Java Platform, Standard Edition

Core Libraries

Release 14
F23119-02
April 2020

Java Platform, Standard Edition Core Libraries, Release 14

F23119-02

Contents

Preface

This guide provides information about the Java core libraries.

Audience

This document is for Java developers who develop applications that require functionality such as threading, process control, I/O, monitoring and management of the JVM, serialization, concurrency, and other functionality close to the VM. .

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

See JDK 14 Documentation.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Java Core Libraries

The core libraries consist of classes which are used by many portions of the JDK. They include functionality which is close to the VM and is not explicitly included in other areas, such as security. Here you will find current information that will help you use some of the core libraries.

Topics in this Guide

- Serialization Filtering
- Enhanced Deprecation
- XML Catalog API
- Creating Unmodifiable Lists, Sets, and Maps
- Process API
- Preferences API
- Java Logging Overview

Other Core Libraries Guides

- Internationalization Overview in *Java Platform, Standard Edition Internationalization Guide*

Security Related Topics

- Serialization Filtering
- RMI:
  - RMI Security Recommendations in *Java Platform, Standard Edition Java Remote Method Invocation User's Guide*
  - Using Custom Socket Factories with Java RMI in the Java Tutorials
- JAXP:
  - JAXP Processing Limits in the Java Tutorials
  - External Access Restriction Properties in the Java Tutorials

2
Serialization Filtering

You can use the Java serialization filtering mechanism to help prevent deserialization vulnerabilities. You can define pattern-based filters or you can create custom filters.

Topics:

Addressing Deserialization Vulnerabilities

An application that accepts untrusted data and deserializes it is vulnerable to attacks. You can create filters to screen incoming streams of serialized objects before they are deserialized.

An object is serialized when its state is converted to a byte stream. That stream can be sent to a file, to a database, or over a network. A Java object is serializable if its class or any of its superclasses implements either the `java.io.Serializable` interface or the `java.io.Externalizable` subinterface. In the JDK, serialization is used in many areas, including Remote Method Invocation (RMI), custom RMI for interprocess communication (IPC) protocols (such as the Spring HTTP invoker), Java Management Extensions (JMX), and Java Messaging Service (JMS).

An object is deserialized when its serialized form is converted to a copy of the object. It is important to ensure the security of this conversion. Deserialization is code execution, because the `readObject` method of the class that is being deserialized can contain custom code. Serializable classes, also known as "gadget classes", can do arbitrary reflective actions such as create classes and invoke methods on them. If your application deserializes these classes, they can cause a denial of service or remote code execution.

When you create a filter, you can specify which classes are acceptable to an application, and which should be rejected. You can control the object graph size and complexity during deserialization so that the object graph doesn't exceed reasonable limits. Filters can be configured as properties, or implemented programmatically.

Besides creating filters, you can take the following actions to help prevent deserialization vulnerabilities:

- Do not deserialize untrusted data.
- Use SSL to encrypt and authenticate the connections between applications.

- Validate field values before assignment, including checking object invariants by using the `readObject` method.

> **Note:**
>
> Built-in filters are provided for RMI. However, you should use these built-in filters as starting points only. Configure blacklists and/or extend the whitelist to add additional protection for your application that uses RMI. See Built-in Filters.

For more information about these and other strategies, see "Serialization and Deserialization" in Secure Coding Guidelines for Java SE.

Java Serialization Filters

The Java serialization filtering mechanism screens incoming streams of serialized objects to help improve security and robustness. Filters can validate incoming classes before they are deserialized.

As stated in JEP 290, the goals of the Java serialization filtering mechanism are to:

- Provide a way to narrow the classes that can be deserialized down to a context-appropriate set of classes.
- Provide metrics to the filter for graph size and complexity during deserialization to validate normal graph behaviors.
- Allow RMI-exported objects to validate the classes expected in invocations.

You can implement serialization filters in the following ways:

- Pattern-based filters do not require you to modify your application. They consist of a sequence of patterns that are defined in properties, in a configuration file or on the command line. Pattern-based filters can accept or reject specific classes, packages, or modules. They can place limits on array sizes, graph depth, total references, and stream size. A typical use case is to blacklist classes that have been identified as potentially compromising the Java runtime. Pattern-based filters are defined for one application or all applications in a process.
- Custom filters are implemented using the `ObjectInputFilter` API. They allow an application to integrate finer control than pattern-based filters, because they can be specific to each `ObjectInputStream`. Custom filters are set on an individual input stream or on all streams in a process.

The filter mechanism is called for each new object in the stream. If more than one active filter (process-wide filter, application filter, or stream-specific filter) exists, only the most specific filter is called.

In most cases, a custom filter should check if a process-wide filter is set. If one exists, the custom filter should invoke it and use the process-wide filter's result, unless the status is `UNDECIDED`.

Support for serialization filters is included starting with JDK 9, and in Java CPU releases starting with 8u121, 7u131, and 6u141.

Whitelists and Blacklists

Whitelists and blacklists can be implemented using pattern-based filters or custom filters. These lists allow you to take proactive and defensive approaches to protect your applications.

The proactive approach uses whitelists to accept only the classes that are recognized and trusted. You can implement whitelists in your code when you develop your application, or later by defining pattern-based filters. If your application only deals with a small set of classes then this approach can work very well. You can implement whitelists by specifying the classes, packages, or modules that are allowed.

The defensive approach uses blacklists to reject classes that are not trusted. Usually, blacklists are implemented after an attack that reveals that a class is a problem. A class can be added to a blacklist, without a code change, by defining a pattern-based filter.

Creating Pattern-Based Filters

Pattern-based filters are filters that you define without changing your application code. You add process-wide filters in properties files, or application-specific filters on the `java` command line.

A pattern-based filter is a sequence of patterns. Each pattern is matched against the name of a class in the stream or a resource limit. Class-based and resource limit patterns can be combined in one filter string, with each pattern separated by a semicolon (;).

Pattern-based Filter Syntax

When you create a filter that is composed of patterns, use the following guidelines:

- Separate patterns by semicolons. For example:

  ```
  pattern1.*;pattern2.*
  ```

- White space is significant and is considered part of the pattern.
- Put the limits first in the string. They are evaluated first regardless of where they are in the string, so putting them first reinforces the ordering. Otherwise, patterns are evaluated from left to right.
- A class that matches a pattern that is preceded by `!` is rejected. A class that matches a pattern without `!` is accepted. The following filter rejects `pattern1.MyClass` but accepts `pattern2.MyClass`:

  ```
  !pattern1.*;pattern2.*
  ```

- Use the wildcard symbol (*) to represent unspecified classes in a pattern as shown in the following examples:
  - To match every class, use *
  - To match every class in `mypackage`, use `mypackage.*`
  - To match every class in `mypackage` and its subpackages, use `mypackage.**`

  - To match every class that starts with `text`, use `text*`

If a class doesn't match any filter, then it is accepted. If you want to accept only certain classes, then your filter must reject everything that doesn't match. To reject all classes other than those specified, include `!*` as the last pattern in a class filter.

For a complete description of the syntax for the patterns, see the `conf/security/java.security` file, or see JEP 290.

Pattern-Based Filter Limitations

Pattern-based filters are used for simple acceptance or rejection. These filters have some limitations. For example:

- Patterns can't allow different sizes of arrays based on the class.
- Patterns can't match classes based on the supertype or interfaces of the class.
- Patterns have no state and can't make choices depending on the earlier classes deserialized in the stream.

Define a Pattern-Based Filter for One Application

You can define a pattern-based filter as a system property for one application. A system property supersedes a Security Property value.

To create a filter that only applies to one application, and only to a single invocation of Java, define the `jdk.serialFilter` system property in the command line.

The following example shows how to limit resource usage for an individual application:

```
java -
Djdk.serialFilter=maxarray=100000;maxdepth=20;maxrefs=500 com.example.test.
Application
```

Define a Pattern-Based Filter for All Applications in a Process

You can define a pattern-based filter as a Security Property, for all applications in a process. A system property supersedes a Security Property value.

1. Edit the `java.security` properties file.
   - JDK 9 and later: `$JAVA_HOME/conf/security/java.security`
   - JDK 8,7,6: `$JAVA_HOME/lib/security/java.security`
2. Add the pattern to the `jdk.serialFilter` Security Property.

Define a Class Filter

You can create a pattern-based class filter that is applied globally. For example, the pattern might be a class name or a package with wildcard.

In the following example, the filter rejects one class from a package (`!example.somepackage.SomeClass`), and accepts all other classes in the package:

```
jdk.serialFilter=!example.somepackage.SomeClass;example.somepackage.*;
```

The previous example filter accepts all other classes, not just those in `example.somepackage.*`. To reject all other classes, add `!*`:

```
jdk.serialFilter=!example.somepackage.SomeClass;example.somepackage.*;!*
```

Define a Resource Limit Filter

A resource filter limits graph complexity and size. You can create filters for the following parameters to control the resource usage for each application:

- Maximum allowed array size. For example: `maxarray=100000;`
- Maximum depth of a graph. For example: `maxdepth=20;`
- Maximum references in a graph between objects. For example: `maxrefs=500;`
- Maximum number of bytes in a stream. For example: `maxbytes=500000;`

Creating Custom Filters

Custom filters are filters you specify in your application's code. They are set on an individual stream or on all streams in a process. You can implement a custom filter as a pattern, a method, a lambda expression, or a class.

Reading a Stream of Serialized Objects

You can set a custom filter on one `ObjectInputStream`, or, to apply the same filter to every stream, set a process-wide filter. If an `ObjectInputStream` doesn't have a filter defined for it, the process-wide filter is called, if there is one.

While the stream is being decoded, the following actions occur:

- For each new object in the stream, the filter is called before the object is instantiated and deserialized.
- For each class in the stream, the filter is called with the resolved class. It is called separately for each supertype and interface in the stream.
- The filter can examine each class referenced in the stream, including the class of objects to be created, supertypes of those classes, and their interfaces.
- For each array in the stream, whether it is an array of primitives, array of strings, or array of objects, the filter is called with the array class and the array length.
- For each reference to an object already read from the stream, the filter is called so it can check the depth, number of references, and stream length. The depth starts at 1 and increases for each nested object and decreases when each nested call returns.
- The filter is not called for primitives or for `java.lang.String` instances that are encoded concretely in the stream.
- The filter returns a status of accept, reject, or undecided.
- Filter actions are logged if logging is enabled.

Unless a filter rejects the object, the object is accepted.

Setting a Custom Filter for an Individual Stream

You can set a filter on an individual `ObjectInputStream` when the input to the stream is untrusted and the filter has a limited set of classes or constraints to enforce. For example, you could ensure that a stream only contains numbers, strings, and other application-specified types.

A custom filter is set using the `setObjectInputFilter` method. The custom filter must be set before objects are read from the stream.

In the following example, the `setObjectInputFilter` method is invoked with the `dateTimeFilter` method. This filter only accepts classes from the `java.time` package. The `dateTimeFilter` method is defined in a code sample in Setting a Custom Filter as a Method.

```
    LocalDateTime readDateTime(InputStream is) throws IOException {
        try (ObjectInputStream ois = new ObjectInputStream(is)) {
            ois.setObjectInputFilter(FilterClass::dateTimeFilter);
            return (LocalDateTime) ois.readObject();
        } catch (ClassNotFoundException ex) {
            IOException ioe = new StreamCorruptedException("class
missing");
            ioe.initCause(ex);
            throw ioe;
        }
    }
```

Setting a Process-Wide Custom Filter

You can set a process-wide filter that applies to every use of `ObjectInputStream` unless it is overridden on a specific stream. If you can identify every type and condition that is needed by the entire application, the filter can allow those and reject the rest. Typically, process-wide filters are used to reject specific classes or packages, or to limit array sizes, graph depth, or total graph size.

A process-wide filter is set once using the methods of the `ObjectInputFilter.Config` class. The filter can be an instance of a class, a lambda expression, a method reference, or a pattern.

```
    ObjectInputFilter filter = ...
    ObjectInputFilter.Config.setSerialFilter(filter);
```

In the following example, the process-wide filter is set by using a lambda expression.

```
    ObjectInputFilter.Config.setSerialFilter(info -> info.depth() > 10 ?
Status.REJECTED : Status.UNDECIDED);
```

In the following example, the process-wide filter is set by using a method reference:

```
    ObjectInputFilter.Config.setSerialFilter(FilterClass::dateTimeFilter);
```

Setting a Custom Filter Using a Pattern

A pattern-based custom filter, which is convenient for simple cases, can be created by using the `ObjectInputFilter.Config.createFilter` method. You can create a pattern-based filter as a system property or Security Property. Implementing a pattern-based filter as a method or a lambda expression gives you more flexibility.

The filter patterns can accept or reject specific classes, packages, modules, and can place limits on array sizes, graph depth, total references, and stream size. Patterns cannot match the supertype or interfaces of the class.

In the following example, the filter allows `example.File` and rejects `example.Directory` classes.

```
    ObjectInputFilter filesOnlyFilter =
ObjectInputFilter.Config.createFilter("example.File;!example.Directory");
```

This example allows only `example.File`. All other classes are rejected.

```
    ObjectInputFilter filesOnlyFilter =
ObjectInputFilter.Config.createFilter("example.File;!*");
```

Setting a Custom Filter as a Class

A custom filter can be implemented as a class implementing the `java.io.ObjectInputFilter` interface, as a lambda expression, or as a method.

A filter is typically stateless and performs checks solely on the input parameters. However, you may implement a filter that, for example, maintains state between calls to the `checkInput` method to count artifacts in the stream.

In the following example, the `FilterNumber` class allows any object that is an instance of the `Number` class and rejects all others.

```
    class FilterNumber implements ObjectInputFilter {
        public Status checkInput(FilterInfo filterInfo) {
            Class<?> clazz = filterInfo.serialClass();
            if (clazz != null) {
                return (Number.class.isAssignableFrom(clazz)) ?
Status.ALLOWED : Status.REJECTED;
            }
            return Status.UNDECIDED;
        }
    }
```

In the example:

- The `checkInput` method accepts an `ObjectInputFilter.FilterInfo` object. The object's methods provide access to the class to be checked, array size, current depth, number of references to existing objects, and stream size read so far.
- If `serialClass` is not null, indicating that a new object is being created, the value is checked to see if the class of the object is `Number`. If so, it is accepted, otherwise it is rejected.

