Java Platform, Standard Edition Core Libraries



Release 23 F98492-03 October 2024

ORACLE

Java Platform, Standard Edition Core Libraries, Release 23

F98492-03

Copyright © 2017, 2024, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	ix
Documentation Accessibility	ix
Diversity and Inclusion	ix
Related Documents	ix
Conventions	ix

1 Java Core Libraries

2 Serialization Filtering

Addressing Deserialization Vulnerabilities	2-1
Java Serialization Filters	2-2
Filter Factories	2-3
Allow-Lists and Reject-Lists	2-3
Creating Pattern-Based Filters	2-4
Creating Custom Filters	2-6
Reading a Stream of Serialized Objects	2-6
Setting a Custom Filter for an Individual Stream	2-7
Setting a JVM-Wide Custom Filter	2-7
Setting a Custom Filter Using a Pattern	2-7
Setting a Custom Filter as a Class	2-8
Setting a Custom Filter as a Method	2-9
Creating a Filter with ObjectInputFilter Methods	2-9
Setting a Filter Factory	2-10
Setting a Filter Factory with setSerialFilterFactory	2-11
Specifying a Filter Factory in a System or Security Property	2-13
Built-in Filters	2-13
Logging Filter Actions	2-15

3 Enhanced Deprecation

Deprecation in the JDK

3-1



How to Deprecate APIs	3-1
Notifications and Warnings	3-3
Running jdeprscan	3-5

4 XML Catalog API

Purpose of XML Catalog API	4-1
XML Catalog API Interfaces	4-2
Using the XML Catalog API	4-3
System Reference	4-3
Public Reference	4-5
URI Reference	4-5
Java XML Processors Support	4-7
Enable Catalog Support	4-7
Use Catalog with XML Processors	4-8
Calling Order for Resolvers	4-12
Detecting Errors	4-12

5 Java Collections Framework

Creating Unmodifiable Lists, Sets, and Maps	5-2
Use Cases	5-2
Syntax	5-3
Unmodifiable List Static Factory Methods	5-3
Unmodifiable Set Static Factory Methods	5-3
Unmodifiable Map Static Factory Methods	5-4
Creating Unmodifiable Copies of Collections	5-5
Creating Unmodifiable Collections from Streams	5-5
Randomized Iteration Order	5-6
About Unmodifiable Collections	5-7
Space Efficiency	5-8
Thread Safety	5-9
Creating Sequenced Collections, Sets, and Maps	5-10
SequencedCollection	5-12
SequencedSet	5-13
SequencedMap	5-14
Demonstrating ArrayList and LinkedHashMap Reversed Views	5-15
Demonstrating a Reverse-Ordered View of a Collection	5-15
Demonstrating Composition of LinkedHashMap Views	5-16
Demonstrating SequencedMap Does Not Support Mutation of the Underlying Map	5-17

6 Process API

Process API Classes and Interfaces	6-1
ProcessBuilder Class	6-2
Process Class	6-2
ProcessHandle Interface	6-3
ProcessHandle.Info Interface	6-4
Creating a Process	6-4
Getting Information About a Process	6-5
Redirecting Output from a Process	6-6
Filtering Processes with Streams	6-6
Handling Processes When They Terminate with the onExit Method	6-7
Controlling Access to Sensitive Process Information	6-10

7 Preferences API

Comparing the Preferences API to Other Mechanisms	7-1
Usage Notes	7-2
Obtain Preferences Objects for an Enclosing Class	7-2
Obtain Preferences Objects for a Static Method	7-3
Atomic Updates	7-3
Determine Backing Store Status	7-4
Design FAQ	7-4

8 Java Logging Overview

Java Logging Examples	8-7
Appendix A: DTD for XMLFormatter Output	8-9

9 Java NIO

9-4
9-5
9-6
9-8
9-9
9-10
9-17
9-22
9-26
9-26



10 Java Networking

Networking System Properties	10-1
HTTP Client Properties	10-1
HTTP Server Properties	10-5
IPv4 and IPv6 Protocol Properties	10-7
HTTP Proxy Properties	10-8
HTTPS Proxy Properties	10-9
FTP Proxy Properties	10-9
SOCKS Proxy Properties	10-10
Acquiring the SOCKS User Name and Password	10-11
Other Proxy-Related Properties	10-11
UNIX Domain Sockets Properties	10-12
Other HTTP URL Stream Protocol Handler Properties	10-13
System Properties That Modify the Behavior of HTTP Digest Authentication	
Mechanism	10-18
Specify Mappings from Host Names to IP Addresses	10-18
Address Cache Properties	10-19
Enhanced Exception Messages	10-20

11 Pseudorandom Number Generators

Characteristics of PRNGs	11-1
Generating Pseudorandom Numbers with RandomGenerator Interface	11-2
Generating Pseudorandom Numbers in Multithreaded Applications	11-3
Dynamically Creating New Generators	11-3
Creating Stream of Generators	11-3
Choosing a PRNG Algorithm	11-4

12 Foreign Function and Memory API

On-Heap and Off-Heap Memory	12-1
Memory Segments and Arenas	12-1
Allocating a Memory Segment with an Arena and Storing a String in It	12-3
Printing the Contents of Off-Heap Memory	12-3
Closing an Arena	12-4
Calling a C Library Function with the Foreign Function and Memory API	12-4
Obtaining an Instance of the Native Linker	12-5
Locating the Address of the C Function	12-5
Describing the C Function Signature	12-6
Creating the Downcall Handle for the C Function	12-7
Calling the C Function Directly from Java	12-7
Upcalls: Passing Java Code as a Function Pointer to a Foreign Function	12-7



Defining the Java Method That Compares Two Elements	12-10
Creating a Downcall Method Handle for the qsort Function	12-10
Creating a Method Handle to Represent the Comparison Method qsortCompare	12-10
Creating a Function Pointer from the Method Handle compareHandle	12-11
Allocating Off-Heap Memory to Store the int Array	12-11
Calling the qsort Function	12-12
Copying the Sorted Array Values from Off-Heap to On-Heap Memory	12-12
Foreign Functions That Return Pointers	12-12
Memory Layouts and Structured Access	12-15
Checking for Native Errors Using errno	12-17
Slicing Allocators and Slicing Memory Segments	12-19
Slicing Allocators	12-20
Slicing Memory Segments	12-21
Restricted Methods	12-22
Calling Native Functions with jextract	12-24
Run a Python Script in a Java Application	12-24
Call the qsort Function from a Java Application	12-26

13 Scoped Values

14 Concurrency

Virtual Threads	14-3
What is a Platform Thread?	14-4
What is a Virtual Thread?	14-4
Why Use Virtual Threads?	14-4
Creating and Running a Virtual Thread	14-4
Creating a Virtual Thread with the Thread Class and the Thread.Builder Interface	14-5
Creating and Running a Virtual Thread with the Executors.newVirtualThreadPerTaskExecutor() Method	14-6
Multithreaded Client Server Example	14-6
Scheduling Virtual Threads and Pinned Virtual Threads	14-8
Debugging Virtual Threads	14-8
JDK Flight Recorder Events for Virtual Threads	14-8
Viewing Virtual Threads in jcmd Thread Dumps	14-9
Virtual Threads: An Adoption Guide	14-9
Write Simple, Synchronous Code Employing Blocking I/O APIs in the Thread-Per- Request Style	14-9
Represent Every Concurrent Task as a Virtual Thread; Never Pool Virtual Threads	14-10
Use Semaphores to Limit Concurrency	14-12
Don't Cache Expensive Reusable Objects in Thread-Local Variables	14-13

Avoid Lengthy and Frequent Pinning	14-14
Structured Concurrency	14-15
Basic Usage of the StructuredTaskScope Class	14-16
Common Shutdown Policies: ShutdownOnSuccess and ShutdownOnFailure	14-18
Implement Your Own StructuredTaskScope Policies	14-20
Debugging StructuredTaskScope and Its Forked Subtasks with the jcmd Command	14-22
Thread-Local Variables	14-25
Inheriting Thread-Local Variables	14-27
Issues with Thread-Local Variables	14-29

15 Stream Gatherers

What Is a Gatherer?	15-1
Creating a Gatherer	15-2
Creating Gatherers with Factory Methods	15-6
Built-In Gatherers	15-7
Composing Gatherers	15-9



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 Java Virtual Machine (JVM), serialization, concurrency, and other functionality close to the JVM.

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 customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

See JDK 23 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.



Convention	Meaning
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
- Java Collections Framework
- Process API
- Preferences API
- Java Logging Overview
- Java NIO
- Java Networking
- Pseudorandom Number Generators
- Foreign Function and Memory API
- Scoped Values
- Concurrency
- Stream Gatherers

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
- Java Serialization Filters
- Filter Factories
- Allow-Lists and Reject-Lists
- Creating Pattern-Based Filters
- Creating Custom Filters
- Setting a Filter Factory
- Built-in Filters
- Logging Filter Actions

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.

Inherent Dangers of Deserialization

Deserializing untrusted data, especially from an unknown, untrusted, or unauthenticated client, is an inherently dangerous activity because the content of the incoming data stream determines the objects that are created, the values of their fields, and the references between them. By careful construction of the stream, an adversary can run code in arbitrary classes with malicious intent.

For example, if object construction has side effects that change state or invoke other actions, then those actions can compromise the integrity of application objects, library objects, and even the Java runtime. "Gadget classes," which can perform arbitrary reflective actions such as create classes and invoke methods on them, can be deserialized maliciously to cause a denial of service or remote code execution.

The key to disabling deserialization attacks is to prevent instances of arbitrary classes from being deserialized, thereby preventing the direct or indirect execution of their methods. You can do this through serialization filters.

Java Serialization and Deserialization Overview

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 <code>java.io.Serializable</code> interface or the <code>java.io.Externalizable</code> 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), and Java Management Extensions (JMX).

An object is descrialized when its serialized form is converted to a copy of the object. It is important to ensure the security of this conversion. Descrialization is code execution because the readObject method of the class that is being descrialized can contain custom code.

Serialization Filters

A serialization filter enables you to specify which classes are acceptable to an application and which should be rejected. Filters also enable you to control the object graph size and complexity during deserialization so that the object graph doesn't exceed reasonable limits. You can configure filters as properties or implement them programmatically.

Note:

A serialization filter is not enabled or configured by default. Serialization filtering doesn't occur unless you have specified the filter in a system property or a Security Property or set it with the <code>ObjectInputFilter class</code>.

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, for example, 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 reject-lists and/or extend the allow-list 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 instances of classes before they are deserialized.

As stated in JEP 290 and JEP 415, the goals of the Java serialization filtering mechanism are to:

- Provide a way to narrow the classes that can be deserialized down to a contextappropriate 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.



There are two kinds of filters:

- **JVM-wide filter**: Is applied to every deserialization in the JVM. However, whether and how a JVM-wide filter validates classes in a particular deserialization depends on how it's combined with other filters.
- Stream-specific filter: Validates classes from one specific ObjectInputStream.

You can implement a serialization filter in the following ways:

- Specify a JVM-wide, pattern-based filter with the jdk.serialFilter property: A pattern-based filter consists of a sequence of patterns that can accept or reject the name of specific classes, packages, or modules. It can place limits on array sizes, graph depth, total references, and stream size. A typical use case is to add classes that have been identified as potentially compromising the Java runtime to a reject-list. If you specify a pattern-based filter with the jdk.serialFilter property, then you don't have to modify your application.
- Implement a custom or pattern-based stream-specific filter with the ObjectInputFilter API: You can implement a filter with the ObjectInputFilter API, which you then set on an ObjectInputStream. You can create a pattern-based filter with the ObjectInputFilter API by calling the Config.createFilter(String) method.

Note:

A serialization filter is not enabled or configured by default. Serialization filtering doesn't occur unless you have specified the filter in a system property or a Security Property or set it with the <code>ObjectInputFilter class</code>.

For every new object in the stream, the filter mechanism applies only one filter to it. However, this filter might be a combination of filters.

In most cases, a stream-specific filter should check if a JVM-wide filter is set, especially if you haven't specified a filter factory. If a JVM-wide filter does exist, then the stream-specific filter should invoke it and use the JVM-wide filter's result unless the status is UNDECIDED.

Filter Factories

A filter factory selects, chooses, or combines filters into a single filter to be used for a stream. When you specify one, a deserialization operation uses it when it encounters a class for the first time to determine whether to allow it. (Subsequent instances of the same class aren't filtered.) It's implemented as a BinaryOperator<ObjectInputFilter> and specified with the ObjectInputFilter.Config.setSerialFilterFactory method or in a system or Security property; see Setting a Filter Factory. Whenever an ObjectInputStream is created, the filter factory selects an ObjectInputFilter. However, you can have a different filter created based on the characteristics of the stream and the filter that the filter factory previously created.

Allow-Lists and Reject-Lists

Allow-lists and reject-lists 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 allow-lists to allow only class names that are recognized and trusted and to reject all others. You can implement allow-lists 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 allow-lists by specifying the names of classes, packages, or modules that are allowed.

The defensive approach uses reject-lists to reject instances of classes that are not trusted. Usually, reject-lists are implemented after an attack that reveals that a class is a problem. A class name can be added to a reject-list, without a code change, by adding it to a pattern-based filter that's specified in the jdk.serialFilter property.

Creating Pattern-Based Filters

Pattern-based filters are filters that you define without changing your application code. You add JVM-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 name that matches a pattern that is preceded by ! is rejected. A class name that matches a pattern without ! is allowed. The following filter rejects pattern1.MyClass but allows pattern2.MyClass:

!pattern1.*;pattern2.*

- Use the wildcard symbol (*) to represent unspecified class names in a pattern as shown in the following examples:
 - To match every class name, use *
 - To match every class name in mypackage, use mypackage.*
 - To match every class name in mypackage and its subpackages, use mypackage.**
 - To match every class name that starts with text, use text*

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

For a complete description of the syntax for the patterns, see JEP 290.

Pattern-Based Filter Limitations

The following are some of the limitations of pattern-based filters:

ORACLE

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

Note:

A pattern-based filter doesn't check interfaces that are implemented by classes being deserialized. The filter is invoked for interfaces explicitly referenced in the stream; it isn't invoked for interfaces implemented by classes for objects being deserialized.

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.App
lication
```

Define a Pattern-Based Filter for All Applications

You can define a pattern-based, JVM-wide filter that affects every application run with a Java runtime from <code>\$JAVA_HOME</code> by specifying it as a Security Property. (Note that a system property supersedes a Security Property value.) Edit the file <code>\$JAVA_HOME/conf/security/java.security</code> and add the pattern-based filter to the <code>jdk.serialFilter</code> 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 name from a package (! example.somepackage.SomeClass), and allows all other class names in the package:

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

The previous example filter allows all other class names, not just those in example.somepackage.*. To reject all other class names, 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.

Topics

- Reading a Stream of Serialized Objects
- Setting a Custom Filter for an Individual Stream
- Setting a JVM-Wide Custom Filter
- Setting a Custom Filter Using a Pattern
- Setting a Custom Filter as a Class
- Setting a Custom Filter as a Method
- Creating a Filter with ObjectInputFilter Methods

Reading a Stream of Serialized Objects

You can set a custom filter on one <code>ObjectInputStream</code>, or, to apply the same filter to every stream, set a JVM-wide filter. If an <code>ObjectInputStream</code> doesn't have a filter defined for it, the JVM-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 and before the object is instantiated and deserialized, the filter is called when it encounters a class for the first time. (Subsequent instances of the same class aren't filtered.)
- 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 JVM-Wide Custom Filter

You can set a JVM-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, JVM-wide filters are used to reject specific classes or packages, or to limit array sizes, graph depth, or total graph size.

A JVM-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 JVM-wide filter is set by using a lambda expression.

In the following example, the JVM-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 names of classes, packages, and modules and can place limits on array sizes, graph depth, total references, and stream size. Patterns cannot match the names of the supertype or interfaces of the class.

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

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

This example allows only example. File. All other class names 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))
               ? ObjectInputFilter.Status.ALLOWED
               : ObjectInputFilter.Status.REJECTED;
      }
      return ObjectInputFilter.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, then the value is checked to see if the class of the object is Number. If so, it is accepted and returns ObjectInputFilter.Status.ALLOWED. Otherwise, it is rejected and returns ObjectInputFilter.Status.REJECTED.
- Any other combination of arguments returns <code>ObjectInputFilter.Status.UNDECIDED.</code> 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;
    }
}
```

This custom filter allows only the classes found in the base module of the JDK:

Creating a Filter with ObjectInputFilter Methods

The <code>ObjectInputFilter</code> interface includes the following static methods that enable you to quickly create filters:

- allowFilter(Predicate<Class<?>>, ObjectInputFilter.Status)
- rejectFilter(Predicate<Class<?>>, ObjectInputFilter.Status)
- rejectUndecidedClass(ObjectInputFilter)
- merge(ObjectInputFilter, ObjectInputFilter)

The allowFilter method creates a filter based on a Predicate that takes a Class as its argument. The created filter returns <code>ObjectInputFilter.Status.ALLOWED</code> if the predicate is true. Otherwise, it returns the value of the allowFilter method's second argument. The



following creates a filter that accepts the Integer class. All other classes are considered undecided:

The rejectFilter method is the inverse of allowFilter: It creates a filter based on a Predicate that takes a Class as its argument. The created filter returns ObjectInputFilter.Status.REJECTED if the predicate is true. Otherwise, it returns the value of the rejectFilter method's second argument. The following creates a filter that rejects any class loaded from the application class loader:

The rejectUndecidedClass method creates a new filter based on an existing filter by rejecting any class that the existing filter considers as undecided. The following creates a filter based on intFilter. It accepts the Integer class but rejects all other (undecided) classes:

```
ObjectInputFilter rejectUndecidedFilter =
    ObjectInputFilter.rejectUndecidedClass(intFilter);
```

The merge method creates a new filter by merging two filters. The following merges the filters intFilter and f. It accepts the Integer class but rejects any class loaded from the application class loader:

```
ObjectInputFilter mergedFilter = ObjectInputFilter.merge(intFilter, f);
```

A merged filter follows these steps when it filters a class:

- 1. Return Status.REJECTED if either of its filters return Status.REJECTED.
- 2. Return Status.ACCEPTED if either of its filters return Status.ACCEPTED.
- 3. Return Status.UNDECIDED (both of its filters return Status.UNDECIDED).

The merge method is useful in filter factories. Every time a filter is set on a stream, you can append that filter to the one that the filter factory creates with the merge method. See the ObjectInputFilter API documentation for an example.

