {
   //Ask the user to ensure Bluetooth in on
} catch (IOException e) {
   //Handle general IO exception
}
```

NOKIA

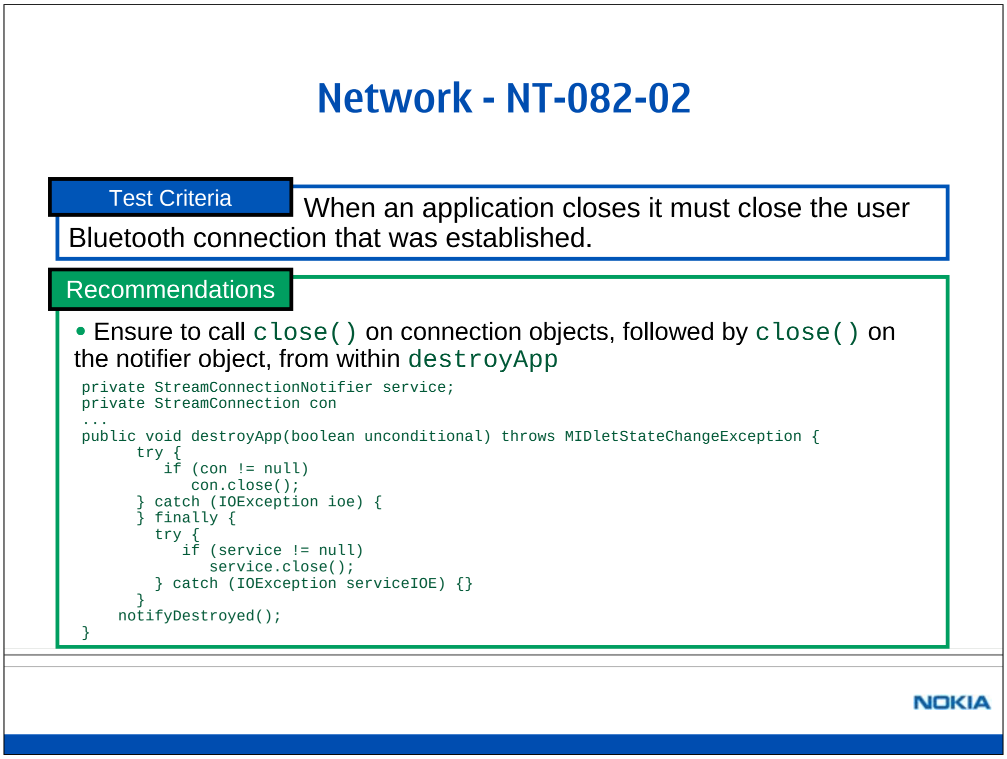
The user of the mobile device may have the option to turn off Bluetooth. This often done by user to conserve battery power or for security reasons (i.e. to prevent 'Bluejacking', where messages can be received from an unknown sender) This means that the local device may be non-connectable. When calling the API, a request is made to the Bluetooth Connection Centre (BCC) to make the local device connectable, but this request might not be satisfied if the device user has chosen to make the local device non-connectable.

A BluetoothStateException is thrown if the server attempts to make itself connectable, but this request conflicts with the device settings, established by the user. This exception is thrown when a device cannot honor a request that it normally supports because of the state of the Bluetooth settings on the device.

Although a device in non-connectable mode will not respond to connection attempts by remote devices, it could initiate connection attempts of its own. That is, a non-connectable device can be a client, but not a server. Therefore, when calling the acceptAndOpen method of a notifier, catching this exception is particular important, as it is likely this is due to Bluetooth being turned off on this device.

The code below shows an example of where the BluetoothStateException will be thrown if the Bluetooth is turned off a device. Notice in the code that the BluetoothStateException is caught before the IOException is caught. This is to provide specific information to the user, since BluetoothStateException actually extends IOException

```
private L2CAPConnection connection = null;
private  L2CAPConnectionNotifier server = null;
...
try {
       server = (L2CAPConnectionNotifier) Connector.open("btl2cap://"+url);
       connection = server.acceptAndOpen();
       ...
} catch (BluetoothStateException bse) {
       displayError("There was a problem establishing a Bluetooth connection", "Please ensure
Bluetooth is turned on", bse);
} catch (IOException ioe) {
       //General error
```



Network - NT-082-02

Test Criteria

When an application closes it must close the user Bluetooth connection that was established.

Recommendations

- Ensure to call `close()` on connection objects, followed by `close()` on the notifier object, from within `destroyApp`

```
private StreamConnectionNotifier service;
private StreamConnection con
...
public void destroyApp(boolean unconditional) throws MIDletStateChangeException {
      try {
         if (con != null)
            con.close();
      } catch (IOException ioe) {
      } finally {
        try {
           if (service != null)
              service.close();
        } catch (IOException serviceIOE) {}
      }
    notifyDestroyed();
}
```

To ensure a Bluetooth connection is closed when the application closes, you must call the close() method on any connection objects you have been using, followed by close() on any associated notifier objects.

For example, the method close() in the StreamConnection object, that represents an SPP server-side connection, is used to close the connection. When the close() method is called on the StreamConnectionNotifier the service record associated with that notifier becomes inaccessible to clients through service discovery. If the StreamConnection to this service remains open when the StreamConnectionNotifier is closed, the RFCOMM server channel that is assigned to this service will not be released.

Therefore, to ensure that the RFCOMM server channel is released, all StreamConnections to this service must be closed, followed by the associated notifier.