- Any other combination of arguments returns `UNDECIDED`. Deserialization continues, and any remaining filters are run until the object is accepted or rejected. If there are no other filters, the object is accepted.

Setting a Custom Filter as a Method

A custom filter can also be implemented as a method. The method reference is used instead of an inline lambda expression.

The `dateTimeFilter` method that is defined in the following example is used by the code sample in Setting a Custom Filter for an Individual Stream.

```
    public class FilterClass {
        static ObjectInputFilter.Status
dateTimeFilter(ObjectInputFilter.FilterInfo info) {
            Class<?> serialClass = info.serialClass();
            if (serialClass != null) {
                return serialClass.getPackageName().equals("java.time")
                        ? ObjectInputFilter.Status.ALLOWED
                        : ObjectInputFilter.Status.REJECTED;
            }
            return ObjectInputFilter.Status.UNDECIDED;
        }
    }
```

Example: Filter for Classes in the java.base Module

This custom filter, which is also implemented as a method, allows only the classes found in the base module of the JDK. This example works with JDK 9 and later.

```
        static ObjectInputFilter.Status
baseFilter(ObjectInputFilter.FilterInfo info) {
            Class<?> serialClass = info.serialClass();
            if (serialClass != null) {
                return
serialClass.getModule().getName().equals("java.base")
                        ? ObjectInputFilter.Status.ALLOWED
                        : ObjectInputFilter.Status.REJECTED;
            }
            return ObjectInputFilter.Status.UNDECIDED;
       }
```

Built-in Filters

The Java Remote Method Invocation (RMI) Registry, the RMI Distributed Garbage Collector, and Java Management Extensions (JMX) all have filters that are included in

the JDK. You should specify your own filters for the RMI Registry and the RMI Distributed Garbage Collector to add additional protection.

Filters for RMI Registry

> **Note:**
>
> Use these built-in filters as starting points only. Edit the `sun.rmi.registry.registryFilter` system property to configure blacklists and/or extend the whitelist to add additional protection for the RMI Registry. To protect the whole application, add the patterns to the `jdk.serialFilter` global system property to increase protection for other serialization users that do not have their own custom filters.

The RMI Registry has a built-in whitelist filter that allows objects to be bound in the registry. It includes instances of the `java.rmi.Remote`, `java.lang.Number`, `java.lang.reflect.Proxy`, `java.rmi.server.UnicastRef`, `java.rmi.activation.ActivationId`, `java.rmi.server.UID`, `java.rmi.server.RMIClientSocketFactory`, and `java.rmi.server.RMIServerSocketFactory` classes.

The built-in filter includes size limits:

```
maxarray=1000000,maxdepth=20
```

Supersede the built-in filter by defining a filter using the `sun.rmi.registry.registryFilter` system property with a pattern. If the filter that you define either accepts classes passed to the filter, or rejects classes or sizes, the built-in filter is not invoked. If your filter does not accept or reject anything, the built-filter is invoked.

Filters for RMI Distributed Garbage Collector

> **Note:**
>
> Use these built-in filters as starting points only. Edit the `sun.rmi.transport.dgcFilter` system property to configure blacklists and/or extend the whitelist to add additional protection for Distributed Garbage Collector. To protect the whole application, add the patterns to the `jdk.serialFilter` global system property to increase protection for other serialization users that do not have their own custom filters.

The RMI Distributed Garbage Collector has a built-in whitelist filter that accepts a limited set of classes. It includes instances of the `java.rmi.server.ObjID`, `java.rmi.server.UID`, `java.rmi.dgc.VMID`, and `java.rmi.dgc.Lease` classes.

The built-in filter includes size limits:

```
maxarray=1000000,maxdepth=20
```

Supersede the built-in filter by defining a filter using the `sun.rmi.transport.dgcFilter` system property with a pattern. If the filter accepts classes passed to the filter, or rejects classes or sizes, the built-in filter is not invoked. If the superseding filter does not accept or reject anything, the built-filter is invoked.

Filters for JMX

> **Note:**
>
> Use these built-in filters as starting points only. Edit the `jmx.remote.rmi.server.serial.filter.pattern` management property to configure blacklists and/or extend the whitelist to add additional protection for JMX. To protect the whole application, add the patterns to the `jdk.serialFilter` global system property to increase protection for other serialization users that do not have their own custom filters.

JMX has a built-in filter to limit a set of classes allowed to be sent as a deserializing parameters over RMI to the server. That filter is disabled by default. To enable the filter, define the `jmx.remote.rmi.server.serial.filter.pattern` management property with a pattern.

The pattern must include the types that are allowed to be sent as parameters over RMI to the server and all types they depends on, plus `javax.management.ObjectName` and `java.rmi.MarshalledObject` types. For example, to limit the allowed set of classes to Open MBean types and the types they depend on, add the following line to `management.properties` file.

```
com.sun.management.jmxremote.serial.filter.pattern=java.lang.*;java.math.Bi
gInteger;java.math.BigDecimal;java.util.*;javax.management.openmbean.*;java
x.management.ObjectName;java.rmi.MarshalledObject;!*
```

Logging Filter Actions

You can turn on logging to record the initialization, rejections, and acceptances of calls to serialization filters. Use the log output as a diagnostic tool to see what's being deserialized, and to confirm your settings when you configure whitelists and blacklists.

When logging is enabled, filter actions are logged to the `java.io.serialization` logger.

To enable serialization filter logging, edit the `$JDK_HOME/conf/logging.properties` file.

To log calls that are rejected, add

```
java.io.serialization.level = FINER
```

To log all filter results, add

```
java.io.serialization.level = FINEST
```

3
Enhanced Deprecation

The semantics of what deprecation means has been clarified, including whether an API may be removed in the near future.

If you are a library maintainer, you can take advantage of the updated deprecation syntax to inform users of your library about the status of APIs provided by your library.

If you are a library or application developer, you can use the `jdeprscan` tool to find uses of deprecated JDK API elements in your applications or libraries.

Topics

Deprecation in the JDK

Deprecation is a notification to library consumers that they should migrate code from a deprecated API.

In the JDK, APIs have been deprecated for widely varying reasons, such as:

- The API is dangerous (for example, the `Thread.stop` method).
- There is a simple rename (for example, AWT `Component.show/hide` replaced by `setVisible`).
- A newer, better API can be used instead.
- The deprecated API is going to be removed.

In prior releases, APIs were deprecated but virtually never removed. Starting with JDK 9, APIs may be marked as deprecated for removal. This indicates that the API is eligible to be removed in the next release of the JDK platform. If your application or library consumes any of these APIs, then you should make a plan to migrate from them soon.

For a list of deprecated APIs in the current release of the JDK, see the Deprecated API page in the API specification.

How to Deprecate APIs

Deprecating an API requires using two different mechanisms: the `@Deprecated` annotation and the `@deprecated` Javadoc tag.

The `@Deprecated` annotation marks an API in a way that is recorded in the class file and is available at runtime. This allows various tools, such as `javac` and `jdeprscan`, to detect and flag usage of deprecated APIs. The `@deprecated` Javadoc tag is used in

documentation of deprecated APIs, for example, to describe the reason for deprecation, and to suggest alternative APIs.

Note the capitalization: the annotation starts with an uppercase *D* and the Javadoc tag starts with a lowercase *d*.

Using the @Deprecated Annotation

To indicate deprecation, precede the module, class, method, or member declaration with `@Deprecated`. The annotation contains these elements:

- `@Deprecated(since="<version>")`
  - <*version*> is the version when the API was deprecated. This is for informational purposes. The default is the empty string ("").
- `@Deprecated(forRemoval=<boolean>)`
  - `forRemoval=true` indicates that the API is subject to removal in a future release.
  - `forRemoval=false` recommends that code should no longer use this API; however, there is no current intent to remove the API. This is the default value.

For example: `@Deprecated(since="9", forRemoval=true)`

The `@Deprecated` annotation causes the Javadoc-generated documentation to be marked with one of the following, wherever that program element appears:

- **Deprecated.**
- **Deprecated, for removal: This API element is subject to removal in a future version.**

The `javadoc` tool generates a page named `deprecated-list.html` which contains the list of deprecated APIs, and adds a link in the navigation bar to that page.

The following is a simple example of using the `@Deprecated` annotation from the `java.lang.Thread` class:

```
public class Thread implements Runnable {
    ...
    @Deprecated(since="1.2")
    public final void stop() {
        ...
    }
    ...
```

Semantics of Deprecation

The two elements of the `@Deprecated` annotation give developers the opportunity to clarify what deprecation means for their exported APIs.

For the JDK platform:

- `@Deprecated(forRemoval=true)` indicates that the API is eligible to be removed in a future release of the JDK platform.
- `@Deprecated(since="<version>")` contains the JDK version string that indicates when the API element was deprecated, for those deprecated in JDK 9 and beyond.

If you maintain libraries and produce your own APIs, then you probably use the `@Deprecated` annotation. You should determine and communicate your policy around API removals. For example, if you release a new library every 6 weeks, then you may choose to deprecate an API for removal, but not remove it for several months to give your customers time to migrate.

Using the @deprecated Javadoc Tag

Use the `@deprecated` tag in the javadoc comment of any deprecated program element to indicate that it should no longer be used (even though it may continue to work). This tag is valid in all class, method, or field documentation comments. The `@deprecated` tag must be followed by a space or a newline. In the paragraph following the `@deprecated` tag, explain why the item was deprecated, and suggest what to use instead. Mark the text that refers to new versions of the same functionality with an `@link` tag.

When it encounters an `@deprecated` tag, the `javadoc` tool moves the text following the `@deprecated` tag to the front of the description and precedes it with a warning. For example, this source:

```
/**
 * ...
 * @deprecated  This method does not properly convert bytes into
 * characters.  As of JDK 1.1, the preferred way to do this is via the
 * {@code String} constructors that take a {@link
 * java.nio.charset.Charset}, charset name, or that use the platform's
 * default charset.
 * ...
 */
 @Deprecated(since="1.1")
 public String(byte ascii[], int hibyte) {
    ...
```

generates the following output:

```
@Deprecated(since="1.1")
public String(byte[] ascii,
              int hibyte)
Deprecated. This method does not properly convert bytes into characters.
As of
JDK 1.1, the preferred way to do this is via the String constructors that
take a
Charset, charset name, or that use the platform's default charset.
```

If you use the `@deprecated` Javadoc tag without the corresponding `@Deprecated` annotation, a warning is generated.

Notifications and Warnings

When an API is deprecated, developers must be notified. The deprecated API may cause problems in your code, or, if it is eventually removed, cause failures at run time.

The Java compiler generates warnings about deprecated APIs. There are options to generate more information about warnings, and you can also suppress deprecation warnings.

Compiler Deprecation Warnings

If the deprecation is `forRemoval=false`, the Java compiler generates an "ordinary deprecation warning". If the deprecation is `forRemoval=true`, the compiler generates a "removal warning".

The two kinds of warnings are controlled by separate `-Xlint` flags: `-Xlint:deprecation` and `-Xlint:removal`. The `javac -Xlint:removal` option is enabled by default, so removal warnings are shown.

The warnings can also be turned off independently (note the "–"): `-Xlint:-deprecation` and `-Xlint:-removal`.

This is an example of an ordinary deprecation warning.

```
$ javac src/example/DeprecationExample.java
Note: src/example/DeprecationExample.java uses or overrides a deprecated
API.
Note: Recompile with -Xlint:deprecation for details.
```

Use the `javac -Xlint:deprecation` option to see what API is deprecated.

```
$ javac -Xlint:deprecation src/example/DeprecationExample.java
src/example/DeprecationExample.java:12: warning: [deprecation]
getSelectedValues() in JList has been deprecated
   Object[] values = jlist.getSelectedValues();
                     ^
1 warning
```

Here is an example of a removal warning.

```
public class RemovalExample {
    public static void main(String[] args) {
        System.runFinalizersOnExit(true);
    }
}
$ javac RemovalExample.java
RemovalExample.java:3: warning: [removal] runFinalizersOnExit(boolean) in
System
has been deprecated and marked for removal
        System.runFinalizersOnExit(true);
              ^
1 warning
==========
```

Suppressing Deprecation Warnings

The `javac -Xlint` options control warnings for all files compiled in a particular run of `javac`. You may have identified specific locations in source code that generate warnings that you no longer want to see. You can use the `@SuppressWarnings`

annotation to suppress warnings whenever that code is compiled. Place the `@SuppressWarnings` annotation at the declaration of the class, method, field, or local variable that uses a deprecated API.

The `@SuppressWarnings` options are:

- `@SuppressWarnings("deprecation")` — Suppresses only the ordinary deprecation warnings.
- `@SuppressWarnings("removal")` — Suppresses only the removal warnings.
- `@SuppressWarnings({"deprecation","removal"})` — Suppresses both types of warnings.

Here's an example of suppressing a warning.

```
@SuppressWarnings("deprecation")
Object[] values = jlist.getSelectedValues();
```

With the `@SuppressWarnings` annotation, no warnings are issued for this line, even if warnings are enabled on the command line.

Running jdeprscan

`jdeprscan` is a static analysis tool that reports on an application's use of deprecated JDK API elements. Run `jdeprscan` to help identify possible issues in compiled class files or jar files.

You can find out about deprecated JDK APIs from the compiler notifications. However, if you don't recompile with every JDK release, or if the warnings were suppressed, or if you depend on third-party libraries that are distributed as binary artifacts, then you should run `jdeprscan`.

It's important to discover dependencies on deprecated APIs before the APIs are removed from the JDK. If the binary uses an API that is deprecated for removal in the current JDK release, and you don't recompile, then you won't get any notifications. When the API is removed in a future JDK release, then the binary will simply fail at runtime. `jdeprscan` lets you detect such usage now, well before the API is removed.