Note:

It's a good idea to merge the JVM-wide filter with the requested, stream-specific filter in your filter factory. If you just return the requested filter, then you effectively disable the JVM-wide filter, which will lead to security gaps.

Setting a Filter Factory

A filter factory is a BinaryOperator, which is a function of two operands that chooses the filter for a stream. You can set a filter factory by calling the method

ObjectInputFilter.Config.setSerialFilterFactory or specifying it in a system or Security property.

Note:

You can set a filter factory exactly once, either with the method setSerialFilterFactory, in the system property jdk.serialFilterFactory, or in the Security Property jdk.serialFilterFactory.

Topics:

- Setting a Filter Factory with setSerialFilterFactory
- Specifying a Filter Factory in a System or Security Property

Setting a Filter Factory with setSerialFilterFactory

When you set a filter factory by calling the method

ObjectInputFilter.Config.setSerialFilterFactory, the filter factory's method BinaryOperator<ObjectInputFilter>.apply(ObjectInputFilter t, ObjectInputFilter u) will be invoked when an ObjectInputStream is constructed and when a stream-specific filter is set on an ObjectInputStream. The parameter t is the current filter and u is the requested filter. When apply is first invoked, t will be null. If a JVM-wide filter has been set, then when apply is first invoked, u will be the JVM-wide filter. Otherwise, u will be null. The apply method (which you must implement yourself) returns the filter to be used for the stream. If apply is invoked again, then the parameter t will be this returned filter. When you set a filter with the method ObjectInputStream.setObjectInputFilter(ObjectInputFilter), then parameter u will be this filter.

The following example implements a simple filter factory that prints its <code>ObjectInputFilter</code> parameters every time its <code>apply</code> method is invoked, merges these parameters into one combined filter, then returns this merged filter.

```
public class SimpleFilterFactory {
    static class MySimpleFilterFactory implements
BinaryOperator<ObjectInputFilter> {
        public ObjectInputFilter apply(
            ObjectInputFilter curr, ObjectInputFilter next) {
            System.out.println("Current filter: " + curr);
            System.out.println("Requested filter: " + next);
            return ObjectInputFilter.merge(next, curr);
        }
    }
    private static byte[] createSimpleStream(Object obj) {
        ByteArrayOutputStream boas = new ByteArrayOutputStream();
        try (ObjectOutputStream ois = new ObjectOutputStream(boas)) {
            ois.writeObject(obj);
            return boas.toByteArray();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        throw new RuntimeException();
```

```
}
    public static void main(String[] args) throws IOException {
        // Set a filter factory
        MySimpleFilterFactory contextFilterFactory = new
MySimpleFilterFactory();
ObjectInputFilter.Config.setSerialFilterFactory(contextFilterFactory);
        // Set a stream-specific filter
        ObjectInputFilter filter1 =
            ObjectInputFilter.Config.createFilter("example.*;java.base/*;!*");
        ObjectInputFilter.Config.setSerialFilter(filter1);
        // Create another filter
        ObjectInputFilter intFilter = ObjectInputFilter.allowFilter(
          cl -> cl.equals(Integer.class),
ObjectInputFilter.Status.UNDECIDED);
        // Create input stream
        byte[] intByteStream = createSimpleStream(42);
        InputStream is = new ByteArrayInputStream(intByteStream);
        ObjectInputStream ois = new ObjectInputStream(is);
        ois.setObjectInputFilter(intFilter);
        try {
            Object obj = ois.readObject();
            System.out.println("Read obj: " + obj);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

This example prints output similar to the following (line breaks have been added for clarity):

```
Current filter: null
Requested filter: example.*;java.base/*;!*
Current filter: example.*;java.base/*;!*
Requested filter:
    merge(
        predicate(
            SimpleFilterFactory$$Lambda$8/0x000000800c00c60@76ed5528,
            ifTrue: ALLOWED, ifFalse: UNDECIDED),
        predicate(
            SimpleFilterFactory$$Lambda$9/0x000000800c01800@2c7b84de,
            ifTrue: REJECTED, ifFalse: UNDECIDED))
Read obj: 42
```



The apply method is invoked twice: when the ObjectInputStream ois is created and when the method setObjectInputFilter is called.

Note:

- You can set a filter on an ObjectInputStream only once. An IllegalStateException will be thrown otherwise.
- To protect against unexpected deserializations, ensure that security experts thoroughly review how your filter factories select and combine filters.

Specifying a Filter Factory in a System or Security Property

You can set a filter factory that applies to only one application and to only a single invocation of Java by specifying it in the jdk.serialFilterFactory system property in the command line:

java -Djdk.serialFilterFactory=FilterFactoryClassName YourApplication

The value of jdk.serialFilterFactory is the class name of the filter factory to be set before the first deserialization. The class must be public and accessible to the application class loader (which the method java.lang.ClassLoader.getSystemClassLoader() returns).

You can set a JVM-wide filter factory that affects every application run with a Java runtime from <code>\$JAVA_HOME</code> by specifying it in a Security Property. Note that a system property supersedes a Security Property value. Edit the file <code>\$JAVA_HOME/conf/security/java.security</code> and specify the filter factory's class name in the <code>jdk.serialFilterFactory</code> Security Property.

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 <code>sun.rmi.registry.registryFilter</code> system property to configure reject-lists and/or extend the allow-list to add additional protection for the RMI Registry. To protect the whole application, add the patterns to the <code>jdk.serialFilter</code> 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 allow-list 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.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 reject-lists and/or
extend the allow-list 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 allow-list 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

com.sun.management.jmxremote.serial.filter.pattern management property to configure reject-lists and/or extend the allow-list 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. The filter's pattern must include the types that are allowed to be sent as parameters over RMI to the server and all types that they depend on, plus javax.management.ObjectName and java.rmi.MarshalledObject types. The default filter



covers any type that Open MBeans and MXBeans might use. Applications that register their own MBeans in the platform MBean server may need to extend the filter to support any additional types that their MBeans accept as parameters. Specify the filter's pattern with the com.sun.management.jmxremote.serial.filter.pattern management property in the *\$JAVA_HOME*/conf/management/management.properties file. You can override the value of this property with the java command-line option –

Dcom.sun.management.jmxremote.serial.filter.pattern=<pattern>.

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 allow-lists and reject-lists.

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

To enable serialization filter logging, edit the <code>\$JDK HOME/conf/logging.properties</code> file.

To log calls that are rejected, add

```
java.io.serialization.level = FINE
```

To log all filter results, add

java.io.serialization.level = FINEST



3 Enhanced Deprecation

The semantics of what deprecation means includes 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
- How to Deprecate APIs
- Notifications and Warnings
- Running jdeprscan

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 API is going to be removed.

In prior releases, APIs were deprecated but rarely ever 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 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 <code>@Deprecated</code>. The annotation contains these elements:

- @Deprecated(since="<version>")
 - <version> identifies the version in which 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 containing 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 <code>@Deprecated</code> 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 (which are APIs that are provided by a library that are accessible to code outside of that library, such as applications or other libraries).

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 <code>@Deprecated</code> annotation. You should determine and communicate your policy around API removals. For

example, if you release a new library every six 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:

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.

Here is an example of a removal 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 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 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 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 manage these external resources.

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

- Availability: If a resource is remote, then XML processors must be able to connect to the remote server hosting the resource. 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, 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 resource, 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

Application developers can create a local catalog of all external references for the 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 can establish a local catalog for the system and configure the Java VM to use the catalog. Then, all of the applications on the system may share the same catalog without any code changes to the applications, assuming that 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 provides 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 jaxp.properties file is typically in the \$JAVA HOME/conf directory.

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 the value continue. 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">
<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, whereas 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.

Consider the following XML file:

```
<?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 file defines an entity example:

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

However, the URI to the example.dtd file in the XML file 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:

With this catalog entry file and the system entry, all you need to do is get a default CatalogFeatures object and set the URI to the catalog entry 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 <code>jaxp.properties</code> 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:
```

Additionally, given another XML file XI test2.xml:

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:



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.

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.

Table 4-1 RESOLVE Settings

5 Java Collections Framework

The Java platform includes a collections framework that provides developers with a unified architecture for representing and manipulating collections, enabling them to be manipulated independently of the details of their representation. A collection is an object that represents a group of objects (such as the classic ArrayList class).

The Java Collections Framework enables interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse. The framework is based on more than a dozen collection interfaces, and includes implementations of these interfaces with the algorithms to manipulate them.

Overview

The Java Collections Framework consists of:

- **Collection interfaces**: Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
- General-purpose implementations: Primary implementations of the collection interfaces.
- **Legacy implementations**: The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
- **Special-purpose implementations**: Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations**: Implementations designed for highly concurrent use.
- Wrapper implementations: Add functionality, such as synchronization, to other implementations.
- **Convenience implementations**: High-performance "mini-implementations" of the collection interfaces.
- Abstract implementations: Partial implementations of the collection interfaces to facilitate custom implementations.
- Algorithms: Static methods that perform useful functions on collections, such as sorting a list.
- Infrastructure: Interfaces that provide essential support for the collection interfaces.
- Array Utilities: Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

See Java Collections Framework for detailed information about the interfaces and implementations contained in the Java Collections Framework.

See The Java[™] Tutorials for basic information about using the Java Collections Framework.

Topics

Creating Unmodifiable Lists, Sets, and Maps



Creating Sequenced Collections, Sets, and Maps

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
- Syntax
- Creating Unmodifiable Copies of Collections
- Creating Unmodifiable Collections from Streams
- Randomized Iteration Order
- About Unmodifiable Collections
- Space Efficiency
- Thread Safety

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 collections is simple, especially for small numbers of elements.

Topics

- Unmodifiable List Static Factory Methods
- Unmodifiable Set Static Factory Methods
- Unmodifiable Map Static Factory Methods

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 in which 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:



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, 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 modifiability 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. (See Unmodifiable View Collections in the Collection interface JavaDoc API documentation). 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 of the collection. 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.



Creating Sequenced Collections, Sets, and Maps

Three interfaces introduced in JDK 21 represent collections with a defined encounter order. Each collection has a well-defined first element, second element, and so forth, up to the last element. They provide uniform APIs for accessing their first and last elements, and processing their elements in forward and reverse order.

Prior to JDK 21, the Java Collections Framework lacked a collection type that represented a sequence of elements with a defined encounter order. For example, List and Deque defined an encounter order but their common supertype, Collection, did not. Similarly, Set and subtypes such as HashSet do not define an encounter order, while subtypes such as SortedSet and LinkedHashSet do. Given the lack of a collection type with a defined encounter order, there is no uniform set of operations that respect encounter order. While there are operations that respect encounter order, they're not uniform.

An example of where a common order-significant operation is missing in the Collections Framework is to get the first element of a Deque and of a List. To get the first element of a Deque, you use the getFirst() method. However, to get the first element of a List, you use get(0).

Support for encounter order was spread across the type hierarchy, making it difficult to express certain useful concepts in APIs. Neither Collection nor List could describe a parameter or return value that had an encounter order. Collection was too general, relegating such constraints to the specification, and possibly leading to hard-to-debug errors. If an API wanted to receive a collection with a defined encounter order, then using List was too specific, because it excluded SortedSet and LinkedHashSet. A related problem was that view collections were often forced to downgrade to weaker semantics. For example, wrapping a LinkedHashSet with Collections::unmodifiableSet yields a Set that discards the information about encounter order.

Without interfaces to define them, operations related to encounter order were either inconsistent or missing. Many implementations support getting the first or last element, but each collection defines its own approach, and some are not obvious or are missing entirely.

Retrofitting the Collections Framework with Sequenced Interfaces

Beginning with JDK 21, JEP 431 introduces three Java Collections Framework interfaces for creating sequenced collections, sequenced sets, and sequenced maps:

- SequencedCollection
- SequencedSet
- SequencedMap

These three interfaces provide the Java Collections Framework with a collection type that represents a sequence of elements with a defined encounter order and with a uniform set of operations applied across the collections. The interfaces fit into the collections type hierarchy as shown in the following diagram.



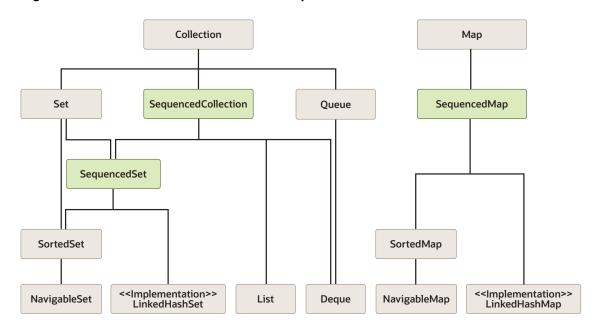


Figure 5-1 Collections Framework with Sequenced Interfaces

The diagram shows the following adjustments that integrated the SequencedCollection, SequencedSet, and SequencedMap interfaces into the Java Collections Framework hierarchy of classes and interfaces:

- List has SequencedCollection as its immediate superinterface.
- Deque has SequencedCollection as its immediate superinterface.
- LinkedHashSet implements SequencedSet.
- SortedSet has SequencedSet as its immediate superinterface.
- LinkedHashMap implements SequencedMap.
- SortedMap has SequencedMap as its immediate superinterface.
- Covariant overrides for the reversed() method are defined in the appropriate places. For example, List::reversed is overridden to return a value of type List rather than a value of type SequencedCollection.
- Methods added to the Collections utility class create unmodifiable wrappers for three new types:
 - Collections.unmodifiableSequencedCollection(sequencedCollection)
 - Collections.unmodifiableSequencedSet(sequencedSet)
 - Collections.unmodifiableSequencedMap(sequencedMap)

See JEP 431 for background information about the interfaces for sequenced collections, sequenced sets, and sequenced maps.

Topics

- SequencedCollection
- SequencedSet
- SequencedMap



SequencedCollection

A SequencedCollection is a collection type added in JDK 21 that represents a sequence of elements with a defined encounter order.

A <u>SequencedCollection</u> has first and last elements with the elements between them having successors and predecessors. A <u>SequencedCollection</u> supports common operations at either end, and it supports processing the elements from first to last and from last to first (such as, forward and reverse).

```
interface SequencedCollection<E> extends Collection<E> {
   SequencedCollection<E> reversed();
   // methods promoted from Deque
   void addFirst(E);
   void addLast(E);
   E getFirst();
   E getLast();
   E removeFirst();
   E removeLast();
}
```

The reversed() method provides a reverse-ordered view of the original collection. Any modifications to the original collection are visible in the view.

The encounter order of elements in the returned view is the inverse of the encounter order of elements in this collection. The reverse ordering affects all order-sensitive operations, including those on the view collections of the returned view.

Changes to the underlying collection might or might not be visible in the reversed view, depending upon the implementation. If permitted, modifications to the view "write through" to the original collection. The reverse-ordered view enables all the different sequenced types to process elements in both directions, using all the usual iteration mechanisms:

- Enhanced for loops
- Explicit iterator() loops
- forEach()
- stream()
- parallelStream()
- toArray()

For example, obtaining a reverse-ordered stream from a LinkedHashSet was previously quite difficult; now it is simply:

```
linkedHashSet.reversed().stream()
```

Note:

The reversed() method is essentially a renamed NavigableSet::descendingSet, promoted to SequencedCollection.

The following methods of SequencedCollection are promoted from Deque. They support adding, getting, and removing elements at both ends:

- void addFirst(E)
- void addLast(E)
- E getFirst()
- E getLast()
- E removeFirst()
- E removeLast()

The add*(E) and remove*() methods are optional, primarily to support the case of unmodifiable collections. The get*() and remove*() methods throw a NoSuchElementException if the collection is empty. There are no definitions of equals() and hashCode() in SequencedCollection because its subinterfaces have conflicting definitions.

SequencedSet

A SequencedSet is both a SequencedCollection and a Set.

A SequencedSet can be thought of either as a Set that also has a well-defined encounter order, or as a SequencedCollection that also has unique elements.

```
interface SequencedSet<E> extends Set<E>, SequencedCollection<E> {
    SequencedSet<E> reversed(); // covariant override
}
```

This interface has the same requirements on the equals and hashCode methods as defined by Set.equals and Set.hashCode. A Set and a SequencedSet compare equals if and only if they have equal elements, irrespective of ordering.

SequencedSet defines the reversed() method, which provides a reverse-ordered view of this set. The only difference from the SequencedCollection.reversed method is that the return type of SequencedSet.reversed is SequencedSet.

In SequencedSet, the add* (E) methods of the SequencedCollection perform the following:

- addFirst(E) Adds an element as the first element of the collection.
- addLast (E) Adds an element as the last element of the collection.

The add*(E) methods of the SequencedCollection also have the following special-case behaviors for LinkedHashSet and SortedSet.

Special-case behaviors for LinkedHashSet:

• The addFirst(E) and addLast(E) methods have special-case semantics for collections such as LinkedHashSet. LinkedHashSet repositions the entry if it is already present in the set. If the element is already present in the set then it is moved to the appropriate position. This remedies a long-standing deficiency in LinkedHashSet, namely the inability to reposition elements.

Special-case behaviors for SortedSet:



 Collections such as SortedSet, which position elements by relative comparison, cannot support explicit-positioning operations such as the addFirst(E) and addLast(E) methods declared in the SequencedCollection superinterface. These methods throw an UnsupportedOperationException.

SequencedMap

A SequencedMap provides methods to add mappings, to retrieve mappings, and to remove mappings at either end of the map's encounter order. This interface also defines the reversed() method, which provides a reverse-ordered view of this map.

A SequencedMap has a well-defined encounter order that supports operations at both ends and is reversible. A map's reverse-ordered view is generally not serializable, even if the original map is serializable. The encounter order of a SequencedMap is similar to that of the elements of a SequencedCollection, but the ordering applies to mappings instead of individual elements:

```
interface SequencedMap<K,V> extends Map<K,V> {
   SequencedMap<K,V> reversed();
   SequencedSet<K> sequencedKeySet();
   SequencedCollection<V> sequencedValues();
   SequencedSet<Entry<K,V>> sequencedEntrySet();
   V putFirst(K, V);
   V putLast(K, V);
   // methods promoted from NavigableMap
   Entry<K, V> firstEntry();
   Entry<K, V> lastEntry();
   Entry<K, V> pollFirstEntry();
   Entry<K, V> pollFirstEntry();
   Entry<K, V> pollLastEntry();
}
```

The sequencedKeySet(), sequencedValues(), and sequencedEntrySet() methods are exactly analogous to the keySet(), values(), and entrySet() methods of Map interface. All of these methods return views of the underlying collection; where modifications to the view are visible in the underlying collection and vice versa. The encounter order of these views exactly corresponds to the encounter order of the underlying map.

The difference between the SequencedMap interface methods and the methods of Map is that the $sequenced^*$ () methods have a sequenced return type:

- In SequencedSet<K> sequencedKeySet(), the implemention returns a SequencedSet view
 of the map's keySet and behaves as follows:
 - add and addAll methods throw UnsupportedOperationException.
 - reversed method returns the sequencedKeySet view of the reversed view of the map.
 - Its other methods call the corresponding methods of the keySet view of the map.
- In SequencedCollection<V> sequencedValues(), the implemention returns a SequencedCollection view of the map's values collection and behaves as follows:
 - add and addAll methods throw UnsupportedOperationException.
 - reversed method returns the sequencedValues view of the reversed view of the map.



- equals and hashCode methods are inherited from Object.
- Its other methods call the corresponding methods of the values view of the map.
- In SequencedSet<Entry<K,V>> sequencedEntrySet(), the implemention returns a SequencedSet view of the map's entrySetand behaves as follows:
 - add and addAll methods throw UnsupportedOperationException.
 - reversed method returns the sequencedEntrySet view of the reversed view of the map.
 - Its other methods call the corresponding methods of the entrySet view of the map.

The put*(K, V) methods have special-case semantics, similar to the corresponding add*(E) methods of SequencedSet:

- For maps such as LinkedHashMap, they have the additional effect of repositioning the entry if it is already present in the map.
- For maps such as SortedMap, these methods throw UnsupportedOperationException.

The following methods of SequencedMap are promoted from NavigableMap. They support getting and removing entries at both ends:

- Entry<K, V> firstEntry()
- Entry<K, V> lastEntry()
- Entry<K, V> pollFirstEntry()
- Entry<K, V> pollLastEntry()

The methods firstEntry(), lastEntry(), pollFirstEntry(), and pollLastEntry() return Map.Entry instances that represent snapshots of mappings as of the time of the call. They do *not* support mutation of the underlying map via the optional setValue method.

Demonstrating ArrayList and LinkedHashMap Reversed Views

Several scenarios are provided of using the sequenced interfaces in the Collections Framework.

Topics

- Demonstrating a Reverse-Ordered View of a Collection
- Demonstrating Composition of LinkedHashMap Views
- Demonstrating SequencedMap Does Not Support Mutation of the Underlying Map

Demonstrating a Reverse-Ordered View of a Collection

The following example demonstrates how the reversed() method of the sequenced interfaces produces a reverse-ordered view of a collection, how modifications to a reversed view affect the original collection, and how modifications to the original collection are visible in the reversed view.

The reversed view is "live" and not a snapshot of a collection. This characteristic is illustrated in the following examples by using an ArrayList and its reversed view.



Note:

Unessential jshell output is not included in the following example code.

Start a jshell session and use the ArrayList class to create a list of String objects.

