Java

New features in JDK 1.4

Presented by Bartosz Sakowicz

Overview of new features

- **New Input/Output**
- **Java Web Start**
- **Logging**
- **Assertion facility**
- **Exceptions**
- **Collections**
- **Regular Expressions**

Presented by Bartosz Sakowicz

DMCS TUL

New Input/Output

The New I/O (NIO) API introduced in JDK 1.4 provides a completely new model of low-level I/O.

Unlike the original I/O libraries in the java.io package, which were strongly stream-oriented, the New I/O API in the java.nio package is *block-oriente*d. This means that I/O operations, wherever possible, are performed on large blocks of data in a single step, rather than on one byte or character at a time.

Channels and buffers represent the two basic abstractions within the New I/O API.

Presented by Bartosz Sakowicz

DMCS TUL

Channels

Channels correspond roughly to input and output streams: they are sources and sinks for sequential data.

However, whereas input and output streams deal most directly with single bytes, channels read and write data in chunks.

Additionally, **a channel can be bidirectional**, in which case it corresponds to *both* an input stream and an output stream.

Buffers

The chunks of data that are written to and read from channels are contained in objects called ***buffer*s**.

A buffer is an array of data enclosed in an abstraction that makes reading from, and writing to, channels easy and convenient.

Copying files example

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class CopyFile
{
static public void main( String args[] ) throws Exception {
String infile = args[0], outfile = args[1];
FileInputStream fin = new FileInputStream( infile );
FileOutputStream fout = new FileOutputStream( outfile );
```

Copying files example(2)

```
FileChannel inc = fin.getChannel();
FileChannel outc = fout.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
while (true) {
int ret = inc.read( buffer );
if (ret==-1) // nothing left to read
break;
buffer.flip();          // prepare buffer to reading
outc.write( buffer );
buffer.clear();         // Make room for the next read
} } }
```

Java Web Start

• **Java Web Start (JAWS)** is a new application deployment system.

• It allows you to install software with a single click within a web browser that has been enhanced with the **Java Web Start plug-in**.

• It transparently handles complex installation procedures, and it caches software on the local hard drive so that successive executions of the program are as fast as possible.

• It can even use different versions of the JDK for different applications, and will download new versions if that is required by the applications.

• Java Web Start does not require that any special modifications be made to an application before it can be used within the system.

Java Web Start(2)

JAWS execute applications within a sandbox, for security reasons. This security barrier can be bypassed by *trusted* applications (applications that have been digitally signed by a trusted party).

Untrusted applications are provided with limited system access via the j**avax.jnlp package**.

JAWS is based on the **Java Network Launching Protocol** & API (JNLP).

JNLP is the underlying technology that defines the underlying abstractions; JAWS is the reference implementation.

JAWS represents a bridge between applets and full-fledged applications, combining, as much as possible, the best features of both.

JAWS execution model

The application files—data and Java classes—are placed on the web server along with a special **launch file (JNLP file)**

When a JAWS application is launched, it is downloaded from the web server, unless it has already been downloaded. The exact way the user initiates the application depends on the operating system, but for all platforms, there is a central client program called the **Application Manager.** This program displays a list of JAWS applications that it knows about, including those that were first launched via a link in a web browser.

JAWS applications can also be run in offline mode. If a JAWS application is fully downloaded and does not itself require network access, then it can be run without access to the server from which it was downloaded.

JLNP launch file example

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://server/PicoDraw/"
<!-- The relative URL of this file -->
href="PicoDraw.jnlp">
<information>
<title>PicoDraw</title>
<!-- A web page containing more information about the
application. This URL will be displayed in
the JAWS Application Manager -->
<homepage href="http://www.manning.com/"/>
```

JLNP launch file example(2)

```
<description>PicoDraw</description>
<description kind="short">
A *very* tiny draw program</description>
<!-- A URL pointing at a GIF or JPG icon file -->
<icon href="images/picodraw.jpg"/>
<!-- Declares that the application can run without
access to the server it was downloaded from -->
<offline-allowed/>
</information>
```

JLNP launch file example(3)

<security>

<!-- Request that the application be given full access to the local (executing) machine, as if it were a regular Java application. Requires that all JAR files be signedby a trusted party -->

<all-permissions/>

</security>

<resources>

<!-- Specify the versions of the Java Runtime Environment (JRE) that are supported by the application. Multiple entries of this kind are allowed, in which case they are considered to be in order of preference -->

<j2se version="1.4"/>

JLNP launch file example(4)

```
<!-- Specify the relative URL of a JAR file containing code or data.
Specifying lazy tells the JAWS system that the file does not need
to be downloaded before the application can be run -->

<jar href="lib/classes.jar"/>

<jar href="lib/backgrounds.jar"/>

</resources>

<!-- Declare the class containing main() -->

<application-desc main-class="PicoDraw"/>


</jnlp>
```

Web Server configuration

The only requirement is that the web server recognize the ***application/x-java-jnlp-file*** MIME type.