For the complete syntax of how to run the tool and how to interpret the output, see The jdeprscan Command in the *Java Development Kit Tool Specifications*.

4

XML Catalog API

Use the XML Catalog API to implement a local XML catalog.

Java SE 9 introduced a new XML Catalog API to support the Organization for the Advancement of Structured Information Standards (OASIS) XML Catalogs, OASIS Standard V1.1, 7 October 2005. This chapter of the Core Libraries Guide describes the API, its support by the Java XML processors, and usage patterns.

The XML Catalog API is a straightforward API for implementing a local catalog, and the support by the JDK XML processors makes it easier to configure your processors or the entire environment to take advantage of the feature.

Learning More About Creating Catalogs

To learn about creating catalogs, see the XML Catalogs, OASIS Standard V1.1, 7 October 2005. The XML catalogs under the directory `/etc/xml/catalog` on some Linux distributions can also be a good reference for creating a local catalog.

Purpose of XML Catalog API

The XML Catalog API and the Java XML processors provide an option for developers and system administrators to better manage external resources.

The XML Catalog API provides an implementation of OASIS XML Catalogs v1.1, a standard designed to address issues caused by external resources.

Problems Caused by External Resources

XML, XSD and XSL documents may contain references to external resources that the Java XML processors need to retrieve to process the documents. External resources can cause a problem for the applications or the system. The Catalog API and the Java XML processors provide an option for developers and system administrators to better manage these external resources.

External resources can cause a problem for the applications or the system in these areas:

- Availability. When the resources are remote, the XML processors must be able to connect to the remote server. Even though connectivity is rarely an issue, it's still a factor in the stability of an application. Too many connections can be a hazard to servers that hold the resources (such as the well-documented case involving excessive DTD traffic directed to the W3C's servers), and this in turn could affect your applications. See Use Catalog with XML Processors) for an example that solves this issue using the XML Catalog API.
- Performance. Although in most cases connectivity isn't an issue, a remote fetch can still cause a performance issue for an application. Furthermore, there may be multiple applications on the same system attempting to resolve the same source, and this would be a waste of system resources.

- Security. Allowing remote connections can pose a security risk if the application processes untrusted XML sources.
- Manageability. If a system processes a large number of XML documents, then externally referenced documents, whether local or remote, can become a maintenance hassle.

How XML Catalog API Addresses Problems Caused by External Resources

The XML Catalog API and the Java XML processors provide an option for developers and system administrators to better manage the external resources.

- Application developers – You can create a local catalog of all external references for your application, and let the Catalog API resolve them for the application. This not only avoids remote connections but also makes it easier to manage these resources.
- System administrators – You can establish a local catalog for your system and configure the Java VM to point to the catalog. Then, all of your applications on the system may share the same catalog without any code changes to the applications, assuming they're compatible with Java SE 9. To establish a catalog, you may take advantage of existing catalogs such as those included with some Linux distributions.

XML Catalog API Interfaces

Access the XML Catalog API through its interfaces.

XML Catalog API Interfaces

The XML Catalog API defines the following interfaces:

- The `Catalog` interface represents an entity catalog as defined by XML Catalogs, OASIS Standard V1.1, 7 October 2005. A `Catalog` object is immutable. After it's created, the `Catalog` object can be used to find matches in a `system`, `public`, or `uri` entry. A custom resolver implementation may find it useful to locate local resources through a catalog.
- The `CatalogFeatures` class holds all of the features and properties the Catalog API supports, including `javax.xml.catalog.files`, `javax.xml.catalog.defer`, `javax.xml.catalog.prefer`, and `javax.xml.catalog.resolve`.
- The `CatalogManager` class manages the creation of XML catalogs and catalog resolvers.
- The `CatalogResolver` interface is a catalog resolver that implements SAX `EntityResolver`, StAX `XMLResolver`, DOM LS `LSResourceResolver` used by schema validation, and transform `URIResolver`. This interface resolves external references using catalogs.

Details on the CatalogFeatures Class

The catalog features are collectively defined in the `CatalogFeatures` class. The features are defined at the API and system levels, which means that they can be set through the API, system properties, and JAXP properties. To set a feature through the API, use the `CatalogFeatures` class.

The following code sets `javax.xml.catalog.resolve` to `"continue"` so that the process continues even if no match is found by the `CatalogResolver`:

```
CatalogFeatures f = CatalogFeatures.builder().with(Feature.RESOLVE,
"continue").build();
```

To set this`"continue"` functionality system-wide, use the Java command line or `System.setProperty` method:

```
System.setProperty(Feature.RESOLVE.getPropertyName(), "continue");
```

To set this`"continue"` functionality for the whole JVM instance, enter a line in the `jaxp.properties` file:

```
javax.xml.catalog.resolve = "continue"
```

The `resolve` property, as well as the `prefer` and `defer` properties, can be set as an attribute of the catalog or group entry in a catalog file. For example, in the following catalog, the `resolve` attribute is set with a value `"continue"` on the catalog entry that instructs the processor to continue when the no match is found through this catalog. The attribute can also be set on the `group` entry as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog
  xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
  resolve="continue"
  xml:base="http://local/base/dtd/">
  <group resolve="continue">
    <system
      systemId="http://remote/dtd/alice/docAlice.dtd"
      uri="http://local/dtd/docAliceSys.dtd"/>
  </group>
</catalog>
```

Properties set in a narrower scope override those that are set in a wider one. Therefore, a property set through the API always takes preference.

Using the XML Catalog API

Resolve DTD, entity, and alternate URI references in XML source documents using the various entry types of the XML Catalog standard.

The XML Catalog Standard defines a number of entry types. Among them, the system entries, including `system`, `rewriteSystem`, and `systemSuffix` entries, are used for resolving DTD and entity references in XML source documents, while `uri` entries are for alternate URI references.

System Reference

Use a `CatalogResolver` object to locate a local resource.

Locating a Local Resource

The following example demonstrates how to use a `CatalogResolver` object to locate a local resource using a `system` entry, given an XML file that contains a reference to `example.dtd` property:

```
<?xml version="1.0"?>
<!DOCTYPE catalogtest PUBLIC "-//OPENJDK//XML CATALOG DTD//1.0"
  "http://openjdk.java.net/xml/catalog/dtd/example.dtd">

<catalogtest>
  Test &example; entry
</catalogtest>
```

The `example.dtd` defines an entity `"example"`:

```
<!ENTITY example "system">
```

The URI to the `example.dtd` in the XML doesn't need to exist. The purpose is to provide a unique identifier for the `CatalogResolver` object to locate a local resource. To do this, create a catalog entry file called `catalog.xml` with a `system` entry to refer to the local resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <system
    systemId="http://openjdk.java.net/xml/catalog/dtd/example.dtd"
    uri="example.dtd"/>
</catalog>
```

With this catalog and the `system` entry, all you need to do is get a default `CatalogFeatures` object, and set the URI to the catalog file to create a `CatalogResolver` object:

```
CatalogResolver cr =
  CatalogManager.catalogResolver(CatalogFeatures.defaults(), catalogUri);
```

`catalogUri` must be a valid URI. For example:

```
URI.create("file:///users/auser/catalog/catalog.xml")
```

The `CatalogResolver` object can now be used as a JDK XML resolver. In the following example, it's used as a SAX `EntityResolver`:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
```

```
XMLReader reader = factory.newSAXParser().getXMLReader();
reader.setEntityResolver(cr);
```

Notice that in the example the system identifier is given an absolute URI. That makes it easy for the resolver to find the match with exactly the same `systemId` in the catalog's `system` entry.

If the `system` identifier in the XML is relative, then it may complicate the matching process because the XML processor may have made it absolute with a specified base URI or the source file's URI. In that situation, the `systemId` of the system entry would need to match the anticipated absolute URI. An easier solution is to use the `systemSuffix` entry, for example:

```
<systemSuffix systemIdSuffix="example.dtd" uri="example.dtd"/>
```

The `systemSuffix` entry matches any reference that ends with `example.dtd` in an XML source and resolves it to a local `example.dtd` file as specified in the `uri` attribute. You may add more to the `systemId` to ensure that it's unique or the correct reference. For example, you may set the `systemIdSuffix` to `xml/catalog/dtd/example.dtd`, or rename the `id` in both the XML source file and the `systemSuffix` entry to make it a unique match, for example `my_example.dtd`.

The URI of the `system` entry can be absolute or relative. If the external resources have a fixed location, then an absolute URI is more likely to guarantee uniqueness. If the external resources are placed relative to your application or the catalog entry file, then a relative URI may be more effective, allowing the deployment of your application without knowing where it's installed. Such a relative URI then is resolved using the base URI or the catalog file's URI if the base URI isn't specified. In the previous example, `example.dtd` is assumed to have been placed in the same directory as the catalog file.

Public Reference

Use a `public` entry instead of a `system` entry to find a desired resource.

If no `system` entry matches the desired resource, and the `PREFER` property is specified to match `public`, then a `public` entry can do the same as a `system` entry. Note that `public` is the default setting for the `PREFER` property.

Using a Public Entry

When the DTD reference in the parsed XML file contains a public identifier such as `"-//OPENJDK//XML CATALOG DTD//1.0"`, a `public` entry can be written as follows in the catalog entry file:

```
<public publicId="-//OPENJDK//XML CATALOG DTD//1.0" uri="example.dtd"/>
```

When you create and use a `CatalogResolver` object with this entry file, the `example.dtd` resolves through the `publicId` property. See System Reference for an example of creating a `CatalogResolver` object.

URI Reference

Use a `uri` entry to find a desired resource.

The URI type entries, including `uri`, `rewriteURI`, and `uriSuffix`, can be used in a similar way as the system type entries.

Using URI Entries

While the XML Catalog Standard gives a preference to the `system` type entries for resolving DTD references, and `uri` type entries for everything else, the Java XML Catalog API doesn't make that distinction. This is because the specifications for the existing Java XML Resolvers, such as `XMLResolver` and `LSResourceResolver`, doesn't give a preference. The `uri` type entries, including `uri`, `rewriteURI`, and `uriSuffix`, can be used in a similar way as the `system` type entries. The `uri` elements are defined to associate an alternate URI reference with a URI reference. In the case of `system` reference, this is the `systemId` property.

You may therefore replace the `system` entry with a `uri` entry in the following example, although `system` entries are more generally used for DTD references.

```
<system
  systemId="http://openjdk.java.net/xml/catalog/dtd/example.dtd"
  uri="example.dtd"/>
```

A `uri` entry would look like the following:

```
<uri name="http://openjdk.java.net/xml/catalog/dtd/example.dtd"
uri="example.dtd"/>
```

While `system` entries are frequently used for DTDs, `uri` entries are preferred for URI references such as XSD and XSL import and include. The next example uses a `uri` entry to resolve a XSL import.

As described in XML Catalog API Interfaces, the XML Catalog API defines the `CatalogResolver` interface that extends Java XML Resolvers including `EntityResolver`, `XMLResolver`, `URIResolver`, and `LSResolver`. Therefore, a `CatalogResolver` object can be used by SAX, DOM, StAX, Schema Validation, as well as XSLT Transform. The following code creates a `CatalogResolver` object with default feature settings:

```
CatalogResolver cr =
  CatalogManager.catalogResolver(CatalogFeatures.defaults(), catalogUri);
```

The code then registers this `CatalogResolver` object on a `TransformerFactory` class where a `URIResolver` object is expected:

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setURIResolver(cr);
```

Alternatively the code can register the `CatalogResolver` object on the `Transformer` object:

```
Transformer transformer = factory.newTransformer(xslSource);
transformer.setURIResolver(cur);
```

Assuming the XSL source file contains an `import` element to import the `xslImport.xsl` file into the XSL source:

```
<xsl:import href="pathto/xslImport.xsl"/>
```

To resolve the `import` reference to where the import file is actually located, a `CatalogResolver` object should be set on the `TransformerFactory` class before creating the `Transformer` object, and a `uri` entry such as the following must be added to the catalog entry file:

```
<uri name="pathto/xslImport.xsl" uri="xslImport.xsl"/>
```

The discussion about absolute or relative URIs and the use of `systemSuffix` or `uriSuffix` entries with the system reference applies to the `uri` entries as well.

Java XML Processors Support

Use the XML Catalogs features with the standard Java XML processors.

The XML Catalogs features are supported throughout the Java XML processors, including SAX and DOM (`javax.xml.parsers`), and StAX parsers (`javax.xml.stream`), schema validation (`javax.xml.validation`), and XML transformation (`javax.xml.transform`).

This means that you don't need to create a `CatalogResolver` object outside an XML processor. Catalog files can be registered directly to the Java XML processor, or specified through system properties, or in the `jaxp.properties` file. The XML processors perform the mappings through the catalogs automatically.

Enable Catalog Support

To enable the support for the XML Catalogs feature on a processor, the `USE_CATALOG` feature must be set to `true`, and at least one catalog entry file specified.

USE_CATALOG

A Java XML processor determines whether the XML Catalogs feature is supported based on the value of the `USE_CATALOG` feature. By default, `USE_CATALOG` is set to `true` for all JDK XML Processors. The Java XML processor further checks for the availability of a catalog file, and attempts to use the XML Catalog API only when the `USE_CATALOG` feature is `true` and a catalog is available.