```
jshell> var list = new ArrayList<>(Arrays.asList("a", "b", "c", "d", "e"))
list ==> [a, b, c, d, e]
```

Next, use the reversed() method to produce a reverse-ordered view of the collection.

```
jshell> var rev = list.reversed()
rev ==> [e, d, c, b, a]
```

When you modify the reversed view, it affects the original collection. Add f as an entry to the reverse-ordered view and then verify that it is added to the original collection.

```
jshell> rev.add(1, "f")
jshell> rev
rev ==> [e, f, d, c, b, a]
jshell> list
list ==> [a, b, c, d, f, e]
```

When you modify the original collection, your modifications are visible in the reversed view. Set the element at index 2 to x, verify it is added to the collection, and then produce a reverse-ordered view of the modified collection.

```
jshell> list.set(2, "X")
jshell> list
list ==> [a, b, X, d, f, e]
jshell> rev
rev ==> [e, f, d, X, b, a]
```

Demonstrating Composition of LinkedHashMap Views

In addition to using ArrayList, a reversed() view can also be composed of other views such
as List.subList().reversed() or SequencedMap.sequencedKeySet().reversed() and
SequencedMap.reversed().sequencedKeySet().

The SequencedMap.sequencedKeySet().reversed() and SequencedMap.reversed().sequencedKeySet() views are functionally equivalent and are illustrated by using the LinkedHashMap class in the following example code.

Start a jshell session and use the LinkedHashMap class to create a map of String objects.

```
jshell> var map = new LinkedHashMap<String, Integer>()
jshell> map.put("a", 1)
jshell> map.put("b", 2)
jshell> map.put("c", 3)
jshell> map.put("d", 4)
```



```
jshell> map.put("e", 5)
map ==> {a=1, b=2, c=3, d=4, e=5}
```

Next, use the reversed() method to produce a reverse-ordered view of the keySet view of the original collection.

```
jshell> map.sequencedKeySet().reversed()
$17 ==> [e, d, c, b, a]
```

Demonstrating SequencedMap Does Not Support Mutation of the Underlying Map

This demonstration illustrates the final statement in the SequencedMap section that firstEntry(), lastEntry(), pollFirstEntry(), and pollLastEntry() methods do not support mutation of the underlying map through use of the optional setValue method.

Attempting to change an entry in the underlying map by using setValue() with these methods
will throw an UnsupportedOperationException. This is in contrast to changing a map entry
obtained by *iterating* the entrySet. If you call seqmap.entrySet().iterator().next() to
return a map entry and then call setValue() on the entry, it will modify the original map.

Open a jshell session and use the map produced in Demonstrating Composition of LinkedHashMap Views.

Call map.entrySet().iterator().next() to return the first map entry.

```
jshell> var entry = map.entrySet().iterator().next()
entry ==> a=1
```

Use setValue() to change the value of the map entry to 77. The entry was obtained by *iterating* the entrySet so it can be modified in the original map. Verify that the value in map changed to 77.

```
jshell> entry.setValue(77)
$19 ==> 1
jshell> map
map ==> {a=77, b=2, c=3, d=4, e=5}
```

Note:

The ability to call setValue() on the entry returned by the iterator is not a new behavior introduced in JDK 21.

Use setValue() to try and change the map entry to 999. Because the map entry was not obtained by iterating with entrySet, it throws an UnsupportedOperationException.

```
jshell> entry = map.firstEntry()
entry ==> a=77
jshell> entry.setValue(999)
| Exception java.lang.UnsupportedOperationException: not supported
| at NullableKeyValueHolder.setValue (NullableKeyValueHolder.java:126)
| at (#22:1)
```



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 tasks to start when another ends
 - Clean up leftover processes
- Test the running of commands:
 - Run a series of tests
 - Log output
- Monitor commands:
 - Monitor long-running processes and restart them if they terminate
 - Collect usage statistics

Topics

- Process API Classes and Interfaces
- Creating a Process
- Getting Information About a Process
- Redirecting Output from a Process
- Filtering Processes with Streams
- Handling Processes When They Terminate with the onExit Method
- Controlling Access to Sensitive Process Information

Process API Classes and Interfaces

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

Topics

- ProcessBuilder Class
- Process Class
- ProcessHandle Interface
- 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:

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 (see the System.getEnv() method).	 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.getOutputStrea m method.	 redirectInput (ProcessBuilder.Red irect 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(Proc essBuilder.Redirect destination) redirectError(Proce ssBuilder.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

Table 6-1 ProcessBuilder Class Attributes and Related Methods

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.



Method Type	Related Methods
Wait for the process to complete.	waitfor()waitFor(long timeout, TimeUnit unit)
Retrieve information about the process.	<pre>isAlive() pid() info() exitValue()</pre>
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()

Table 6-2 Process Class Methods

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:

Related Methods
• allProcesses()
• current()
• of(long pid)
• parent()
• isAlive()
• pid()
• info()
• children()
• descendants()
• destroy()
• destroyForcibly()

Table 6-3 ProcessHandle Interface Methods



Method Type	Related Methods
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()

Table 6-3 (Cont.) ProcessHandle Interface Methods

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.
<pre>startInstant()</pre>	Returns the start time of the process.
totalCpuDuration()	Returns the process's total accumulated CPU time.
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's 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 doc, 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((Instant i) -> i
                .atZone(ZoneId.systemDefault()).toLocalDateTime().toString())
                .orElse(na));
        System.out.printf("Arguments: %s%n",
            info.arguments().map(
                (String[] 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).redire
ctError(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 {
  // ...
 public static void main(String[] args) {
    ProcessTest.filterProcessesTest();
  }
  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 <code>onExit()</code>.thenAccept(onExitMethod) on each of the processes; <code>onExitMethod</code> 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;
  }
  // ...
1
```

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.java:8
Exit 12403, status 0
ProcessTest.class:0
ProcessTest.java:8
```

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.

WARNING:

The Security Manager and APIs related to it have been deprecated and are subject to removal in a future release. There is no replacement for the Security Manager. See JEP 411 for discussion and alternatives.



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 changes 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
- Usage Notes
- Design FAQ

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 a dynamic 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. Furthermore, 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
- Obtain Preferences Objects for a Static Method
- Atomic Updates
- Determine Backing Store Status

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?
- How does the Preferences API relate to JNDI?
- Why do all of the get methods require the caller to pass in a default?
- How was it decided which methods should throw BackingStoreException?
- 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?
- 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?
- Why doesn't this API use the Java 2 Collections Framework?
- Why don't the put and remove methods return the old values?



- Why does the API permit, but not require, stored defaults?
- Why doesn't this API contain methods to read and write arbitrary serializable objects?
- Why is Preferences an abstract class rather than an interface?
- Where is the default backing store?

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 userhome/.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

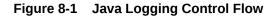
- Overview of Control Flow
- Log Levels
- Loggers
- Logging Methods
- Handlers
- Formatters
- The LogManager
- Configuration File
- Default Configuration
- Dynamic Configuration Updates
- Native Methods
- XML DTD
- Unique Message IDs
- Security
- Configuration Management
- Packaging
- Localization
- Remote Access and Serialization
- Java Logging Examples
- Appendix A: DTD for XMLFormatter Output

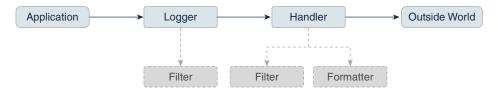
Overview of Control Flow

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.



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.

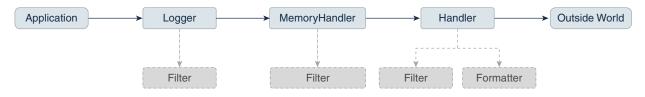




Each Logger object keeps track of a set of output Handler objects. By default all Logger objects also send their output to their parent Logger. But Logger objects may also be configured to ignore Handler objects higher up the tree.

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.





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 to 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, corresponding to 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)>



9 Java NIO

The Java NIO (New Input/Output) API defines buffers, which are containers for data, and other structures, such as charsets, channels, and selectable channels. Charsets are mappings between bytes and Unicode characters. Channels represent connections to entities capable of performing I/O operations. Selectable channels are those that can be multiplexed, which means that they can process multiple I/O operations in one channel.

Java NIO Examples

The following code examples demonstrate the Java NIO API:

- Grep NIO Example
- Checksum NIO Example
- Time Query NIO Example
- Time Server NIO Example
- Non-Blocking Time Server NIO Example
- Internet Protocol and UNIX Domain Sockets NIO Example
- File NIO examples:
 - Chmod File NIO Example
 - Copy File NIO Example
 - Disk Usage File NIO Example
 - User-Defined File Attributes File NIO Example

Buffers

They are containers for a fixed amount of data of a specific primitive type. See the java.nio package and Table 9-1.

Table 3-1 Dullel Classes	Table 9-1	Buffer	Classes
--------------------------	-----------	--------	---------

Buffer Class	Description
Buffer	Base class for buffer classes.
ByteBuffer	Buffer for bytes.
MappedByteBuffer	Buffer for bytes that is mapped to a file.
CharBuffer	Buffer for the char data type.
DoubleBuffer	Buffer for the double data type.
FloatBuffer	Buffer for the float data type.
IntBuffer	Buffer for the int data type.
LongBuffer	Buffer for the long data type.
ShortBuffer	Buffer for the short data type.



Charsets

They are named mappings between sequences of 16-bit Unicode characters and sequences of bytes. Support for charsets include decoders and encoders, which translate between bytes and Unicode characters. See the java.nio.charset package and Table 9-2.

Charset Class	Description
Charset	Named mapping between characters and bytes, for example, US-ASCII and UTF-8.
CharsetDecoder	Decodes bytes into characters.
CharsetEncoder	Encodes characters into bytes.
CoderResult	Describes the result state of an decoder or encoder.
CodingErrorAction	Describes actions to take when coding errors are detected.

Channels

They represent an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, for example reading or writing. See the java.nio.channels package and Table 9-3.

Channel Interface or Class	Description	
Channel	Base interface for channel interfaces and classes.	
ReadableByteChannel	A channel that can read bytes.	
ScatteringByteChannel	A channel that can read bytes into a sequence of buffers. A <i>scattering</i> read operation reads, in a single invocation, a sequence of bytes into one or more of a given sequence of buffers.	
WritableByteChannel	A channel that can write bytes.	
GatheringByteChannel	A channel that can write bytes from a sequence of buffers. A <i>gathering</i> write operation writes, in a single invocation, a sequence of bytes from one or more of a given sequence of buffers.	
ByteChannel	A channel that can read and write bytes. It unifies ReadableByteChannel and WritableByteChannel.	
SeekableByteChannel	A byte channel that maintains a current <i>position</i> and allows the position to be changed. A seekable byte channel is connected to an entity, typically a file, that contains a variable-length sequence of bytes that can be read and written.	
AsynchronousChannel	A channel that supports asynchronous I/O operations.	
AsynchronousByteChannel	An asynchronous channel that can read and write bytes.	
NetworkChannel	A channel to a network socket.	

Table 9-3 Channel Interfaces and Classes



Channel Interface or Class	Description
MulticastChannel	A network channel that supports Internet Protocol (IP) multicasting. IP multicasting is the transmission of IP datagrams to members of a <i>group</i> that is zero or more hosts identified by a single destination address.
FileChannel	A channel for reading, writing, mapping, and manipulating a file. It's a SeekableByteChannel that is connected to a file.
SelectableChannel	A channel that can be multiplexed through a Selector.
	Multiplexing is the ability to process multiple I/O operations in one channel. A selectable channel can be put into blocking or non-blocking mode. In blocking mode, every I/O operation invoked upon the channel will block until it completes. In non- blocking mode, an I/O operation will never block and may transfer fewer bytes than were requested or possibly no bytes at all.
DatagramChannel	A selectable channel that can send and receive UDP (User Datagram Protocol) packets.
	You can create datagram channels with different protocol families:
	 Create channels for Internet Protocol sockets with the INET or INET6 protocol families. These channels support network communication using TCL and UDP. Their addresses are of type InetSocketAddress, which encapsulates an IP address and port number. Create channels for UNIX Domain sockets with the UNIX protocol family. These sockets support local interprocess communication on the same host. Their addresses are of type UnixDomainSocketAddress, which encapsulate a file system path name on the local system.
Pipe.SinkChannel	A channel representing the writable end of a pipe. A Pipe is a pair of channels: A writable sink channel and a readable source channel.
Pipe.SourceChannel	A channel representing the readable end of a pipe.
ServerSocketChannel	A selectable channel for stream-oriented listening sockets. Like datagram channels, you can create server
	socket channels that are for Internet Protocol sockets or Unix Domain sockets.
SocketChannel	A selectable channel for stream-oriented connecting sockets.
	Like datagram channels, you can create socket channels that are for Internet Protocol sockets or Unix Domain sockets.

Table 9-3 (Cont.) Channel Interfaces and Classes



Channel Interface or Class	Description
AsynchronousFileChannel	An asynchronous channel for reading, writing, and manipulating a file.
AsynchronousSocketChannel	An asynchronous channel for stream-oriented connecting sockets.
AsynchronousServerSocketChannel	An asynchronous channel for stream-oriented listening sockets.

Grep NIO Example

This example searches a list of files for lines that match a given regular expression pattern. It demonstrates NIO-mapped byte buffers, charsets, and regular expressions.

```
public class Grep {
    // Charset and decoder for ISO-8859-15
    private static Charset charset = Charset.forName("ISO-8859-15");
    private static CharsetDecoder decoder = charset.newDecoder();
    // Pattern used to parse lines
    private static Pattern linePattern = Pattern.compile(".*\r?\n");
    // The input pattern that we're looking for
    private static Pattern pattern;
    // Compile the pattern from the command line
    private static void compile(String pat) {
        try {
            pattern = Pattern.compile(pat);
        } catch (PatternSyntaxException x) {
            System.err.println(x.getMessage());
            System.exit(1);
        }
    }
    // Use the linePattern to break the given CharBuffer into lines, applying
    // the input pattern to each line to see if we have a match
    private static void grep(File f, CharBuffer cb) {
        Matcher lm = linePattern.matcher(cb); // Line matcher
        Matcher pm = null; // Pattern matcher
        int lines = 0;
        while (lm.find()) {
            lines++;
            CharSequence cs = lm.group(); // The current line
            if (pm == null)
                pm = pattern.matcher(cs);
            else
                pm.reset(cs);
            if (pm.find())
                System.out.print(f + ":" + lines + ":" + cs);
            if (lm.end() == cb.limit())
```

```
break;
        }
    }
    // Search for occurrences of the input pattern in the given file
    private static void grep(File f) throws IOException {
        // Open the file and then get a channel from the stream
        try (FileInputStream fis = new FileInputStream(f);
            FileChannel fc = fis.getChannel()) {
            // Get the file's size and then map it into memory
            int sz = (int) fc.size();
            MappedByteBuffer bb = fc.map(FileChannel.MapMode.READ ONLY, 0,
sz);
            // Decode the file into a char buffer
            CharBuffer cb = decoder.decode(bb);
            // Perform the search
            grep(f, cb);
        }
    }
    public static void main(String[] args) {
        if (args.length < 2) {
            System.err.println("Usage: java Grep pattern file...");
            return;
        }
        compile(args[0]);
        for (int i = 1; i < args.length; i++) {</pre>
            File f = new File(args[i]);
            try {
                grep(f);
            } catch (IOException x) {
                System.err.println(f + ": " + x);
            1
        }
    }
}
```

Checksum NIO Example

This example computes 16-bit checksums for a list of files. It uses NIO-mapped byte buffers for speed.

