Java API Programming and Secure Coding Concepts

Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master.

Streams

Java programs perform I/O through streams.

A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.

All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket.

Likewise, an output stream may refer to the console, a disk file, or a network connection.

Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example.

Java implements streams within class hierarchies defined in the java.io package.

Byte Streams and Character Streams

Java defines two types of streams: byte and character.

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.

Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: InputStream and OutputStream. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 13-1 The Byte Stream Classes in **java.io**

The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement. Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: Reader and Writer. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 13-2 The Character Stream I/O Classes in **java.io**

The abstract classes Reader and Writer define several key methods that the other stream classes implement. Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

Reading Console Input

In Java, console input is accomplished by reading from System.in. To obtain a character-based stream that is attached to the console, wrap System.in in a BufferedReader object. BufferedReader supports a buffered input stream. A commonly used constructor is shown here:

```
BufferedReader(Reader inputReader)
```

Here, inputReader is the stream that is linked to the instance of BufferedReader that is being created. Reader is an abstract class. One of its concrete subclasses is InputStreamReader, which converts bytes to characters. To obtain an InputStreamReader object that is linked to System.in, use the following constructor:

```
InputStreamReader(InputStream inputStream)
```

Because System.in refers to an object of type InputStream, it can be used for inputStream.

Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
```

After this statement executes, br is a character-based stream that is linked to the console through System.in.

Reading Characters

To read a character from a BufferedReader, use read( ). The version of read( ) that we will be using is

```
int read( ) throws IOException
```

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered. As you can see, it can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "q." Notice that any I/O exceptions that might be generated are simply thrown out of main( ). Such an approach is common when reading from the console in simple example programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```
package com.example.io;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

//Use a BufferedReader to read characters from the console.

public class BRRead {

	public static void main(String[] args) throws IOException {
		char c;
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		System.out.println("Enter characters, 'q' to quit.");
		// read characters
		do {
			c = (char) br.read();
			System.out.println(c);
		} while (c != 'q');

	}

}
```

Reading Strings

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

```
String readLine( ) throws IOException
```

As you can see, it returns a String object.

The following program demonstrates BufferedReader and the readLine( ) method; the program reads and displays lines of text until you enter the word "stop":

```
package com.example.io;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

//Read a string from console using a BufferedReader.

public class BRReadLines {

	public static void main(String[] args) throws IOException {
		// create a BufferedReader using System.in
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		String str;
		System.out.println("Enter lines of text.");
		System.out.println("Enter 'stop' to quit.");
		do {
			str = br.readLine();
			System.out.println(str);
		} while (!str.equals("stop"));
	}

}
```

The next example creates a tiny text editor. It creates an array of String objects and then reads in lines of text, storing each line in the array. It will read up to 100 lines or until you enter "stop." It uses a BufferedReader to read from the console.

```
package com.example.io;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

//A tiny editor.

public class TinyEdit {

	public static void main(String[] args) throws IOException {
		// create a BufferedReader using System.in
```

```
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            String str[] = new String[100];

            System.out.println("Enter lines of text.");
            System.out.println("Enter 'stop' to quit.");

            for (int i = 0; i < 100; i++) {
                  str[i] = br.readLine();
                  if (str[i].equals("stop"))
                        break;
            }

            System.out.println("\nHere is your file:");

            // display the lines
            for (int i = 0; i < 100; i++) {
                  if (str[i].equals("stop"))
                        break;
                  System.out.println(str[i]);
            }
      }

}
```

Writing Console Output

Console output is most easily accomplished with print( ) and println( ), described earlier, which are used in most of the examples in this book. These methods are defined by the class PrintStream (which is the type of object referenced by System.out). Even though System.out is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is shown here:

```
void write(int byteval)
```

This method writes the byte specified by byteval. Although byteval is declared as an integer, only the low-order eight bits are written. Here is a short example that uses write( ) to output the character "A" followed by a newline to the screen:

```
package com.example.io;

//Demonstrate System.out.write().

public class WriteDemo {

      public static void main(String[] args) {
            int b;
            b = 'A';
            System.out.write(b);
            System.out.write('\n');
```

```
	}

}
```

You will not often use write( ) to perform console output (although doing so might be useful in some situations) because print( ) and println( ) are substantially easier to use.

The PrintWriter Class

Although using System.out to write to the console is acceptable, its use is probably best for debugging purposes or for sample programs, such as those found in this book.