The `USE_CATALOG` feature is supported by the XML Catalog API, the system property, and the `jaxp.properties` file. For example, if `USE_CATALOG` is set to `true` and it's desirable to disable the catalog support for a particular processor, then this can be done by setting the `USE_CATALOG` feature to `false` through the processor's `setFeature`

method. The following code sets the `USE_CATALOG` feature to the specified value `useCatalog` for an `XMLReader` object:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setNamespaceAware(true);
XMLReader reader = spf.newSAXParser().getXMLReader();
if (setUseCatalog) {
   reader.setFeature(XMLConstants.USE_CATALOG, useCatalog);
}
```

On the other hand, if the entire environment must have the catalog turned off, then this can be done by configuring the `jaxp.properties` file with a line:

```
 javax.xml.useCatalog = false;
```

javax.xml.catalog.files

The `javax.xml.catalog.files` property is defined by the XML Catalog API and supported by the JDK XML processors, along with other catalog features. To employ the catalog feature on a parsing, validating, or transforming process, all that's needed is to set the `FILES` property on the processor, through its system property or using the `jaxp.properties` file.

Catalog URI

The catalog file reference must be a valid URI, such as `file:///users/auser/catalog/catalog.xml`.

The URI reference in a system or a URI entry in the catalog file can be absolute or relative. If they're relative, then they are resolved using the catalog file's URI or a base URI if specified.

Using system or uri Entries

When using the XML Catalog API directly (see XML Catalog API Interfaces for an example), `system` and `uri` entries both work when using the JDK XML Processors' native support of the `CatalogFeatures` class. In general, `system` entries are searched first, then `public` entries, and if no match is found then the processor continues searching `uri` entries. Because both `system` and `uri` entries are supported, it's recommended that you follow the custom of XML specifications when selecting between using a `system` or `uri` entry. For example, DTDs are defined with a `systemId` and therefore `system` entries are preferable.

Use Catalog with XML Processors

Use the XML Catalog API with various Java XML processors.

The XML Catalog API is supported throughout JDK XML processors. The following sections describe how it can be enabled for a particular type of processor.

Use Catalog with DOM

To use a catalog with DOM, set the `FILES` property on a `DocumentBuilderFactory` instance as demonstrated in the following code:

```
static final String CATALOG_FILE =
CatalogFeatures.Feature.FILES.getPropertyName();
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
if (catalog != null) {
   dbf.setAttribute(CATALOG_FILE, catalog);
}
```

Note that `catalog` is a URI to a catalog file. For example, it could be something like `"file:///users/auser/catalog/catalog.xml"`.

It's best to deploy resolving target files along with the catalog entry file, so that the files can be resolved relative to the catalog file. For example, if the following is a `uri` entry in the catalog file, then the `XSLImport_html.xsl` file will be located at `/users/auser/catalog/XSLImport_html.xsl`.

```
<uri name="pathto/XSLImport_html.xsl" uri="XSLImport_html.xsl"/>
```

Use Catalog with SAX

To use the Catalog feature on a SAX parser, set the catalog file to the `SAXParser` instance:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setNamespaceAware(true);
spf.setXIncludeAware(true);
SAXParser parser = spf.newSAXParser();
parser.setProperty(CATALOG_FILE, catalog);
```

In the prior sample code, note the statement `spf.setXIncludeAware(true)`. When this is enabled, any `XInclude` is resolved using the catalog as well.

Given an XML file `XI_simple.xml`:

```
<simple>
  <test xmlns:xinclude="http://www.w3.org/2001/XInclude">
    <latin1>
      <firstElement/>
      <xinclude:include href="pathto/XI_text.xml" parse="text"/>
      <insideChildren/>
      <another>
        <deeper>text</deeper>
      </another>
    </latin1>
    <test2>
      <xinclude:include href="pathto/XI_test2.xml"/>
    </test2>
```

```
  </test>
</simple>
```

Additionally, given another XML file `XI_test2.xml`:

```
<?xml version="1.0"?>
<!-- comment before root -->
<!DOCTYPE red SYSTEM "pathto/XI_red.dtd">
<red xmlns:xinclude="http://www.w3.org/2001/XInclude">
  <blue>
    <xinclude:include href="pathto/XI_text.xml" parse="text"/>
  </blue>
</red>
```

Assume another text file, `XI_text.xml`, contains a simple string, and the file `XI_red.dtd` is as follows:

```
 <!ENTITY red "it is read">
```

In these XML files, there is an `XInclude` element inside an `XInclude` element, and a reference to a DTD. Assuming they are located in the same folder along with the catalog file `CatalogSupport.xml`, add the following catalog entries to map them:

```
<uri name="pathto/XI_text.xml" uri="XI_text.xml"/>
<uri name="pathto/XI_test2.xml" uri="XI_test2.xml"/>
<system systemId="pathto/XI_red.dtd" uri="XI_red.dtd"/>
```

When the `parser.parse` method is called to parse the `XI_simple.xml` file, it's able to locate the `XI_test2.xml` file in the `XI_simple.xml` file, and the `XI_text.xml` file and the `XI_red.dtd` file in the `XI_test2.xml` file through the specified catalog.

Use Catalog with StAX

To use the catalog feature with a StAX parser, set the catalog file on the `XMLInputFactory` instance before creating the `XMLStreamReader` object:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(CatalogFeatures.Feature.FILES.getPropertyName(),
catalog);
XMLStreamReader streamReader =
  factory.createXMLStreamReader(xml, new FileInputStream(xml));
```

When the `XMLStreamReader streamReader` object is used to parse the XML source, external references in the source are then resolved in accordance with the specified entries in the catalog.

Note that unlike the `DocumentBuilderFactory` class that has both `setFeature` and `setAttribute` methods, the `XMLInputFactory` class defines only a `setProperty` method. The XML Catalog API features including `XMLConstants.USE_CATALOG` are all

set through this `setProperty` method. For example, to disable `USE_CATALOG` on a `XMLStreamReader` object, you can do the following:

```
factory.setProperty(XMLConstants.USE_CATALOG, false);
```

Use Catalog with Schema Validation

To use a catalog to resolve any external resources in a schema, such as XSD `import` and `include`, set the catalog on the `SchemaFactory` object:

```
SchemaFactory factory =
  SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
factory.setProperty(CatalogFeatures.Feature.FILES.getPropertyName(),
catalog);
Schema schema = factory.newSchema(schemaFile);
```

The XMLSchema schema document contains references to external DTD:

```
<!DOCTYPE xs:schema PUBLIC "-//W3C//DTD XMLSCHEMA 200102//EN" "pathto/
XMLSchema.dtd" [
   ...
]>
```

And to `xsd` import:

```
<xs:import
  namespace="http://www.w3.org/XML/1998/namespace"
  schemaLocation="http://www.w3.org/2001/pathto/xml.xsd">
  <xs:annotation>
    <xs:documentation>
      Get access to the xml: attribute groups for xml:lang
      as declared on 'schema' and 'documentation' below
    </xs:documentation>
  </xs:annotation>
</xs:import>
```

Following along with this example, to use local resources to improve your application performance by reducing calls to the W3C server:

- Include these entries in the catalog set on the `SchemaFactory` object:

```
<public publicId="-//W3C//DTD XMLSCHEMA 200102//EN" uri="XMLSchema.dtd"/>
<!-- XMLSchema.dtd refers to datatypes.dtd -->
<systemSuffix systemIdSuffix="datatypes.dtd" uri="datatypes.dtd"/>
<uri name="http://www.w3.org/2001/pathto/xml.xsd" uri="xml.xsd"/>
```

- Download the source files `XMLSchema.dtd`, `datatypes.dtd`, and `xml.xsd` and save them along with the catalog file.

As already discussed, the XML Catalog API lets you use any of the entry types that you prefer. In the prior case, instead of the `uri` entry, you could also use either one of the following:

- A `public` entry, because the `namespace` attribute in the `import` element is treated as the `publicId` element:

```
<public publicId="http://www.w3.org/XML/1998/namespace" uri="xml.xsd"/>
```

- A `system` entry:

```
<system systemId="http://www.w3.org/2001/pathto/xml.xsd" uri="xml.xsd"/>
```

> **Note:**
>
> When experimenting with the XML Catalog API, it might be useful to ensure that none of the URIs or system IDs used in your sample files points to any actual resources on the internet, and especially not to the W3C server. This lets you catch mistakes early should the catalog resolution fail, and avoids putting a burden on W3C servers, thus freeing them from any unnecessary connections. All the examples in this topic and other related topics about the XML Catalog API, have an arbitrary string "`pathto`" added to any URI for that purpose, so that no URI could possibly resolve to an external W3C resource.

To use the catalog to resolve any external resources in an XML source to be validated, set the catalog on the `Validator` object:

```
SchemaFactory schemaFactory =
  SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = schemaFactory.newSchema();
Validator validator = schema.newValidator();
validator.setProperty(CatalogFeatures.Feature.FILES.getPropertyName(),
catalog);
StreamSource source = new StreamSource(new File(xml));
validator.validate(source);
```

Use Catalog with Transform

To use the XML Catalog API in a XSLT transform process, set the catalog file on the `TransformerFactory` object.

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setAttribute(CatalogFeatures.Feature.FILES.getPropertyName(),
catalog);
Transformer transformer = factory.newTransformer(xslSource);
```

If the XSL source that the factory is using to create the `Transformer` object contains DTD, import, and include statements similar to these:

```
<!DOCTYPE HTMLlat1 SYSTEM "http://openjdk.java.net/xml/catalog/dtd/
XSLDTD.dtd">
<xsl:import href="pathto/XSLImport_html.xsl"/>
<xsl:include href="pathto/XSLInclude_header.xsl"/>
```

Then the following catalog entries can be used to resolve these references:

```
<system
  systemId="http://openjdk.java.net/xml/catalog/dtd/XSLDTD.dtd"
  uri="XSLDTD.dtd"/>
<uri name="pathto/XSLImport_html.xsl" uri="XSLImport_html.xsl"/>
<uri name="pathto/XSLInclude_header.xsl" uri="XSLInclude_header.xsl"/>
```

Calling Order for Resolvers

The JDK XML processors call a custom resolver before the catalog resolver.

Custom Resolver Preferred to Catalog Resolver

The catalog resolver (defined by the `CatalogResolver` interface) can be used to resolve external references by the JDK XML processors to which a catalog file has been set. However, if a custom resolver is also provided, then it's always be placed ahead of the catalog resolver. This means that a JDK XML processor first calls a custom resolver to attempt to resolve external resources. If the resolution is successful, then the processor skips the catalog resolver and continues. Only when there's no custom resolver or if the resolution by a custom resolver returns null, does the processor then call the catalog resolver.

For applications that use custom resolvers, it's therefore safe to set an additional catalog to resolve any resources that the custom resolvers don't handle. For existing applications, if changing the code isn't feasible, then you may set a catalog through the system property or `jaxp.properties` file to redirect external references to local resources knowing that such a setting won't interfere with existing processes that are handled by custom resolvers.

Detecting Errors

Detect configuration issues by isolating the problem.

The XML Catalogs Standard requires that the processors recover from any resource failures and continue, therefore the XML Catalog API ignores any failed catalog entry files without issuing an error, which makes it harder to detect configuration issues.

Dectecting Configuration Issues

To detect configuration issues, isolate the issues by setting one catalog at a time, setting the `RESOLVE` value to `strict`, and checking for a `CatalogException` exception when no match is found.

Table 4-1 RESOLVE Settings

`RESOLVE` Value	`CatalogResolver` Behavior	Description
`strict` (default)	Throws a `CatalogException` if no match is found with a specified reference	An unmatched reference may indicate a possible error in the catalog or in setting the catalog.

Table 4-1 (Cont.) RESOLVE Settings

RESOLVE Value	CatalogResolver Behavior	Description
`continue`	Returns quietly	This is useful in a production environment where you want the XML processors to continue resolving any external references not covered by the catalog.
`ignore`	Returns quietly	For processors such as SAX, that allow skipping the external references, the `ignore` value instructs the `CatalogResolver` object to return an empty `InputSource` object, thus skipping the external reference.

5

Creating Unmodifiable Lists, Sets, and Maps

Convenience static factory methods on the `List`, `Set`, and `Map` interfaces let you easily create unmodifiable lists, sets, and maps.

A collection is considered *unmodifiable* if elements cannot be added, removed, or replaced. After you create an unmodifiable instance of a collection, it holds the same data as long as a reference to it exists.

A collection that is modifiable must maintain bookkeeping data to support future modifications. This adds overhead to the data that is stored in the modifiable collection. A collection that is unmodifiable does not need this extra bookkeeping data. Because the collection never needs to be modified, the data contained in the collection can be packed much more densely. Unmodifiable collection instances generally consume much less memory than modifiable collection instances that contain the same data.

Topics

Use Cases

Whether to use an unmodifiable collection or a modifiable collection depends on the data in the collection.

An unmodifiable collection provides space efficiency benefits and prevents the collection from accidentally being modified, which might cause the program to work incorrectly. An unmodifiable collection is recommended for the following cases:

- Collections that are initialized from constants that are known when the program is written
- Collections that are initialized at the beginning of a program from data that is computed or is read from something such as a configuration file

For a collection that holds data that is modified throughout the course of the program, a modifiable collection is the best choice. Modifications are performed in-place, so that incremental additions or deletions of data elements are quite inexpensive. If this were

done with an unmodifiable collection, a complete copy would have to be made to add or remove a single element, which usually has unacceptable overhead.

Syntax

The API for these new collections is simple, especially for small numbers of elements.

Topics

Unmodifiable List Static Factory Methods

The `List.of` static factory methods provide a convenient way to create unmodifiable lists.

A list is an ordered collection, where duplicate elements are allowed. Null values are not allowed.

The syntax of these methods is:

```
List.of()
List.of(e1)
List.of(e1, e2)         // fixed-argument form overloads up to 10 elements
List.of(elements...)   // varargs form supports an arbitrary number of
elements or an array
```

Example 5-1 Examples

In JDK 8:

```
List<String> stringList = Arrays.asList("a", "b", "c");
stringList = Collections.unmodifiableList(stringList);
```

In JDK 9 and later:

```
List<String> stringList = List.of("a", "b", "c");
```

See Unmodifiable Lists.

Unmodifiable Set Static Factory Methods

The `Set.of` static factory methods provide a convenient way to create unmodifiable sets.

A set is a collection that does not contain duplicate elements. If a duplicate entry is detected, then an `IllegalArgumentException` is thrown. Null values are not allowed.

The syntax of these methods is:

```
Set.of()
Set.of(e1)
Set.of(e1, e2)         // fixed-argument form overloads up to 10 elements
Set.of(elements...)   // varargs form supports an arbitrary number of
elements or an array
```

Example 5-2 Examples

In JDK 8:

```
Set<String> stringSet = new HashSet<>(Arrays.asList("a", "b", "c"));
stringSet = Collections.unmodifiableSet(stringSet);
```

In JDK 9 and later:

```
Set<String> stringSet = Set.of("a", "b", "c");
```

See Unmodifiable Sets.