```
public class Sum {
    // Compute a 16-bit checksum for all the remaining bytes
    // in the given byte buffer
    private static int sum(ByteBuffer bb) {
        int sum = 0;
        while (bb.hasRemaining()) {
            if ((sum & 1) != 0)
        }
    }
}
```

```
sum = (sum >> 1) + 0x8000;
            else
                sum >>= 1;
            sum += bb.get() & 0xff;
            sum &= 0xffff;
        }
        return sum;
    }
    // Compute and print a checksum for the given file
    private static void sum(File f) throws IOException {
        // Open the file and then get a channel from the stream
        try (
            FileInputStream fis = new FileInputStream(f);
            FileChannel fc = fis.getChannel()) {
            // Get the file's size and then map it into memory
            int sz = (int) fc.size();
            MappedByteBuffer bb = fc.map(FileChannel.MapMode.READ_ONLY, 0,
sz);
            // Compute and print the checksum
            int sum = sum(bb);
            int kb = (sz + 1023) / 1024;
            String s = Integer.toString(sum);
            System.out.println(s + "\t" + kb + "\t" + f);
        }
    }
    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: java Sum file...");
            return;
        }
        for (int i = 0; i < args.length; i++) {
            File f = new File(args[i]);
            try {
                sum(f);
            } catch (IOException e) {
                System.err.println(f + ": " + e);
        }
    }
}
```

Time Query NIO Example

This example asks a list of hosts what time it is. It's a simple, blocking program that demonstrates NIO socket channels (connection and reading), buffer handling, charsets, and regular expressions.

```
public class TimeQuery {
```



```
// The standard daytime port
private static int DAYTIME PORT = 13;
// The port we'll actually use
private static int port = DAYTIME PORT;
// Charset and decoder for US-ASCII
private static Charset charset = Charset.forName("US-ASCII");
private static CharsetDecoder decoder = charset.newDecoder();
// Direct byte buffer for reading
private static ByteBuffer dbuf = ByteBuffer.allocateDirect(1024);
// Ask the given host what time it is
private static void query(String host) throws IOException {
   try (SocketChannel sc = SocketChannel.open()) {
        InetSocketAddress isa = new InetSocketAddress(
            InetAddress.getByName(host), port);
        // Connect
        sc.connect(isa);
        // Read the time from the remote host. For simplicity we assume
        // that the time comes back to us in a single packet, so that we
        // only need to read once.
        dbuf.clear();
        sc.read(dbuf);
        // Print the remote address and the received time
        dbuf.flip();
        CharBuffer cb = decoder.decode(dbuf);
        System.out.print(isa + " : " + cb);
   }
}
public static void main(String[] args) {
   if (args.length < 1) {
        System.err.println("Usage: java TimeQuery [port] host...");
        return;
    ł
   int firstArg = 0;
   // If the first argument is a string of digits then we take that
   // to be the port number
   if (Pattern.matches("[0-9]+", args[0])) {
       port = Integer.parseInt(args[0]);
        firstArg = 1;
    }
   for (int i = firstArg; i < args.length; i++) {</pre>
        String host = args[i];
        try {
            query(host);
        } catch (IOException e) {
```

```
System.err.println(host + ": " + e);
e.printStackTrace();
}
}
}
```

Time Server NIO Example

This example listens for connections and tells callers what time it is. Is a simple, blocking program that demonstrates NIO socket channels (accepting and writing), buffer handling, charsets, and regular expressions.

```
public class TimeServer {
    // We can't use the normal daytime port (unless we're running as root,
    // which is unlikely), so we use this one instead
    private static int PORT = 8013;
    // The port we'll actually use
    private static int port = PORT;
    // Charset and encoder for US-ASCII
    private static Charset charset = Charset.forName("US-ASCII");
    private static CharsetEncoder encoder = charset.newEncoder();
    // Direct byte buffer for writing
    private static ByteBuffer dbuf = ByteBuffer.allocateDirect(1024);
    // Open and bind the server-socket channel
    private static ServerSocketChannel setup() throws IOException {
        ServerSocketChannel ssc = ServerSocketChannel.open();
        InetSocketAddress isa = new InetSocketAddress(
            InetAddress.getLocalHost(), port);
        ssc.socket().bind(isa);
        return ssc;
    }
    // Service the next request to come in on the given channel
    private static void serve(ServerSocketChannel ssc) throws IOException {
        try (SocketChannel sc = ssc.accept()) {
            String now = new Date().toString();
            System.out.println("now: " + now);
            sc.write(encoder.encode(CharBuffer.wrap(now + "\r\n")));
            System.out.println(sc.socket().getInetAddress() + " : " + now);
        }
    }
    public static void main(String[] args) {
        if (args.length > 1) {
            System.err.println("Usage: java TimeServer [port]");
            return;
        }
```

```
// If the first argument is a string of digits then we take that
// to be the port number
if ((args.length == 1) && Pattern.matches("[0-9]+", args[0]))
    port = Integer.parseInt(args[0]);

try {
    ServerSocketChannel ssc = setup();
    for (;;) {
        serve(ssc);
    }
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

Non-Blocking Time Server NIO Example

This example implements a non-blocking internet time server.

```
public class NBTimeServer {
    private static final int DEFAULT TIME PORT = 8900;
    // Constructor with no arguments creates a time server on default port.
    public NBTimeServer() throws Exception {
       acceptConnections(this.DEFAULT TIME PORT);
    }
    // Constructor with port argument creates a time server on specified port.
    public NBTimeServer(int port) throws Exception {
       acceptConnections(port);
    }
    // Accept connections for current time. Lazy Exception thrown.
    private static void acceptConnections(int port) throws Exception {
        // Selector for incoming time requests
       Selector acceptSelector = SelectorProvider.provider().openSelector();
       // Create a new server socket and set to non blocking mode
       ServerSocketChannel ssc = ServerSocketChannel.open();
       ssc.configureBlocking(false);
       // Bind the server socket to the local host and port
       InetAddress lh = InetAddress.getLocalHost();
        InetSocketAddress isa = new InetSocketAddress(lh, port);
       ssc.socket().bind(isa);
       // Register accepts on the server socket with the selector. This
       // step tells the selector that the socket wants to be put on the
       // ready list when accept operations occur, so allowing multiplexed
       // non-blocking I/O to take place.
       SelectionKey acceptKey = ssc.register(acceptSelector,
            SelectionKey.OP ACCEPT);
```

```
int keysAdded = 0;
        // Here's where everything happens. The select method will
        // return when any operations registered above have occurred, the
        // thread has been interrupted, etc.
        while ((keysAdded = acceptSelector.select()) > 0) {
            // Someone is ready for I/O, get the ready keys
            Set<SelectionKey> readyKeys = acceptSelector.selectedKeys();
            Iterator<SelectionKey> i = readyKeys.iterator();
            // Walk through the ready keys collection and process date
requests.
            while (i.hasNext()) {
                SelectionKey sk = (SelectionKey) i.next();
                i.remove();
                // The key indexes into the selector so you
                // can retrieve the socket that's ready for I/O
                ServerSocketChannel nextReady = (ServerSocketChannel) sk
                    .channel();
                // Accept the date request and send back the date string
                Socket s = nextReady.accept().socket();
                // Write the current time to the socket
                PrintWriter out = new PrintWriter(s.getOutputStream(), true);
                Date now = new Date();
                out.println(now);
                out.close();
            }
        }
    }
    // Entry point.
    public static void main(String[] args) {
        // Parse command line arguments and
        // create a new time server (no arguments yet)
        try {
            NBTimeServer nbt = new NBTimeServer();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Internet Protocol and UNIX Domain Sockets NIO Example

This example illustrates how to intermix AF_UNIX and AF_INET/6 channels with the SocketChannel and ServerSocketChannel classes in a non-blocking client/server single-threaded application.

This example mimics some of the capabilities of the socat command-line utility. It can create listeners or clients and connect them to listeners and perform various different types of binding. Run this command with the -h option to print usage information.

Special handling is only required for the different address types at initialization. For the server side, once a listener is created and bound to an address, the code managing the selector can handle the different address families identically.

```
import java.io.IOException;
import java.io.UncheckedIOException;
import java.net.*;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import jdk.net.ExtendedSocketOptions;
import jdk.net.UnixDomainPrincipal;
import static java.net.StandardProtocolFamily.UNIX;
import static java.net.StandardProtocolFamily.INET;
import static java.net.StandardProtocolFamily.INET6;
public class Socat {
    static void usage() {
        String ustring = """
usage: java Socat -s <baddr>...
    java Socat -c [-bind <baddr>] <daddr> N [delay]
    java Socat -h
-s means create one or more listening servers bound to addresses <baddr>...,
then accept all incoming connections and display (counts of) received data. If
more than one <baddr> is supplied, then multiple channels are created, each
bound to one of the supplied addresses. All channels are non-blocking and
managed by one Selector.
-c means create a client, connect it to <daddr> and send N (16 Kb) buffers.
The
client may optionally bind to a given address <baddr>. If a delay is
specified,
then the program pauses for the specified number of milliseconds between each
send. After sending, the client reads until EOF and then exits.
Note: AF UNIX client sockets do not bind to an address by default. Therefore,
the remote address seen on the server side (and the client's local address) is
an empty path. This is slightly different from AF INET/6 sockets, which, if
the
user does not choose a local port, then a randomly chosen one is assigned.
-h means print this message and exit.
<baddr> and <daddr> are addresses specified as follows:
    UNIX: {path}
```



```
INET: {host}:port
    INET6: {host}:port
{path} is the name of a socket file surrounded by curly brackets,
{}, which can be empty when binding a server signifying a randomly chosen
local
address.
{host}:port is an internet address comprising a domain name or IPv4/v6 literal
surrounded by curly brackets, {}, which can be empty when binding (signifying
any local address) and a port number, which can be zero when binding.
""";
        System.out.println(ustring);
    }
    static boolean isClient;
    static boolean initialized = false;
    static final int BUFSIZE = 8 * 1024;
    static int N;
                            // Number of buffers to send
    static int DELAY = 0; // Milliseconds to delay between sends
    static List<AddressAndFamily> locals = new LinkedList<>();
    static AddressAndFamily remote;
    // family is only needed in cases where address is null.
    // It could be a Record type.
    static class AddressAndFamily {
        SocketAddress address;
        ProtocolFamily family;
        AddressAndFamily(ProtocolFamily family, SocketAddress address) {
            this.address = address;
            this.family = family;
        }
    }
    static AddressAndFamily parseAddress(String addr) throws
UnknownHostException {
        char c = addr.charAt(0);
        if (c != 'U' && c != 'I')
            throw new IllegalArgumentException("invalid address");
        String family = addr.substring(0, addr.indexOf(':')).toUpperCase();
        return switch (family) {
            case "UNIX" -> parseUnixAddress(addr);
            case "INET" -> parseInetSocketAddress(INET, addr);
            case "INET6" -> parseInetSocketAddress(INET6, addr);
            default -> throw new IllegalArgumentException();
        };
    }
    static AddressAndFamily parseUnixAddress(String token) {
        String path = getPathDomain(token);
        UnixDomainSocketAddress address;
```

```
if (path.isEmpty())
              address = null;
        else
            address = UnixDomainSocketAddress.of(path);
        return new AddressAndFamily(UNIX, address);
    }
    static AddressAndFamily parseInetSocketAddress(StandardProtocolFamily
family, String token) throws UnknownHostException {
        String domain = getPathDomain(token);
        InetAddress address;
        if (domain.isEmpty()) {
            address = (family == StandardProtocolFamily.INET)
                    ? InetAddress.getByName("0.0.0.0")
                    : InetAddress.getByName("::0");
        } else {
            address = InetAddress.getByName(domain);
        }
        int cp = token.lastIndexOf(':') + 1;
        int port = Integer.parseInt(token.substring(cp));
        var isa = new InetSocketAddress(address, port);
        return new AddressAndFamily(family, isa);
    }
    // Return the token between braces, that is, a domain name or UNIX path.
    static String getPathDomain(String s) {
       int start = s.indexOf('{') + 1;
        int end = s.indexOf('}');
        if (start == -1 || end == -1 || (start > end))
            throw new IllegalArgumentException(s);
       return s.substring(start, end);
    }
    // Return false if the program must exit.
    static void parseArgs(String[] args) throws UnknownHostException {
        if (args[0].equals("-h")) {
            usage();
        } else if (args[0].equals("-c")) {
            isClient = true;
            int nextArg;
            AddressAndFamily local = null;
            if (args[1].equals("-bind")) {
                local = parseAddress(args[2]);
                locals.add(local);
                nextArg = 3;
            } else {
                nextArg = 1;
            remote = parseAddress(args[nextArg++]);
            N = Integer.parseInt(args[nextArg++]);
            if (nextArg == args.length - 1) {
                DELAY = Integer.parseInt(args[nextArg]);
            initialized = true;
```

```
} else if (args[0].equals("-s")) {
            isClient = false;
            for (int i = 1; i < args.length; i++) {</pre>
                locals.add(parseAddress(args[i]));
            l
            initialized = true;
        } else
            throw new IllegalArgumentException();
    }
    public static void main(String[] args) throws Exception {
        try {
            parseArgs(args);
        } catch (Exception e) {
            System.out.printf("\nInvalid arguments supplied. See the
following for usage information\n");
            usage();
        }
        if (!initialized)
            return;
        if (isClient) {
            doClient();
        } else {
            doServer();
        }
    }
    static Map<SocketChannel,Integer> byteCounter = new HashMap<>();
   private static void initListener(AddressAndFamily aaf, Selector selector)
{
        try {
            ProtocolFamily family = aaf.family;
            SocketAddress address = aaf.address;
            ServerSocketChannel server = ServerSocketChannel.open(family);
            server.bind(address);
            server.configureBlocking(false);
            postBind(address);
            server.register(selector, SelectionKey.OP ACCEPT, null);
            System.out.println("Server: Listening on " +
server.getLocalAddress());
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }
    private static void doServer() throws IOException {
        ByteBuffer readBuf = ByteBuffer.allocate(64 * 1024);
        final Selector selector = Selector.open();
        locals.forEach(localAddress -> initListener(localAddress, selector));
        int nextConnectionId = 1;
        while (true) {
            selector.select();
            var keys = selector.selectedKeys();
            for (SelectionKey key : keys) {
                try {
```

```
SelectableChannel c = key.channel();
                    if (c instanceof ServerSocketChannel) {
                        var server = (ServerSocketChannel)c;
                        var ch = server.accept();
                        var userid = "";
                        if (server.getLocalAddress() instanceof
UnixDomainSocketAddress) {
                            // An illustration of additional capability of
UNIX
                            // channels; it's not required behavior.
                            UnixDomainPrincipal pr =
ch.getOption(ExtendedSocketOptions.SO PEERCRED);
                            userid = "user: " + pr.user().toString() + "
group: " +
                                    pr.group().toString();
                        }
                        ch.configureBlocking(false);
                        byteCounter.put(ch, 0);
                        System.out.printf("Server: new connection\n\tfrom {%s}
\n", ch.getRemoteAddress());
                        System.out.printf("\tConnection id: %s\n",
nextConnectionId);
                        if (userid.length() > 0) {
                            System.out.printf("\tpeer credentials: %s\n",
userid);
                        }
                        System.out.printf("\tConnection count: %d\n",
byteCounter.size());
                        ch.register(selector, SelectionKey.OP READ,
nextConnectionId++);
                    } else {
                        var ch = (SocketChannel) c;
                        int id = (Integer)key.attachment();
                        int bytes = byteCounter.get(ch);
                        readBuf.clear();
                        int n = ch.read(readBuf);
                        if (n < 0) {
                            String remote = ch.getRemoteAddress().toString();
                            System.out.printf("Server: closing
connection\n\tfrom: {%s} Id: %d\n", remote, id);
                            System.out.printf("\tBytes received: %d\n",
bytes);
                            byteCounter.remove(ch);
                            ch.close();
                        } else {
                            readBuf.flip();
                            bytes += n;
                            byteCounter.put(ch, bytes);
                            display(ch, readBuf, id);
                        }
                    ł
                } catch (IOException e) {
                    throw new UncheckedIOException(e);
                }
```

```
};
            keys.clear();
        }
    }
    private static void postBind(SocketAddress address) {
        if (address instanceof UnixDomainSocketAddress) {
            var usa = (UnixDomainSocketAddress)address;
            usa.getPath().toFile().deleteOnExit();
        }
    }
    private static void display (SocketChannel ch, ByteBuffer readBuf, int id)
    throws IOException
    {
        System.out.printf("Server: received %d bytes from: {%s} Id: %d\n",
        readBuf.remaining(), ch.getRemoteAddress(), id);
    }
    private static void doClient() throws Exception {
        SocketChannel client;
        if (locals.isEmpty())
            client = SocketChannel.open(remote.address);
        else {
            AddressAndFamily aaf = locals.get(0);
            client = SocketChannel.open(aaf.family);
            client.bind(aaf.address);
            postBind(aaf.address);
            client.connect(remote.address);
        }
        ByteBuffer sendBuf = ByteBuffer.allocate(BUFSIZE);
        for (int i=0; i<N; i++) {</pre>
            fill(sendBuf);
            client.write(sendBuf);
            Thread.sleep(DELAY);
        }
        client.shutdownOutput();
        ByteBuffer rxb = ByteBuffer.allocate(64 * 1024);
        int c;
        while ((c = client.read(rxb)) > 0) {
            rxb.flip();
            System.out.printf("Client: received %d bytes\n", rxb.remaining());
            rxb.clear();
        }
        client.close();
    }
    private static void fill(ByteBuffer sendBuf) {
        // Because this example is for demonstration purposes, this method
        // doesn't fill the ByteBuffer sendBuf with data. Instead, it sets the
        // limits of sendBuf to its capacity and its position to zero.
        // Consequently, when the example writes the contents of sendBuf, it
        // writes the entire contents of whatever happened to be in memory
when
        // sendBuf was allocated.
```

```
sendBuf.limit(sendBuf.capacity());
sendBuf.position(0);
}
```

Example of Running the Socat Example

}

The following is an example of running the Socat example:

1. In a command-line shell, run Socat as follows:

```
$ java Socat -s UNIX:{/tmp/uds.sock}
Server: Listening on /tmp/uds.sock
```

2. In another command-line shell, run Socat as follows:

```
$ java Socat -c UNIX:{/tmp/uds.sock} 1
```

In the first command-line shell, you'll see output similar to the following:

```
Server: new connection
    from {}
    Connection id: 1
    peer credentials: user: yourusername group: yourgroup
    Connection count: 1
Server: received 8192 bytes from: {} Id: 1
Server: closing connection
    from: {} Id: 1
    Bytes received: 8192
```

If you don't specify a file name when you create a UNIX domain socket, then the JVM creates a socket file and automatically binds the socket to it:

```
$ java Socat -s UNIX:{}
Server: Listening on /tmp/socket 837668026
```

This is the same as calling ServerSocketChannel.bind(null). You can change the default directory where the JVM saves automatically generated socket files by setting the jdk.net.unixdomain.tmpdir system property. See Networking System Properties.

Chmod File NIO Example

This example compiles a list of one or more *symbolic mode expressions* that can change a set of file permissions in a manner similar to the UNIX chmod command.

The *symbolic-mode-list* parameter is a comma-separated list of expressions where each expression has the following form:

```
who operator [permissions]
```

• *who*: One or more of the following characters: u, g, o, or a, meaning owner (user), group, others, or all (owner, group, and others), respectively.



- operator: The character +, -, or =, signifying how to change the permissions:
 - +: Permissions are added
 - -: Permissions are removed
 - =: Permissions are assigned absolutely
- *permissions*: A sequence of zero or more of the following:
 - r: Read permission
 - w: Write permission

public class Chmod {

x: Execute permission

If *permissions* is omitted when permissions are assigned absolutely (with the = operator), then the permissions are cleared for the owner, group or others as identified by *who*. When *permissions* is omitted, then the operators + and – are ignored.

The following are examples of the *symbolic-mode-list* parameter:

- u=rw: Sets the owner permissions to read and write.
- ug+w: Sets the owner write and group write permissions.
- u+w, o-rwx: Sets the owner write permission and removes the others read, others write, and others execute permissions.
- o=: Sets the others permission to none (others read, others write, and others executed permissions are removed if set).