For real-world programs, the recommended method of writing to the console when using Java is through a PrintWriter stream. PrintWriter is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier.

PrintWriter defines several constructors. The one we will use is shown here:

```
PrintWriter(OutputStream outputStream, boolean flushingOn)
```

Here, outputStream is an object of type OutputStream, and flushingOn controls whether Java flushes the output stream every time a println( ) method (among others) is called. If flushingOn is true, flushing automatically takes place. If false, flushing is not automatic.

The following application illustrates using a PrintWriter to handle console output:

```
package com.example.io;

import java.io.PrintWriter;

//Demonstrate PrintWriter

public class PrintWriterDemo {

	public static void main(String[] args) {
		PrintWriter pw = new PrintWriter(System.out, true);

		pw.println("This is a string");
		int i = -7;
		pw.println(i);
		double d = 4.5e-7;
		pw.println(d);

	}

}
```

Reading and Writing Files

The following program uses read( ) to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

```
package com.example.io;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

/* Display a text file.

	To use this program, specify the name
	of the file that you want to see.
	For example, to see a file called TEST.TXT,
	use the following command line.

	java ShowFile TEST.TXT
*/

public class ShowFile {

	public static void main(String[] args) {
		int i;
		FileInputStream fin = null;

		// First, confirm that a filename has been specified.
		if (args.length != 1) {
			System.out.println("Usage: ShowFile filename");
			return;
		}

		// The following code opens a file, reads characters until EOF
		// is encountered, and then closes the file via a finally block.
		try {
			fin = new FileInputStream(args[0]);
			do {
				i = fin.read();
				if (i != -1)
					System.out.print((char) i);
			} while (i != -1);
		} catch (FileNotFoundException e) {
			System.out.println("File Not Found.");
		} catch (IOException e) {
			System.out.println("An I/O Error Occurred");
		} finally {
			// Close file in all cases.
			try {
				if (fin != null)
					fin.close();
			} catch (IOException e) {
				System.out.println("Error Closing File");
			}
		}
```

```
	}
}
```

Automatically Closing a File

Automatic resource management is based on an expanded form of the try statement. Here is its general form:

```
try (resource-specification) {
   // use the resource
}
```

Here, resource-specification is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the try block ends, the resource is automatically released.

In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call close( ) explicitly.) Of course, this form of try can also include catch and finally clauses. This new form of try is called the try-with-resources statement.

The try-with-resources statement can be used only with those resources that implement the AutoCloseable interface defined by java.lang. This interface defines the close( ) method. AutoCloseable is inherited by the Closeable interface in java.io. Both interfaces are implemented by the stream classes. Thus, try-with-resources can be used when working with streams, including file streams.

```
package com.example.io;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/* A version of CopyFile that uses try-with-resources.
	It demonstrates two resources (in this case files) being
	managed by a single try statement.
*/

public class CopyFile {

	public static void main(String[] args) {
		int i;

		// First, confirm that both files have been specified.
		if (args.length != 2) {
			System.out.println("Usage: CopyFile from to");
			return;
		}

		// Open and manage two files via the try statement.
		try (FileInputStream fin = new FileInputStream(args[0]);
```

```
                        FileOutputStream fout = new FileOutputStream(args[1]))
{
                do {
                        i = fin.read();
                        if (i != -1)
                                fout.write(i);
                } while (i != -1);
        } catch (IOException e) {
                System.out.println("I/O Error: " + e);
        }

    }

}
```

The NIO Classes

Beginning with version 1.4, Java has provided a second I/O system called NIO (which is short for New I/O). It supports a buffer-oriented, channel-based approach to I/O perations.

With the release of JDK 7, the NIO system was greatly expanded, providing enhanced support for file-handling and file system features. In fact, so significant were the changes that the term NIO.2 is often used. Because of the capabilities supported by the NIO file classes, NIO is expected to become an increasingly important approach to file handling.

The NIO classes are contained in the packages shown here:

Package	Purpose
java.nio	Top-level package for the NIO system. Encapsulates various types of buffers that contain data operated upon by the NIO system.
java.nio.channels	Supports channels, which are essentially open I/O connections.
java.nio.channels.spi	Supports service providers for channels.
java.nio.charset	Encapsulates character sets. Also supports encoders and decoders that convert characters to bytes and bytes to characters, respectively.
java.nio.charset.spi	Supports service providers for character sets.
java.nio.file	Provides support for files.
java.nio.file.attribute	Provides support for file attributes.
java.nio.file.spi	Supports service providers for file systems.