Unmodifiable Map Static Factory Methods

The `Map.of` and `Map.ofEntries` static factory methods provide a convenient way to create unmodifiable maps.

A `Map` cannot contain duplicate keys. If a duplicate key is detected, then an `IllegalArgumentException` is thrown. Each key is associated with one value. Null cannot be used for either `Map` keys or values.

The syntax of these methods is:

```
Map.of()
Map.of(k1, v1)
Map.of(k1, v1, k2, v2)    // fixed-argument form overloads up to 10 key-
value pairs
Map.ofEntries(entry(k1, v1), entry(k2, v2),...)
 // varargs form supports an arbitrary number of Entry objects or an array
```

Example 5-3 Examples

In JDK 8:

```
Map<String, Integer> stringMap = new HashMap<String, Integer>();
stringMap.put("a", 1);
stringMap.put("b", 2);
stringMap.put("c", 3);
stringMap = Collections.unmodifiableMap(stringMap);
```

In JDK 9 and later:

```
Map<String, Integer> stringMap = Map.of("a", 1, "b", 2, "c", 3);
```

Example 5-4 Map with Arbitrary Number of Pairs

If you have more than 10 key-value pairs, then create the map entries using the `Map.entry` method, and pass those objects to the `Map.ofEntries` method. For example:

```
import static java.util.Map.entry;
Map <Integer, String> friendMap = Map.ofEntries(
   entry(1, "Tom"),
   entry(2, "Dick"),
   entry(3, "Harry"),
   ...
   entry(99, "Mathilde"));
```

See Unmodifiable Maps.

Creating Unmodifiable Copies of Collections

Let's consider the case where you create a collection by adding elements and modifying it, and then at some point, you want an unmodifiable snapshot of that collection. Create the copy using the `copyOf` family of methods.

For example, suppose you have some code that gathers elements from several places:

```
List<Item> list = new ArrayList<>();
list.addAll(getItemsFromSomewhere());
list.addAll(getItemsFromElsewhere());
list.addAll(getItemsFromYetAnotherPlace());
```

It's inconvenient to create an unmodifiable collection using the `List.of` method. Doing this would require creating an array of the right size, copying elements from the list into the array, and then calling `List.of(array)` to create the unmodifiable snapshot. Instead, do it in one step using the `copyOf` static factory method:

```
List<Item> snapshot = List.copyOf(list);
```

There are corresponding static factory methods for `Set` and `Map` called `Set.copyOf` and `Map.copyOf`. Because the parameter of `List.copyOf` and `Set.copyOf` is `Collection`, you can create an unmodifiable `List` that contains the elements of a `Set` and an unmodifiable `Set` that contains the elements of a `List`. If you use `Set.copyOf` to create a `Set` from a `List`, and the `List` contains duplicate elements, an exception is not thrown. Instead, an arbitrary one of the duplicate elements is included in the resulting `Set`.

If the collection you want to copy is modifiable, then the `copyOf` method creates an unmodifiable collection that is a copy of the original. That is, the result contains all the same elements as the original. If elements are added to or removed from the original collection, that won't affect the copy.

If the original collection is *already* unmodifiable, then the `copyOf` method simply returns a reference to the original collection. The point of making a copy is to isolate

the returned collection from changes to the original one. But if the original collection cannot be changed, there is no need to make a copy of it.

In both of these cases, if the elements are mutable, and an element is modified, that change causes both the original collection and the copy to appear to have changed.

Creating Unmodifiable Collections from Streams

The Streams library includes a set of terminal operations known as `Collectors`. A `Collector` is most often used to create a new collection that contains the elements of the stream. The `java.util.stream.Collectors` class has `Collectors` that create new unmodifiable collections from the elements of the streams.

If you want to guarantee that the returned collection is unmodifiable, you should use one of the `toUnmodifiable-` collectors. These collectors are:

```
Collectors.toUnmodifiableList()
Collectors.toUnmodifiableSet()
Collectors.toUnmodifiableMap(keyMapper, valueMapper)
Collectors.toUnmodifiableMap(keyMapper, valueMapper, mergeFunction)
```

For example, to transform the elements of a source collection and place the results into an unmodifiable set, you can do the following:

```
Set<Item> unmodifiableSet =
   sourceCollection.stream()
                   .map(...)
                   .collect(Collectors.toUnmodifiableSet());
```

If the stream contains duplicate elements, the `toUnmodifiableSet` collector chooses an arbitrary one of the duplicates to include in the resulting `Set`. For the `toUnmodifiableMap(keyMapper, valueMapper)` collector, if the `keyMapper` function produces duplicate keys, an `IllegalStateException` is thrown. If duplicate keys are a possibility, use the `toUnmodifiableMap(keyMapper, valueMapper, mergeFunction)` collector instead. If duplicate keys occur, the `mergeFunction` is called to merge the values of each duplicate key into a single value.

The `toUnmodifiable-` collectors are conceptually similar to their counterparts `toList`, `toSet`, and the corresponding two `toMap` methods, but they have different characteristics. Specifically, the `toList`, `toSet`, and `toMap` methods make no guarantee about the modifiablilty of the returned collection, however, the `toUnmodifiable-` collectors guarantee that the result is unmodifiable.

Randomized Iteration Order

Iteration order for `Set` elements and `Map` keys is randomized and likely to be different from one JVM run to the next. This is intentional and makes it easier to identify code that depends on iteration order. Inadvertent dependencies on iteration order can cause problems that are difficult to debug.

The following example shows how the iteration order is different after `jshell` is restarted.

```
jshell> var stringMap = Map.of("a", 1, "b", 2, "c", 3);
stringMap ==> {b=2, c=3, a=1}

jshell> /exit
|  Goodbye

C:\Program Files\Java\jdk\bin>jshell

jshell> var stringMap = Map.of("a", 1, "b", 2, "c", 3);
stringMap ==> {a=1, b=2, c=3}
```

Randomized iteration order applies to the collection instances created by the `Set.of`, `Map.of`, and `Map.ofEntries` methods and the `toUnmodifiableSet` and `toUnmodifiableMap` collectors. The iteration ordering of collection implementations such as `HashMap` and `HashSet` is unchanged.

About Unmodifiable Collections

The collections returned by the convenience factory methods added in JDK 9 are unmodifiable. Any attempt to add, set, or remove elements from these collections causes an `UnsupportedOperationException` to be thrown.

However, if the contained elements are mutable, then this may cause the collection to behave inconsistently or make its contents to appear to change.

Let's look at an example where an unmodifiable collection contains mutable elements. Using `jshell`, create two lists of `String` objects using the `ArrayList` class, where the second list is a copy of the first. Trivial `jshell` output was removed.

```
jshell> List<String> list1 = new ArrayList<>();
jshell> list1.add("a")
jshell> list1.add("b")
jshell> list1
list1 ==> [a, b]

jshell> List<String> list2 = new ArrayList<>(list1);
list2 ==> [a, b]
```

Next, using the `List.of` method, create `unmodlist1` and `unmodlist2` that point to the first lists. If you try to modify `unmodlist1`, then you see an exception error because `unmodlist1` is unmodifiable. Any modification attempt throws an exception.

```
jshell> List<List<String>> unmodlist1 = List.of(list1, list1);
unmodlist1 ==> [[a, b], [a, b]]

jshell> List<List<String>> unmodlist2 = List.of(list2, list2);
unmodlist2 ==> [[a, b], [a, b]]

jshell> unmodlist1.add(new ArrayList<String>())
|  java.lang.UnsupportedOperationException thrown:
```

```
|        at ImmutableCollections.uoe (ImmutableCollections.java:71)
|        at ImmutableCollections$AbstractImmutableList.add
(ImmutableCollections
.java:75)
|        at (#8:1)
```

But if you modify the original `list1`, the contents of `unmodlist1` changes, even though `unmodlist1` is unmodifiable.

```
jshell> list1.add("c")
jshell> list1
list1 ==> [a, b, c]
jshell> unmodlist1
ilist1 ==> [[a, b, c], [a, b, c]]

jshell> unmodlist2
ilist2 ==> [[a, b], [a, b]]

jshell> unmodlist1.equals(unmodlist2)
$14 ==> false
```

Unmodifiable Collections vs. Unmodifiable Views

The unmodifiable collections behave in the same way as the unmodifiable views returned by the `Collections.unmodifiable...` methods. However, the unmodifiable collections are not views — these are data structures implemented by classes where any attempt to modify the data causes an exception to be thrown.

If you create a `List` and pass it to the `Collections.unmodifiableList` method, then you get an unmodifiable view. The underlying list is still modifiable, and modifications to it are visible through the `List` that is returned, so it is not actually immutable.

To demonstrate this behavior, create a `List` and pass it to `Collections.unmodifiableList`. If you try to add to that `List` directly, then an exception is thrown.

```
jshell> List<String> list1 = new ArrayList<>();
jshell> list1.add("a")
jshell> list1.add("b")
jshell> list1
list1 ==> [a, b]

jshell> List<String> unmodlist1 = Collections.unmodifiableList(list1);
unmodlist1 ==> [a, b]

jshell> unmodlist1.add("c")
|  Exception java.lang.UnsupportedOperationException
|        at Collections$UnmodifiableCollection.add (Collections.java:1058)
|        at (#8:1)
```

Note that `unmodlist1` is a view of `list1`. You cannot change the view directly, but you can change the original list, which changes the view. If you change the original `list1`, no error is generated, and the `unmodlist1` list has been modified.

```
jshell> list1.add("c")
$19 ==> true
jshell> list1
list1 ==> [a, b, c]

jshell> unmodlist1
unmodlist1 ==> [a, b, c]
```

The reason for an unmodifiable view is that the collection cannot be modified by calling methods on the view. However, anyone with a reference to the underlying collection, and the ability to modify it, can cause the unmodifiable view to change.

Space Efficiency

The collections returned by the convenience factory methods are more space efficient than their modifiable equivalents.

All of the implementations of these collections are private classes hidden behind a static factory method. When it is called, the static factory method chooses the implementation class based on the size. The data may be stored in a compact field-based or array-based layout.

Let's look at the heap space consumed by two alternative implementations. First, here's an unmodifiable `HashSet` that contains two strings:

```
Set<String> set = new HashSet<>(3);   // 3 buckets
set.add("silly");
set.add("string");
set = Collections.unmodifiableSet(set);
```

The set includes six objects: the unmodifiable wrapper; the `HashSet`, which contains a `HashMap`; the table of buckets (an array); and two `Node` instances (one for each element). On a typical VM, with a 12–byte header per object, the total overhead comes to 96 bytes + 28 * 2 = 152 bytes for the set. This is a large amount of overhead compared to the amount of data stored. Plus, access to the data unavoidably requires multiple method calls and pointer dereferences.

Instead, we can implement the set using `Set.of`:

```
Set<String> set = Set.of("silly", "string");
```

Because this is a field-based implementation, the set contains one object and two fields. The overhead is 20 bytes. The new collections consume less heap space, both in terms of fixed overhead and on a per-element basis.

Not needing to support mutation also contributes to space savings. In addition, the locality of reference is improved, because there are fewer objects required to hold the data.

Thread Safety

If multiple threads share a modifiable data structure, steps must be taken to ensure that modifications made by one thread do not cause unexpected side effects for other threads. However, because an immutable object cannot be changed, it is considered thread safe without requiring any additional effort.

When several parts of a program share data structures, a modification to a structure made by one part of the program is visible to the other parts. If the other parts of the program aren't prepared for changes to the data, then bugs, crashes, or other unexpected behavior could occur. However, if different parts of a program share an immutable data structure, such unexpected behavior can never happen, because the shared structure cannot be changed.

Similarly, when multiple threads share a data structure, each thread must take precautions when modifying that data structure. Typically, threads must hold a lock while reading from or writing to any shared data structure. Failing to lock properly can lead to race conditions or inconsistencies in the data structure, which can result in bugs, crashes, or other unexpected behavior. However, if multiple threads share an immutable data structure, these problems cannot occur, even in the absence of locking. Therefore, an immutable data structure is said to be thread safe without requiring any additional effort such as adding locking code.

A collection is considered unmodifiable if elements cannot be added, removed, or replaced. However, an unmodifiable collection is only immutable if the elements contained in the collection are immutable. To be considered thread safe, collections created using the static factory methods and `toUnmodifiable-` collectors must contain only immutable elements.

6
Process API

The Process API lets you start, retrieve information about, and manage native operating system processes.

With this API, you can work with operating system processes as follows:

- Run arbitrary commands:
  - Filter running processes.
  - Redirect output.
  - Connect heterogeneous commands and shells by scheduling processes to start when another ends.
- Test the execution of commands:
  - Run a series of tests.
  - Log output.
  - Cleanup leftover processes.
- Monitor commands:
  - Monitor long-running processes and restart them if they terminate
  - Collect usage statistics

Topics

Process API Classes and Interfaces

The Process API consists of the classes and interfaces `ProcessBuilder`, `Process`, `ProcessHandle`, and `ProcessHandle.Info`.

Topics

- ProcessHandle.Info Interface

ProcessBuilder Class

The `ProcessBuilder` class lets you create and start operating system processes.

See Creating a Process for examples on how to create and start a process. The `ProcessBuilder` class manages various process attributes, which the following table summarizes:

Table 6-1 ProcessBuilder Class Attributes and Related Methods

Process Attribute	Description	Related Methods
Command	Strings that specify the external program file to call and its arguments, if any.	• `ProcessBuilder` constructor • `command(String... command)`
Environment	The environment variables (and their values). This is initially a copy of the system environment of the current process.	• `environment()`
Working directory	By default, the current working directory of the current process.	• `directory()` • `directory(File directory)`
Standard input source	By default, a process reads standard input from a pipe; access this through the output stream returned by the `Process.getOutputStream` method.	• `redirectInput (ProcessBuilder.Redirect source)`
Standard output and standard error destinations	By default, a process writes standard output and standard error to pipes; access these through the input streams returned by the `Process.getInputStream` and `Process.getErrorStream` methods. See Redirecting Output from a Process for an example.	• `redirectOutput(ProcessBuilder.Redirect destination)` • `redirectError(ProcessBuilder.Redirect destination)`
`redirectErrorStream` property	Specifies whether to send standard output and error output as two separate streams (with a value of false) or merge any error output with standard output (with a value of true).	• `redirectErrorStream()` • `redirectErrorStream(boolean redirectErrorStream)`

Process Class

The methods in the `Process` class let you to control processes started by the methods `ProcessBuilder.start` and `Runtime.exec`. The following table summarizes these methods:

The following table summarizes the methods of the `Process` class.

