Java™ Platform, Micro Edition

Part 6 – Record Stores, Distribution and Localization

Disclaimer

- **These slides are provided free of charge at http://www.symbianresources.com and are used during Java ME courses at the University of Applied Sciences in Hagenberg, Austria at the Mobile Computing department (http://www.fh-ooe.at/mc)**
- **Respecting the copyright laws, you are allowed to use them:**
  - **for your own, personal, non-commercial use**
  - **in the academic environment**
- **In all other cases (e.g. for commercial training), please contact andreas.jakl@fh-hagenberg.at**
- The correctness of the contents of these materials cannot be guaranteed. Andreas Jakl is not liable for incorrect information or damage that may arise from using the materials.
- This document contains copyright materials which are proprietary to Sun or various mobile device manufacturers, including Nokia, SonyEricsson and Motorola. Sun, Sun Microsystems, the Sun Logo and the Java™ Platform, Micro Edition are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Contents

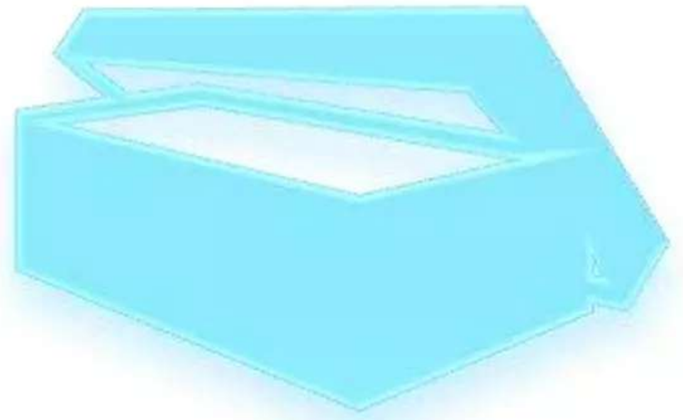
- Record Stores
  - Input/OutputStreams
- Distribution
  - Deployment
  - Pre-Processing
  - Obfuscation
  - OTA-Deployment
- Localization

Save your data

Record Stores

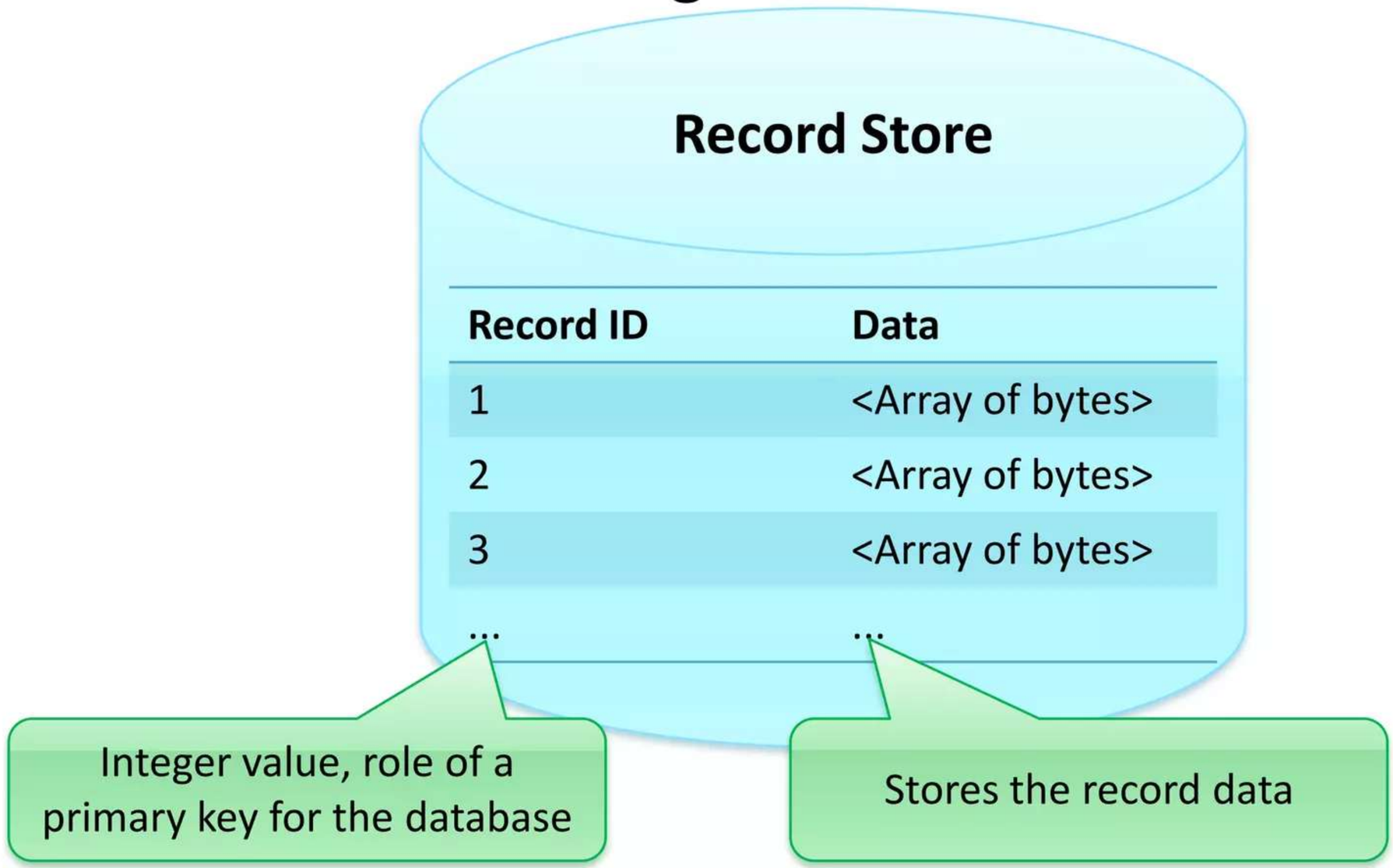
Persistent Data

- **Most applications have to store something, e.g.:**
  - Settings
  - Current status / progress / high scores
  - User documents and results
- **MIDP requires:**
  - **Non-volatile** memory
  - At least **8kB**
- Full file system access is optional (JSR 75)



Record Store & Records

- **Records:** can be thought of rows in a table

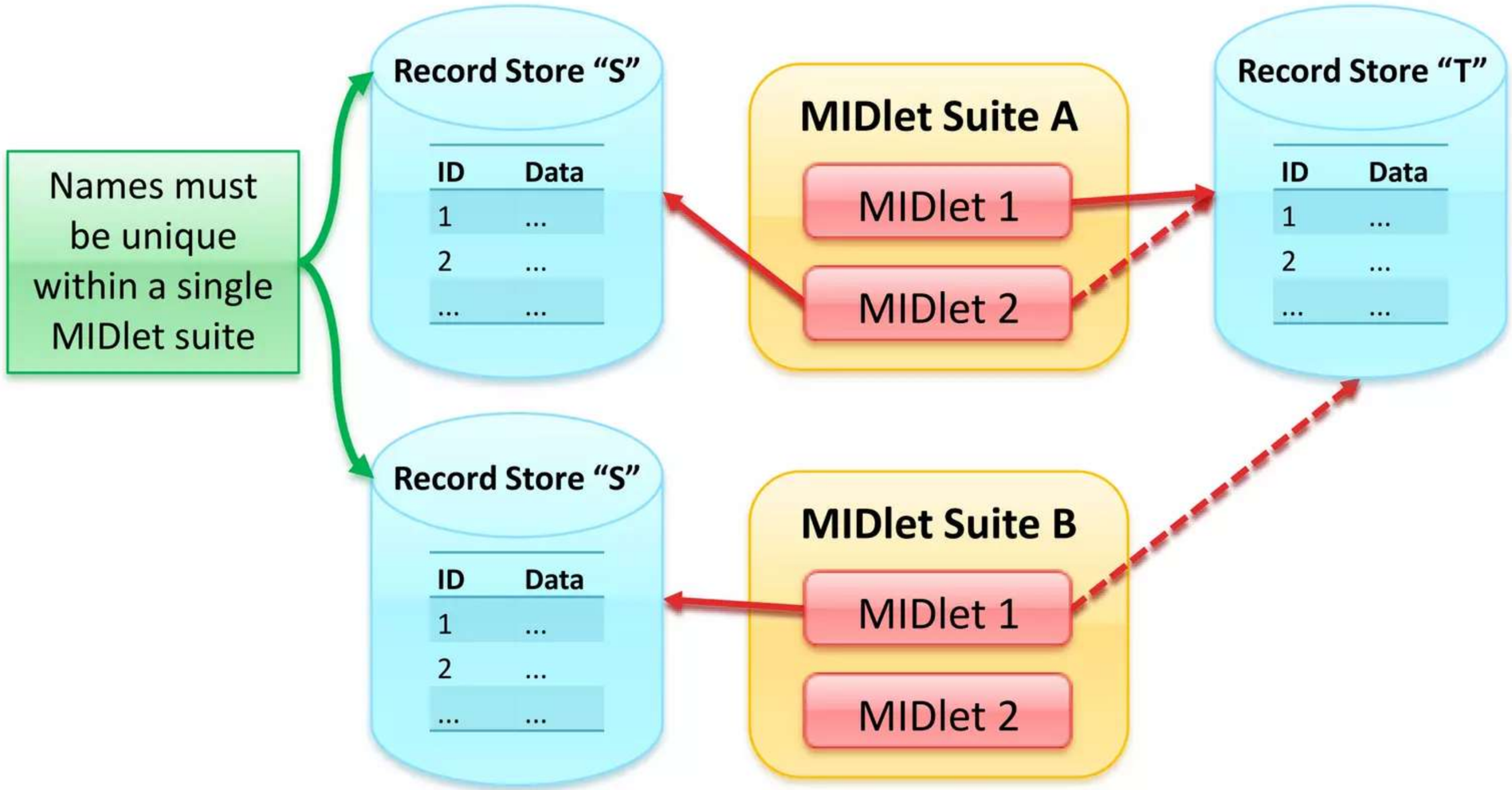


Record Management System

- **MIDP Record Management System (RMS)** works like DBMS
- Device-independent reading and writing,
  hides real database-file and location
- **Database = "Record Store"**
  - **Consists of a number of records** with automatically assigned, **unique IDs**: 1, 2, ...
- **Record = Array of bytes**
  - To **change content**: read byte array → modify it → replace original record