NIO Fundamentals

The NIO system is built on two foundational items: buffers and channels.

A buffer holds data.

A channel represents an open connection to an I/O device, such as a file or a socket.

In general, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold data. You then operate on the buffer, inputting or outputting data as needed.

Buffers

Buffers are defined in the java.nio package. All buffers are subclasses of the Buffer class, which defines the core functionality common to all buffers: current position, limit, and capacity.

The current position is the index within the buffer at which the next read or write operation will take place. The current position is advanced by most read or write operations.

The limit is the index value one past the last valid location in the buffer.

The capacity is the number of elements that the buffer can hold.

Often the limit equals the capacity of the buffer. Buffer also supports mark and reset.

Method	Description
abstract Object array( )	If the invoking buffer is backed by an array, returns a reference to the array. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown.
abstract int arrayOffset( )	If the invoking buffer is backed by an array, returns the index of the first element. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown.
final int capacity( )	Returns the number of elements that the invoking buffer is capable of holding.
final Buffer clear( )	Clears the invoking buffer and returns a reference to the buffer.
final Buffer flip( )	Sets the invoking buffer's limit to the current position and resets the current position to 0. Returns a reference to the buffer.
abstract boolean hasArray( )	Returns **true** if the invoking buffer is backed by a read/write array and **false** otherwise.
final boolean hasRemaining( )	Returns **true** if there are elements remaining in the invoking buffer. Returns **false** otherwise.

Table 21-1 The Methods Defined by **Buffer**

Method	Description
abstract boolean isDirect( )	Returns **true** if the invoking buffer is direct, which means I/O operations act directly upon it. Returns **false** otherwise.
abstract boolean isReadOnly( )	Returns **true** if the invoking buffer is read-only. Returns **false** otherwise.
final int limit( )	Returns the invoking buffer's limit.
final Buffer limit(int *n*)	Sets the invoking buffer's limit to *n*. Returns a reference to the buffer.
final Buffer mark( )	Sets the mark and returns a reference to the invoking buffer.
final int position( )	Returns the current position.
final Buffer position(int *n*)	Sets the invoking buffer's current position to *n*. Returns a reference to the buffer.
int remaining( )	Returns the number of elements available before the limit is reached. In other words, it returns the limit minus the current position.
final Buffer reset( )	Resets the current position of the invoking buffer to the previously set mark. Returns a reference to the buffer.
final Buffer rewind( )	Sets the position of the invoking buffer to 0. Returns a reference to the buffer.

Table 21-1 The Methods Defined by **Buffer** *(continued)*

Channels

Channels are defined in java.nio.channels. A channel represents an open connection to an I/O source or destination. Channels implement the Channel interface. It extends Closeable, and it extends AutoCloseable. By implementing AutoCloseable, channels can be managed with a try-with-resources statement. When used in a try-with-resources block, a channel is closed automatically when it is no longer needed.

Method	Description
abstract byte get( )	Returns the byte at the current position.
ByteBuffer get(byte *vals*[ ])	Copies the invoking buffer into the array referred to by *vals*. Returns a reference to the buffer. If there are not *vals*.**length** elements remaining in the buffer, a **BufferUnderflowException** is thrown.
ByteBuffer get(byte *vals*[ ], int *start*, int *num*)	Copies *num* elements from the invoking buffer into the array referred to by *vals*, beginning at the index specified by *start*. Returns a reference to the buffer. If there are not *num* elements remaining in the buffer, a **BufferUnderflowException** is thrown.
abstract byte get(int *idx*)	Returns the byte at the index specified by *idx* within the invoking buffer.
abstract ByteBuffer put(byte *b*)	Copies *b* into the invoking buffer at the current position. Returns a reference to the buffer. If the buffer is full, a **BufferOverflowException** is thrown.
final ByteBuffer put(byte *vals*[ ])	Copies all elements of *vals* into the invoking buffer, beginning at the current position. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown.
ByteBuffer put(byte *vals*[ ], int *start*, int *num*)	Copies *num* elements from *vals*, beginning at *start*, into the invoking buffer. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown.
ByteBuffer put(ByteBuffer *bb*)	Copies the elements in *bb* to the invoking buffer, beginning at the current position. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. Returns a reference to the buffer.
abstract ByteBuffer put(int *idx*, byte *b*)	Copies *b* into the invoking buffer at the location specified by *idx*. Returns a reference to the buffer.