```
public static Changer compile(String exprs) {
        // minimum is who and operator (u= for example)
        if (exprs.length() < 2)
            throw new IllegalArgumentException("Invalid mode");
        // permissions that the changer will add or remove
        final Set<PosixFilePermission> toAdd = new
HashSet<PosixFilePermission>();
        final Set<PosixFilePermission> toRemove = new
HashSet<PosixFilePermission>();
        // iterate over each of expression modes
        for (String expr: exprs.split(",")) {
            // minimum of who and operator
            if (expr.length() < 2)
                throw new IllegalArgumentException("Invalid mode");
            int pos = 0;
            // who
            boolean u = false;
            boolean g = false;
            boolean o = false;
            boolean done = false;
            for (;;) {
                switch (expr.charAt(pos)) {
                    case 'u' : u = true; break;
```

```
case 'g' : g = true; break;
        case 'o' : o = true; break;
        case 'a' : u = true; g = true; o = true; break;
        default : done = true;
    }
    if (done)
        break;
    pos++;
}
if (!u && !g && !o)
    throw new IllegalArgumentException("Invalid mode");
// get operator and permissions
char op = expr.charAt(pos++);
String mask = (expr.length() == pos) ? "" : expr.substring(pos);
// operator
boolean add = (op == '+');
boolean remove = (op == '-');
boolean assign = (op == '=');
if (!add && !remove && !assign)
    throw new IllegalArgumentException("Invalid mode");
// who= means remove all
if (assign && mask.length() == 0) {
   assign = false;
   remove = true;
   mask = "rwx";
}
// permissions
boolean r = false;
boolean w = false;
boolean x = false;
for (int i=0; i<mask.length(); i++) {</pre>
    switch (mask.charAt(i)) {
        case 'r' : r = true; break;
        case 'w' : w = true; break;
        case 'x' : x = true; break;
        default:
            throw new IllegalArgumentException("Invalid mode");
    }
}
// update permissions set
if (add) {
    if (u) {
        if (r) toAdd.add(OWNER READ);
        if (w) toAdd.add(OWNER WRITE);
        if (x) toAdd.add(OWNER EXECUTE);
    }
    if (g) {
        if (r) toAdd.add(GROUP READ);
        if (w) toAdd.add(GROUP WRITE);
        if (x) toAdd.add(GROUP EXECUTE);
    }
```

```
if (o) {
                    if (r) toAdd.add(OTHERS READ);
                    if (w) toAdd.add(OTHERS WRITE);
                    if (x) toAdd.add(OTHERS EXECUTE);
                }
            if (remove) {
                if (u) {
                    if (r) toRemove.add(OWNER READ);
                    if (w) toRemove.add(OWNER WRITE);
                    if (x) toRemove.add(OWNER EXECUTE);
                }
                if (q) {
                    if (r) toRemove.add(GROUP READ);
                    if (w) toRemove.add(GROUP WRITE);
                    if (x) toRemove.add(GROUP EXECUTE);
                }
                if (0) {
                    if (r) toRemove.add(OTHERS READ);
                    if (w) toRemove.add(OTHERS WRITE);
                    if (x) toRemove.add(OTHERS EXECUTE);
                }
            }
            if (assign) {
                if (u) {
                    if (r) toAdd.add(OWNER READ);
                      else toRemove.add(OWNER READ);
                    if (w) toAdd.add(OWNER WRITE);
                      else toRemove.add(OWNER WRITE);
                    if (x) toAdd.add(OWNER EXECUTE);
                      else toRemove.add(OWNER EXECUTE);
                }
                if (g) {
                    if (r) toAdd.add(GROUP_READ);
                      else toRemove.add(GROUP READ);
                    if (w) toAdd.add(GROUP WRITE);
                      else toRemove.add(GROUP WRITE);
                    if (x) toAdd.add(GROUP EXECUTE);
                      else toRemove.add(GROUP EXECUTE);
                }
                if (o) {
                    if (r) toAdd.add(OTHERS READ);
                      else toRemove.add(OTHERS READ);
                    if (w) toAdd.add(OTHERS WRITE);
                      else toRemove.add(OTHERS WRITE);
                    if (x) toAdd.add(OTHERS EXECUTE);
                      else toRemove.add(OTHERS EXECUTE);
                }
            }
        }
        // return changer
        return new Changer() {
            @Override
            public Set<PosixFilePermission> change(Set<PosixFilePermission>
perms) {
```

```
perms.addAll(toAdd);
                perms.removeAll(toRemove);
                return perms;
            }
        };
    }
    /**
     * A task that <i>changes</i> a set of {@link PosixFilePermission}
elements.
    */
   public interface Changer {
        /**
         * Applies the changes to the given set of permissions.
         *
         * @param perms
                    The set of permissions to change
         *
         * @return The {@code perms} parameter
         */
        Set<PosixFilePermission> change(Set<PosixFilePermission> perms);
    }
    /**
    * Changes the permissions of the file using the given Changer.
    */
    static void chmod(Path file, Changer changer) {
       try {
            Set<PosixFilePermission> perms = Files
                .getPosixFilePermissions(file);
            Files.setPosixFilePermissions(file, changer.change(perms));
        } catch (IOException x) {
            System.err.println(x);
        }
    }
    /**
    * Changes the permission of each file and directory visited
    */
    static class TreeVisitor implements FileVisitor<Path> {
        private final Changer changer;
        TreeVisitor(Changer changer) {
            this.changer = changer;
        }
        @Override
        public FileVisitResult preVisitDirectory(Path dir,
BasicFileAttributes attrs) {
            chmod(dir, changer);
            return CONTINUE;
        }
        @Override
        public FileVisitResult visitFile(Path file, BasicFileAttributes
attrs) {
```

```
chmod(file, changer);
            return CONTINUE;
        }
        @Override
        public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
            if (exc != null)
                System.err.println("WARNING: " + exc);
            return CONTINUE;
        }
        @Override
        public FileVisitResult visitFileFailed(Path file, IOException exc) {
            System.err.println("WARNING: " + exc);
            return CONTINUE;
        }
    }
    static void usage() {
        System.err.println("java Chmod [-R] symbolic-mode-list file...");
        System.exit(-1);
    }
    public static void main(String[] args) throws IOException {
        if (args.length < 2)
            usage();
        int argi = 0;
        int maxDepth = 0;
        if (args[argi].equals("-R")) {
            if (args.length < 3)
                usage();
            argi++;
            maxDepth = Integer.MAX VALUE;
        }
        // compile the symbolic mode expressions
        Changer changer = compile(args[argi++]);
        TreeVisitor visitor = new TreeVisitor(changer);
        Set<FileVisitOption> opts = Collections.emptySet();
        while (argi < args.length) {</pre>
            Path file = Paths.get(args[argi]);
            Files.walkFileTree(file, opts, maxDepth, visitor);
            argi++;
        }
    }
}
```

Copy File NIO Example

This example copies files in a similar manner to the copy command.

```
public class Copy {
    /**
```

```
* Returns {@code true} if okay to overwrite a file ("cp -i")
     */
    static boolean okayToOverwrite(Path file) {
        String answer = System.console().readLine("overwrite %s (yes/no)? ",
file);
        return (answer.equalsIgnoreCase("y") ||
answer.equalsIgnoreCase("yes"));
    }
    /**
     * Copy source file to target location. If {@code prompt} is true then
     * prompt user to overwrite target if it exists. The {@code preserve}
     * parameter determines if file attributes should be copied/preserved.
     */
    static void copyFile(Path source, Path target, boolean prompt, boolean
preserve) {
        CopyOption[] options = (preserve) ?
            new CopyOption[] { COPY ATTRIBUTES, REPLACE EXISTING } :
            new CopyOption[] { REPLACE EXISTING };
        if (!prompt || Files.notExists(target) || okayToOverwrite(target)) {
            try {
                Files.copy(source, target, options);
            } catch (IOException x) {
                System.err.format("Unable to copy: %s: %s%n", source, x);
            }
        }
    }
    /**
     * A {@code FileVisitor} that copies a file-tree ("cp -r")
     */
    static class TreeCopier implements FileVisitor<Path> {
        private final Path source;
        private final Path target;
        private final boolean prompt;
        private final boolean preserve;
        TreeCopier(Path source, Path target, boolean prompt, boolean
preserve) {
            this.source = source;
            this.target = target;
            this.prompt = prompt;
            this.preserve = preserve;
        }
        @Override
        public FileVisitResult preVisitDirectory(Path dir,
BasicFileAttributes attrs) {
            // before visiting entries in a directory we copy the directory
            // (okay if directory already exists).
            CopyOption[] options = (preserve) ?
                new CopyOption[] { COPY ATTRIBUTES } : new CopyOption[0];
            Path newdir = target.resolve(source.relativize(dir));
            try {
                Files.copy(dir, newdir, options);
```

```
} catch (FileAlreadyExistsException x) {
                // ignore
            } catch (IOException x) {
                System.err.format("Unable to create: %s: %s%n", newdir, x);
                return SKIP SUBTREE;
            }
            return CONTINUE;
        }
        @Override
        public FileVisitResult visitFile(Path file, BasicFileAttributes
attrs) {
            copyFile(file, target.resolve(source.relativize(file)),
                     prompt, preserve);
            return CONTINUE;
        }
        @Override
        public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
            // fix up modification time of directory when done
            if (exc == null && preserve) {
                Path newdir = target.resolve(source.relativize(dir));
                try {
                    FileTime time = Files.getLastModifiedTime(dir);
                    Files.setLastModifiedTime(newdir, time);
                } catch (IOException x) {
                    System.err.format("Unable to copy all attributes to: %s:
%s%n", newdir, x);
            }
            return CONTINUE;
        }
        @Override
        public FileVisitResult visitFileFailed(Path file, IOException exc) {
            if (exc instanceof FileSystemLoopException) {
                System.err.println("cycle detected: " + file);
            } else {
                System.err.format("Unable to copy: %s: %s%n", file, exc);
            }
            return CONTINUE;
        }
    }
    static void usage() {
        System.err.println("java Copy [-ip] source... target");
        System.err.println("java Copy -r [-ip] source-dir... target");
        System.exit(-1);
    }
    public static void main(String[] args) throws IOException {
        boolean recursive = false;
        boolean prompt = false;
        boolean preserve = false;
        // process options
```

```
int argi = 0;
        while (argi < args.length) {</pre>
            String arg = args[argi];
            if (!arg.startsWith("-"))
                break;
            if (arg.length() < 2)
                usage();
            for (int i=1; i<arg.length(); i++) {</pre>
                char c = arg.charAt(i);
                switch (c) {
                    case 'r' : recursive = true; break;
                    case 'i' : prompt = true; break;
                    case 'p' : preserve = true; break;
                    default : usage();
                }
            }
            argi++;
        }
        // remaining arguments are the source files(s) and the target location
        int remaining = args.length - argi;
        if (remaining < 2)
            usage();
        Path[] source = new Path[remaining-1];
        int i=0;
        while (remaining > 1) {
            source[i++] = Paths.get(args[argi++]);
            remaining--;
        }
        Path target = Paths.get(args[argi]);
        // check if target is a directory
        boolean isDir = Files.isDirectory(target);
        // copy each source file/directory to target
        for (i=0; i<source.length; i++) {</pre>
            Path dest = (isDir) ? target.resolve(source[i].getFileName()) :
target;
            if (recursive) {
                // follow links when copying files
                EnumSet<FileVisitOption> opts =
EnumSet.of(FileVisitOption.FOLLOW LINKS);
                TreeCopier tc = new TreeCopier(source[i], dest, prompt,
preserve);
                Files.walkFileTree(source[i], opts, Integer.MAX VALUE, tc);
            } else {
                // not recursive so source must not be a directory
                if (Files.isDirectory(source[i])) {
                    System.err.format("%s: is a directory%n", source[i]);
                    continue;
                copyFile(source[i], dest, prompt, preserve);
            }
        }
```

}

Disk Usage File NIO Example

This example prints disk space information in a similar manner to the df command.

```
public class DiskUsage {
    static final long K = 1024;
    static void printFileStore(FileStore store) throws IOException {
        long total = store.getTotalSpace() / K;
        long used = (store.getTotalSpace() - store.getUnallocatedSpace()) / K;
        long avail = store.getUsableSpace() / K;
        String s = store.toString();
        if (s.length() > 20) {
            System.out.println(s);
            s = "";
        System.out.format("%-20s %12d %12d %12d \n", s, total, used, avail);
    }
    public static void main(String[] args) throws IOException {
        System.out.format("%-20s %12s %12s %12s \n", "Filesystem", "kbytes",
"used", "avail");
        if (args.length == 0) {
            FileSystem fs = FileSystems.getDefault();
            for (FileStore store: fs.getFileStores()) {
                printFileStore(store);
            }
        } else {
            for (String file: args) {
                FileStore store = Files.getFileStore(Paths.get(file));
                printFileStore(store);
            }
        }
    }
}
```

User-Defined File Attributes File NIO Example

This example lists, sets, retrieves, and deletes user-defined file attributes.

```
public class Xdd {
    static void usage() {
        System.out.println("Usage: java Xdd <file>");
        System.out.println(" java Xdd -set <name>=<value> <file>");
        System.out.println(" java Xdd -get <name> <file>");
        System.out.println(" java Xdd -del <name> <file>");
        System.exit(-1);
```

```
public static void main(String[] args) throws IOException {
       // one or three parameters
       if (args.length != 1 && args.length != 3)
           usage();
       Path file = (args.length == 1) ? Paths.get(args[0])
           : Paths.get(args[2]);
       // check that user defined attributes are supported by the file store
       FileStore store = Files.getFileStore(file);
       if (!store
           .supportsFileAttributeView(UserDefinedFileAttributeView.class)) {
           System.err.format(
               "UserDefinedFileAttributeView not supported on %s\n", store);
           System.exit(-1);
       }
       UserDefinedFileAttributeView view = Files.getFileAttributeView(file,
           UserDefinedFileAttributeView.class);
       // list user defined attributes
       if (args.length == 1) {
           System.out.println(" Size Name");
           System.out
                                                -----");
                .println("----- -----
           for (String name : view.list()) {
               System.out.format("%8d %s\n", view.size(name), name);
           }
           return;
        }
       // Add/replace a file's user defined attribute
       if (args[0].equals("-set")) {
           // name=value
           String[] s = args[1].split("=");
           if (s.length != 2)
               usage();
           String name = s[0];
           String value = s[1];
           view.write(name, Charset.defaultCharset().encode(value));
           return;
       }
       // Print out the value of a file's user defined attribute
       if (args[0].equals("-get")) {
           String name = args[1];
           int size = view.size(name);
           ByteBuffer buf = ByteBuffer.allocateDirect(size);
           view.read(name, buf);
           buf.flip();
System.out.println(Charset.defaultCharset().decode(buf).toString());
           return;
       }
```

}

