Java Platform, Standard Edition

Core Libraries

Release 9
E74190-02
September 2017

Java Platform, Standard Edition Core Libraries, Release 9

E74190-02

Contents

4 Process API

Preface

This guide provides information about the Java core libraries.

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.

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

Enhanced Deprecation

JDK 9 clarifies the semantics of what deprecation means, 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 JDK 9 release, 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 `jdeprscan` in the *Java Platform, Standard Edition Tools Reference*.

2
XML Catalog API

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

Java SE 9 introduces a new XML Catalog API to support the Organization for the Advancement of Structured Information Standards (OASIS) XML Catalogs, OASIS Standard V1.1. This chapter of the Oracle JDK 9 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 Catalog Standard. 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 Schema Validation) 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 The XML Catalog API, 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 The XML Catalog API 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 2-1 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.
`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.

3

Creating Immutable Lists, Sets, and Maps

Convenience static factory methods on the List, Set, and Map interfaces, which were added in JDK 9, let you easily create immutable lists, sets, and maps.

An object is considered *immutable* if its state cannot change after it is constructed. After you create an immutable instance of a collection, it holds the same data as long as a reference to it exists.

If the collections created using these methods contain immutable objects, then they are automatically thread safe after construction. Because the structures do not need to support mutation, they can be made much more space efficient. Immutable collection instances generally consume much less memory than their mutable counterparts.

As discussed in About Immutability, an immutable collection can contain mutable objects, and if it does, the collection is neither immutable nor thread safe.

Topics

Use Cases

The common use case for the immutable methods is a collection that is initialized from known values, and that never changes. Also consider using these methods if your data changes infrequently.

For optimal performance, the immutable collections store a data set that never changes. However, you may be able to take advantage of the performance and space-saving benefits even if your data is subject to change. These collections may provide better performance than the mutable collections, even if your data changes occasionally.

If you have a large number of values, you may consider storing them in a HashMap. If you are constantly adding and removing entries, then this is a good choice. But, if you have a set of values that never change, or rarely change, and you read from that set a lot, then the immutable `Map` is a more efficient choice. If the data set is read frequently, and the values change only rarely, then you may find that the overall speed is faster, even when you include the performance impact of destroying and rebuilding an immutable `Map` when a value changes.

Syntax

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

Topics

Immutable List Static Factory Methods

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

A list is an ordered collection, where duplicate elements are typically 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 3-1 Examples

In JDK 8:

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

In JDK 9:

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

See Immutable List Static Factory Methods.

Immutable Set Static Factory Methods

The `Set.of` static factory methods provide a convenient way to create immutable 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 3-2 Examples

In JDK 8:

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

In JDK 9:

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

See Immutable Set Static Factory Methods.

Immutable Map Static Factory Methods

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

A `Map` cannot contain duplicate keys; each key can map to at most one value. If a duplicate key is detected, then an `IllegalArgumentException` is thrown. Null values cannot be used as `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 3-3 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:

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

Example 3-4 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 Immutable Map Static Factory Methods.

Randomized Iteration Order

The iteration order for `Set` elements and `Map` keys is randomized: it is likely to be different from one JVM run to the next. This is intentional — it makes it easier for you to identify code that depends on iteration order. Sometimes dependencies on iteration order inadvertently creep into code, and cause problems that are difficult to debug.

You can see how the iteration order is the same until `jshell` is restarted.

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

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

jshell> /exit
|  Goodbye

C:\Program Files\Java\jdk-9\bin>jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro

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

The collection instances created by the `Set.of`, `Map.of`, and `Map.ofEntries` methods are the only ones whose iteration orders are randomized. The iteration ordering of collection implementations such as `HashMap` and `HashSet` is unchanged.

About Immutability

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

These collections are not "immutable persistent" or "functional" collections. If you are using one of those collections, then you can modify it, but when you do, you are returned a new updated collection that may share the structure of the first one.

One advantage of an immutable collection is that it is automatically thread safe. After you create a collection, you can hand it to multiple threads, and they will all see a consistent view.

However, an immutable collection of objects is not the same as a collection of immutable objects. 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 immutable 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 `ilist1` and `ilist2` that point to the first lists. If you try to modify `ilist1`, then you see an exception error because `ilist1` is immutable. Any modification attempt throws an exception.

```
jshell> List<List<String>> ilist1 = List.of(list1, list1);
ilist1 ==> [[a, b], [a, b]]

jshell> List<List<String>> ilist2 = List.of(list2, list2);
ilist2 ==> [[a, b], [a, b]]

jshell> ilist1.add(new ArrayList<String>())
|  java.lang.UnsupportedOperationException thrown:
|        at ImmutableCollections.uoe (ImmutableCollections.java:70)
|        at ImmutableCollections$AbstractImmutableList.add (ImmutableCollections
.java:76)
|        at (#10:1)
```

But if you modify the original `list1`, `ilist1` and `ilist2` are no longer equal.

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

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

jshell> ilist1.equals(ilist2)
$14 ==> false
```

Immutable and Unmodifiable Are Not the Same

The immutable collections behave in the same way as the `Collections.unmodifiable...` wrappers. However, these collections are not wrappers — 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> unmodlist1 = Collections.unmodifiableList(list1);
unmodlist1 ==> [a, b, c]

jshell> unmodlist1.add("d")
|  java.lang.UnsupportedOperationException thrown:
|        at Collections$UnmodifiableCollection.add (Collections.java:1056)
|        at (#17:1)
```

But, if you change the original `list1`, no error is generated, and the `unmodlist1` list has been modified.

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

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

Space Efficiency

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

4
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

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 processes attributes, which the following table summarizes:

Table 4-1 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 4-2 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 4-3 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()`

Table 4-3 (Cont.) ProcessHandle Interface Methods

Method Type	Related Methods
Retrieve streams of direct and indirect child processes.	• `children()` • `descendants()`
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 4-4 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.