Table 21-2 The **get( )** and **put( )** Methods Defined for **ByteBuffer**

One way to obtain a channel is by calling getChannel( ) on an object that supports channels. For example, getChannel( ) is supported by the following I/O classes:

DatagramSocket	FileInputStream	FileOutputStream
RandomAccessFile	ServerSocket	Socket

Enhancements Added to NIO by JDK 7

Beginning with JDK 7, the NIO system was substantially expanded and enhanced. In addition to support for the try-with-resources statement (which provides automatic resource management), the improvements included three new packages (java.nio.file, java.nio.file.attribute, and java.nio.file.spi); several new classes, interfaces, and methods; and direct support for stream-based I/O. The additions have greatly expanded the ways in which NIO can be used, especially with files.

The Path Interface

Perhaps the single most important addition to the NIO system is the Path interface because it encapsulates a path to a file. As you will see, Path is the glue that binds together many of the NIO.2 file-based features. It describes a file's location within the directory structure.

Path is packaged in java.nio.file, and it inherits the following interfaces: Watchable, Iterable<Path>, and Comparable<Path>. Watchable describes an object that can be monitored for changes.

Method	Description
boolean endsWith(String *path*)	Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**.
boolean endsWith(Path *path*)	Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**.
Path getFileName( )	Returns the filename associated with the invoking **Path**.
Path getName(int *idx*)	Returns a **Path** object that contains the name of the path element specified by *idx* within the invoking object. The leftmost element is at index 0. This is the element nearest the root. The rightmost element is at **getNameCount( ) – 1**.
int getNameCount( )	Returns the number of elements beyond the root directory in the invoking **Path**.
Path getParent( )	Returns a **Path** that contains the entire path except for the name of the file specified by the invoking **Path**.
Path getRoot( )	Returns the root of the invoking **Path**.

Table 21-3 A Sampling of Methods Specified by **Path**

Method	Description
boolean isAbsolute( )	Returns **true** if the invoking **Path** is absolute. Otherwise, returns **false**.
Path resolve(Path *path*)	If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified.
Path resolve(String *path*)	If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified.
boolean startsWith(String *path*)	Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**.
boolean startsWith(Path *path*)	Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**.
Path toAbsolutePath( )	Returns the invoking **Path** as an absolute path.
String toString( )	Returns a string representation of the invoking **Path**.

Table 21-3 A Sampling of Methods Specified by **Path** *(continued)*

The Files Class

Many of the actions that you perform on a file are provided by static methods within the Files class. The file to be acted upon is specified by its Path. Thus, the Files methods use a Path to specify the file that is being operated upon. Files contains a wide array of functionality.

For example, it has methods that let you open or create a file that has the specified path. You can obtain information about a Path, such as whether it is executable, hidden, or read-only. Files also supplies methods that let you copy or move files.

JDK 8 adds these four methods to Files: list( ), walk( ), lines( ), and find( ). All return a Stream object.

The Paths Class

Because Path is an interface, not a class, you can't create an instance of Path directly through the use of a constructor. Instead, you obtain a Path by a calling a method that returns one. Frequently, you do this by using the get( ) method defined by the Paths class.

There are two forms of get( ). The one used in this chapter is shown here:

```
static Path get(String pathname, String ... parts)
```

It returns a Path that encapsulates the specified path.

The path can be specified in two ways. First, if parts is not used, then the path must be specified in its entirety by pathname.

Alternatively, you can pass the path in pieces, with the first part passed in pathname and the subsequent elements specified by the parts varargs parameter. In either case, if the path specified is syntactically invalid, get( ) will throw an InvalidPathException.

The second form of get( ) creates a Path from a URI. It is shown here:

```
static Path get(URI uri)
```

The Path corresponding to uri is returned.

It is important to understand that creating a Path to a file does not open or create a file. It simply creates an object that encapsulates the file's directory path.

The File Attribute Interfaces

Associated with a file is a set of attributes. These attributes include such things as the file's time of creation, the time of its last modification, whether the file is a directory, and its size. NIO organizes file attributes into several different interfaces. Attributes are represented by a hierarchy of interfaces defined in java.nio.file.attribute. At the top is BasicFileAttributes. It encapsulates the set of attributes that are commonly found in a variety of file systems.