MIDlets and Record Stores

- MIDlets can access all record stores in their suite
- With a fully qualified name also from other suites



Record Store Features

- **Record Store saves:**
  - **Time + Date** of last modification
  - **Version number** (integer), incremented for each operation that modified the contents
- **Individual operations** (write, ...) are atomic, synchronous and serialized
  - When MIDlet uses multiple threads to access a single record store, it is responsible for synchronization

RecordStore API

- **Opening / Creating a Record Store**
  - RecordStore rs = RecordStore.openRecordStore(
    String recordStoreName, boolean createIfNecessary);
    - recordStoreName: MIDlet suite-unique name, up to 32 (Unicode) chars
    - createIfNecessary: if it does not exist, create it or throw an exception
- **Add Records**
  - int newId = rs.addRecord(byte[] data, int offset, int numBytes);
    - newId: ID of the record that was created (1, 2, ...)
    - data: byte-array to add as a new record
    - offset: index into data buffer (first byte to write), usually 0
    - numBytes: number of bytes to use (length of data to write)

RecordStore API

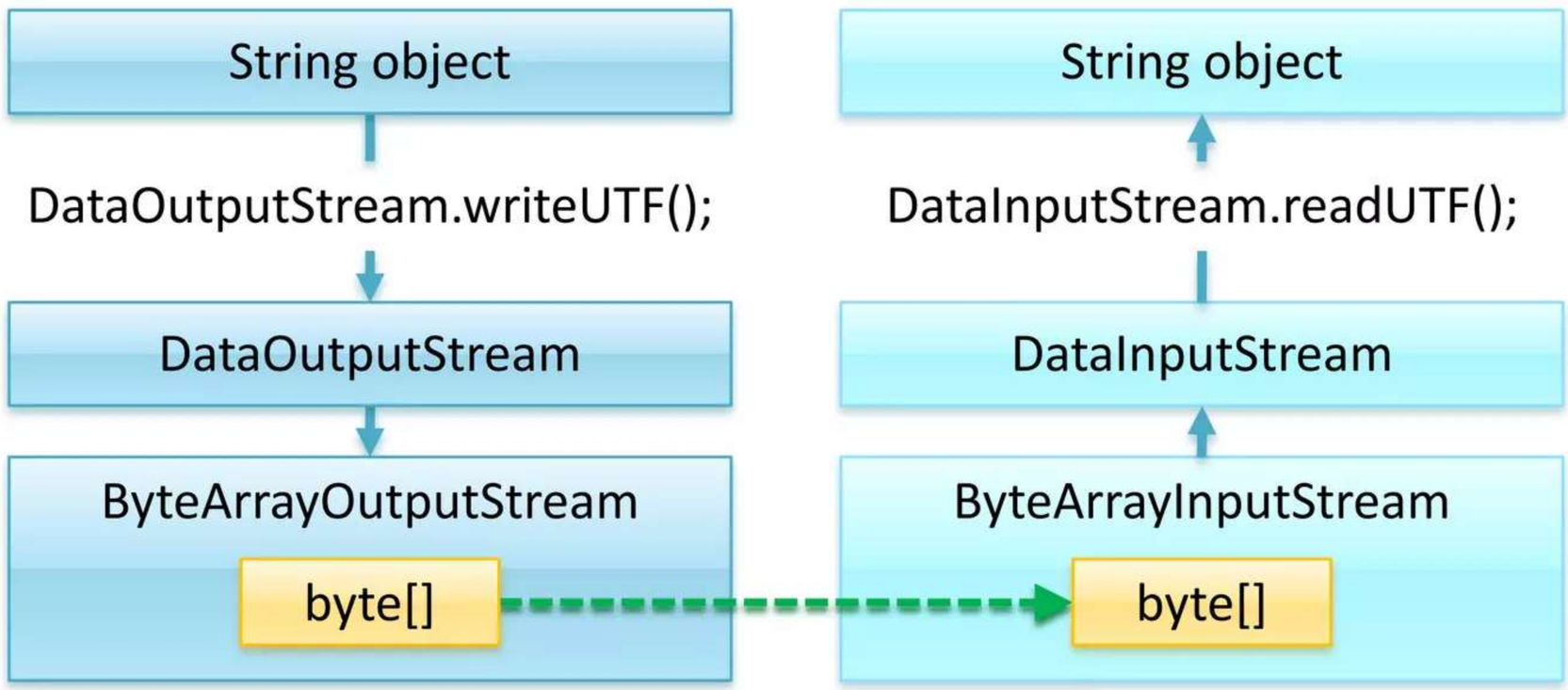
- **Reading a record**
  - byte[] record = rs.getRecord(int recordId);
    - record: copy of the data stored in the record
    - recordId: ID of the record to retrieve
- **Close the record store**
  - rs.closeRecordStore();
- **Delete a record store**
  - RecordStore.deleteRecordStore(String recordStoreName);

Working with

Byte-Arrays

Streams

- **Java offers comfortable streams to:**
  - **Create byte arrays** on the fly (ByteArrayOutputStream)
  - Write **standard data types** to the stream (DataOutputStream)



Output Stream Example

```
// Open (+ create) the record store with the name "MyData"
RecordStore rs = RecordStore.openRecordStore("MyData", true);
// First create the byte array output stream which can generate the array
ByteArrayOutputStream bos = new ByteArrayOutputStream();
// On top of that, create the data output stream,
// which can serialize several standard Java data types
DataOutputStream dos = new DataOutputStream(bos);
// Write data to the stream
dos.writeInt(667487);
dos.writeBoolean(true);
dos.writeUTF("Test");
// Flush data to make sure everything is commited down streams
dos.flush();
// Grab byte array from the stream
byte[] recordOut = bos.toByteArray();
// Add a new record to the record store, write the whole array from 0 to its length
int newRecordId = rs.addRecord(recordOut, 0, recordOut.length);
// Finished working on the record store – close it
dos.close();                    // Closes underlying output stream as well
rs.closeRecordStore();
```

(example omits catching Exceptions for clarity)

Byte Array

- Contents of the byte array:

recordOut	byte[]	#229415(length=11)
[0]	byte	0
[1]	byte	10
[2]	byte	47
[3]	byte	95
[4]	byte	1
[5]	byte	0
[6]	byte	4
[7]	byte	84
[8]	byte	101
[9]	byte	115
[10]	byte	116

```
dos.writeInt(667487);
dos.writeBoolean(true);
dos.writeUTF("Test");
```

Int (4 bytes)

Boolean (1 byte)

String

- Number of bytes used (2 bytes)
- UTF-8 encoded String (4 bytes)

DataOutputStream-Methods only save data, no type information

→ you have to remember what you wrote and read it in exactly the same order

Reading

- Rather similar to writing:

```
// Open the record store with the name "MyData"
RecordStore rs = RecordStore.openRecordStore("MyData", false);
// Get record contents
byte[] record = rs.getRecord(newRecordId);
// First create the byte array input stream which accesses the byte array
ByteArrayInputStream bis = new ByteArrayInputStream(record);
// On top of that, create the data input stream,
// which can interpret the contents byte array as Java data types
DataInputStream dis = new DataInputStream(bis);
// Read data from the stream
int version = dis.readInt();
boolean firstStart = dis.readBoolean();
String userName = dis.readUTF();
// Finished working on the record store – close it
dis.close();                    // Closes underlying input stream as well
rs.closeRecordStore();
```

(example omits catching Exceptions for clarity)

Enough reading and writing ...

