



This article describes how to create text resources for a MIDP application, and how to load them at run-time. This technique is unicode safe, and so is suitable for all languages. The run-time code is small, fast, and uses relatively little memory.

## Contents

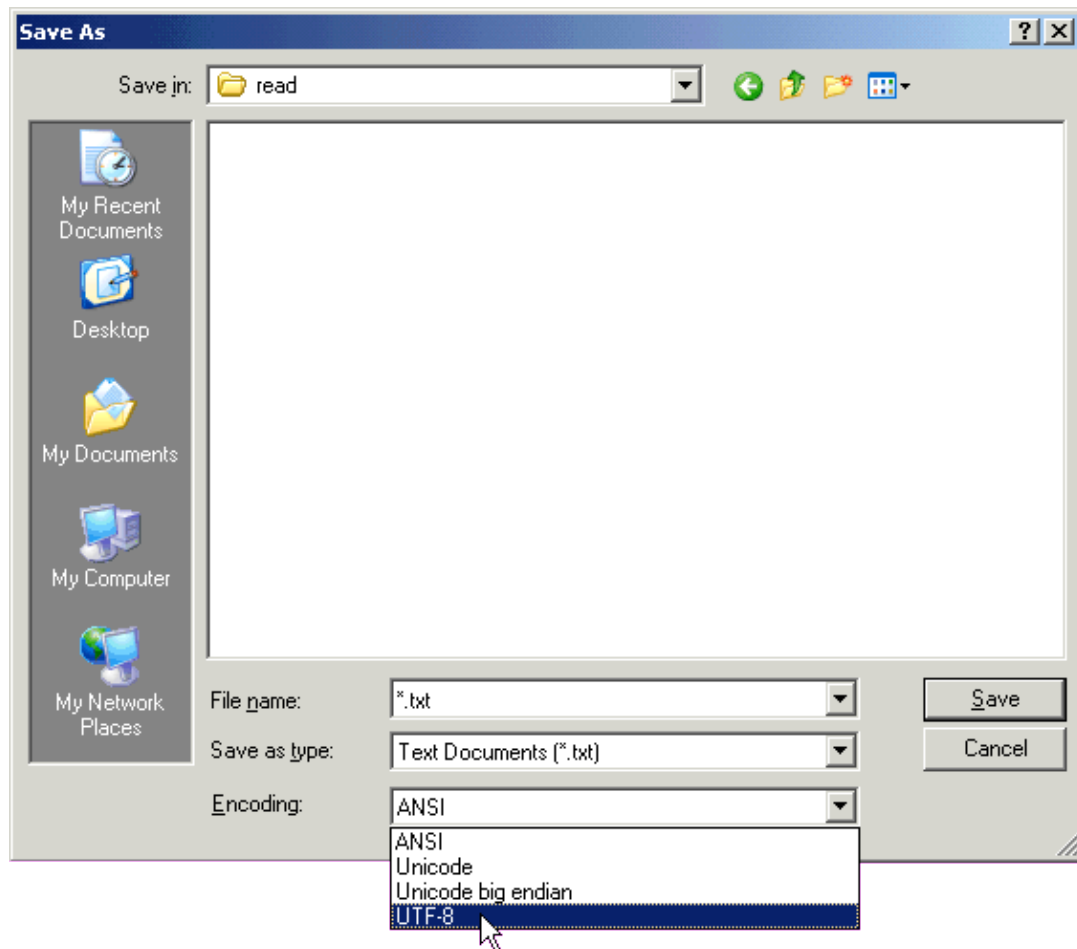
- [1 Creating the Text Source](#)
- [2 Converting the Text to a Resource File](#)
- [3 Using the Text at Runtime](#)
- [4 Comparison to Other Techniques](#)

## Creating the Text Source

The process starts with creating a text file. When the file is loaded, each line becomes a separate String object, so you can create a file like:

```
My Game  
OK  
Cancel  
Copyright © 2009 Me
```

This needs to be in UTF-8 format. On Windows, you can create UTF-8 files in Notepad. Make sure you use Save As..., and select UTF-8 encoding.