Method	Description
FileTime creationTime( )	Returns the time at which the file was created. If creation time is not provided by the file system, then an implementation-dependent value is returned.
Object fileKey( )	Returns the file key. If not supported, **null** is returned.
boolean isDirectory( )	Returns **true** if the file represents a directory.
boolean isOther( )	Returns **true** if the file is not a file, symbolic link, or a directory.
boolean isRegularFile( )	Returns **true** if the file is a normal file, rather than a directory or symbolic link.
boolean isSymbolicLink( )	Returns **true** if the file is a symbolic link.
FileTime lastAccessTime( )	Returns the time at which the file was last accessed. If the time of last access is not provided by the file system, then an implementation-dependent value is returned.
FileTime lastModifiedTime( )	Returns the time at which the file was last modified. If the time of last modification is not provided by the file system, then an implementation-dependent value is returned.
long size( )	Returns the size of the file.

Table 21-6 The Methods Defined by **BasicFileAttributes**

The FileSystem , FileSystems, and FileStore Classes

You can easily access the file system through the FileSystem and FileSystems classes packaged in java.nio.file. In fact, by using the newFileSystem( ) method defined by FileSystems, it is even possible to obtain a new file system. The FileStore class encapsulates the file storage system. Although these classes are not used directly in this chapter, you may find them helpful in your own applications.

Secure Coding

Refer to page [Secure Coding Guidelines for Java SE (oracle.com)]

Localization

Why Localize?

The decision to create a version of an application for international use often happens at the start of a development project.

- Region– and language-aware software.
- Dates, numbers, and currencies formatted for specific countries.
- Ability to plug in country-specific data without changing code.

Localization is the process of adapting software for specific region or language by adding locale-specific components and translating text.

The goal is to design for localization so that no coding changes are required.

A Sample Application

Localize a sample application.

- Text-based user interface.
- Localize menus.
- Display currency and date localizations.

```
=== Localization App ===
1. Set to English
2. Set to French
3. Set to Chinese
4. Set to Russian
5. Show me the date
6. Show me the money!
q. Enter q to quit
Enter a command:
```

Locale

In Java, a locale is specified by using two values: language and country.

- Language
  - An alpha-2 or alpha-3 ISO 639 code
  - "en" for English, "es" for Spanish
  - Always uses lowercase
- Country
  - Uses the ISO 3166 alpha-2 country code or UN M.49 numeric area code
  - "US" for United States, "ES" for Spain
  - Always uses uppercase

Reference on localization [Creating a Locale (The Java™ Tutorials > Internationalization > Setting the Locale) (oracle.com)]

Properties

- The java.util.Properties class is used to load and save key-value pairs in Java.
- Can be stored in a simple text file.
- File name ends in .properties
- File can be anywhere that compiler can find it.

```
hostName = www.example.com
userName = user
password = pass
```

The benefit of a properties file is the ability to set values for your application externally. The properties file is typically read at the start of the application and is used for default values. But the properties file can also be an integral part of a localization scheme, where you store the values of menu labels and text for various languages that your application may support.

The convention for a properties file is <filename>.properties, but the file can have any extension you want. The file can be located anywhere that the application can find it.

Loading and Using a Properties File

```
1   public static void main(String[] args) {
2     Properties myProps = new Properties();
3     try {
4       FileInputStream fis = new FileInputStream("ServerInfo.properties");
5       myProps.load(fis);
6     } catch (IOException e) {
7       System.out.println("Error: " + e.getMessage());
8     }
9
10     // Print Values
11     System.out.println("Server: " + myProps.getProperty("hostName"));
12     System.out.println("User: " + myProps.getProperty("userName"));
13     System.out.println("Password: " + myProps.getProperty("password"));
14  }
```

In the code fragment, you create a Properties object. Then, using a try statement, you open a file relative to the source files in your NetBeans project. When it is loaded, the name-value pairs are available for use in your application.

Properties files enable you to easily inject configuration information or other application data into the application.

Loading Properties from the Command Line

- Property information can also be passed on the command line.
- Use the –D option to pass key-value pairs:

```
java -Dpropertyname=value -Dpropertyname=value myApp
```

- For example, pass one of the previous values:

```
java -Dusername=user myApp
```

- Get the Properties data from the System object:

```
String userName = System.getProperty("username");
```

Property information can also be passed on the command line. The advantage to passing properties from the command line is simplicity. You do not have to open a file and read from it. However, if you have more than a few parameters, a properties file is preferable.