... back to the RecordStore

RecordStore API

- **Modify (= overwrite, replace) existing record**
  - rs.setRecord(int recordId, byte[] newData, int offset, int numBytes);
    - recordId: ID of the record to replace
    - data: byte-array to add as a the new record
    - offset: index into data buffer (first byte to write), usually 0
    - numBytes: number of bytes to use (length of data to write)

Strategy – Settings

- **Saving application settings**
  - Write all settings to a **byte array** into **one record** of a **single record store**
  - If possible: **Write directly after settings change**, not when exiting the application
    - You never know when and how the program is closed (battery removed?)
- **Read on application start-up**
  - Use default settings for “file not found”-Exception

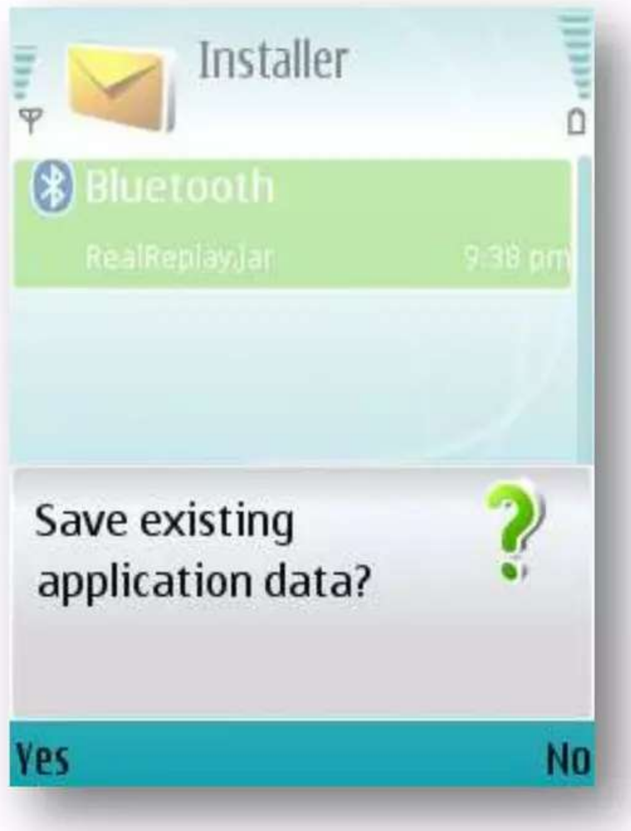
Settings – Existing Record

- When saving settings, **check:**
  - **New** record store was created → **add record**
  - Record store **already exists** → **replace record**

```
if (rs.getNumRecords () == 0) {                  // Check if record store already contains our settings record
   rs.addRecord (recordOut, 0, recordOut.length);   // Add new record to the store, will get position 1
} else {
   rs.setRecord (1, recordOut, 0, recordOut.length);   // Replace previous settings-record
}
```

Settings – Version

- **Save version of data structure** to prevent problems after updating application
  - (Version of Record Store only saves # of modifications!)
- **Store version** at the beginning of the stream
  - dos.writeShort (**1**); // Version
- **Check version** when reading stream
  - int version = dis.readShort();
    if (version == **1**) { /* Continue parsing */ }



Strategy – Database

- **Application with database** (e.g. time management)
  - Multiple records per record store.
    Each record = one entry, which consists of the data, written as a byte array as usual
- **Serializing Objects**
  - Used to **save object's state** to a sequence of bytes
  - Allows **rebuilding an identical object** later on
  - **Not supported** in current CLDC
  - Write it **yourself:** Use a stream to put all relevant data into a stream; read and create obj. in a static method

Example – Database I

ContactData class, contains information about a person

```
public class ContactData {
   private String name;    // Saves name of the contact
   private int age;        // Saves age

   public ContactData (String name, int age) {
         this.name = name;
         this.age = age;
   }

   /** Write all relevant data to the stream, which will allow rebuilding the object when reading the stream. */
   public void externalize(DataOutputStream dos) throws IOException {
         dos.writeUTF (name);
         dos.writeInt (age);
   }

   /** Read data from the stream and use it to create and return a new instance of this object. */
   public static ContactData internalize(DataInputStream dis) throws IOException {
         String tmpName = dis.readUTF ();
         int tmpAge = dis.readInt ();
         return new ContactData(tmpName, tmpAge);
   }

   /** Return relevant information in string form. */
   public String toString() {
         return ("Contact data: Name = " + name + ", Age = " + age);
   }
}
```

Example – Database II

Main application – save contacts to the database

```
// Create new vector that will contain all contact data
Vector names = new Vector();
names.addElement (new ContactData("Valerie Dimeling", 20));
names.addElement (new ContactData("Ginger Hay", 54));
try
{
   RecordStore.deleteRecordStore ("myRs");                    // Make sure no record store already exists
   RecordStore rs = RecordStore.openRecordStore("myRs", true);  // Create a new record store
   ByteArrayOutputStream bos = new ByteArrayOutputStream();   // Create streams for writing data
   DataOutputStream dos = new DataOutputStream(bos);
   // Go through the vector and create a record for each element
   for (Enumeration e = names.elements (); e.hasMoreElements (); )
   {
        ContactData cd = (ContactData) e.nextElement ();    // Get next element of the vector
        cd.externalize (dos);                               // Let the instance externalize itself into our stream
        dos.flush();                                        // Make sure everything is written
        byte[] record = bos.toByteArray ();                 // Get byte array from the record
        rs.addRecord (record, 0, record.length);            // Add the record to the store
        bos.reset ();                                       // Clear the output stream so that we can start from scratch
   }
   dos.close ();

[…]
```

Example – Database III

Main application – restore database from the record store

```
// Read back data using a record enumerator to go through all elements
// Record enumerator would be more powerful, we do not use the advanced features here
RecordEnumeration renum = rs.enumerateRecords (null, null, false);
while (renum.hasNextElement ())
{
    byte[] record = renum.nextRecord ();                          // Get data of the next record
    ByteArrayInputStream bis = new ByteArrayInputStream(record);  // Input streams for parsing data
    DataInputStream dis = new DataInputStream(bis);
    ContactData cd = ContactData.internalize (dis);   // Internalize and create new instance (static method)
    System.out.println(cd);                           // Print information to console
    dis.close();
  }
  rs.closeRecordStore ();                 // Close the record store
} catch (RecordStoreException ex) {
  ex.printStackTrace();
} catch (IOException ex) {
  ex.printStackTrace();
}
```

Advanced Access

- **Sorting**
  - RecordComparator
- **Searching**
  - RecordFilter
- **Notification of changes**
  - RecordListener

Let others experience your fabulous applications!

Distributing MIDlets

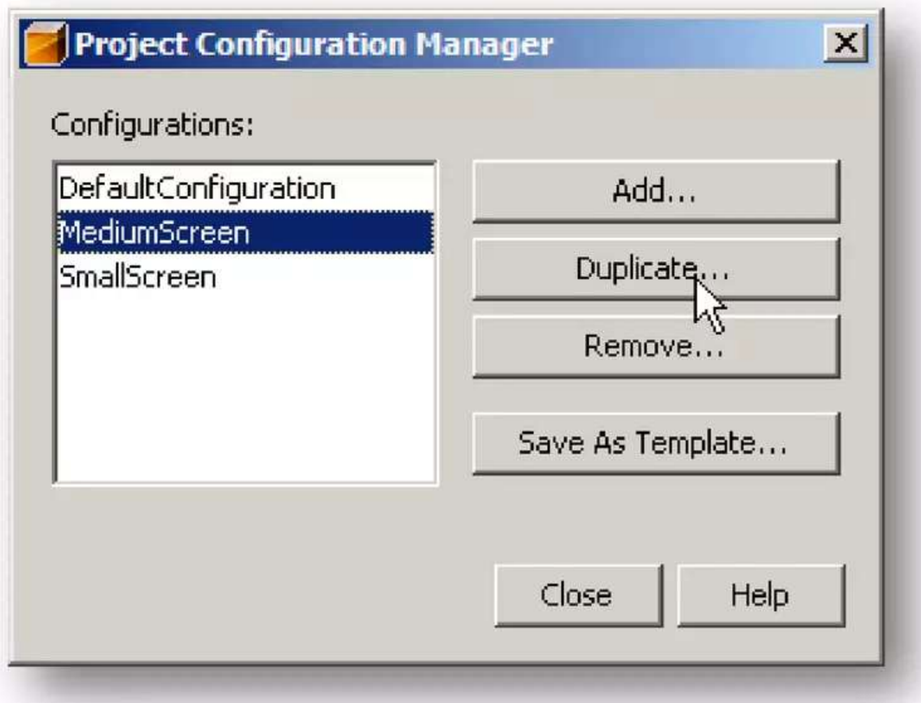
Device Fragmentation

- **Mobile devices vary in:**
  - Screen size
  - Colour depth
  - Speed
  - Optional API support (Bluetooth, ...)
  - Bugs (+ their workarounds)
- **Solution (in NetBeans):**
  - Project configurations + pre-processing (like in C++)

Project Configurations

- **Project Configurations used for**
  - **Setting variables** that are used for pre-processing
  - Including required **files to .jar-archive**
  - Modify **build settings** (Obfuscation, ...)