When the web server is asked for a JNLP file, it must send the file as this type.

Sandbox restrictions

The sandbox offers the JAWS application restricted access to system resources via the ***javax.jnlp*** package.

For example, it is possible for the application to read and write files, but each time a file is opened for reading or writing, the user must explicitly approve this through a Load or Save dialog box.

This feature is intended to allow applications to load or save documents, rather than to allow free access to any files on the system.

Logging

The Logging API within JDK 1.4, in the java.util.logging package, provides a flexible and powerful system for logging messages, and for turning these messages on and off, without the need for recompiling.

This means that logging messages can be turned on by the end user, long after the software has shipped.

The Logging API uses a system-wide configuration file to define default settings, and additional configuration files can be used to provide greater control.

Programmatic control is also available—you can configure the logging system directly from your program.

Presented by Bartosz Sakowicz

DMCS TUL

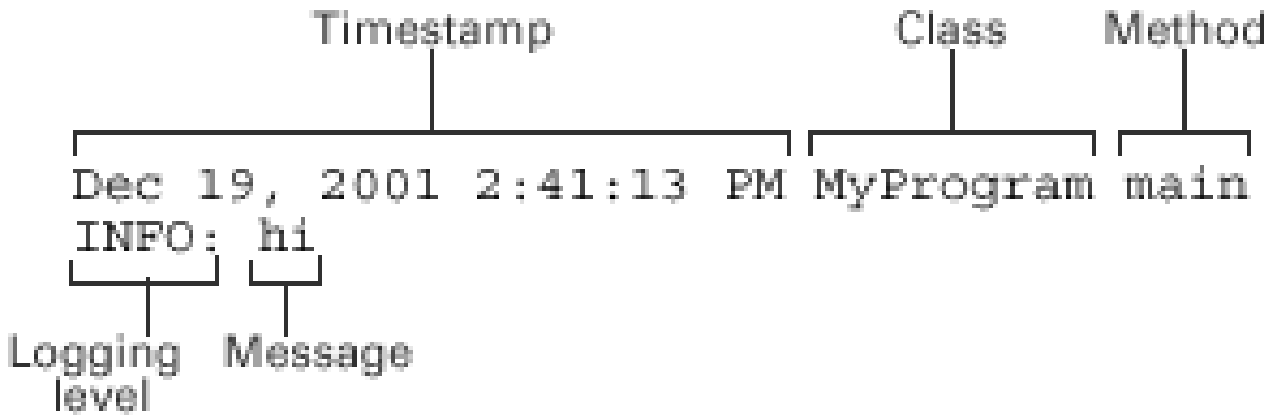
Logging (2)

The simplest way to log a message is as follows:

Logger logger = Logger.getLogger("current.package");

logger.info("hi");

This results, under the default JDK configuration, in the following output:



Logging levels

SEVERE—Used for catastrophic errors—conditions from which the program may not recover, and which, in any case, require immediate attention.

WARNING—Used for serious problems that may or may not be catastrophic. These do not necessarily require immediate attention, but they should definitely be noted.

INFO—Used for run-of-the-mill messages. The INFO level is the default, and so messages logged at the INFO level are seen during normal runs.

CONFIG—Used for logging configuration settings, generally at startup.

FINE, FINER, FINEST—Used for detailed logging. This information is generally logged for debugging. The finer the level, the more information is logged at that level.