Resource Bundle

Design for localization begins by designing the application so that all the text, sounds, and images can be replaced at run time with the appropriate elements for the region and culture desired. Resource bundles contain key-value pairs that can be hard-coded within a class or located in a .properties file.

- The ResourceBundle class isolates locale-specific data:
  - Returns key/value pairs stored separately.
  - Can be a class or a .properties file.
- Steps to use:
  - Create bundle files for each locale.
  - Call a specific locale from your application.

Resource Bundle File

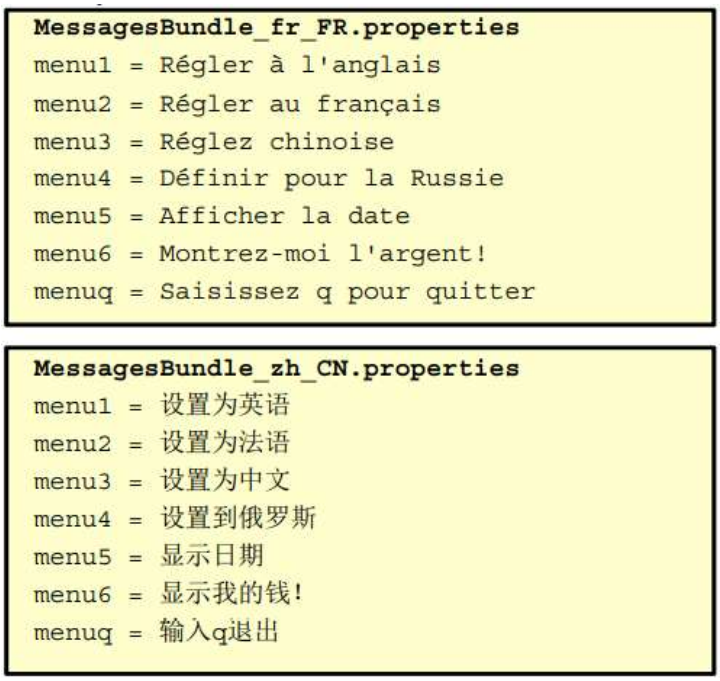
- Properties file contains a set of key-value pairs.
  - Each key identifies a specific application component.
  - Special file names use language and country codes.
- Default for sample application.
  - Menu converted into resource bundle.

The diagram shows a sample resource bundle file for this application. Each menu option has been converted into a name/value pair. This is the default file for the application. For alternative languages, a special naming convention is used:

`MessageBundle_xx_YY.properties`

where xx is the language code and YY is the country code.

Samples for French and Chinese.



The diagram shows the resource bundle files for French and Chinese. Note that the file names include both language and country. The English menu item text has been replaced with French and Chinese alternatives.

Initializing the Sample Application

```
PrintWriter pw = new PrintWriter(System.out, true);
    // More init code here

    Locale usLocale = Locale.US;
    Locale frLocale = Locale.FRANCE;
    Locale zhLocale = new Locale("zh", "CN");
    Locale ruLocale = new Locale("ru", "RU");
    Locale currentLocale = Locale.getDefault();

    ResourceBundle messages = ResourceBundle.getBundle("MessagesBundle",
    currentLocale);

    // more init code here

    public static void main(String[] args){
        SampleApp ui = new SampleApp();
        ui.run();
    }
```

With the resource bundles created, you simply need to load the bundles into the application.

The source code in the diagram shows the steps. First, create a Locale object that specifies the language and country. Then load the resource bundle by specifying the base file name for the bundle and the current Locale.

Note that there are a couple of ways to define a Locale. The Locale class includes default constants for some countries. If a constant is not available, you can use the language code with the country code to define the location. Finally, you can use the getDefault() method to get the default location.

Sample Application: Main Loop

For this application, a run method contains the main loop. The loop runs until the letter "q" is typed in as input. A string switch is used to perform an operation based on the number entered.

A simple call is made to each method to make locale changes and display a formatted output.

```
public void run(){
    String line = "";
    while (!(line.equals("q"))){
        this.printMenu();
        try { line = this.br.readLine(); }
        catch (Exception e){ e.printStackTrace(); }

        switch (line){
            case "1": setEnglish(); break;
            case "2": setFrench(); break;
            case "3": setChinese(); break;
            case "4": setRussian(); break;
            case "5": showDate(); break;
            case "6": showMoney(); break;
        }
    }
}
```

The printMenu Method

Instead of text, a resource bundle is used.