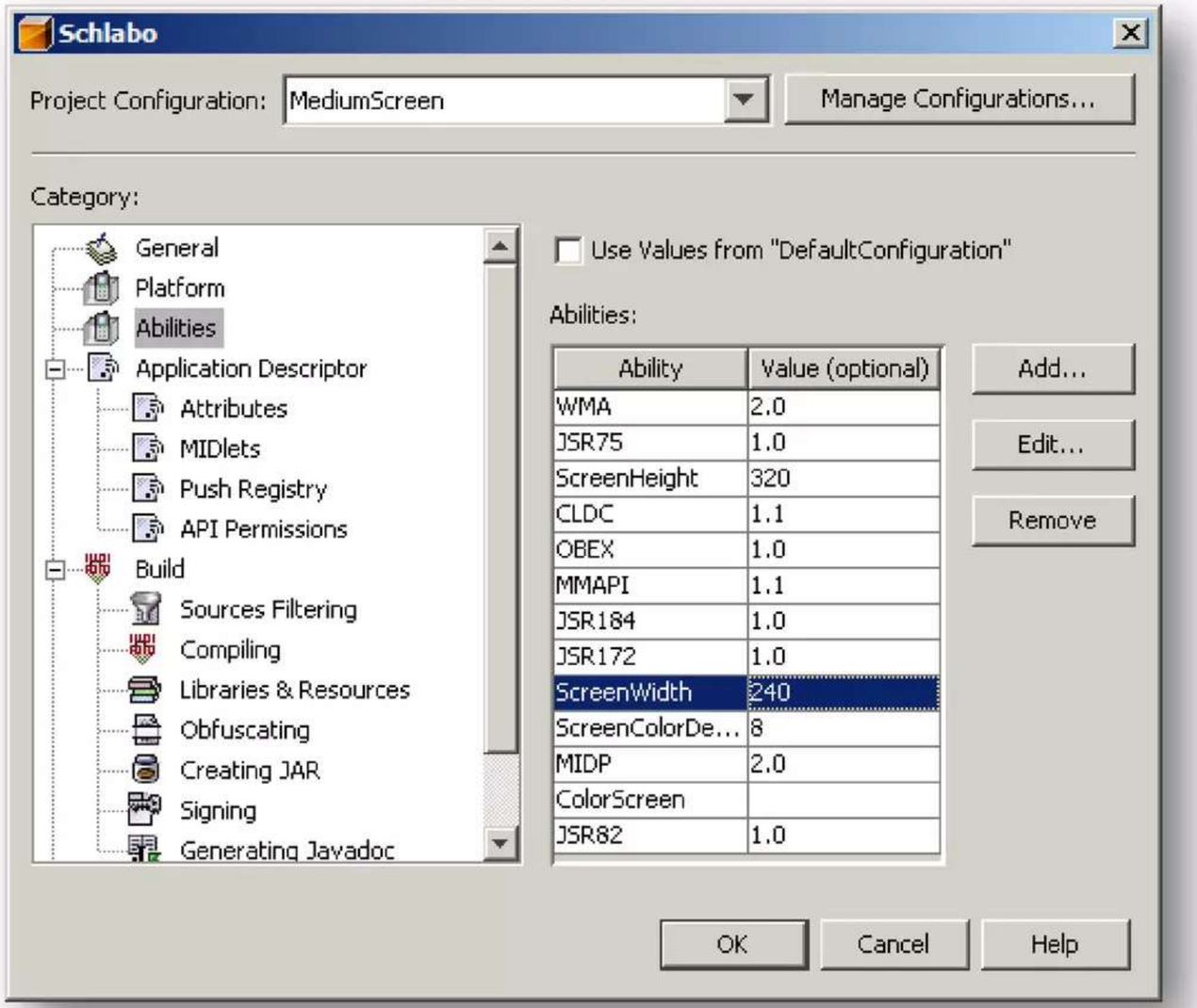
1. **Create new configuration**
   - Project Properties → Manage Configurations ... → Duplicate one of the configurations



Adapt Abilities

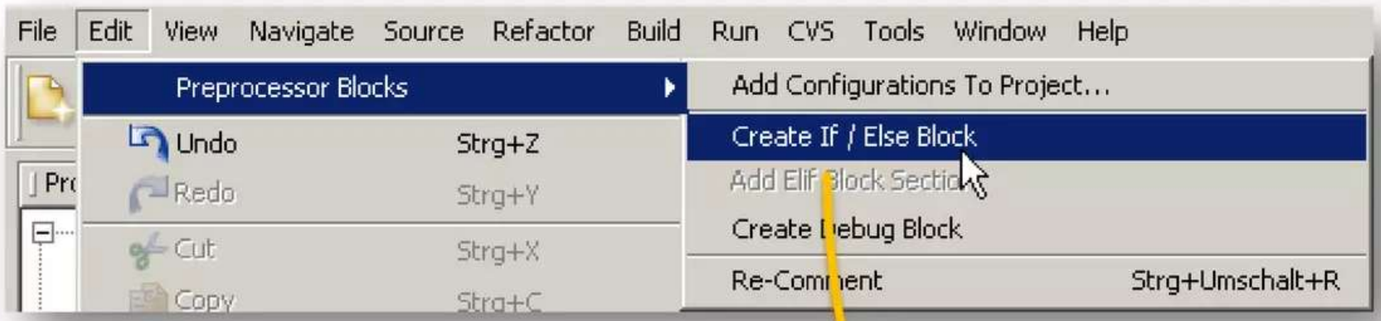
2. **Define variables for pre-processing**

   – e.g. ScreenWidth, support for APIs, ...



Fragment your Code

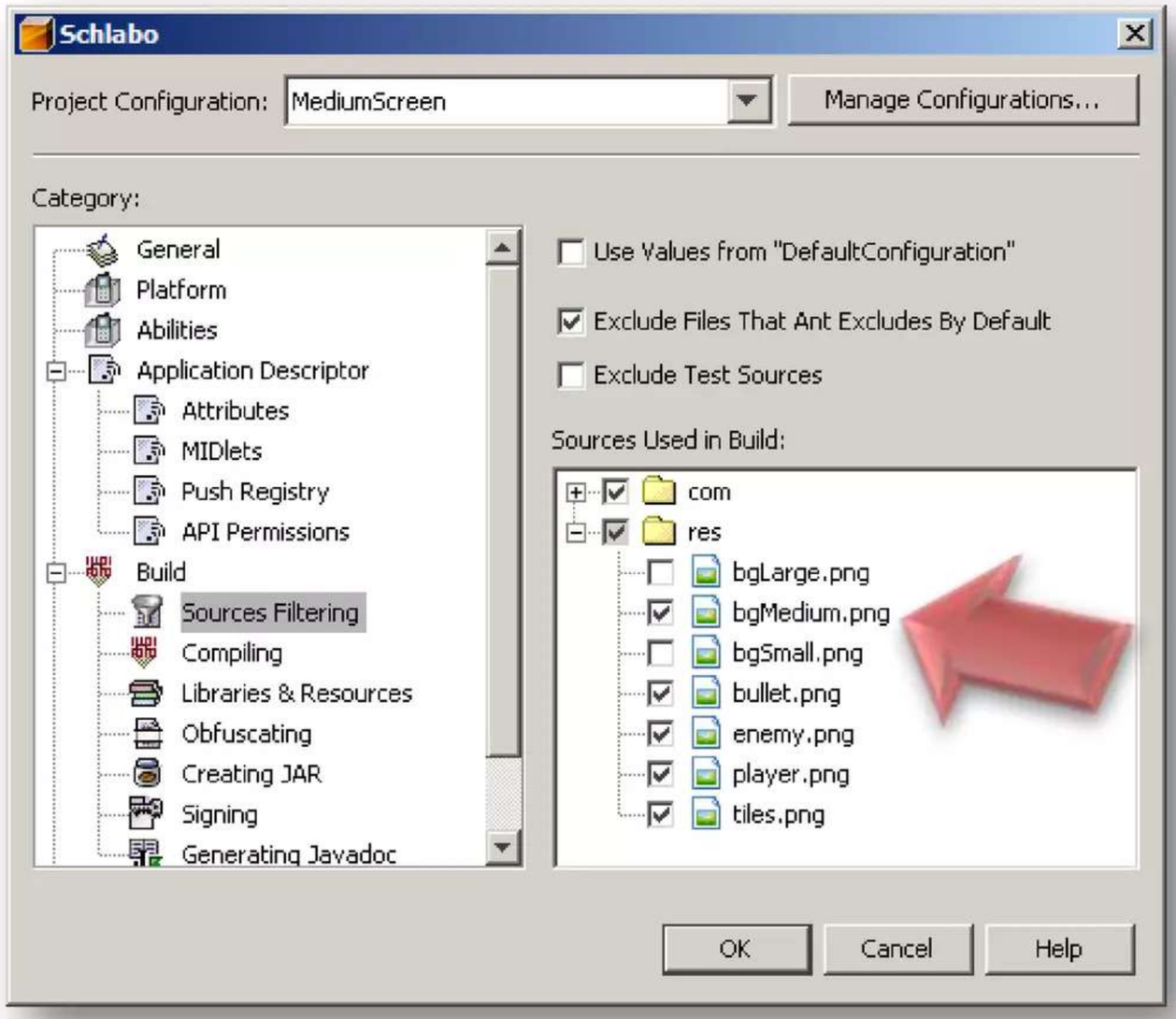
- Use **pre-processor directives** to adapt code depending on abilities of current project configuration




```
            String loadFile;
//#if ScreenHeight<260
//#          loadFile = "bgSmall.png";
//#elif ScreenHeight<450
            loadFile = "bgMedium.png";
//#else
//#          loadFile = "bgLarge.png";
//#endif
```

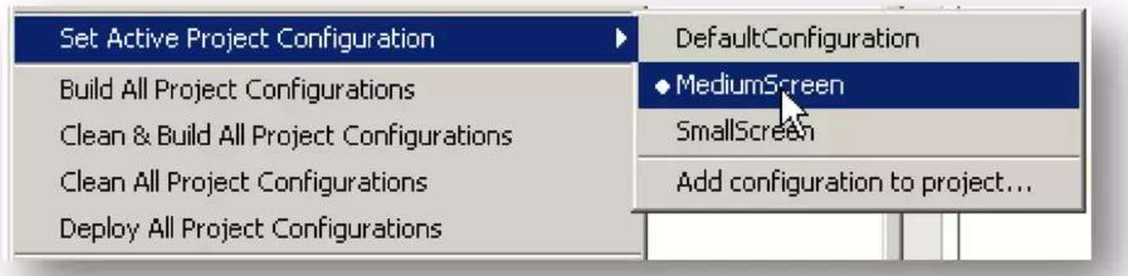
Adapt Included Resources

- Choose which files to include for each project configuration:



Set Active Configuration

- Right click on the project → “Set Active Project Configuration”: Defines which pre-processor block is active




```
//#if ScreenHeight<260
            loadFile = "bgSmall.png";
//#elif ScreenHeight<450
//#          loadFile = "bgMedium.png";
//#else
//#          loadFile = "bgLarge.png";
//#endif
```

“SmallScreen”
ScreenHeight = 128

```
//#if ScreenHeight<260
//#          loadFile = "bgSmall.png";
//#elif ScreenHeight<450
            loadFile = "bgMedium.png";
//#else
//#          loadFile = "bgLarge.png";
//#endif
```

“MediumScreen”
ScreenHeight = 320

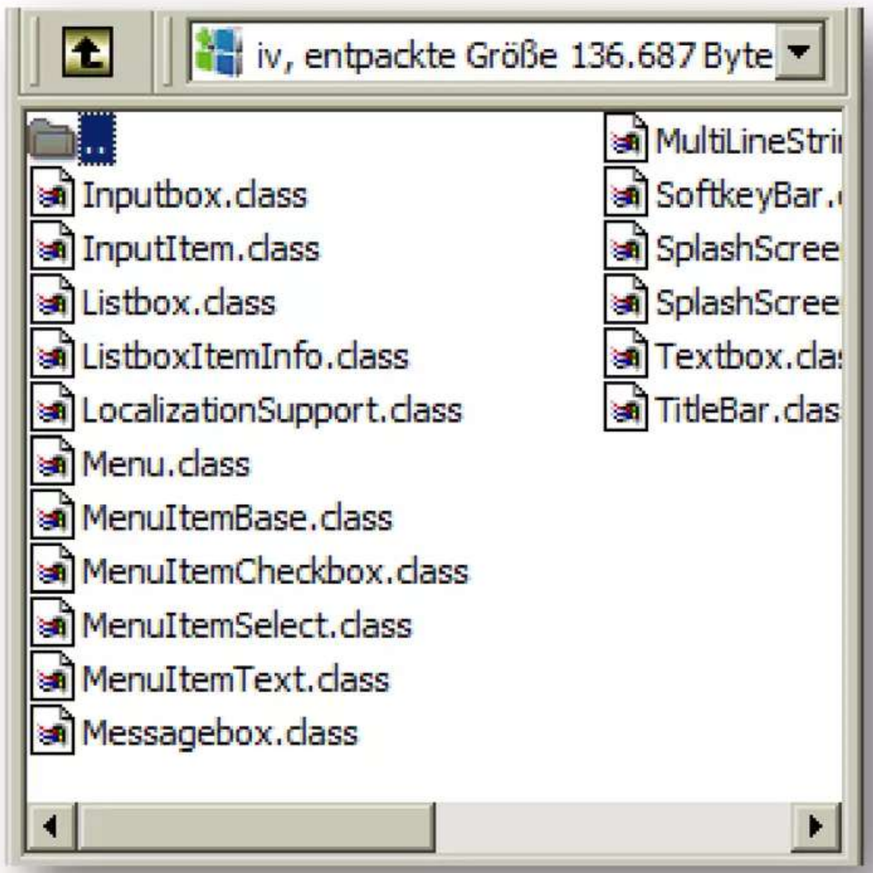
Optimization – Obfuscation

- **Original intention:**
  - Make reverse engineering more difficult
  - Code more difficult to read after de-compilation
- Renames classes to “a.class, b.class, …”
- Removes non-used methods, variables, classes
- → **Significant size reduction**
  - Over-the-Air = expensive!
  - MIDlet size restrictions in many phones
  - Improves speed (less code to load / parse)

Obfuscation

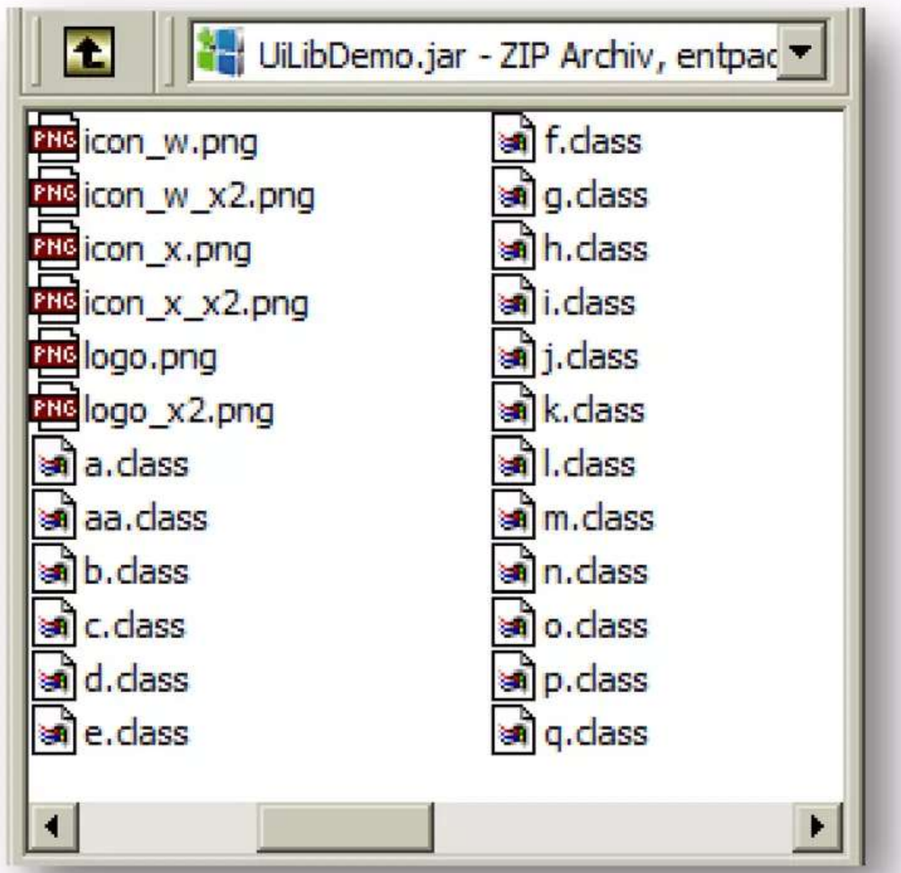
Original archive

79.2 kB



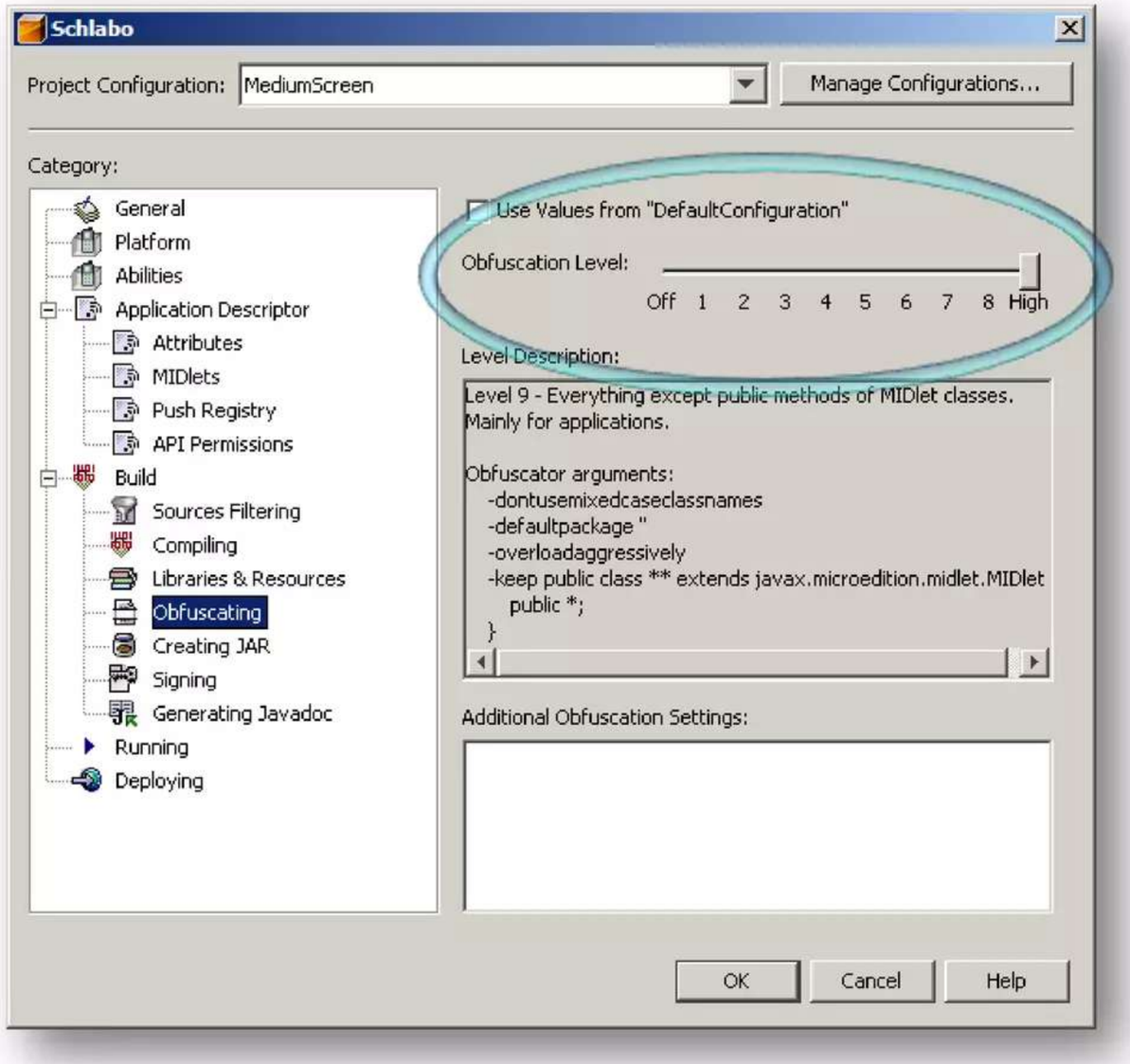
Obfuscated

42.9 kB = 45% smaller!