Table 6-2 Process Class Methods

Method Type	Related Methods
Wait for the process to complete.	• `waitfor()` • `waitFor(long timeout, TimeUnit unit)`
Retrieve information about the process.	• `isAlive()` • `pid()` • `info()` • `exitValue()`
Retrieve input, output, and error streams. See Handling Processes When They Terminate with the onExit Method for an example.	• `getInputStream()` • `getOutputStream()` • `getErrorStream()`
Retrieve direct and indirect child processes.	• `children()` • `descendants()`
Destroy or terminate the process.	• `destroy()` • `destroyForcibly()` • `supportsNormalTermination()`
Return a `CompletableFuture` instance that will be completed when the process exits. See Handling Processes When They Terminate with the onExit Method for an example.	• `onExit()`

ProcessHandle Interface

The `ProcessHandle` interface lets you identify and control native processes. The `Process` class is different from `ProcessHandle` because it lets you control processes started only by the methods `ProcessBuilder.start` and `Runtime.exec`; however, the `Process` class lets you access process input, output, and error streams.

See Filtering Processes with Streams for an example of the `ProcessHandle` interface. The following table summarizes the methods of this interface:

Table 6-3 ProcessHandle Interface Methods

Method Type	Related Methods
Retrieve all operating system processes.	• `allProcesses()`
Retrieve process handles.	• `current()` • `of(long pid)` • `parent()`
Retrieve information about the process.	• `isAlive()` • `pid()` • `info()`
Retrieve streams of direct and indirect child processes.	• `children()` • `descendants()`

Table 6-3 (Cont.) ProcessHandle Interface Methods

Method Type	Related Methods
Destroy processes.	• `destroy()` • `destroyForcibly()`
Return a `CompletableFuture` instance that will be completed when the process exits. See Handling Processes When They Terminate with the onExit Method for an example.	• `onExit()`

ProcessHandle.Info Interface

The `ProcessHandle.Info` interface lets you retrieve information about a process, including processes created by the `ProcessBuilder.start` method and native processes.

See Getting Information About a Process for an example of the `ProcessHandle.Info` interface. The following table summarizes the methods in this interface:

Table 6-4 ProcessHandle.Info Interface Methods

Method	Description
`arguments()`	Returns the arguments of the process as a `String` array.
`command()`	Returns the executable path name of the process.
`commandLine()`	Returns the command line of the process.
`startInstant()`	Returns the start time of the process.
`totalCpuDuration()`	Returns the total CPU time accumulated of the process.
`user()`	Returns the user of the process.

Creating a Process

To create a process, first specify the attributes of the process, such as the command name and its arguments, with the `ProcessBuilder` class. Then, start the process with the `ProcessBuilder.start` method, which returns a `Process` instance.

The following lines create and start a process:

```
ProcessBuilder pb = new ProcessBuilder("echo", "Hello World!");
Process p = pb.start();
```

In the following excerpt, the `setEnvTest` method sets two environment variables, `horse` and `oats`, then prints the value of these environment variables (as well as the system environment variable `HOME`) with the `echo` command:

```
  public static void setEnvTest() throws IOException, InterruptedException
{
    ProcessBuilder pb =
      new ProcessBuilder("/bin/sh", "-c",
"echo $horse $dog $HOME").inheritIO();
    pb.environment().put("horse", "oats");
    pb.environment().put("dog", "treats");
    pb.start().waitFor();
  }
```

This method prints the following (assuming that your home directory is `/home/admin`):

```
oats treats /home/admin
```

Getting Information About a Process

The method `Process.pid` returns the native process ID of the process. The method `Process.info` returns a `ProcessHandle.Info` instance, which contains additional information about the process, such as its executable path name, start time, and user.

In the following excerpt, the method `getInfoTest` starts a process and then prints information about it:

```
  public static void getInfoTest() throws IOException {
    ProcessBuilder pb = new ProcessBuilder("echo", "Hello World!");
    String na = "<not available>";
    Process p = pb.start();
    ProcessHandle.Info info = p.info();
    System.out.printf("Process ID: %s%n", p.pid());
    System.out.printf("Command name: %s%n", info.command().orElse(na));
    System.out.printf("Command line: %s%n", info.commandLine().orElse(na));

    System.out.printf("Start time: %s%n",
      info.startInstant().map(i -> i.atZone(ZoneId.systemDefault())
                                    .toLocalDateTime().toString())
                         .orElse(na));

    System.out.printf("Arguments: %s%n",
      info.arguments().map(a -> Stream.of(a)
                                      .collect(Collectors.joining(" ")))
                      .orElse(na));

    System.out.printf("User: %s%n", info.user().orElse(na));
  }
```

This method prints output similar to the following:

```
Process ID: 18761
Command name: /usr/bin/echo
Command line: echo Hello World!
Start time: 2017-05-30T18:52:15.577
Arguments: <not available>
User: administrator
```

> **Note:**
>
> - The attributes of a process vary by operating system and are not available in all implementations. In addition, information about processes is limited by the operating system privileges of the process making the request.
> - All the methods in the interface `ProcessHandle.Info` return instances of `Optional<T>`; always check if the returned value is empty.

Redirecting Output from a Process

By default, a process writes standard output and standard error to pipes. In your application, you can access these pipes through the input streams returned by the methods `Process.getOutputStream` and `Process.getErrorStream`. However, before starting the process, you can redirect standard output and standard error to other destinations, such as a file, with the methods `redirectOutput` and `redirectError`.

In the following excerpt, the method `redirectToFileTest` redirects standard input to a file, `out.tmp`, then prints this file:

```
  public static void redirectToFileTest() throws IOException,
InterruptedException {
    File outFile = new File("out.tmp");
    Process p = new ProcessBuilder("ls", "-la")
      .redirectOutput(outFile)
      .redirectError(Redirect.INHERIT)
      .start();
    int status = p.waitFor();
    if (status == 0) {
      p = new ProcessBuilder("cat" , outFile.toString())
        .inheritIO()
        .start();
      p.waitFor();
    }
  }
```

The excerpt redirects standard output to the file `out.tmp`. It redirects standard error to the standard error of the invoking process; the value `Redirect.INHERIT` specifies that the subprocess I/O source or destination is the same as that of the current

process. The call to the `inheritIO()` method is equivalent to `redirectInput(Redirect.INHERIT).redirectOuput(Redirect.INHERIT).redirectError(Redirect.INHERIT)`.

Filtering Processes with Streams

The method `ProcessHandle.allProcesses` returns a stream of all processes visible to the current process. You can filter the `ProcessHandle` instances of this stream the same way that you filter elements from a collection.

In the following excerpt, the method `filterProcessesTest` prints information about all the processes owned by the current user, sorted by the process ID of their parent's process:

```
public class ProcessTest {

  // ...

  static void filterProcessesTest() {
    Optional<String> currUser = ProcessHandle.current().info().user();
    ProcessHandle.allProcesses()
        .filter(p1 -> p1.info().user().equals(currUser))
        .sorted(ProcessTest::parentComparator)
        .forEach(ProcessTest::showProcess);
  }

  static int parentComparator(ProcessHandle p1, ProcessHandle p2) {
    long pid1 = p1.parent().map(ph -> ph.pid()).orElse(-1L);
    long pid2 = p2.parent().map(ph -> ph.pid()).orElse(-1L);
    return Long.compare(pid1, pid2);
  }

  static void showProcess(ProcessHandle ph) {
    ProcessHandle.Info info = ph.info();
    System.out.printf("pid: %d, user: %s, cmd: %s%n",
      ph.pid(), info.user().orElse("none"), info.command().orElse("none"));
  }

  // ...
}
```

Note that the `allProcesses` method is limited by native operating system access controls. Also, because all processes are created and terminated asynchronously, there is no guarantee that a process in the stream is alive or that no other processes may have been created since the call to the `allProcesses` method.

Handling Processes When They Terminate with the onExit Method

The `Process.onExit` and `ProcessHandle.onExit` methods return a `CompletableFuture` instance, which you can use to schedule tasks when a process

terminates. Alternatively, if you want your application to wait for a process to terminate, then you can call `onExit().get()`.

In the following excerpt, the method `startProcessesTest` creates three processes and then starts them. Afterward, it calls `onExit().thenAccept(onExitMethod)` on each of the processes; `onExitMethod` prints the process ID (PID), exit status, and output of the process.

```
public class ProcessTest {

  // ...

  static public void startProcessesTest() throws IOException,
InterruptedException {
    List<ProcessBuilder> greps = new ArrayList<>();
    greps.add(new ProcessBuilder("/bin/sh", "-c", "grep -c \"java\" *"));
    greps.add(new ProcessBuilder("/bin/sh", "-c", "grep -c \"Process\"
*"));
    greps.add(new ProcessBuilder("/bin/sh", "-c", "grep -c \"onExit\" *"));
    ProcessTest.startSeveralProcesses (greps,
ProcessTest::printGrepResults);
    System.out.println("\nPress enter to continue ...\n");
    System.in.read();
  }

  static void startSeveralProcesses (
    List<ProcessBuilder> pBList,
    Consumer<Process> onExitMethod)
    throws InterruptedException {
    System.out.println("Number of processes: " + pBList.size());
    pBList.stream().forEach(
      pb -> {
        try {
          Process p = pb.start();
          System.out.printf("Start %d, %s%n",
            p.pid(), p.info().commandLine().orElse("<na>"));
          p.onExit().thenAccept(onExitMethod);
        } catch (IOException e) {
          System.err.println("Exception caught");
          e.printStackTrace();
        }
      }
    );
  }

  static void printGrepResults(Process p) {
    System.out.printf("Exit %d, status %d%n%s%n%n",
      p.pid(), p.exitValue(), output(p.getInputStream()));
  }

  private static String output(InputStream inputStream) {
    String s = "";
    try (BufferedReader br = new BufferedReader(new
InputStreamReader(inputStream))) {
      s =
```

```
br.lines().collect(Collectors.joining(System.getProperty("line.separator"))
);
    } catch (IOException e) {
      System.err.println("Caught IOException");
      e.printStackTrace();
    }
    return s;
  }

  // ...
}
```

The output of the method `startProcessesTest` is similar to the following. Note that the processes might exit in a different order than the order in which they were started.

```
Number of processes: 3
Start 12401, /bin/sh -c grep -c "java" *
Start 12403, /bin/sh -c grep -c "Process" *
Start 12404, /bin/sh -c grep -c "onExit" *

Press enter to continue ...

Exit 12401, status 0
ProcessTest.class:0
ProcessTest.java:16

Exit 12404, status 0
ProcessTest.class:0
ProcessTest.java:8

Exit 12403, status 0
ProcessTest.class:0
ProcessTest.java:38
```

This method calls the `System.in.read()` method to prevent the program from terminating before all the processes have exited (and have run the method specified by the `thenAccept` method).

If you want to wait for a process to terminate before proceeding with the rest of the program, then call `onExit().get()`:

```
  static void startSeveralProcesses (
    List<ProcessBuilder> pBList, Consumer<Process> onExitMethod)
    throws InterruptedException {
    System.out.println("Number of processes: " + pBList.size());
    pBList.stream().forEach(
      pb -> {
        try {
          Process p = pb.start();
          System.out.printf("Start %d, %s%n",
            p.pid(), p.info().commandLine().orElse("<na>"));
          p.onExit().get();
          printGrepResults(p);
        } catch (IOException|InterruptedException|ExecutionException e ) {
```

```
            System.err.println("Exception caught");
            e.printStackTrace();
          }
        }
      );
    }
```

The `ComputableFuture` class contains a variety of methods that you can call to schedule tasks when a process exits including the following:

- `thenApply`: Similar to `thenAccept`, except that it takes a lambda expression of type `Function` (a lambda expression that returns a value).
- `thenRun`: Takes a lambda expression of type `Runnable` (no formal parameters or return value).
- `thenApplyAsyc`: Runs the specified `Function` with a thread from `ForkJoinPool.commonPool()`.

Because `ComputableFuture` implements the `Future` interface, this class also contains synchronous methods:

- `get(long timeout, TimeUnit unit)`: Waits, if necessary, at most the time specified by its arguments for the process to complete.
- `isDone`: Returns true if the process is completed.

Controlling Access to Sensitive Process Information

Process information may contain sensitive information such as user IDs, paths, and arguments to commands. Control access to process information with a security manager.

When running as a normal application, a `ProcessHandle` has the same operating system privileges to information about other processes as a native application; however, information about system processes may not be available.

If your application uses the `SecurityManager` class to implement a security policy, then to enable it to access process information, the security policy must grant `RuntimePermission("manageProcess")`. This permission enables native process termination and access to the process `ProcessHandle` information. Note that this permission enables code to identify and terminate processes that it did not create.

7

Preferences API

The Preferences API enables applications to manage preference and configuration data.

Applications require preference and configuration data to adapt to the needs of different users and environments. The `java.util.prefs` package provides a way for applications to store and retrieve user and system preference and configuration data. The data is stored persistently in an implementation-dependent backing store. There are two separate trees of preference nodes, one for user preferences and one for system preferences.

All of the methods that modify preference data are permitted to operate asynchronously. They may return immediately, and changes will eventually propagate to the persistent backing store. The `flush` method can be used to force updates to the backing store.

The methods in the `Preferences` class may be invoked concurrently by multiple threads in a single JVM without the need for external synchronization, and the results will be equivalent to some serial execution. If this class is used concurrently by multiple JVMs that store their preference data in the same backing store, the data store will not be corrupted, but no other guarantees are made concerning the consistency of the preference data.