- `messages` is a resource bundle.
- A key is used to retrieve each menu item.
- Language is selected based on the Locale setting.

```
public void printMenu(){
    pw.println("=== Localization App ===");
    pw.println("1. " + messages.getString("menu1"));
    pw.println("2. " + messages.getString("menu2"));
    pw.println("3. " + messages.getString("menu3"));
    pw.println("4. " + messages.getString("menu4"));
    pw.println("5. " + messages.getString("menu5"));
    pw.println("6. " + messages.getString("menu6"));
    pw.println("q. " + messages.getString("menuq"));
    System.out.print(messages.getString("menucommand")+" ");
}
```

Instead of printing text, the resource bundle (messages) is called and the current Locale determines what language is presented to the user.

Changing the Locale

To change the Locale:

- Set currentLocale to the desired language.
- Reload the bundle by using the current locale.

```
public void setFrench(){
    currentLocale = frLocale;
    messages = ResourceBundle.getBundle("MessagesBundle",
    currentLocale);
}
```

After the menu bundle is updated with the correct locale, the interface text is output by using the currently selected language.

Sample Interface with French

After the French option is selected, the updated user interface looks like the following:

```
=== Localization App ===
1. Régler à l'anglais
2. Régler au français
3. Réglez chinoise
4. Définir pour la Russie
5. Afficher la date
6. Montrez-moi l'argent!
q. Saisissez q pour quitter
Entrez une commande:
```

Format Date and Currency

Changing text is not the only available localization tool. Dates and numbers can also be formatted based on local standards.

- Numbers can be localized and displayed in their local format.
- Special format classes include:
    - java.time.format.DateTimeFormatter
    - java.text.NumberFormat
- Create objects using Locale.

Displaying Currency

Create a NumberFormat object by using the selected locale and get a formatted output.

- Format currency:
  - Get a currency instance from NumberFormat.
  - Pass the Double to the format method.
- Sample currency output:

```
1 000 000 руб. ru_RU
1 000 000,00 € fr_FR
￥1,000,000.00 zh_CN
£1,000,000.00 en_GB
```

Formatting Currency with NumberFormat

Set the location and a numeric value to be displayed. Then, set up a NumberFormat object with a specified location. Pass the Double to the format method to print the formatted currency.

```
 1 package com.example.format;
 2
 3 import java.text.NumberFormat;
 4 import java.util.Locale;
 5
 6 public class NumberTest {
 7
 8   public static void main(String[] args) {
 9
10     Locale loc = Locale.UK;
11     NumberFormat nf = NumberFormat.getCurrencyInstance(loc);
12     double money = 1_000_000.00d;
13
14     System.out.println("Money: " + nf.format(money) + " in
    Locale: " + loc);
15   }
16 }
```

Displaying Dates

Create a date format object by using the locale and the date is formatted for the selected locale.

- Format a date:
  - Get a DateTimeFormatter object based on the Locale.
  - From the LocalDateTime variable, call the format method passing the formatter.
- Sample dates:

```
20 juil. 2011 fr_FR
20.07.2011 ru_RU
```

Displaying Dates with DateTimeFormatter

The setup of the DateTimeFormatter is a bit verbose, but fairly clear. A factory is used to specify a style and a locale. Then the formatter is passed to the LocalDateTime object's format method.

```
 3 import java.time.LocalDateTime;
 4 import java.time.format.DateTimeFormatter;
 5 import java.time.format.FormatStyle;
 6 import java.util.Locale;
 7
 8 public class DateFormatTest {
 9   public static void main(String[] args) {
10
11     LocalDateTime today = LocalDateTime.now();
12     Locale loc = Locale.FRANCE;
13
14     DateTimeFormatter df =
15       DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
16         .withLocale(loc);
17     System.out.println("Date: " + today.format(df)
18         + " Locale: " + loc.toString());
19   }
```

Format Styles

- DateTimeFormatter uses the FormatStyle enumeration to determine how the data is formatted.
- Enumeration values
    - SHORT: Is completely numeric, such as 12.13.52 or 3:30 pm
    - MEDIUM: Is longer, such as Jan 12, 1952
    - LONG: Is longer, such as January 12, 1952 or 3:30:32 pm
    - FULL: Is completely specified date or time, such as Tuesday, April 12, 1952 AD or 3:30:42 pm PST