```
// Delete a file's user defined attribute
if (args[0].equals("-del")) {
    view.delete(args[1]);
    return;
}
// option not recognized
usage();
}
```

}

10 Java Networking

The Java networking API provides classes for networking functionality, including addressing, classes for using URLs and URIs, socket classes for connecting to servers, networking security functionality, and more. It consists of these packages and modules:=

- java.net: Classes for implementing networking applications.
- java.net.http: Contains the API for the HTTP Client, which provides high-level client interfaces to HTTP (versions 1.1 and 2) and low-level client interfaces to WebSocket instances. See Java HTTP Client for more information about this API, including videos and sample code.

Note:

You can use the jwebserver tool for testing and debugging your client application.

- javax.net: Classes for creating sockets.
- javax.net.ssl: Secure socket classes.
- jdk.httpserver: Platform-specific APIs for building HTTP servers for educational and testing purposes, as well as the jwebserver tool for running a minimal HTTP server.
- jdk.net: Platform-specific socket options for the java.net and java.nio.channels socket classes.

Networking System Properties

You can set the following networking system properties in one of three ways:

- Using the -D option of the java command
- Using the System.setProperty(String, String) method
- Specifying them in the <code>\$JAVA_HOME/conf/net.properties</code> file. Note that you can specify only proxy-related properties in this file.

Unless specified otherwise, a property value is checked every time it's used.

See Networking Properties and the java.net.http and jdk.httpserver modules in the Java SE API Specification for more information.

HTTP Client Properties

Some of the following properties are subject to predefined minimum and maximum values that override any user-specified values. Note that the default value of boolean values is true if the property exists but has no value.



Property	Default Value	Description
jdk.httpclient.allowRestri ctedHeaders	No default value	A comma-separated list of normally restricted HTTP header names that users may set in HTTP requests or by user code in HttpRequest instances.
		By default, user code cannot set the following request headers: connection, content-length, expect, host, and upgrade. Yo can override this behavior with this property.
		Header names specified in this property are case-insensitive, and whitespace is ignored. Note that this property is intended for testing and not for real-world deployments. Protocol errors or other undefined behavior are likely to occur when using this property.
		There may be other headers that are restricted from being set depending on the context. This includes the Authorization header when the relevant HttpClient has an authenticator set. These restrictions cannot be overridden by this property.
jdk.httpclient.auth.retryl imit	3	The number of attempts the Basi authentication filter will attempt to retry a failed authentication.
jdk.httpclient.bufsize	16384 (16 kB)	The size to use for internal allocated buffers in bytes.
jdk.httpclient.connectionP oolSize	0	The maximum number of connections to keep in the HTTP/1.1 keep alive cache. A value of 0 means that the cache is unbounded.
jdk.httpclient.connectionW indowSize	2^26	The HTTP/2 client connection window size in bytes.
		The maximum size is 2^31-1. This value cannot be smaller that the stream window size.
jdk.httpclient.disableRetr yConnect	false	Whether automatic retry of connection failures is disabled. If false, then retries are attempted (subject to the retry limit).
jdk.httpclient.enableAllMe thodRetry	false	Whether it is permitted to automatically retry non-idempotent HTTP requests.

Table 10-1 HTTP Client Properties



Property	Default Value	Description	
jdk.httpclient.enablepush	1	Whether HTTP/2 push promis enabled. A value of 1 enables push promise; a value of 0 disables it.	
jdk.httpclient.hpack.maxhe adertablesize	16384 (16 kB)	The HTTP/2 client maximum HPACK header table size in bytes.	
jdk.httpclient.HttpClient. log	No default value	Enables high-level logging of various events through the Java Logging API (which is contained in the package java.util.logging). The value contains a comma-	
		<pre>separated list of any of the following items: errors requests headers content frames ssl trace channel You can append the frames item with a colon-separated list of any of the following items: control data window all Specifying an item adds it to the HTTP client's log. For example, it you specify the following value, then the Java Logging API logs all possible HTTP Client events: errors, requests, headers, f ames:control:data:window, sl,trace, channel Note that you can replace control:data:window with all. The name of the logger is jdk.httpclient.HttpClient.</pre>	
jdk.httpclient.keepalive.t imeout	30	The number of seconds to keep idle HTTP/1.1 connections alive in the keep alive cache.	

Table 10-1 (Cont.) HTTP Client Properties



Property	Default Value	Description
<pre>jdk.httpclient.keepalive.t imeout.h2</pre>	See the description	The number of seconds to keep idle HTTP/2 connections alive. If not set, then the jdk.httpclient.keepalive.t imeout setting is used.
jdk.httpclient.maxframesiz e	16384 (16 kB)	The HTTP/2 client maximum frame size in bytes. The server is not permitted to send a frame larger than this.
jdk.httpclient.maxLiteralW ithIndexing	512	The number of additions a server may request a client to make to the HPack dynamic table when decoding a set of headers.
		If this limit is exceeded, then the request fails with a protocol exception.
		If this property has a zero or negative value, then there's no limit.
jdk.httpclient.maxNonFinal Responses	8	The number of interim responses the client will accept before receiving a final response. An interim response is considered informational and is a response whose status is in the range [100, 199]. These responses are typically either handled internally or simply discarded by the implementation.
		If this limit is exceeded, then the request fails with a protocol exception.
		If this property has a zero or negative value, then there's no limit.
jdk.httpclient.maxstreams	100	The maximum number of concurrent HTTP/2 streams per connection.
jdk.httpclient.receiveBuff erSize	The operating system's default value	The HTTP client socket receive buffer size in bytes. See StandardSocketOptions. SO_RCVBUF.
jdk.httpclient.redirects.r etrylimit	5	The maximum number of attempts to send a HTTP request when redirected or any failure occurs for any reason.
jdk.httpclient.sendBufferS ize	Operating system default	The HTTP client socket send buffer size. See StandardSocketOptions. SO_SNDBUF. Values less than or equal to zero are ignored.

Table 10-1 (Cont.) HTTP Client Properties



Property	Default Value	Description	
jdk.httpclient.websocket.w riteBufferSize	16384 (16 kB)	The buffer size used by the web socket implementation for socket writes.	
jdk.httpclient.windowsize	16777216 (16 MB)	The HTTP/2 client stream window size in bytes.	
jdk.internal.httpclient.di sableHostnameVerification	false	If true (or set to an empty string), hostname verification in SSL certificates is disabled. This is a system property only and not available in conf/ net.properties. It is provided for testing purposes only.	

Table 10-1 (Cont.) HTTP Client Properties

Note:

The properties jdk.http.auth.proxying.disabledSchemes and jdk.http.auth.tunneling.disabledSchemes, described in Other Proxy-Related Properties, are also taken into account by HttpClient.

HTTP Server Properties

The following are JDK-specific system properties used by the default HTTP server implementation in the JDK. Any of these properties that take a numeric value assume the default value if given a string that does not parse as a number.

Property	Default Value	Description
jdk.http.maxHeaderSize	393216 (384 kB)	The maximum response header size that the JDK built-in implementation of the legacy URL protocol handler for HTTP, java.net.HttpURLConnec tion and the newer HTTP client, java.net.http.HttpClie nt, will accept from a remote party. This limit is computed as the cumulative size of all header names and header values plus an overhead of 32 bytes per header name-value pair.
		If this limit is exceeded, then the request fails with a protocol exception.
		If this property has a zero or negative value, then there's no limit.

Table 10-2 HTTP Server Properties



Property	Default Value	Description
jdk.httpserver.maxConnecti ons	-1	The maximum number of open connections at a time. This includes active and idle connections. If this property has a zero or negative value, then no limit is enforced.
<pre>sun.net.httpserver.drainAm ount</pre>	65536	The maximum number of bytes that will be automatically read and discarded from a request body that has not been completely consumed by its HttpHandler. If the number of remaining unread bytes are less than this limit, then the connection will be put in the idle connection cache. If not, then it will be closed.
<pre>sun.net.httpserver.idleInt erval</pre>	30	The maximum duration in seconds which an idle connection is kept open. This timer has an implementation-specific granularity that may mean that idle connections are closed later than the specified interval. If this property has a zero or negative value, then the default value is used.
<pre>sun.net.httpserver.maxIdle Connections</pre>	200	The maximum number of idle connections that may exist at the same time. If this property has a zero or negative value, then connections are closed after use.
<pre>sun.net.httpserver.maxReqH eaders</pre>	200	The maximum number of header fields accepted in a request. If this limit is exceeded while the headers are being read, then the connection is terminated and the request ignored. If this property has a zero or negative value, then the default value is used.
sun.net.httpserver.maxReqH eaderSize	393216 (384 kB)	The maximum request header size that the JDK built-in implementation of com.sun.net.httpserver .HttpServer will accept. This limit is computed the same way as jdk.http.maxHeaderSize. If the limit is exceeded. then the connection is closed. If this property has a zero or negative value, then there's no limit.

Table 10-2 (Cont.) HTTP Server Properties



Property	Default Value	Description
sun.net.httpserver.maxReqT ime	-1	The maximum time in milliseconds allowed to receive a request headers and body. In practice, the actual time is a function of request size, network speed, and handler processing delays. If this property has a zero or negative value, then the time i not limited. If the limit is exceeded, then the connection is terminated and the handler will receive an IOException. This timer has an implementation- specific granularity that may mean requests are aborted later than the specified interval.
sun.net.httpserver.maxRspT ime	-1	The maximum time in milliseconds allowed to receive a response headers and body. In practice, the actual time is a function of response size, network speed, and handler processing delays. If this propert has a zero or negative value, the the time is not limited. If the limit is exceeded then the connection is terminated and the handler wil receive an IOException. This timer has an implementation- specific granularity that may mean responses are aborted later than the specified interval.
sun.net.httpserver.nodelay	false	A boolean value, which if true, sets the TCP_NODELAY socket option on all incoming connections.

Table 10-2 (Cont.) HTTP Server Properties	Table 10-2	(Cont.)	HTTP	Server	Properties
---	------------	---------	------	--------	------------

IPv4 and IPv6 Protocol Properties

These two properties are checked only once, at startup.

Property Default Value		Description	
java.net.preferIPv4Stack	false	If IPv6 is available on the operating system, then the underlying native socket will be, by default, an IPv6 socket, which lets applications connect to, and accept connections from, both IPv4 and IPv6 hosts.	
		Set this property to true if you want your application use IPv4- only sockets. This implies that it won't be possible for the application to communicate with IPv6-only hosts.	
java.net.preferIPv6Address es	false	When dealing with a host which has both IPv4 and IPv6 addresses, and if IPv6 is available on the operating system, the default behavior is to prefer using IPv4 addresses ove IPv6 ones. This is to ensure backward compatibility, for example, for applications that depend on the representation of an IPv4 address (such as 192.168.1.1). Set this property to true to	
		change this preference and use IPv6 addresses over IPv4 ones where possible.	
		Set this property to system to preserve the order of the addresses as returned by the operating system.	

Table 10-3 IPv4 and IPv6 Protocol Properties

HTTP Proxy Properties

The following proxy settings are used by the HTTP protocol handler and the default proxy selector.

Table 10-4 HTTP Proxy Properties

Property	Default Value	Description
http.proxyHost	No default value	Proxy server that the HTTP protocol handler will use.
http.proxyPort	80	Port that the HTTP protocol handler will use.



Property	Default Value	Description
http.nonProxyHosts	localhost 127.* [::1]	Indicates the hosts that should be accessed without going through the proxy. Typically, this defines internal hosts. The value of this property is a list of hosts, separated by the vertical bar (1) character. In addition, you can use the asterisk (*) for pattern matching. For example, the following specifies that every host in the exmaple.com domain and localhost should be accessed directly even if a proxy server is specified: -
		<pre>mple.com/localhost" The default value evaluates all</pre>
		The default value excludes all common variations of the loopback address.

Table 10-4 (Cont.) HTTP Proxy Properties

HTTPS Proxy Properties

HTTPS, HTTP over SSL, is a secure version of HTTP mainly used when confidentiality is needed (such as payment web sites). The following proxy settings are used by the HTTPS protocol handler and the default proxy selector.

Note:

The HTTPS protocol handler uses the same http.nonProxyHosts property as the HTTP protocol.

Table 10-5 HTTPS Proxy Properties

Property	Default Value	Description
https.proxyHost	No default value	Proxy server that the HTTPS protocol handler will use.
https.proxyPort	443	Port that the HTTPS protocol handler will use.

FTP Proxy Properties

The following proxy settings are used by the FTP protocol handler.

System Property	Default Value	Description
ftp.proxyHost	No default value	Proxy server that the FTP protocol handler will use.
ftp.proxyPort	80	Port that the FTP protocol handler will use.
ftp.nonProxyHosts	localhost 127.* [::1]	Similar to http.nonProxyHosts, this property indicates the hosts that should be accessed without going through the proxy. Typically, this defines internal hosts. The value of this property is a list of hosts, separated by the vertical bar () character. In addition, you can use the asterisk (*) for pattern matching. For example, the following specifies that every hos in the exmaple.com domain and localhost should be accessed directly even if a proxy server is specified: - Dftp.nonProxyHosts="*.exam ple.com localhost" The default value excludes all common variations of the loopback address.

Table 10-6 FTP Proxy Properties

SOCKS Proxy Properties

The SOCKS proxy enables a lower-level type of tunneling because it works at the TCP level. Specifying a SOCKS proxy server results in all TCP connections going through that proxy server unless other proxies are specified. The following proxy settings are used by the SOCKS protocol handler.

Table 10-7	SOCKS Proxy Properties
------------	------------------------

Property	Default Value	Description
java.net.socks.username	No default value	See Acquiring the SOCKS User Name and Password
java.net.socks.password	No default value	See Acquiring the SOCKS User Name and Password
socksProxyHost	No default value	SOCKS proxy server that the SOCKS protocol handler will use.
socksProxyPort	1080	Port that the SOCKS protocol handler will use.



Property	Default Value	Description
socksProxyVersion	5	The version of the SOCKS protocol supported by the server. The default is 5 indicating SOCKS V5; alternatively 4 can be specified for SOCKS V4. Setting the property to values other than these leads to unspecified behavior.

Table 10-7 (Cont.) SOCKS Proxy Properties

Acquiring the SOCKS User Name and Password

The SOCKS user name and password are acquired in the following way:

- 1. First, if the application has registered a java.net.Authenticator default instance, then this will be queried with the protocol set to the string SOCKS5, and the prompt set to the string SOCKS authentication.
- 2. If the authenticator does not return a user name/password or if no authenticator is registered, then the system checks the values of properties java.net.socks.username and java.net.socks.password.
- 3. If these values don't exist, then the system property user.name is checked for a user name. In this case, no password is supplied.

Other Proxy-Related Properties

Property	Default Value	Description
jdk.http.auth.proxying.dis abledSchemes	See conf/net.properties	Lists the authentication schemes that will be disabled when proxying HTTP.
		The value of this property is a comma-separated list of case- insensitive authentication scheme names, as defined by their relevant RFCs. Schemes include Basic, Digest, NTLM, Kerberos, and Negotiate. A scheme that is not known or supported is ignored.
		In some environments, certain authentication schemes may be undesirable when proxying HTTP or HTTPS. For example, Basic results in effectively the cleartext transmission of the user's password over the physical network.

Table 10-8 Other Proxy-Related Properties



Property	Default Value	Description
jdk.http.auth.tunneling.di sabledSchemes	See conf/net.properties	Lists the authentication schemes that will be disabled when tunneling HTTPS over a proxy with the HTTP CONNECT method.
		The value of this property is a comma-separated list of case- insensitive authentication schem names, as defined by their relevant RFCs. Schemes include Basic, Digest, NTLM, Kerberos, and Negotiate. A scheme that is not known or supported is ignored.
java.net.useSystemProxies	false	If true, then the operating system's proxy settings are used
		Note that the system properties that explicitly set proxies like http.proxyHost take precedence over the system settings even if java.net.useSystemProxies is set to true.
		This property is checked only once, at startup.

Table 10-8 (Cont.) Other Proxy-Related Properties

UNIX Domain Sockets Properties

Calling ServerSocketChannel.bind with a null address parameter will bind the channel's socket to an automatically assigned socket address. For UNIX domain sockets, this means a unique path in some predefined system temporary directory.

Use these properties to control the selection of this directory:

Property	Default Value	Description
java.io.tmpdir	Dependent on the operating system	If the temporary directory can't be determined with the jdk.net.unixdomain.tmpdir system property, then the directory specified by the java.io.tmpdir system property is used.
jdk.net.unixdomain.tmpdir	On some platforms, (for example, some UNIX systems) this will have a predefined default value. On others, (for example, Windows) there is no default value.	Specifies the directory to use for automatically bound server socket addresses.

 Table 10-9
 UNIX Doman Sockets Properties



On Linux and macOS, the search order to determine this directory is as follows:

- The system property jdk.net.unixdomain.tmpdir (set on the command line or by System.setProperty(String, String))
- 2. The same property set in the \$JAVA HOME/conf/net.properties file
- 3. The system property java.io.tmpdir

On Windows, the search order to determine this directory is as follows:

- The system property jdk.net.unixdomain.tmpdir (set on the command line or by System.setProperty (String, String))
- 2. The same property set in the %JAVA HOME%\conf\net.properties file
- 3. The TEMP environment variable
- 4. The system property java.io.tmpdir

Because UNIX domain socket addresses are limited in length to approximately 100 bytes (depending on the platform), it is important to ensure that the temporary directory's name together with the file name used for the socket does not exceed this limit.

Note:

If a client socket is connected to a remote destination without calling bind first, then the socket is implicitly bound. In this case, UNIX domain sockets are unnamed (that is, their path is empty). This behavior is not affected by any system or networking properties.

Other HTTP URL Stream Protocol Handler Properties

These properties are checked only once, at startup.

Table 10-10 Other HTTP ORL Stream Protocol Handler Properties	Table 10-10	Other HTTP URL Stream Protocol Handler Properties
---	-------------	---

Property	Default Value	Description
http.agent	Java/ <version></version>	Defines the string sent in the User-Agent request header in HTTP requests. Note that the string Java/ <version> will be appended to the one provided in the property. For example, if - Dhttp.agent="example" is specified, the User-Agent header will contain example Java/ 1.8.0 if the version of the JVM is 1.8.0).</version>
http.auth.digest.cnonceRep eat	5	See System Properties That Modify the Behavior of HTTP Digest Authentication Mechanism.



Property	Default Value	Description
http.auth.digest.reEnabled Algorithms	No default value	By default, certain message digest algorithms are disabled for use in HTTP Digest authentication due to their prover security limitations. This only applies to proxy authentication and plain-text HTTP server authentication. Disabled algorithms are still usable for HTTPS server authentication. The default list of disabled algorithms is specified in the java.security properties file and currently comprises MD5 and SHA-1. If it is still required to use one of these algorithms, then they can be re-enabled by setting this property to a comma- separated list of the algorithm names.
<pre>http.auth.digest.validateP roxy</pre>	false	See System Properties That Modify the Behavior of HTTP Digest Authentication Mechanism.
http.auth.digest.validateS erver	false	See System Properties That Modify the Behavior of HTTP Digest Authentication Mechanism.

Property	Default Value	Description
http.auth.ntlm.domain	No default value	Similar to other HTTP authentication schemes, New Technology LAN Manager (NTLM) uses the java.net.Authenticator class to acquire user names and passwords when they are needed. However, NTLM also needs the NT domain name. There are three options for specifying the domain:
		 Do not specify it. In some environments, the domain is not actually required and the application does not have to specify it.
		 The domain name can be encoded within the user name by prefixing the domain name followed by a backslash (\) before the use name. With this method, existing applications that us the Authenticator class do not need to be modified, as long as users are made aware that this notation must be used.
		3. If a domain name is not specified as in the second option and the system property http.auth.ntlm.domain is defined, then the value of this property will be used as the domain name.
http.keepAlive	true	Indicates if persistent (keep-alive connections should be supporte They improve performance by allowing the underlying socket connection to be reused for multiple HTTP requests. If this is set to true, then persistent connections will be requested with HTTP 1.1 servers.
		Set this property to false to disable the use of persistent connections.
http.KeepAlive.queuedCon ctions	ne 10	The maximum number of keep- alive connections to be on the queue for clean up.



Property	Default Value	Description
http.KeepAlive.remainingDa ta	512	The maximum amount of data in kilobytes that will be cleaned off the underlying socket so that it can be reused.
http.keepAlive.time.server and http.keepAlive.time.proxy	5	These properties modify the behavior of the HTTP keepalive cache in the case where the server (or proxy) has not specified a keepalive time. If the property is set in this case, then idle connections will be closed after the specified number of seconds. If the property is set, and the server does specify a keepalive time in a "Keep-Alive" response header, then the time specified by the server is used. If the property is not set and also the server does not specify a keepalive time, then connections are kept alive for an implementation defined time, assuming http.keepAlive is true.
http.maxConnections	5	If HTTP persistent connections (see the http.keepAlive property) are enabled, then this value determines the maximum number of idle connections that will be simultaneously kept alive per destination.
http.maxRedirects	20	Integer value that determines the maximum number, for a given request, of HTTP redirects that will be automatically followed by the protocol handler.

Property	Default Value	Description
jdk.https.negotiate.cbt	never	Controls the generation and sending of TLS channel binding tokens (CBT) when Kerberos or the Negotiate authentication scheme using Kerberos are employed over HTTPS with HttpsURLConnection. There are three possible settings • never: This is also the
		 default value if the property is not set. In this case, CBTs are never sent. always: CBTs are sent for all Kerberos authentication attempts over HTTPS. domain:<comma- separated domain list> Each domain in the list specifies the destination hos or hosts for which a CBT is sent. Domains can be: Single hosts like example or example.com </comma- Literal IP addresses as specified in RFC 2732 Hostnames that contain wildcards like *.example.com; this example matches all hosts under example.com and its
		subdomains. The channel binding tokens generated are of the type tls- server-end-point as defined in RFC 5929.



Property	Default Value	Description
jdk.http.ntlm.transparentA uth	No default value	Enables transparent New Technology LAN Manager (NTLM) HTTP authentication on Windows.
		Transparent authentication can be used for the NTLM scheme, where the security credentials based on the currently logged in user's name and password can be obtained directly from the operating system, without prompting the user.
		If this value is not set, then transparent authentication is never used.
		This property has three possible values:
		 disabled: Transparent authentication is never used
		 allHosts: Transparent. authentication is used for all hosts
		 trustedHosts: Transparer authentication is enabled for hosts that are trusted in Windows Internet settings.
		Note that NTLM is not a strongly secure authentication scheme; care should be taken before enabling it.

System Properties That Modify the Behavior of HTTP Digest Authentication Mechanism

The system properties http.auth.digest.validateServer and

http.auth.digest.validateProxy modify the behavior of the HTTP digest authentication mechanism. Digest authentication provides a limited ability for the server to authenticate itself to the client (that is, by proving that it knows the user's password). However, not all servers support this capability and by default the check is switched off. To enforce this check for authentication with an origin, set http.auth.digest.validateServer to true; with a proxy server, set http.auth.digest.validateProxy to true.

It is usually not necessary to set the system property http://www.auth.digest.cnonceRepeat. This determines how many times a cnonce value is reused. This can be useful when the MD5-sess algorithm is being used. Increasing the value reduces the computational overhead on both the client and the server by reducing the amount of material that has to be hashed for each HTTP request.

Specify Mappings from Host Names to IP Addresses

You can customize the mapping from host names to IP addresses by deploying a system-wide resolver. See the InetAddressResolverProvider class in the Java SE API Specification for more information. In cases where this is not practical, such as testing, you can configure InetAddress to use a specific hosts file, rather than the system-wide resolver, to map host names to IP addresses. Specify this hosts file with the system property jdk.net.hosts.file.

Note:

Use a specific hosts file for testing; it's not intended as a general purpose solution because the complete list of host names is not always known in advance.

By default, the system property jdk.net.hosts.file is not set. If it's set, then name service lookups are obtained from the file specified by this system property. If this system property specifies a file that doesn't exist, then it treats it as an empty file, and a name/address lookup throws an UnknownHostException.

The structure of the hosts file is similar to a Linux or macOS /etc/hosts file. Each line of this text file has the following syntax:

IPAddress hostname [host aliases...]

- IPAddress: IP address
- hostname: Host name to which the IP address is mapped
 - A host name should have the syntax and structure of a fully qualified domain name (FQDN), composed of alphanumeric characters, hyphens (-), and periods (.). It should begin and end with an alphanumeric character.
 - Note that no syntax checking or host name validation is performed.
- [host aliases...]: An optional list of host aliases

The fields of an entry are separated by any number of whitespace (spaces and tabs).

A comment, which starts with a number sign (#) and followed by text until the end of the line, is ignored.

The following is an example of a hosts file:

```
# sample jdk.net.hosts.file entries
127.0.0.1 localhost
127.0.0.1 host.rabbit.hole
127.0.0.1 c1.this.domain
192.0.2.0 testhost.testdomain
192.0.2.255 testhost2.testdomain
```

Address Cache Properties

The java.net package, when performing name resolution, uses an address cache for both security and performance reasons. Any address resolution attempt, be it forward (name to IP address) or reverse (IP address to name), will have its result cached, whether it was successful or not, so that subsequent identical requests will not have to access the naming service. These properties enable you to tune how the address cache operates.



Note:

The following properties are part of the security policy. They are not set by the -D option or the <code>System.setProperty()</code> method. Instead, they are set as Security Properties.

Table 10-11 Address Cache Properties

Property	Default Value	Description
networkaddress.cache.ttl	-1	Specified in the \$JAVA_HOME/ conf/security/ java.security file to indicate the caching policy for successful name lookups from the name service. The value is a integer corresponding to the number of seconds successful name lookups will be kept in the cache.
		A value of -1 (or any other negative value) indicates a "cache forever" policy, while a value of 0 (zero) means no caching.
		The default value is -1 (forever) a security manager is installed and implementation-specific if no security manager is installed.
networkaddress.cache.negat ive.ttl	: 10	Specified in the \$JAVA_HOME/ conf/security/ java.security file to indicate the caching policy for unsuccessful name lookups from the name service. The value is an integer corresponding to the number of seconds an unsuccessful name lookup will be kept in the cache. A value of -1 (or any negative value) means "cache forever," while a value of 0 (zero) means no caching.

Enhanced Exception Messages

By default, for security reasons, exception messages do not include potentially sensitive security information such as hostnames or UNIX domain socket address paths. Use the jdk.includeInExceptions to relax this restriction for debugging and other purposes.



Property	Default Value	Description
jdk.includeInExceptions	No default value	The value is a omma-separated list of keywords that refer to exception types whose messages may be enhanced with more detailed information.
		In particular, if the value includes the string hostInfo, then socket addresses will be included in exception message texts (for example, hostnames and UNIX domain socket address paths).

	Table 10-12	Enhanced Exception Messages Property
--	-------------	--------------------------------------

11 Pseudorandom Number Generators

Random number generators included in Java SE are more accurately called pseudorandom number generators (PRNGs). They create a series of numbers based on a deterministic algorithm.

The most important interfaces and classes are RandomGenerator, which enables you to generate random numbers of various primitive types given a PRNG algorithm, and RandomGeneratorFactory, which enables you to create PRNGs based on characteristics other than the algorithm's name.

See the java.util.random package for more detailed information about the PRNGs implemented in Java SE.

Topics

- Characteristics of PRNGs
- Generating Pseudorandom Numbers with RandomGenerator Interface
- Generating Pseudorandom Numbers in Multithreaded Applications
 - Dynamically Creating New Generators
 - Creating Stream of Generators
- Choosing a PRNG Algorithm

Characteristics of PRNGs

Because PRNGs generate a sequence of values based on an algorithm instead of a "random" physical source, this sequence will eventually restart. The number of values a PRNG generates before it restarts is called a period.

The state cycle of a PRNG consists of the sequence of all possible values a PRNG can generate. The state of a PRNG is the position of the last generated value in its state cycle.

In general, to generate a value, the PRNG bases it on the previously generated value. However, some PRNGs can generate a value many values further down the sequence without calculating any intermediate values. These are called jumpable PRNGs because they could jump far ahead in the sequence of values, usually by a fixed distance, typically 2⁶⁴. A leapable PRNG can jump even further, typically 2¹²⁸ values. An arbitrarily jumpable PRNG can jump to any value in the generated sequence of values.

The java.util.Random Class Compared to Other PRNGs

The java.util.random.RandomGeneratorFactory class enables you to create various PRNGs, many of which are in the jdk.random package. The most significant difference between the PRNGs in jdk.random and the java.util.Random class is that Random has a very short period: only 2⁴⁸ values.



Generating Pseudorandom Numbers with RandomGenerator Interface

The following example demonstrates the basic way to create a PRNG and use it to generate a random number:

```
RandomGenerator random1 = RandomGenerator.of("Random");
long value1 = random1.nextLong();
System.out.println(value1);
```

It uses the method RandomGenerator.of (String). The argument of this method is the algorithm name of the PRNG. Java SE contains many PRNG classes. Unlike Random, however, most of them are in the jdk.random package.

The RandomGenerator interface contains many methods such as nextLong(), nextInt(), nextDouble(), and nextBoolean() to generate a random number of various primitive data types.

The following example demonstrates how to create a PRNG using the RandomGeneratorFactory Class:

```
RandomGeneratorFactory<RandomGenerator> factory2 =
    RandomGeneratorFactory.of("SecureRandom");
RandomGenerator random2 = factory2.create();
long value2 = random2.nextLong();
System.out.println(value2);
```

To obtain a list of PRNGs implemented by Java SE, call the RandomGeneratorFactory.all() method:

```
RandomGeneratorFactory.all()
.map(f -> f.name())
.sorted()
.forEach(n -> System.out.println(n));
```

This method returns a stream of all the available RandomGeneratorFactory instances available.

You can use the RandomGeneratorFactory class to create PRNGs based on characteristics other than an algorithm's name. The following example finds the PRNG with the longest period, and creates a RandomGeneratorFactory based on this characteristic:

```
RandomGeneratorFactory<RandomGenerator> greatest =
    RandomGeneratorFactory
    .all()
    .sorted((f, g) -> g.period().compareTo(f.period()))
    .findFirst()
    .orElse(RandomGeneratorFactory.of("Random"));
System.out.println(greatest.name());
System.out.println(greatest.group());
System.out.println(greatest.create().nextLong());
```

Generating Pseudorandom Numbers in Multithreaded Applications

If multiple threads in your application are generating sequences of values using PRNGs, then you want to ensure that there's no chance that these sequences contain values that coincide with each other, especially if they're using the same PRNG algorithm. (You would want to use the same PRNG algorithm to ensure that all your application's pseudorandom number sequences have the same statistical properties.) Splittable, jumpable, and leapable PRNGs are ideal for this; they can create a stream of generators that have the same statistical properties and are statistically independent.

There are two techniques you can use to incorporate PRNGs into your applications. You can dynamically create a new generator when an application needs to fork a new thread. Alternatively, you can create a stream of RandomGenerator objects based on an initial RandomGenerator, then map each RandomGenerator object from the stream to its own thread.

Dynamically Creating New Generators

If you're using a PRNG that implements the RandomGenerator.SplittableGenerator interface, then when a thread running in your application needs to fork a new thread, call the split() method. It creates a new generator with the same properties as the original generator. It does this by partitioning the original generator's period into two; each partition is for the exclusive use of either the original or new generator.

The following example uses the L128X1024MixRandom PRNG, which implements the RandomGenerator.SplittableGenerator interface. The IntStream processes stream represents tasks intended to be run on different threads.

```
int NUM_PROCESSES = 100;
RandomGeneratorFactory<SplittableGenerator> factory =
    RandomGeneratorFactory.of("L128X1024MixRandom");
SplittableGenerator random = factory.create();
IntStream processes = IntStream.rangeClosed(1, NUM_PROCESSES);
processes.parallel().forEach(p -> {
    RandomGenerator r = random.split();
    System.out.println(p + ": " + r.nextLong());
});
```

Splittable PRNGs generally have large periods to ensure that new objects resulting from a split use different state cycles. But even if two instances "accidentally" use the same state cycle, they are highly likely to traverse different regions of that shared state cycle.

Creating Stream of Generators

If the initial generator implements the interface

RandomGenerator.StreamableGenerator, then call the method rngs(), jumps() (for jumpable generators), or leaps() (for leapable generators) to create a stream of generators. Call the map() method on the stream to assign each generator to its own thread.

When you call the jumps() method, the generator changes its state by jumping forward a large fixed distance within its state cycle, then creates a new generator based on the generator's new state. The generator repeatedly jumps and creates generators, creating a stream of generators. The leaps() method is similar; the size of the jump is much larger.

The following example creates a jumpable generator, then creates a stream of generators based on this initial generator by calling the jumps() method. The first several generators in the stream (defined by NUM_TASKS) are wrapped in a Task instance, then each Task is run in its own thread.

```
int NUM TASKS = 10;
RandomGeneratorFactory<JumpableGenerator> factory =
     RandomGeneratorFactory.of("Xoshiro256PlusPlus");
JumpableGenerator random = factory.create();
class Task implements Runnable {
    private int p;
     private RandomGenerator r;
     public Task(RandomGenerator prng) {
         r = prng;
     }
     public void run() {
         System.out.println(r.nextLong());
     }
 }
List<Thread> taskList = random
     .jumps()
     .limit(NUM TASKS)
     .map(prng -> new Thread(new Task(prng)))
     .collect(Collectors.toList());
taskList.stream().forEach(t -> t.start());
```

Choosing a PRNG Algorithm

For applications (such as physical simulation, machine learning, and games) that don't require a cryptographically secure algorithm, the java.util.random package provides multiple implementations of interface RandomGenerator that focus on one or more PRNG properties, which include speed, space, period, accidental correlation, and equidistribution.

Note:

As PRNG algorithms evolve, Java SE may add new PRNG algorithms and deprecate older ones. It's recommended that you don't use deprecated algorithms; they may be removed from a future Java SE release. Check if an algorithm has been deprecated by calling either the RandomGenerator.isDeprecated() or RandomGeneratorFactory.isDeprecated() method.

Cryptographically Secure

For applications that require a random number generator algorithm that is cryptographically secure, use the SecureRandom class in the java.security package.

See The SecureRandom Class in Java Platform, Standard Edition Security Developer's Guide for more information.

General Purpose

For applications with no special requirements, L64X128MixRandom balances speed, space, and period well. It's suitable for both single-threaded and multithreaded applications when used properly (a separate instance for each thread).

Single-Threaded, High Performance

For single-threaded applications, Xoroshiro128PlusPlus is small, fast, and has a sufficiently long period.

32-Bit Applications

For applications running in a 32-bit environment and using only one or a small number of threads, L32X64StarStarRandom or L32X64MixRandom are good choices.

Multithreaded Applications with Static Threads

For applications that use many threads that are allocated in one batch at the start of computation, consider a jumpable generator such as Xoroshiro128PlusPlus or Xoshiro256PlusPlus or a splittable generator such as L64X128MixRandom or L64X256MixRandom. If your application uses only floating-point values from a uniform distribution where no more than 32 bits of floating-point precision is required and exact equidistribution is not required, then MRG32k3a, a classic and well-studied algorithm, may be appropriate.

Multithreaded Applications with Dynamic Threads

For applications that create many threads dynamically, perhaps through the use of spliterators, a splittable generator such as L64X128MixRandom or L64X256MixRandom is recommended.

If the number of generators created dynamically may be very large (millions or more), then using generators such as L128X128MixRandom or L128X256MixRandom will make it much less likely that two instances use the same state cycle.

Tuples of Consecutively Generated Values

For applications that use tuples of consecutively generated values, consider a generator that is k-equidistributed such that k is at least as large as the length of the tuples being generated. For example, the generator L64X256MixRandom is shown to be 4-equidistributed, which means that you can have a sequence of tuples that contain four values, and these tuples will be uniformly distributed (there's an equal chance that any 4-tuple will appear in the sequence). It's also shown that L64X1024MixRandom is 16-equidistributed.

Large Permutations

For applications that generate large permutations, consider a generator whose period is much larger than the total number of possible permutations; otherwise, it will be impossible to generate some of the intended permutations. For example, if the goal is to shuffle a deck of 52 cards, the number of possible permutations is 52! (52 factorial), which is approximately 2^{225.58}, so it may be best to use a generator whose period is roughly 2²⁵⁶ or larger, such as L64X256MixRandom, L64X1024MixRandom, L128X256MixRandom, or L128X1024MixRandom.



12 Foreign Function and Memory API

The Foreign Function and Memory (FFM) API enables Java programs to interoperate with code and data outside the Java runtime. This API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI. The API invokes *foreign functions*, code outside the JVM, and safely accesses *foreign memory*, memory not managed by the JVM.

For background information about the FFM API, see JEP 454.

The FFM API is contained in the package java.lang.foreign.

Topics

- On-Heap and Off-Heap Memory
- Memory Segments and Arenas
- Calling a C Library Function with the Foreign Function and Memory API
- Upcalls: Passing Java Code as a Function Pointer to a Foreign Function
- Foreign Functions That Return Pointers
- Memory Layouts and Structured Access
- Checking for Native Errors Using errno
- Slicing Allocators and Slicing Memory Segments
- Restricted Methods
- Calling Native Functions with jextract

On-Heap and Off-Heap Memory

On-heap memory is memory in the Java heap, which is a region of memory managed by the garbage collector. Java objects reside in the heap. The heap can grow or shrink while the application runs. When the heap becomes full, garbage collection is performed: The JVM identifies the objects that are no longer being used (unreachable objects) and recycles their memory, making space for new allocations.

Off-heap memory is memory outside the Java heap. To invoke a function or method from a different language such as C from a Java application, its arguments must be in off-heap memory. Unlike heap memory, off-heap memory is not subject to garbage collection when no longer needed. You can control how and when off-heap memory is deallocated.

You interact with off-heap memory through a MemorySegment object. You allocate a MemorySegment object with an *arena*, which enables you to specify *when* the off-heap memory associated with the MemorySegment object is deallocated.

Memory Segments and Arenas

You can access off-heap or on-heap memory with the Foreign Function and Memory (FFM) API through the MemorySegment interface. Each memory segment is associated with, or *backed by*, a contiguous region of memory. There are two kinds of memory segments:

- Heap segment: This is a memory segment backed by a region of memory inside the Java heap, an *on-heap* region.
- Native segment: This is a memory segment backed by a region of memory outside the Java heap, an *off-heap* region. The examples in this chapter demonstrate how to allocate and access native segments.

An *arena* controls the lifecycle of native memory segments. To create an arena, use one of the methods in the <u>Arena</u> interface, such as <u>Arena.ofConfined()</u>. You use an arena to allocate a memory segment. Each arena has a *scope*, which specifies when the region of memory that backs the memory segment will be deallocated and is no longer valid. A memory segment can only be accessed if the scope associated with it is still valid or *alive*.

Most of the examples described in this chapter use a *confined arena*, which is created with Arena::ofConfined. A confined arena provides a bounded and deterministic lifetime. Its scope is alive from when it's created to when it's closed. A confined arena has an owner thread. This is typically the thread that created it. Only the owner thread can access the memory segments allocated in a confined arena. You'll get an exception if you try to close a confined arena with a thread other than the owner thread.

There are other kinds of arenas:

- A shared arena, which is created with Arena::ofShared, has no owner thread. Multiple threads may access the memory segments allocated in a shared arena. In addition, any thread may close a shared arena, and the closure is guaranteed to be safe and atomic. See Slicing Memory Segments for an example of a shared arena.
- An automatic arena, which is created with Arena::ofAuto. This is an area that's managed, automatically, by the garbage collector. Any thread can access memory segments allocated by an automatic arena. If you call Arena::close on an automatic arena, you'll get a UnsupportedOperationException.
- A global arena, which is created with Arena::global. Any thread can access memory segments allocated with this arena. In addition, the region of memory of these memory segments is never deallocated; if you call Arena::close on a global arena, you'll get a UnsupportedOperationException.

The following example allocates a memory segment with an arena, stores a Java String in the off-heap memory associated with the memory segment, and then prints the contents of the off-heap memory. At the end of the try-with-resources block, the arena is closed, and the off-heap memory associated with the memory segment is deallocated.