Presented by Bartosz Sakowicz

DMCS TUL

The LogRecord class

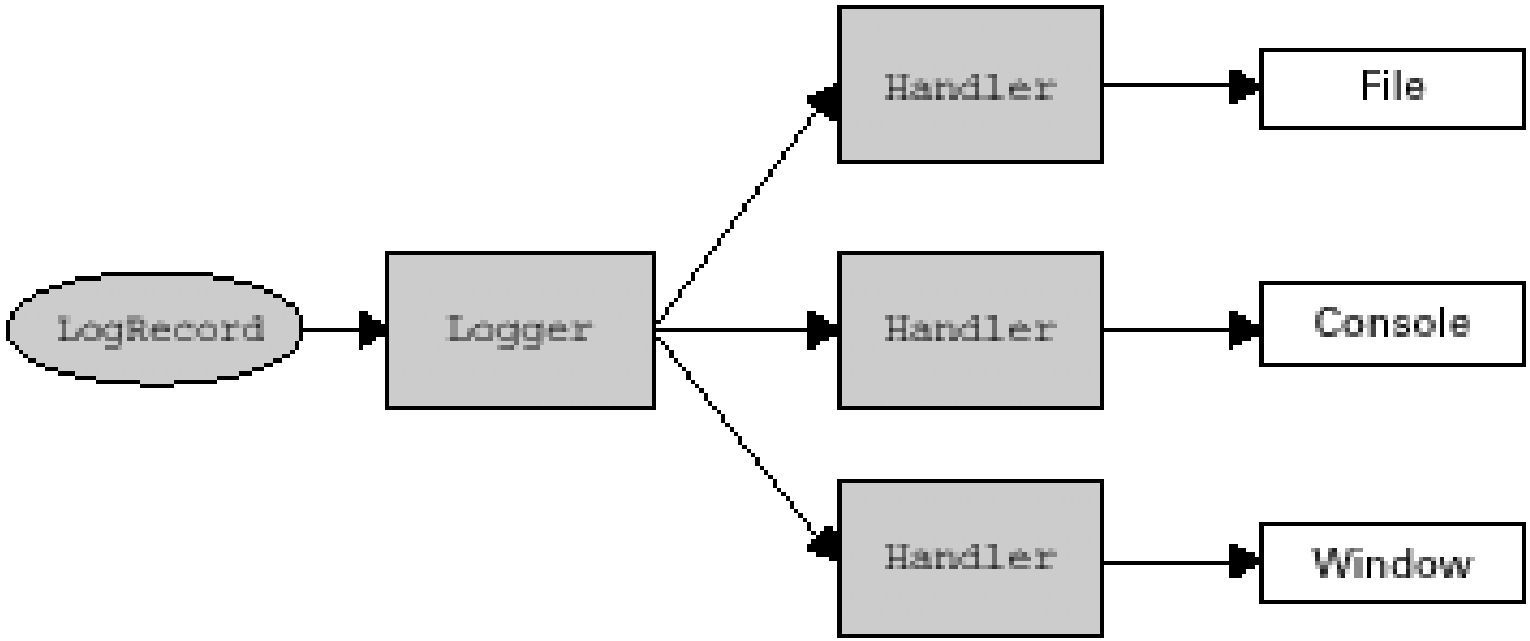
A **LogRecord** object encapsulates a message sent to the logging system. Most of the Logger.log() methods take a String as an argument, but this string is encapsulated by the logger within a LogRecord.

A LogRecord example elements:

- The raw message string
- A logging level
- The logger's name
- A timestamp
- A unique sequence number
- The thread ID of the Thread that generated the LogRecord

Handlers

Each logger is assigned one or more Handlers. A logger is assigned a handler by using its addHandler() method. A handler takes a LogRecord and sends it some-where—the destination depends on the handler . Each logger can have any number of handlers installed.



Handlers(2)

The following handlers are included in the ***java.util.logging*** package:

FileHandler—Writes log messages to a particular file

ConsoleHandler—Writes log messages to the console (or command-line shell)

MemoryHandler—Writes log messages to a circular buffer; can dump recent messages, on command, to another handler

SocketHandler—Writes log messages to a server listening on a particular host and port

StreamHandler—Writes log messages to a particular stream

Formatters

A Formatter is used by a Handler to turn a LogRecord into a String so that it can be displayed or stored in some way. There are two Formatters in the java.util.logging package:

SimpleFormatter—Produces the default format

XMLFormatter—Transforms a LogRecord into a standard XML format

Configuring the logging system

All loggers are defined within the context of a LogManager.The default LogManager reads its configuration from ***jre/lib/logging.properties***, formatted as a properties file, within the JDK installation directory.