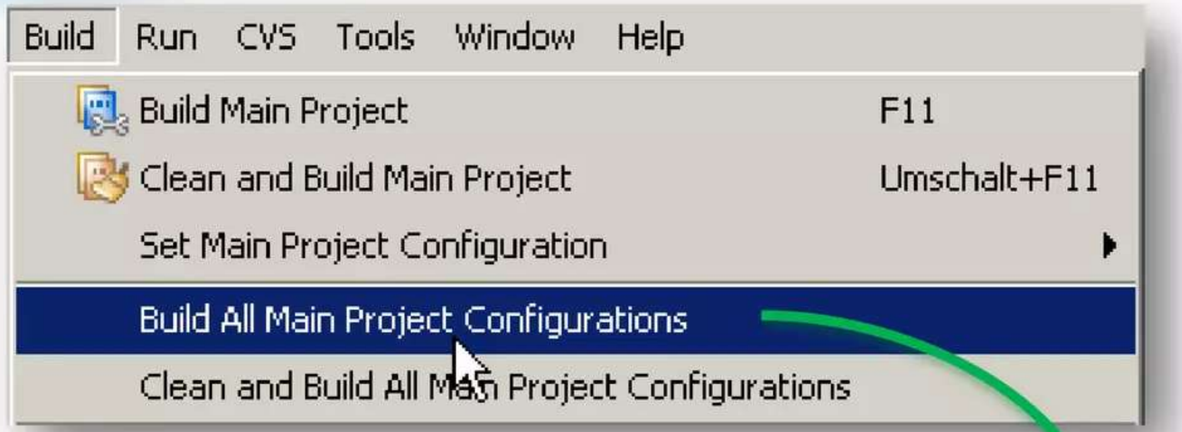


Define Obfuscation Level

- **For release builds:** Activate obfuscation (max. level)
- Don’t use it for debugging

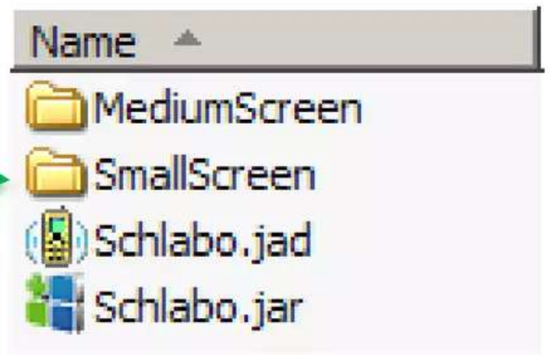


Build All Configurations



... creates own directory and .jar/.jad for each configuration

Default configuration located in base directory



Transmitting MIDlets to phones

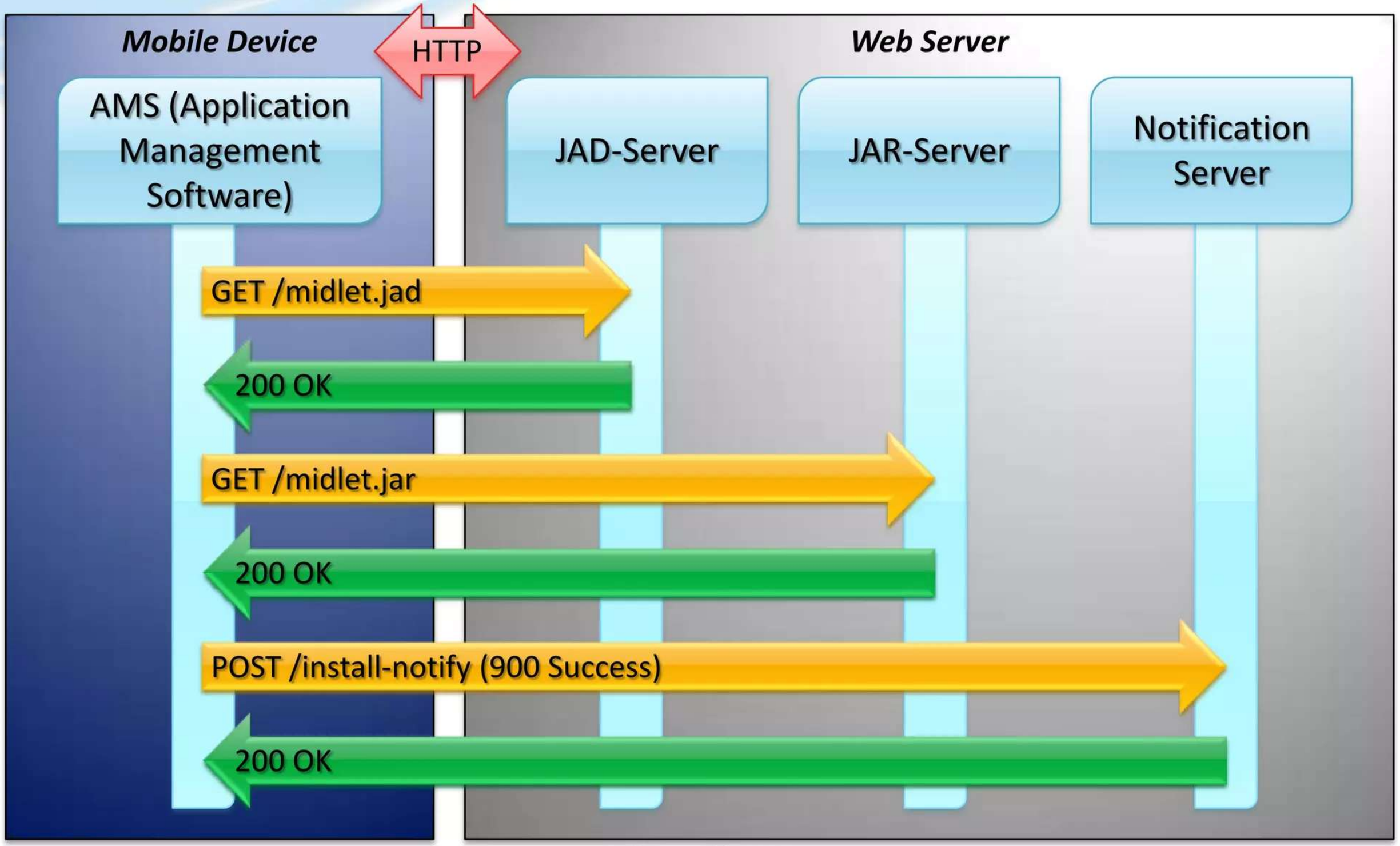
Over-The-Air Deployment

Why OTA for Deployment?

- Some phones (Samsung, Sagem, BREW,...) do not support installing MIDlets through the PC or Bluetooth
- **Only alternative:**
  - Download directly through mobile phone (UTMS)