Topics:

Comparing the Preferences API to Other Mechanisms

Prior to the introduction of the Preferences API, developers could choose to manage preference and configuration data in an ad hoc fashion by using the Properties API or the Java Naming and Directory Interface (JNDI) API.

Often, preference and configuration data was stored in properties files, accessed through the `java.util.Properties` API. However, there are no standards as to where such files should reside on disk, or what they should be called. Using this mechanism, it is extremely difficult to back up a user's preference data, or transfer it from one machine to another. As the number of applications increases, the possibility of file name conflicts increases. Also, this mechanism is of no help on platforms that lack a local disk, or where it is desirable that the data be stored in an external data store, such as an enterprise-wide LDAP directory service.

Less frequently, developers stored user preference and configuration data in a directory service accessed through the JNDI API. Unlike the Properties API, JNDI allows the use of arbitrary data stores (back-end neutrality). While JNDI is extremely powerful, it is also rather large, consisting of 5 packages and 83 classes. JNDI

provides no policy as to where in the directory name space the preference data should be stored, or in which name space.

Neither Properties nor JNDI provide a simple, ubiquitous, back-end neutral preferences management facility. The Preferences API does provide such a facility, combining the simplicity of Properties with the back-end neutrality of JNDI. It provides sufficient built-in policy to prevent name clashes, foster consistency, and encourage robustness in the face of inaccessibility of the backing data store.

Usage Notes

The information in this section is not part of the Preferences API specification. It is intended to provide some examples of how the Preferences API might be used.

Topics:

Obtain Preferences Objects for an Enclosing Class

The examples in this section show how you can obtain the system and user Preferences objects pertaining to the enclosing class. These examples only work inside instance methods.

The following example obtains per-user preferences. Reasonable defaults are provided for each of the preference values obtained. These defaults are returned if no preference value has been set, or if the backing store is inaccessible.

Note that static final fields, rather than inline `String` literals, are used for the key names (`NUM_ROWS` and `NUM_COLS`). This reduces the likelihood of runtime bugs from typographical errors in key names.

```
package com.greencorp.widget;
import  java.util.prefs.*;

public class Gadget {
    // Preference keys for this package
    private static final String NUM_ROWS = "num_rows";
    private static final String NUM_COLS = "num_cols";

    void getPrefs() {
        Preferences prefs = Preferences.userNodeForPackage(Gadget.class);

        int numRows = prefs.getInt(NUM_ROWS, 40);
        int numCols = prefs.getInt(NUM_COLS, 80);

        ...
    }
```

The previous example obtains per-user preferences. If a single, per-system value is desired, replace the first line in `getPrefs` with the following:

```
Preferences prefs = Preferences.systemNodeForPackage(Gadget.class);
```

Obtain Preferences Objects for a Static Method

The examples in this section show how you can obtain the system and user Preferences objects in a static method.

In a static method (or static initializer), you need to explicitly provide the name of the package:

```
static String ourNodeName = "/com/greencorp/widget";
static void getPrefs() {
    Preferences prefs = Preferences.userRoot().node(ourNodeName);

    ...
}
```

It is always acceptable to obtain a system preferences object once, in a static initializer, and use it whenever system preferences are required:

```
static Preferences prefs =  Preferences.systemRoot().node(ourNodeName);
```

In general, it is acceptable to do the same thing for a user preferences object, but not if the code in question is to be used in a server, wherein multiple users are running concurrently or serially. In such a system, `userNodeForPackage` and `userRoot` return the appropriate node for the calling user, thus it's critical that calls to `userNodeForPackage` or `userRoot` be made from the appropriate thread at the appropriate time. If a piece of code may eventually be used in such a server environment, it is a good, conservative practice to obtain user preferences objects immediately before they are used, as in the prior example.

Atomic Updates

The Preferences API does not provide database-like "transactions" wherein multiple preferences are modified atomically. Occasionally, it is necessary to modify two or more preferences as a unit.

For example, suppose you are storing the x and y coordinates where a window is to be placed. The only way to achieve atomicity is to store both values in a single preference. Many encodings are possible. Here's a simple one:

```
int x, y;
...
prefs.put(POSITION, x + "," + y);
```

When such a "compound preference" is read, it must be decoded. For robustness, allowances should be made for a corrupt (unparseable) value:

```
static int X_DEFAULT = 50, Y_DEFAULT = 25;
void parsePrefs() {
    String position = prefs.get(POSITION, X_DEFAULT + "," + Y_DEFAULT);
    int x, y;
    try {
        int i = position.indexOf(',');
        x = Integer.parseInt(coordinates.substring(0, i));
        y = Integer.parseInt(position.substring(i + 1));
    } catch(Exception e) {
        // Value was corrupt, just use defaults
        x = X_DEFAULT;
        y = Y_DEFAULT;
    }
    ...
}
```

Determine Backing Store Status

Typical application code has no need to know whether the backing store is available. It should almost always be available, but if it isn't, the code should continue to execute using default values in place of preference values from the backing store.

Very rarely, some advanced program might want to vary its behavior, or simply refuse to run, if the backing store is unavailable. Following is a method that determines whether the backing store is available by attempting to modify a preference value and flush the result to the backing store.

```
private static final String BACKING_STORE_AVAIL = "BackingStoreAvail";

private static boolean backingStoreAvailable() {
    Preferences prefs = Preferences.userRoot().node("<temporary>");
    try {
        boolean oldValue = prefs.getBoolean(BACKING_STORE_AVAIL, false);
        prefs.putBoolean(BACKING_STORE_AVAIL, !oldValue);
        prefs.flush();
    } catch(BackingStoreException e) {
        return false;
    }
    return true;
}
```

Design FAQ

This section provides answers to frequently asked questions about the design of the Preferences API.

Topics:

How does this Preferences API relate to the Properties API?

It is intended to replace most common uses of Properties, rectifying many of its deficiencies, while retaining its light weight. When using Properties, the programmer must explicitly specify a path name for each properties file, but there is no standard location or naming convention. Properties files are "brittle", as they are hand-editable but easily corrupted by careless editing. Support for non-string data types in properties is non-existent. Properties cannot easily be used with a persistence mechanism other than the file system. In sum, the Properties facility does not scale.

How does the Preferences API relate to JNDI?

Like JNDI, it provides back-end neutral access to persistent key-value data. JNDI, however, is far more powerful, and correspondingly heavyweight. JNDI is appropriate for enterprise applications that need its power. The Preferences API is intended as a simple, ubiquitous, back-end neutral preferences-management facility, enabling any Java application to easily tailor its behavior to user preferences and maintain small amounts of state from run to run.

Why do all of the get methods require the caller to pass in a default?

This forces the application authors to provide reasonable default values, so that applications have a reasonable chance of running even if the repository is unavailable.

How was it decided which methods should throw BackingStoreException?

Only methods whose semantics absolutely require the ability to communicate with the backing store throw this exception. Typical applications will have no need to call these methods. As long as these methods are avoided, applications will be able to run even if the backing store is unavailable, which was an explicit design goal.

Why doesn't this API provide stronger guarantees concerning concurrent access by multiple VMs? Similarly, why doesn't the API allow multiple

Preferences updates to be combined into a single "transaction", with all or nothing semantics?

While the API does provide rudimentary persistent data storage, it is not intended as a substitute for a database. It is critical that it be possible to implement this API atop standard preference/configuration repositories, most of which do not provide database-like guarantees and functionality. Such repositories have proven adequate for the purposes for which this API is intended.

Why does this API have case-sensitive keys and node-names, while other APIs playing in a similar space (such as the Microsoft Windows Registry and LDAP) do not?

In the Java programming universe, case-sensitive String keys are ubiquitous. In particular, they are provided by the Properties class, which this API is intended to replace. It is not uncommon for people to use Properties in a fashion that demands case-sensitivity. For example, Java package names (which are case-sensitive) are sometimes used as keys. It is recognized that this design decision complicates the life of the systems programmer who implements Preferences atop a backing store with case-insensitive keys, but this is considered an acceptable price to pay, as far more programmers will use the Preferences API than will implement it.

Why doesn't this API use the Java 2 Collections Framework?

This API is designed for a very particular purpose, and is optimized for that purpose. In the absence of generic types (see JSR-14), the API would be less convenient for typical users. It would lack compile-time type safety, if forced to conform to the Map API. Also, it is not anticipated that interoperability with other Map implementations will be required (though it would be straightforward to implement an adapter class if this assumption turned out to be wrong). The Preferences API is, by design, so similar to Map that programmers familiar with the latter should have no difficulties using the former.

Why don't the put and remove methods return the old values?

It is desirable that both of these methods be executable even if the backing store is unavailable. This would not be possible if they were required to return the old value. Further, it would have negative performance impact if the API were implemented atop some common back-end data stores.

Why does the API permit, but not require, stored defaults?

This functionality is required in enterprise settings for scalable, cost-effective administration of preferences across the enterprise, but would be overkill in a self-administered single-user setting.

Why doesn't this API contain methods to read and write arbitrary serializable objects?

Serialized objects are somewhat fragile: if the version of the program that reads such a property differs from the version that wrote it, the object may not deserialize properly (or at all). It is not impossible to store serialized objects using this API, but we do not encourage it, and have not provided a convenience method.

Why is Preferences an abstract class rather than an interface?

It was decided that the ability to add new methods in an upward compatible fashion outweighed the disadvantage that Preferences cannot be used as a "mixin". That is to say, arbitrary classes cannot also be made to serve as Preferences objects. Also, this obviates the need for a separate class for the static methods. Interfaces cannot contain static methods.

Where is the default backing store?

System and user preference data is stored persistently in an implementation-dependent backing store. Typical implementations include flat files, OS-specific registries, directory servers and SQL databases. For example, on Windows systems the data is stored in the Windows registry.

On Linux systems, the system preferences are typically stored at *`java-home`*`/.systemPrefs` in a network installation, or `/etc/.java/.systemPrefs` in a local installation. If both are present, `/etc/.java/.systemPrefs` takes precedence. The system preferences location can be overridden by setting the system property `java.util.prefs.systemRoot`. The user preferences are typically stored at *`user-home`*`/.java/.userPrefs`. The user preferences location can be overridden by setting the system property `java.util.prefs.userRoot`.

8

Java Logging Overview

The Java Logging APIs, contained in the package `java.util.logging`, facilitate software servicing and maintenance at customer sites by producing log reports suitable for analysis by end users, system administrators, field service engineers, and software development teams. The Logging APIs capture information such as security failures, configuration errors, performance bottlenecks, and/or bugs in the application or platform.

The core package includes support for delivering plain text or XML-formatted log records to memory, output streams, consoles, files, and sockets. In addition, the logging APIs are capable of interacting with logging services that already exist on the host operating system.

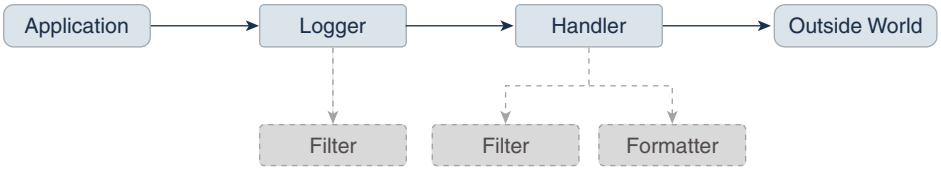
Topics

Overview of Control Flow

Applications make logging calls on `Logger` objects. `Logger` objects are organized in a hierarchical namespace and child `Logger` objects may inherit some logging properties from their parents in the namespace.

Applications make logging calls on `Logger` objects. These `Logger` objects allocate `LogRecord` objects which are passed to `Handler` objects for publication. Both `Logger` and `Handler` objects may use logging `Level` objects and (optionally) `Filter` objects to decide if they are interested in a particular `LogRecord` object. When it is necessary to publish a `LogRecord` object externally, a `Handler` object can (optionally) use a `Formatter` object to localize and format the message before publishing it to an I/O stream.

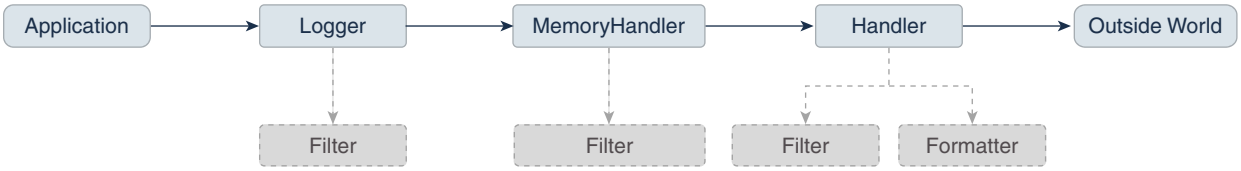
Figure 8-1 Java Logging Control Flow



Each `Logger` object keeps track of a set of output `Handler` objects. By default all `Logger` objects also send their output to their parent `Logger`. But `Logger` objects may also be configured to ignore `Handler` objects higher up the tree.

Some `Handler` objects may direct output to other `Handler` objects. For example, the `MemoryHandler` maintains an internal ring buffer of `LogRecord` objects, and on trigger events, it publishes its `LogRecord` object through a target `Handler`. In such cases, any formatting is done by the last `Handler` in the chain.

Figure 8-2 Java Logging Control Flow with MemoryHandler



The APIs are structured so that calls on the Logger APIs can be cheap when logging is disabled. If logging is disabled for a given log level, then the Logger can make a cheap comparison test and return. If logging is enabled for a given log level, the Logger is still careful to minimize costs before passing the `LogRecord` into the `Handler`. In particular, localization and formatting (which are relatively expensive) are deferred until the `Handler` requests them. For example, a `MemoryHandler` can maintain a circular buffer of `LogRecord` objects without having to pay formatting costs.

Log Levels

Each log message has an associated log `Level` object. The `Level` gives a rough guide to the importance and urgency of a log message. Log `Level` objects encapsulate an integer value, with higher values indicating higher priorities.

The `Level` class defines seven standard log levels, ranging from `FINEST` (the lowest priority, with the lowest value) to `SEVERE` (the highest priority, with the highest value).