The **java.util.logging.config.file** system property allows a different file to be specified,

Configuring handlers

The configuration file allows properties of **each** handler to be set:

java.util.logging.FileHandler.limit = 50000

#The maximum number of bytes to write to file, or zero for no limit

java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

#A Formatter object to format LogRecords before output.
#java.util.logging.SimpleFormatter or *java.util.logging.XML*
#Formatter, depending on the handler

Configuring loggers

The configuration file permits the log level of a particular Logger to be set. Since a Logger corresponds to a package within the larger application, this allows you to configure your application's logging on a package-by-package basis. **Example:**

com.xyz.myapp.level = SEVERE

com.xyz.myapp.io.level = INFO

com.xyz.myapp.net.level = WARNING

A logger's level can be set at runtime using the **Logger.setLevel()** method.

Logging usage example

```
import java.util.logging.*;

...

static private Logger logger;

static {

logger = Logger.getLogger( "myapp.package.com" );

//logger.setLevel( Level.ALL );

}

...
```

Logging usage example

```
public MultiplexingChatServer( int port ) {
new LoggerGUI();
this.port = port;
logger.config( "Will listen on port "+port );
Thread t = new Thread( this, "MultiplexingChatServer" );
t.start();
logger.fine( "Started background I/O thread" );
}
```

Assertion facility

The **assertion facility** provides a mechanism for adding optional “sanity checks” to your code.

These checks are used during the development and testing phases, but are turned off when the software is deployed.

This allows the programmer to insert debugging checks that might be too slow or memory-intensive to use in a real context, but that help during development.

In a sense, assertions are a lot like error checks, except that they are turned off for deployment.

Assertion example

```
public class aClass {

        public void aMethod( int argument ) {

                Foo foo = null;

                // ... somehow get a Foo object

                // Check to make sure we've managed to get one:

                assert foo != null;

        }

}
```

This asserts that foo is not null. If foo is in fact null, an **AssertionError** is thrown. **Any code that executes after this line can safely assume that foo is not null.**

Assertion vs. other error code

The programmer's decision to use an assertion instead of other error-handling code is often based on a general rule of programming psychology:

the less likely a programmer thinks an error is, the less code she will write to deal with it.

An assertion is easier to write than a RuntimeException; a RuntimeException is easier to write than a regular Exception.

A good rule of thumb is that **you should use an assertion for exceptional cases that you would like to forget about**. An assertion is the quickest way to deal with, and forget, a condition or state that you don't expect to have to deal with.

Assertion vs. other error ... (2)

Example:

An application might have a hidden configuration file that it never deletes. Since it's *possible, but unlikel*y, that the user will ever delete this hidden configuration file, it might be a good idea, after trying toopen the file, to assert that the open worked. It almost certainly will, but it's a good idea to check.

An assertion is a convenient syntax for checking for an error. In a sense, **it's really just a shorthand for a full error check.**

Assertion syntax

assert expression;

Meaning: if expression isn’t true throw an error

OR:

assert expression_1 : expression_2;

Meaning: if expression_1 isn’t true, throw an error containing the value of expression_2. **Example:**

assert foo != null : "Can't get a Foo, argument="+argument;

Compiling with assertions

One of the most useful features of assertions is that they can be turned off during normal usage, so that they don't incur any speed penalty. **Assertions are off by default.**

Assertions are enabled on the command line via the *-ea* switch, which is an abbreviation for the *-enableassertions* switch. The following two commands are equivalent:

java -ea myPackage.myProgram

java -enableassertions myPackage.myProgram

Assertions are similarly disabled with either the *-da* or -disableassertions commands:

java -da myPackage.myProgram

java -disableassertions myPackage.myProgram

Compiling with assertions(2)

One of the most useful features of assertions is that they can be turned off during normal usage, so that they don't incur any speed penalty. **Assertions are off by default.**

Assertions are enabled on the command line via the *-ea* switch, which is an abbreviation for the *-enableassertions* switch. The following two commands are equivalent:

java -ea myPackage.myProgram

java -enableassertions myPackage.myProgram

Assertions are similarly disabled with either the *-da* or -isableassertions commands:

java -da myPackage.myProgram

java -disableassertions myPackage.myProgram

Other features

• It is possible enable/disable assertions for particular class or packages

• It is possible to enable/disable assertions programatically:

ClassLoader methods:

public void setClassAssertionStatus(String className,

boolean enabled);

public void setPackageAssertionStatus(String packageName,

boolean enabled);

• It is possible to catch AssertionError (as a normal Error) but than it is recommended to rethrow it.