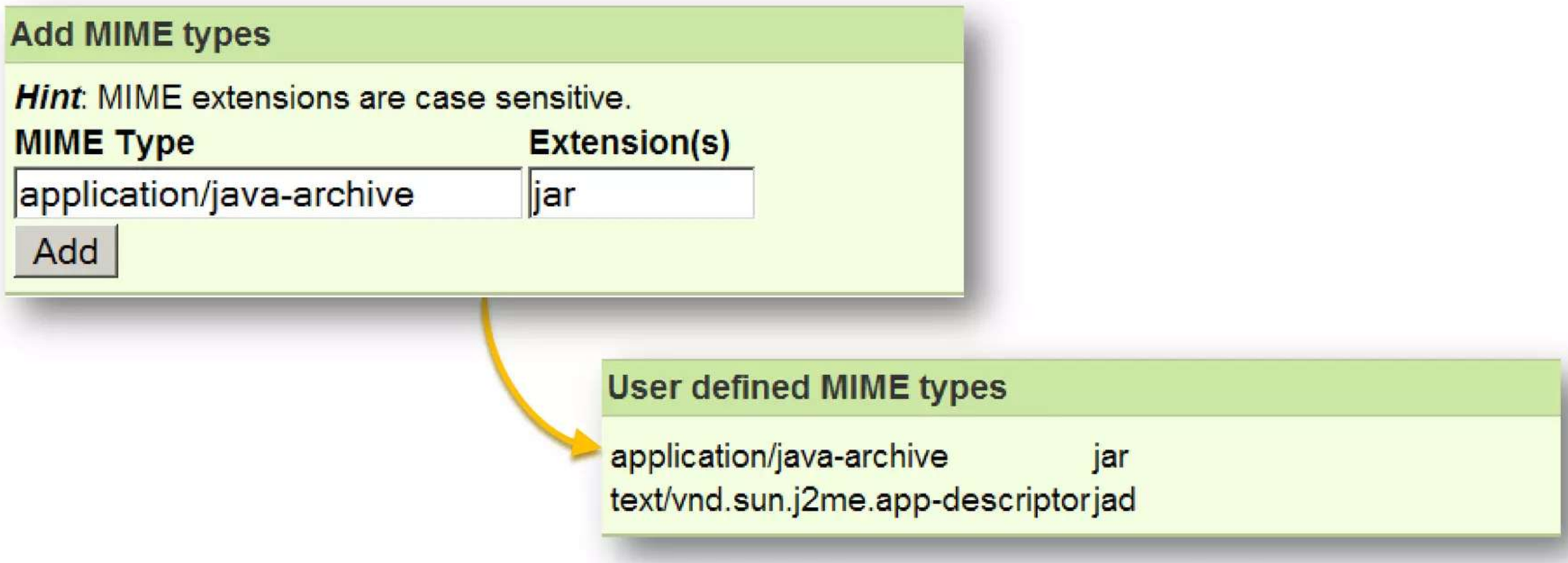
    → Over-the-Air (OTA) delivery

Over-the-Air


Mobile Device
HTTP
Web Server
AMS (Application Management Software)
JAD-Server
JAR-Server
Notification Server
GET /midlet.jad
200 OK
GET /midlet.jar
200 OK
POST /install-notify (900 Success)
200 OK

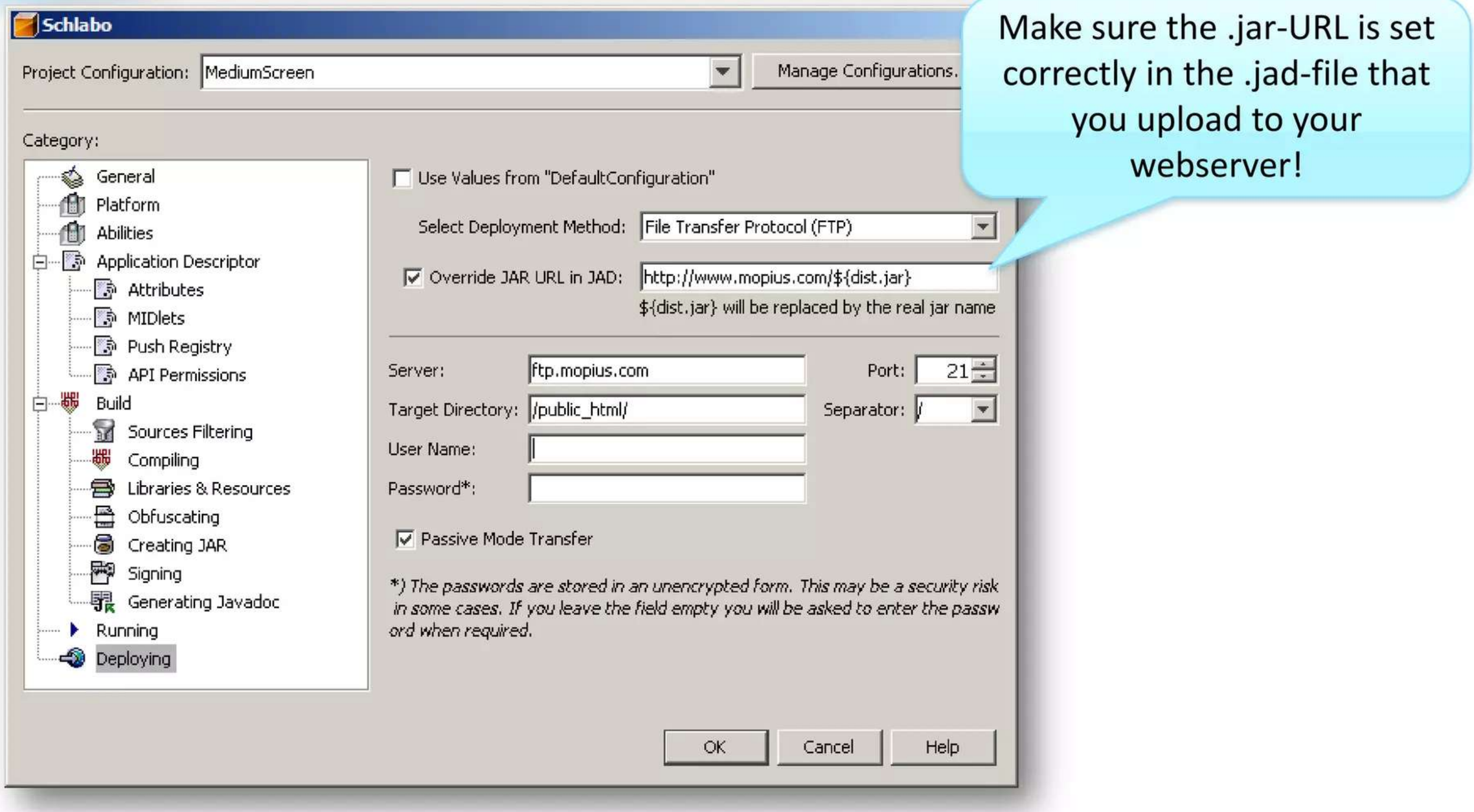
Web Server – MIME Types

- **Add MIME-Types for .jad and .jar to your web server**
  - Required for the phone to correctly handle the files
  - Then, simply provide a link to the .jad-file on your website
  - The .jad-file contains the link to the .jar archive