Loggers

As stated earlier, client code sends log requests to `Logger` objects. Each logger keeps track of a log level that it is interested in, and discards log requests that are below this level.

`Logger` objects are normally named entities, using dot-separated names such as `java.awt`. The namespace is hierarchical and is managed by the `LogManager`. The namespace should typically be aligned with the Java packaging namespace, but is not required to follow it exactly. For example, a Logger called `java.awt` might handle logging requests for classes in the `java.awt` package, but it might also handle logging for classes in `sun.awt` that support the client-visible abstractions defined in the `java.awt` package.

In addition to named `Logger` objects, it is also possible to create anonymous `Logger` objects that don't appear in the shared namespace. See the Security section.

Loggers keep track of their parent loggers in the logging namespace. A logger's parent is its nearest extant ancestor in the logging namespace. The root logger (named "") has no parent. Anonymous loggers are all given the root logger as their parent. Loggers may inherit various attributes from their parents in the logger namespace. In particular, a logger may inherit:

- **Logging level**: If a logger's level is set to be null, then the logger will use an effective `Level` that will be obtained by walking up the parent tree and using the first non-null `Level`.
- **Handlers**: By default, a `Logger` will log any output messages to its parent's handlers, and so on recursively up the tree.
- **Resource bundle names**: If a logger has a null resource bundle name, then it will inherit any resource bundle name defined for its parent, and so on recursively up the tree.

Logging Methods

The `Logger` class provides a large set of convenience methods for generating log messages. For convenience, there are methods for each logging level, named after the logging level name. Thus rather than calling `logger.log(Level.WARNING, ...)`, a developer can simply call the convenience method `logger.warning(...)`.

There are two different styles of logging methods, to meet the needs of different communities of users.

First, there are methods that take an explicit source class name and source method name. These methods are intended for developers who want to be able to quickly locate the source of any given logging message. An example of this style is:

```
void warning(String sourceClass, String sourceMethod, String msg);
```

Second, there are a set of methods that do not take explicit source class or source method names. These are intended for developers who want easy-to-use logging and do not require detailed source information.

```
void warning(String msg);
```

For this second set of methods, the Logging framework will make a "best effort" to determine which class and method called into the logging framework and will add this information into the `LogRecord`. However, it is important to realize that this automatically inferred information may only be approximate. Virtual machines perform extensive optimizations when just-in-time compiling and may entirely remove stack frames, making it impossible to reliably locate the calling class and method.

Handlers

Java SE provides the following `Handler` classes:

- `StreamHandler`: A simple handler for writing formatted records to an `OutputStream`.
- `ConsoleHandler`: A simple handler for writing formatted records to `System.err`
- `FileHandler`: A handler that writes formatted log records either to a single file, or to a set of rotating log files.
- `SocketHandler`: A handler that writes formatted log records to remote TCP ports.
- `MemoryHandler`: A handler that buffers log records in memory.

It is fairly straightforward to develop new `Handler` classes. Developers requiring specific functionality can either develop a handler from scratch or subclass one of the provided handlers.

Formatters

Java SE also includes two standard `Formatter` classes:

- `SimpleFormatter`: Writes brief "human-readable" summaries of log records.
- `XMLFormatter`: Writes detailed XML-structured information.

As with handlers, it is fairly straightforward to develop new formatters.

The LogManager

There is a global `LogManager` object that keeps track of global logging information. This includes:

- A hierarchical namespace of named Loggers.
- A set of logging control properties read from the configuration file. See the section Configuration File.

There is a single `LogManager` object that can be retrieved using the static `LogManager.getLogManager` method. This is created during `LogManager` initialization, based on a system property. This property allows container applications (such as EJB containers) to substitute their own subclass of `LogManager` in place of the default class.

Configuration File

The logging configuration can be initialized using a logging configuration file that will be read at startup. This logging configuration file is in standard `java.util.Properties` format.

Alternatively, the logging configuration can be initialized by specifying a class that can be used for reading initialization properties. This mechanism allows configuration data to be read from arbitrary sources, such as LDAP and JDBC.

There is a small set of global configuration information. This is specified in the description of the `LogManager` class and includes a list of root-level handlers to install during startup.

The initial configuration may specify levels for particular loggers. These levels are applied to the named logger and any loggers below it in the naming hierarchy. The levels are applied in the order they are defined in the configuration file.

The initial configuration may contain arbitrary properties for use by handlers or by subsystems doing logging. By convention, these properties should use names starting with the name of the handler class or the name of the main `Logger` for the subsystem.

For example, the `MemoryHandler` uses a property `java.util.logging.MemoryHandler.size` to determine the default size for its ring buffer.

Default Configuration

The default logging configuration that ships with the JDK is only a default and can be overridden by ISVs, system administrators, and end users. This file is located at *`java-home`*`/conf/logging.properties`.

The default configuration makes only limited use of disk space. It doesn't flood the user with information, but does make sure to always capture key failure information.

The default configuration establishes a single handler on the root logger for sending output to the console.

Dynamic Configuration Updates

Programmers can update the logging configuration at run time in a variety of ways:

- `FileHandler`, `MemoryHandler`, and `ConsoleHandler` objects can all be created with various attributes.
- New `Handler` objects can be added and old ones removed.
- New `Logger` object can be created and can be supplied with specific Handlers.
- `Level` objects can be set on target `Handler` objects.

Native Methods

There are no native APIs for logging.

Native code that wishes to use the Java Logging mechanisms should make normal JNI calls into the Java Logging APIs.

XML DTD

The XML DTD used by the `XMLFormatter` is specified in Appendix A: DTD for XMLFormatter Output.

The DTD is designed with a `<log>` element as the top-level document. Individual log records are then written as `<record>` elements.

Note that in the event of JVM crashes it may not be possible to cleanly terminate an `XMLFormatter` stream with the appropriate closing `</log>`. Therefore, tools that are analyzing log records should be prepared to cope with un-terminated streams.

Unique Message IDs

The Java Logging APIs do not provide any direct support for unique message IDs. Those applications or subsystems requiring unique message IDs should define their own conventions and include the unique IDs in the message strings as appropriate.

Security

The principal security requirement is that untrusted code should not be able to change the logging configuration. Specifically, if the logging configuration has been set up to log a particular category of information to a particular Handler, then untrusted code should not be able to prevent or disrupt that logging.

The security permission `LoggingPermission` controls updates to the logging configuration.

Trusted applications are given the appropriate `LoggingPermission` so they can call any of the logging configuration APIs. Untrusted applets are a different story. Untrusted applets can create and use named loggers in the normal way, but they are not allowed to change logging control settings, such as adding or removing handlers, or changing log levels. However, untrusted applets are able to create and use their own "anonymous" loggers, using `Logger.getAnonymousLogger`. These anonymous loggers are not registered in the global namespace, and their methods are not access-checked, allowing even untrusted code to change their logging control settings.

The logging framework does not attempt to prevent spoofing. The sources of logging calls cannot be determined reliably, so when a `LogRecord` is published that claims to be from a particular source class and source method, it may be a fabrication. Similarly, formatters such as the `XMLFormatter` do not attempt to protect themselves against nested log messages inside message strings. Thus, a spoof `LogRecord` might contain a spoof set of XML inside its message string to make it look as if there was an additional XML record in the output.

In addition, the logging framework does not attempt to protect itself against denial of service attacks. Any given logging client can flood the logging framework with meaningless messages in an attempt to conceal some important log message.

Configuration Management

The APIs are structured so that an initial set of configuration information is read as properties from a configuration file. The configuration information may then be changed programatically by calls on the various logging classes and objects.

In addition, there are methods on `LogManager` that allow the configuration file to be re-read. When this happens, the configuration file values will override any changes that have been made programatically.

Packaging

All of the logging class are in the `java.*` part of the namespace, in the `java.util.logging` package.

Localization

Log messages may need to be localized.

Each logger may have a `ResourceBundle` name associated with it. The corresponding `ResourceBundle` can be used to map between raw message strings and localized message strings.

Normally, formatters perform localization. As a convenience, the `Formatter` class provides a `formatMessage` method that provides some basic localization and formatting support.

Remote Access and Serialization

As with most Java platform APIs, the logging APIs are designed for use inside a single address space. All calls are intended to be local. However, it is expected that some handlers will want to forward their output to other systems. There are a variety of ways of doing this:

Some handlers (such as the `SocketHandler`) may write data to other systems using the `XMLFormatter`. This provides a simple, standard, inter-change format that can be parsed and processed on a variety of systems.

Some handlers may wish to pass `LogRecord` objects over RMI. The `LogRecord` class is therefore serializable. However, there is a problem in how to deal with the `LogRecord` parameters. Some parameters may not be serializable and other parameters may have been designed to serialize much more state than is required for logging. To avoid these problems, the `LogRecord` class has a custom `writeObject` method that converts the parameters to strings (using `Object.toString()`) before writing them out.

Most of the logging classes are not intended to be serializable. Both loggers and handlers are stateful classes that are tied into a specific virtual machine. In this respect they are analogous to the `java.io` classes, which are also not serializable.

Java Logging Examples

Simple Use

The following is a small program that performs logging using the default configuration.

This program relies on the root handlers that were established by the `LogManager` based on the configuration file. It creates its own `Logger` object and then makes calls to that `Logger` object to report various events.

```
package com.wombat;
import java.util.logging.*;
```

```
public class Nose {
    // Obtain a suitable logger.
    private static Logger logger = Logger.getLogger("com.wombat.nose");
    public static void main(String argv[]) {
        // Log a FINE tracing message
        logger.fine("doing stuff");
        try {
            Wombat.sneeze();
        } catch (Exception ex) {
            // Log the exception
            logger.log(Level.WARNING, "trouble sneezing", ex);
        }
        logger.fine("done");
    }
}
```

Changing the Configuration

Here's a small program that dynamically adjusts the logging configuration to send output to a specific file and to get lots of information on wombats. The pattern `%t` means the system temporary directory.

```
public static void main(String[] args) {
    Handler fh = new FileHandler("%t/wombat.log");
    Logger.getLogger("").addHandler(fh);
    Logger.getLogger("com.wombat").setLevel(Level.FINEST);
    ...
}
```

Simple Use, Ignoring Global Configuration

Here's a small program that sets up its own logging `Handler` and ignores the global configuration.

```
package com.wombat;

import java.util.logging.*;

public class Nose {
    private static Logger logger = Logger.getLogger("com.wombat.nose");
    private static FileHandler fh = new FileHandler("mylog.txt");
    public static void main(String argv[]) {
        // Send logger output to our FileHandler.
        logger.addHandler(fh);
        // Request that every detail gets logged.
        logger.setLevel(Level.ALL);
        // Log a simple INFO message.
        logger.info("doing stuff");
        try {
            Wombat.sneeze();
        } catch (Exception ex) {
            logger.log(Level.WARNING, "trouble sneezing", ex);
        }
```

```
        logger.fine("done");
    }
}
```

Sample XML Output

Here's a small sample of what some `XMLFormatter` XML output looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
  <record>
    <date>2015-02-27T09:35:44.885562Z</date>
    <millis>1425029744885</millis>
    <nanos>562000</nanos>
    <sequence>1256</sequence>
    <logger>kgh.test.fred</logger>
    <level>INFO</level>
    <class>kgh.test.XMLTest</class>
    <method>writeLog</method>
    <thread>10</thread>
    <message>Hello world!</message>
  </record>
</log>
```

Appendix A: DTD for XMLFormatter Output

```
<!-- DTD used by the java.util.logging.XMLFormatter -->
<!-- This provides an XML formatted log message. -->

<!-- The document type is "log" which consists of a sequence
of record elements -->
<!ELEMENT log (record*)>

<!-- Each logging call is described by a record element. -->
<!ELEMENT record (date, millis, nanos?, sequence, logger?, level,
class?, method?, thread?, message, key?, catalog?, param*, exception?)>

<!-- Date and time when LogRecord was created in ISO 8601 format -->
<!ELEMENT date (#PCDATA)>

<!-- Time when LogRecord was created in milliseconds since
midnight January 1st, 1970, UTC. -->
<!ELEMENT millis (#PCDATA)>

<!-- Nano second adjustement to add to the time in milliseconds.
This is an optional element, added since JDK 9, which adds further
precision to the time when LogRecord was created.
 -->
<!ELEMENT nanos (#PCDATA)>

<!-- Unique sequence number within source VM. -->
```

```
<!ELEMENT sequence (#PCDATA)>

<!-- Name of source Logger object. -->
<!ELEMENT logger (#PCDATA)>

<!-- Logging level, may be either one of the constant
names from java.util.logging.Level (such as "SEVERE"
or "WARNING") or an integer value such as "20". -->
<!ELEMENT level (#PCDATA)>

<!-- Fully qualified name of class that issued
logging call, e.g. "javax.marsupial.Wombat". -->
<!ELEMENT class (#PCDATA)>

<!-- Name of method that issued logging call.
It may be either an unqualified method name such as
"fred" or it may include argument type information
in parenthesis, for example "fred(int,String)". -->
<!ELEMENT method (#PCDATA)>

<!-- Integer thread ID. -->
<!ELEMENT thread (#PCDATA)>

<!-- The message element contains the text string of a log message. -->
<!ELEMENT message (#PCDATA)>

<!-- If the message string was localized, the key element provides
the original localization message key. -->
<!ELEMENT key (#PCDATA)>

<!-- If the message string was localized, the catalog element provides
the logger's localization resource bundle name. -->
<!ELEMENT catalog (#PCDATA)>

<!-- If the message string was localized, each of the param elements
provides the String value (obtained using Object.toString())
of the corresponding LogRecord parameter. -->
<!ELEMENT param (#PCDATA)>

<!-- An exception consists of an optional message string followed
by a series of StackFrames. Exception elements are used
for Java exceptions and other java Throwables. -->
<!ELEMENT exception (message?, frame+)>

<!-- A frame describes one line in a Throwable backtrace. -->
<!ELEMENT frame (class, method, line?)>

<!-- an integer line number within a class's source file. -->
<!ELEMENT line (#PCDATA)>
```