Exceptions

When a piece of code catches one exception, only to throw another exception, the first exception can be thought of as the cause of the second one. The **chained exception** feature provides a formal recognition of this programming pattern.

Example (old technique):

```
public void write( byte b[] ) throws IOException {
        try {
         // ...
        } catch( SomeInternalException e ) {
                throw new IOException( e.toString() );
        }
}
```

Exceptions(2)

In order to formalize this technique and achieve consistency in the way that these things are handled and displayed, every Throwable now officially has a **cause**, which itself is another Throwable.

The value of a Throwable's cause can be set in its constructor:

new Exception("message", oldException);

Or it can be set using its *initCause()* method:

```
Exception e = new Exception( "message" );
e.initCause( oldException );
```

You can get the cause of an exception by calling its **getCause()** method.

Collections

There are three new classes : **LinkedHashMap, LinkedHashSet,** and **IdentityHashMap.**

The first two are variations on the HashMap and HashSet classes, and they preserve the order in which objects are added to them.

The third, IdentityHashMap, is a Map that overrides the hashCode() methods of its keys, allowing objects to be differentiated based on identity rather than on content.

There is also a new marker interface, **RandomAccess**, which allows a class to declare that it is suitable for fast random access. This means that it has efficient implementations of the get() and set() methods.

Collections(2)

There are new methods in Collection interface:

- Collections.rotate() method rotates the elements of a list, which means that it advances each element a certain number of steps. Elements that are advanced off the end of the list are wrapped around to the beginning of the list.
- Collections.sublist()
- Collections.indexOfSublist()
- Collections.lastIndexOfSublist()
- Collections.swap()

Regular Expressions

A **regular expression**, or **regex**, is an expression that defines a subset of the space of all possible text strings. A regex is said to match a string if the string is within that subset.

Regexes can be used to distinguish correct input from incorrect input, or to look for particular kinds of text within a larger body of text.

Since JDK 1.4, Java includes regexes as part of the core platform, in the ***java.util.regex*** package.

Regular Expressions(2)

A **literal** is any character from the character set that doesn't have a special meaning within a regex. A literal in a regular expression matches only itself.

The **.** character matches *any* single character, except newline. Newlines are also recognized if the DOTALL flag is specified.

The **+** character, when placed after a regular expression, matches one or more copies of that expression. The * character matches zero or more copies.

Parentheses can be placed around any regular expression in order to treat it as a unit. This does not change what it matches, but it does allow modifiers to affect an entire expression, rather than a single character. For example, just as g* means zero or more occurrences of g, (gh)* means zero or more occurrences of gh.

Presented by Bartosz Sakowicz

DMCS TUL

Regular Expressions(3)

A **character class** defines a set of individual characters. For example, the regex [abc] matches an a, a b, or a c; the regex [abc]+ matches one or more characters, each of which is an a, b, or c. You can also use the notation a-z to specify, in this case, all the characters between a and z.

Putting a **caret (^)** at the start of such an expression *negates* the category, which means the expression matches only characters that are *not* in the set. Thus, the regex [^abc] matches any character which is not an a, b, or c.

There are also predefined characters. Examples:

\s Any white space character [\t\n\x0B\f\r]

\S Any character except a white space character [^ \t\n\x0B\f\r]

Presented by Bartosz Sakowicz

DMCS TUL

Regular Expressions(4)

The regex facility in java.util.regex is defined by only three classes:

Pattern — Represents a regular expression

Matcher — Matches a Pattern against a string

PatternSyntaxException — Exception thrown while attempting to compile a regular expression

Creating a Pattern object :

Pattern pattern = Pattern.compile("\\S+\\s+\\S+");

This regex matches two words (made of any non-white space characters) separated by some white space.

Regular Expressions(4)

Once you have a Pattern object, you create a **Matcher** object for a particular input string:

Matcher matcher = pattern.matcher("hey there");

The argument to Pattern.matcher() does not have to be a String—it has to be an object that implements the java.lang.CharSequence interface. The following classes implement CharSequence:

java.lang.String

java.lang.StringBuffer

java.nio.CharBuffer

Matcher has special useful methods for finding and replacing.