Example screenshot: SiteAdmin-Tool used at: http://www.site5.com/

Automated Deployment



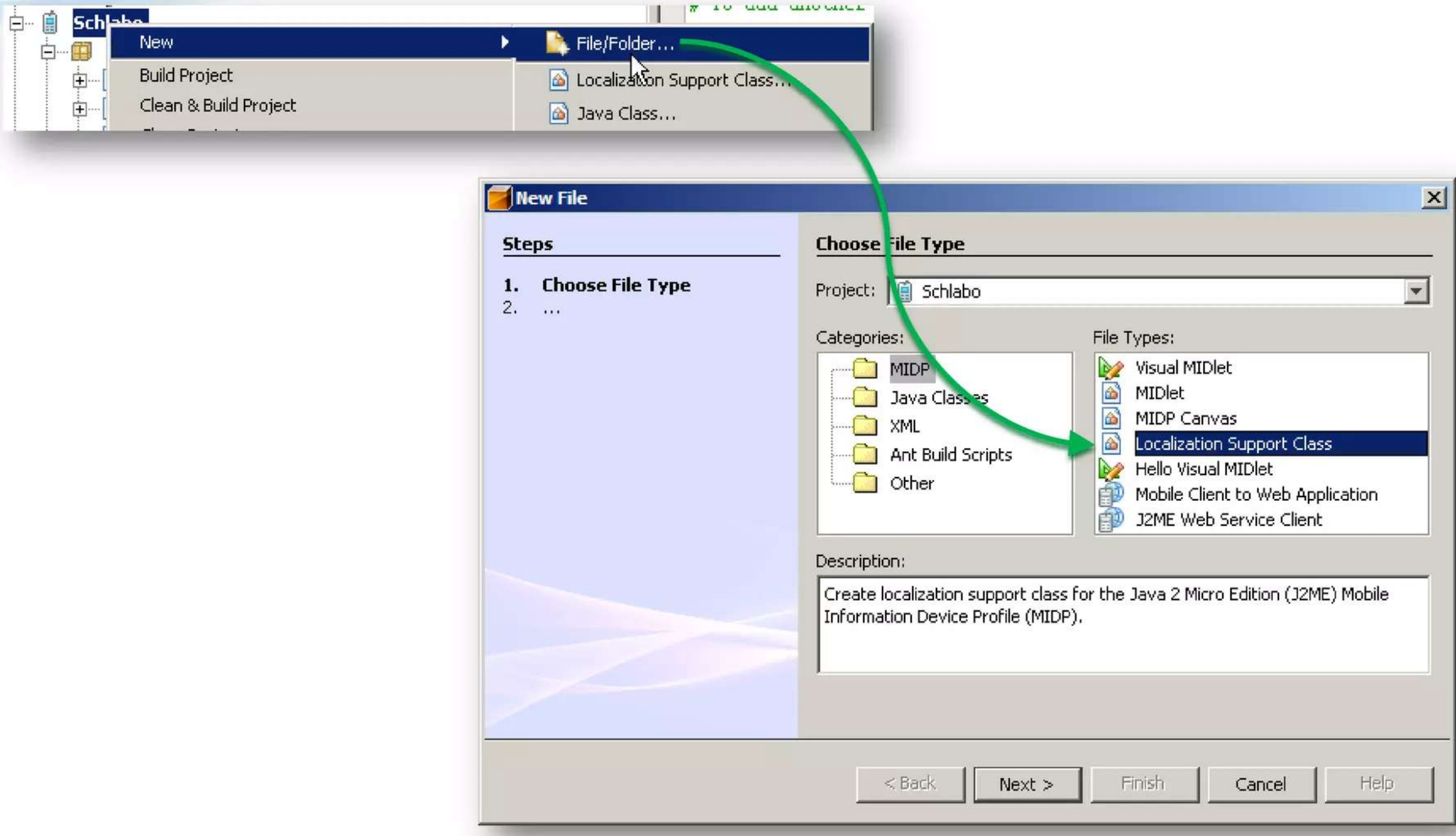
Prepare for the international market

Localization

Localization

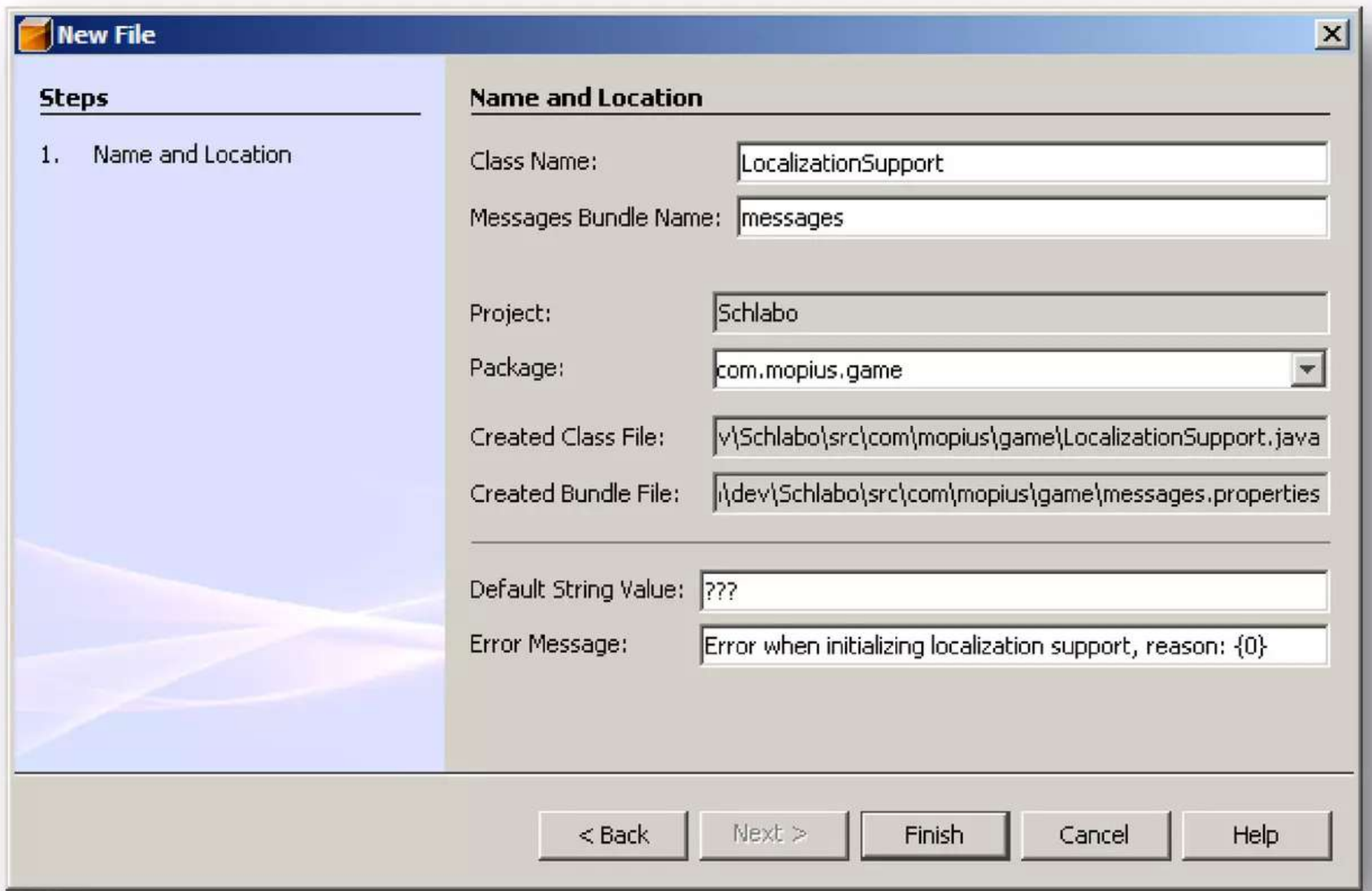
- Mobile phones available on a **global market**
- **Goal:** Make source code **independent of language**, load text for user language from a file
- **Query current language:**
  - String locale = System.getProperty ("microedition.locale");
  - **Examples:**
    en-US, en-GB, de-DE
- **NetBeans:**
  - Provides ready-made LocalizationSupport-class & tools for managing message bundles

Add Support Class



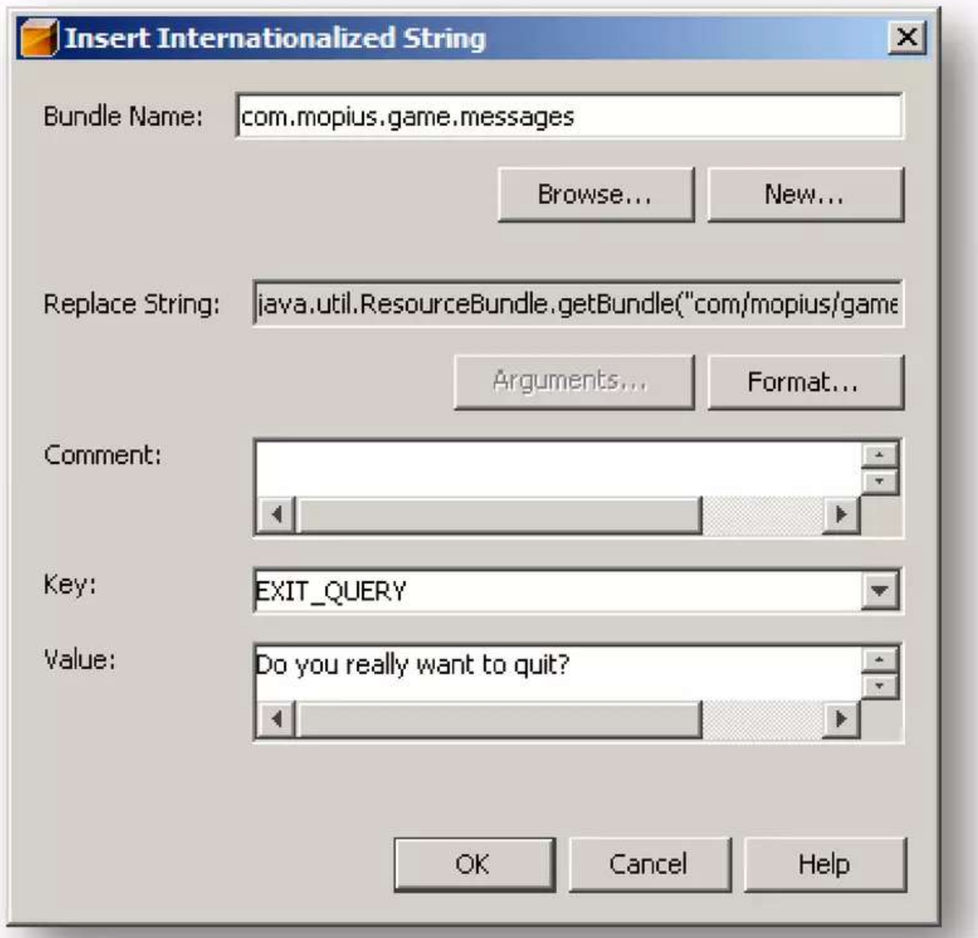
Define Settings

- You can usually accept the default values:



Add Localized String

- Tools → Internationalization → Insert Internationalized String...



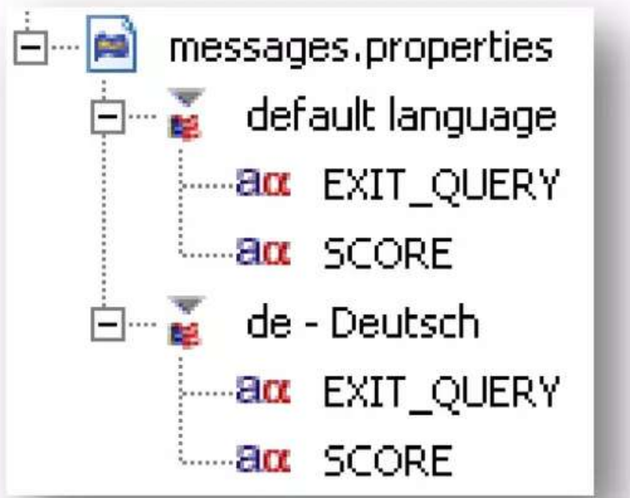
Get the localized string from your source code using the static function:

```
String exitQuery = LocalizationSupport.getMessage("EXIT_QUERY");
```

Initialization will be done automatically when you retrieve the first message.

Message Bundle

- Add additional locales for new languages
- Define same keys in all files




```
SCORE=Score\:
EXIT_QUERY=Do you really want to quit?
```

```
SCORE=Punkte\:
EXIT_QUERY=Wollen Sie das Programm wirklich beenden?
```

That’s it!

Thanks for your attention