Make the name *language.utf8*, where *language* is the appropriate code for this language (en, fr, de, ru, zh, etc.). This way, you can create separate files for each localization.

(I recommend using [ISO 639 codes](#) for languages.)

## Converting the Text to a Resource File

This needs to be converted to a format that can be read easily by the MIDP application. MIDP does not provide convenient ways to read text files, like J2SE's `BufferedReader`. Unicode support can also be a problem when converting between bytes and characters. The easiest way to read text is to use `DataInput.readUTF()`. But to use this, we need to have written the text using `DataOutput.writeUTF()`.

Below is a simple J2SE, command-line program that will read the `.uft8` file you saved from notepad, and create a `.res` file to go in the JAR.

```
import java.io.*;
import java.util.*;

public class TextConverter {

    public static void main(String[] args) {
        if (args.length == 1) {
            String language = args[0];
```

## How\_to\_create\_localized\_text\_resources

```
List<String> text = new Vector<String>();

try {
    // read text from Notepad UTF-8 file
    InputStream in = new FileInputStream(language + ".utf8");
    try {
        BufferedReader bufin = new BufferedReader(new InputStreamReader(in, "UTF-8"))
        String s;
        while ( (s = bufin.readLine()) != null ) {
            // remove formatting character added by Notepad
            s = s.replaceAll("\ufffe", "");
            text.add(s);
        }
    } finally {
        in.close();
    }

    // write it for easy reading in J2ME
    OutputStream out = new FileOutputStream(language + ".res");
    DataOutputStream dout = new DataOutputStream(out);
    try {
        // first item is the number of strings
        dout.writeShort(text.size());
        // then the string themselves
        for (String s: text) {
            dout.writeUTF(s);
        }
    } finally {
        dout.close();
    }
} catch (Exception e) {
    System.err.println("TextConverter: " + e);
}
} else {
    System.err.println("syntax: TextConverter <language-code>");
}
}
```

To convert en.utf8 to en.res, run the converter as:

```
java TextConverter en
```

## Using the Text at Runtime

Place the .res file in the JAR. You can add as many as you like (typically, for each language, en.res, ro.res, ru.res, zh.res, etc.).

In the MIDP application, the text can be read with this method:

```
public String[] loadText(String resName) throws IOException {
    String[] text;
    InputStream in = getClass().getResourceAsStream(resName);
    try {
        DataInputStream din = new DataInputStream(in);
        int size = din.readShort();
        text = new String[size];
    }
```

## How\_to\_create\_localized\_text\_resources

```
        for (int i = 0; i < size; i++) {
            text[i] = din.readUTF();
        }
    } finally {
        in.close();
    }
    return text;
}
```

This returns an array, with one element for each line in the original text file.

Add a class to contain index constants.

```
public class Text {
    public static final int GAME_NAME = 0;
    public static final int OK = 1;
    public static final int CANCEL = 2;
    public static final int COPYRIGHT = 3;
}
```

Since all these members are constants, they will be inlined by the compiler, and the class will be stripped from the build by Proguard.

Load and use text like this:

```
String[] text = loadText(language + ".res");

Command OK = new Command(text[Text.OK], Command.OK, 1);
```

Since the `TextConverter` doesn't really care what the "language code" is, you can just as easily choose some other naming convention. For example, you might choose to have separate text resources for different purposes (such as `help_en`, `menu_en`, etc.). This means you don't have to keep unwanted text in memory, and can help on low-memory devices.

## Comparison to Other Techniques

This is not the only way to solve the problem of localized text, and is not always the best. However, it has the following benefits.

- It supports unicode, so works in all languages.
- It will behave correctly on the vast majority of devices.
- The device-side code is very simple, with a very low overhead.
- The JAR space requirements are minimal.
- The memory overhead is relatively small.
- Text is located very quickly and reliably.

Techniques that involve storing text in Hashtables tend to require more memory, more JAR space, and are prone to problems where text item names are misspelled.

Techniques that rely on one-byte-per-character encoding tend to result in portability issues, and usually support only a minimal number of languages.