```
String s = "My string";
try (Arena arena = Arena.ofConfined()) {
    // Allocate off-heap memory
    MemorySegment nativeText = arena.allocateFrom(s);
    // Access off-heap memory
    for (int i = 0; i < s.length(); i++ ) {
        System.out.print((char)nativeText.get(ValueLayout.JAVA_BYTE, i));
    }
} // Off-heap memory is deallocated
```



The following sections describe this example in detail:

- Allocating a Memory Segment with an Arena and Storing a String in It
- Printing the Contents of Off-Heap Memory
- Closing an Arena

Allocating a Memory Segment with an Arena and Storing a String in It

The Arena interface extends the SegmentAllocator interface, which contains methods that both allocate off-heap memory and copy Java data into it. The previous example calls the method SegmentAllocator.allocateFrom(String), which allocates a memory segment with an arena, converts a string into a UTF-8 encoded, null-terminated C string, and then stores the string into the memory segment.

```
String s = "My string";
try (Arena arena = Arena.ofConfined()) {
    // Allocate off-heap memory
    MemorySegment nativeText = arena.allocateFrom(s);
    // ...
```

Tip:

You can call SegmentAllocator.allocateFrom(String, Charset) to store a string with a different charset. The SegmentAllocator interface contains several allocateFrom methods that enable you to store data of various data types in a memory segment.

See Memory Layouts and Structured Access for information about allocating and accessing more complicated native data types such as C structures.

Printing the Contents of Off-Heap Memory

The following code prints the characters stored in the MemorySegment named nativeText:

```
// Access off-heap memory
for (int i = 0; i < s.length(); i++ ) {
    System.out.print((char)nativeText.get(ValueLayout.JAVA_BYTE,
i));
}</pre>
```

The MemorySegment interface contains various access methods that enable you to read from or write to memory segments. Each access method takes as an argument a *value layout*, which models the memory layout associated with values of basic data types such as primitives. A value layout encodes the size, the endianness or byte order, the bit alignment of the piece of memory to be accessed, and the Java type to be used for the access operation.

For example, MemoryLayout.get (ValueLayout.OfByte, long) takes as an argument ValueLayout.JAVA BYTE. This value layout has the following characteristics:

The same size as a Java byte

- Byte alignment set to 1: This means that the memory layout is stored at a memory address that's a multiple of 8 bits.
- Byte order set to ByteOrder.nativeOrder(): A system can order the bytes of a multibyte value from most significant to least significant (big-endian) or from least significant to most significant (little-endian).

Closing an Arena

When an arena is closed, such as through a try-with-resources statement, then the arena's scope is no longer alive: All memory segments associated with its scope are invalidated, and the memory regions backing them are deallocated.

If you try to access a memory segment associated with an arena scope that's closed, you'll get an IllegalStateException, which the following example demonstrates:

```
String s = "My String";
MemorySegment nativeText;
try (Arena arena = Arena.ofConfined()) {
    // Allocate off-heap memory
    nativeText = arena.allocateFrom(s);
}
for (int i = 0; i < s.length(); i++ ) {
    // Exception in thread "main" java.lang.IllegalStateException:
Already closed
    System.out.print((char)nativeText.get(ValueLayout.JAVA_BYTE, i));
}
```

Calling a C Library Function with the Foreign Function and Memory API

The following example calls strlen with the Foreign Function and Memory API:

```
static long invokeStrlen(String s) throws Throwable {
  try (Arena arena = Arena.ofConfined()) {
    // Allocate off-heap memory and
    // copy the argument, a Java string, into off-heap memory
    MemorySegment nativeString = arena.allocateFrom(s);
    // Link and call the C function strlen
    // Obtain an instance of the native linker
    Linker linker = Linker.nativeLinker();
    // Locate the address of the C function signature
    SymbolLookup stdLib = linker.defaultLookup();
    MemorySegment strlen_addr = stdLib.find("strlen").get();
    // Create a description of the C function
    FunctionDescriptor strlen sig =
```



The following is the declaration of the strlen C standard library function:

```
size t strlen(const char *s);
```

It takes one argument, a string, and returns the length of the string. To call this function from a Java application, you would follow these steps:

- 1. Allocate off-heap memory, which is memory outside the Java runtime, for the strlen function's argument.
- 2. Store the Java string in the off-heap memory that you allocated.

The invokeStrlen example performs the previous step and this step with the following statement:

MemorySegment nativeString = arena.allocateFrom(s);

3. Build and then call a method handle that points to the strlen function. The topics in this section show you how to do this.

The following sections describe this example in detail:

- Obtaining an Instance of the Native Linker
- Locating the Address of the C Function
- Describing the C Function Signature
- Creating the Downcall Handle for the C Function
- Calling the C Function Directly from Java

Obtaining an Instance of the Native Linker

The following statement obtains an instance of the *native linker*, which provides access to the libraries that adhere to the calling conventions of the platform in which the Java runtime is running. These libraries are referred to as "native" libraries.

Linker linker = Linker.nativeLinker();

Locating the Address of the C Function

To call a native method such as strlen, you need a *downcall method handle*, which is a MethodHandle instance that points to a native function. This instance requires the native



function's address. To obtain this address, you use a *symbol lookup*, which enables you to retrieve the address of a symbol (such as the strlen function) in one or more libraries.

The following statements obtain the address of the strlen function:

```
// Obtain an instance of the C function
Linker linker = Linker.nativeLinker();
// Locate the address of the C function signature
SymbolLookup stdLib = linker.defaultLookup();
MemorySegment strlen addr = stdLib.find("strlen").get();
```

Because strlen is part of the C standard library, this example uses the native linker's *default lookup* by calling Linker.defaultLookup(). A default lookup is a symbol lookup for symbols in a set of commonly used libraries (including the C standard library).

Note:

Call the method SymbolLookup.libraryLookup (String, Arena) to create a symbol lookup from the name of a library. This method loads the specified library and associates it with an arena, which controls the symbol lookup's lifetime. The following example specifies <code>libc.so.6</code>, which is the name of the C standard library for many Linux systems.

```
SymbolLookup stdLib =
SymbolLookup.libraryLookup("libc.so.6", arena);
MemorySegment strlen addr = stdLib.find("strlen").get();
```

🖓 Tip:

Call SymbolLookup.loaderLookup() to find symbols in libraries that are loaded with System.loadLibrary(String).

Describing the C Function Signature

A downcall method handle also requires a description of the native function's signature, which is represented by a FunctionDescriptor instance. A *function descriptor* describes the layouts of the native function's arguments and its return value, if any.

Each layout in a function descriptor maps to a Java type, which is the type that should be used when invoking the resulting downcall method handle. Most value layouts map to a Java primitive type. For example, ValueLayout.JAVA_INT maps to an int value. However, ValueLayout.ADDRESS maps to a pointer.

Composite types such as struct and union types are modeled with the GroupLayout interface, which is a supertype of StructLayout and UnionLayout. See Memory Layouts and Structured Access for an example of how to initialize and access a C structure.



The following creates a function descriptor for the strlen function:

```
// Create a description of the C function signature
FunctionDescriptor strlen_sig =
FunctionDescriptor.of(ValueLayout.JAVA_LONG,
ValueLayout.ADDRESS);
```

The first argument of the FunctionDescriptor::of method is the layout of the native function's return value. Native primitive types are modeled using value layouts whose size matches that of such types. This means that a function descriptor is platform-specific. For example, size_t has a layout of JAVA_LONG on 64-bit or x64 platforms but a layout of JAVA INT on 32-bit or x86 platforms.

Tip: To determine the layout of a native primitive type that the native linker uses for your platform, call the method Linker::canonicalLayouts.

The subsequent arguments of FunctionDescriptor::of are the layouts of the native function's arguments. In this example, there's only one subsequent argument, a ValueLayout.ADDRESS. This represents the only argument for strlen, a pointer to a string.

Creating the Downcall Handle for the C Function

The following statement creates a downcall method handle for the strlen function with its address and function descriptor.

```
// Create a downcall handle for the C function
MethodHandle strlen = linker.downcallHandle(strlen_addr,
strlen sig);
```

Calling the C Function Directly from Java

The following statement calls the strlen function with a memory segment that contains the function's argument:

// Call the C function directly from Java
return (long)strlen.invokeExact(nativeString);

You need to cast a method handle invocation with the expected return type; in this case, it's long.

Upcalls: Passing Java Code as a Function Pointer to a Foreign Function

An *upcall* is a call from native code back to Java code. An *upcall stub* enables you to pass Java code as a function pointer to a foreign function.



Consider the standard C library function qsort, which sorts the elements of an array:

It takes four arguments:

- base: Pointer to the first element of the array to be sorted
- nbemb: Number of elements in the array
- size: Size, in bytes, of each element in the array
- compar: Pointer to the function that compares two elements

The following example calls the <code>qsort</code> function to sort an <code>int</code> array. However, this method requires a pointer to a function that compares two array elements. The example defines a comparison method named <code>Qsort::qsortCompare</code>, creates a method handle to represent this comparison method, and then creates a function pointer from this method handle.

```
import java.lang.foreign.*;
import java.lang.invoke.*;
import java.lang.foreign.ValueLayout.*;
public class InvokeQsort {
    class Qsort {
        static int qsortCompare(MemorySegment elem1, MemorySegment elem2) {
            return Integer.compare(elem1.get(ValueLayout.JAVA INT, 0),
elem2.get(ValueLayout.JAVA INT, 0));
        }
    }
    // Obtain instance of native linker
    final static Linker linker = Linker.nativeLinker();
    static int[] qsortTest(int[] unsortedArray) throws Throwable {
        int[] sorted = null;
        // Create downcall handle for qsort
        MethodHandle gsort = linker.downcallHandle(
            linker.defaultLookup().find("gsort").get(),
            FunctionDescriptor.ofVoid(ValueLayout.ADDRESS,
                                      ValueLayout.JAVA LONG,
                                      ValueLayout.JAVA LONG,
                                      ValueLayout.ADDRESS));
        // Create method handle for qsortCompare
        MethodHandle comparHandle = MethodHandles.lookup()
            .findStatic(Qsort.class,
                        "qsortCompare",
                        MethodType.methodType(int.class,
                                              MemorySegment.class,
                                              MemorySegment.class));
```

// Create a Java description of a C function implemented by a Java

```
method
        FunctionDescriptor gsortCompareDesc = FunctionDescriptor.of(
            ValueLayout.JAVA INT,
            ValueLayout.ADDRESS.withTargetLayout(ValueLayout.JAVA INT),
            ValueLayout.ADDRESS.withTargetLayout(ValueLayout.JAVA INT));
        // Create function pointer for qsortCompare
        MemorySegment compareFunc = linker.upcallStub(comparHandle,
                                                       qsortCompareDesc,
                                                       Arena.ofAuto());
        try (Arena arena = Arena.ofConfined()) {
            // Allocate off-heap memory and store unsortedArray in
it
            MemorySegment array = arena.allocateArray(ValueLayout.JAVA INT,
                                                       unsortedArray);
            // Call qsort
            qsort.invoke(array,
                        (long)unsortedArray.length,
                        ValueLayout.JAVA INT.byteSize(),
                        compareFunc);
            // Access off-heap memory
            sorted = array.toArray(ValueLayout.JAVA INT);
        }
        return sorted;
    }
    public static void main(String[] args) {
        try {
            int[] sortedArray = InvokeQsort.gsortTest(new int[] { 0, 9, 3, 4,
6, 5, 1, 8, 2, 7 });
            for (int num : sortedArray) {
                System.out.print(num + " ");
            }
            System.out.println();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

The following sections describe this example in detail:

- Defining the Java Method That Compares Two Elements
- Creating a Downcall Method Handle for the qsort Function
- Creating a Method Handle to Represent the Comparison Method qsortCompare
- Creating a Function Pointer from the Method Handle compareHandle
- Allocating Off-Heap Memory to Store the int Array
- Calling the qsort Function



Copying the Sorted Array Values from Off-Heap to On-Heap Memory

Defining the Java Method That Compares Two Elements

The following class defines the Java method that compares two elements, in this case two int values:

```
class Qsort {
    static int qsortCompare(MemorySegment elem1, MemorySegment elem2) {
        return Integer.compare(elem1.get(ValueLayout.JAVA_INT, 0),
    elem2.get(ValueLayout.JAVA_INT, 0));
    }
}
```

In this method, the int values are represented by <u>MemorySegment</u> objects. A *memory segment* provides access to a contiguous region of memory. To obtain a value from a memory segment, call one of its get methods. This example calls the get (ValueLayout.OfInt, long), where the second argument is the offset in bytes relative to the memory address's location. The second argument is 0 because the memory segments in this example store only one value.

Creating a Downcall Method Handle for the qsort Function

The following statements create a downcall method handle for the qsort function:

```
// Obtain instance of native linker
final static Linker linker = Linker.nativeLinker();
static int[] qsortTest(int[] unsortedArray) throws Throwable {
    int[] sorted = null;
    // Create downcall handle for qsort
    MethodHandle qsort = linker.downcallHandle(
        linker.defaultLookup().find("qsort").get(),
        FunctionDescriptor.ofVoid(ValueLayout.ADDRESS,
            ValueLayout.JAVA_LONG,
            ValueLayout.JAVA_LONG,
            ValueLayout.ADDRESS));
```

Creating a Method Handle to Represent the Comparison Method qsortCompare

The following statement creates a method handle to represent the comparison method Qsort::qsortCompare:



MemorySegment.class, MemorySegment.class));

The MethodHandles.Lookup.findStatic(Class, String, MethodType) method creates a method handle for a static method. It takes three arguments:

- The method's class
- The method's name
- The method's type: The first argument of MethodType::methodType is the method's return value's type. The rest are the types of the method's arguments.

Creating a Function Pointer from the Method Handle compareHandle

The following statement creates a function pointer from the method handle compareHandle:

The Linker::upcallStub method takes three arguments:

- The method handle from which to create a function pointer
- The function pointer's function descriptor; in this example, the arguments for FunctionDescriptor.of correspond to the return value type and arguments of Qsort::gsortCompare
- The arena to associate with the function pointer. The static method Arena.ofAuto() creates a new arena that is managed, automatically, by the garbage collector.

Allocating Off-Heap Memory to Store the int Array

The following statements allocate off-heap memory, then store the int array to be sorted in it:



Calling the qsort Function

The following statement calls the qsort function:

In this example, the arguments of MethodHandle::invoke correspond to those of the standard C library qsort function.

Copying the Sorted Array Values from Off-Heap to On-Heap Memory

Finally, the following statement copies the sorted array values from off-heap to on-heap memory:

```
// Access off-heap memory
sorted = array.toArray(ValueLayout.JAVA INT);
```

Foreign Functions That Return Pointers

Sometimes foreign functions allocate a region of memory, then return a pointer to that region. For example, the C standard library function void *malloc(size_t) allocates the requested amount of memory, in bytes, and returns a pointer to it. However, when you invoke a native function that returns a pointer, like malloc, the Java runtime has no insight into the size or the lifetime of the memory segment the pointer points to. Consequently, the FFM API uses a zero-length memory segment to represent this kind of pointer.

The following example invokes the C standard library function malloc. It prints a diagnostic message immediately after, which demonstrates that the pointer returned by malloc is a zero-length memory segment.

```
static MemorySegment allocateMemory(long byteSize, Arena arena) throws
Throwable {
    // Obtain an instance of the native linker
    Linker linker = Linker.nativeLinker();
    // Locate the address of malloc()
    var malloc_addr = linker.defaultLookup().find("malloc").orElseThrow();
    // Create a downcall handle for malloc()
    MethodHandle malloc = linker.downcallHandle(
        malloc_addr,
        FunctionDescriptor.of(ValueLayout.ADDRESS, ValueLayout.JAVA_LONG)
    );
    // Invoke malloc(), which returns a pointer
    MemorySegment segment = (MemorySegment) malloc.invokeExact(byteSize);
```



```
// The size of the memory segment created by malloc() is zero bytes!
        System.out.println(
            "Size, in bytes, of memory segment created by calling
malloc.invokeExact(" +
            byteSize + "): " + segment.byteSize());
        // Localte the address of free()
        var free addr = linker.defaultLookup().find("free").orElseThrow();
        // Create a downcall handle for free()
        MethodHandle free = linker.downcallHandle(
            free addr,
            FunctionDescriptor.ofVoid (ValueLayout.ADDRESS)
        );
        // This reintepret method:
        // 1. Resizes the memory segment so that it's equal to byteSize
        // 2. Associates it with an existing arena
        // 3. Invokes free() to deallocate the memory allocated by malloc()
        11
             when its arena is closed
        Consumer<MemorySegment> cleanup = s -> {
            try {
                free.invokeExact(s);
            } catch (Throwable e) {
                throw new RuntimeException(e);
        };
        return segment.reinterpret(byteSize, arena, cleanup);
    }
```

The example prints a message similar to the following:

```
Size, in bytes, of memory segment created by calling malloc.invokeExact(100): 0
```

The FFM API uses zero-length memory segments to represent the following:

- Pointers returned from a foreign function
- Pointers passed by a foreign function to an upcall
- Pointers read from a memory segment

If you try to access a zero-length memory segment, the Java runtime will throw an IndexOutOfBoundsException because the Java runtime can't safely access or validate any access operation of a region of memory whose size is unknown. In addition, zero-length memory segments are associated with a fresh scope that's always alive. Consequently, even though you can't directly access zero-length memory segments, you can pass them to other pointer-accepting foreign functions.

However, the MemorySegment::reinterpret method enables you to work with zero length memory segments so that you can safely access them and attach them to an existing arena so that the lifetime of the region of memory backing the segment can be managed automatically. This method takes three arguments:

- The number of bytes to resize the memory segment: The example resizes it to the value of the parameter byteSize.
- The arena with which to associate the memory segment: The example associates it to the arena specified by the parameter arena.
- The action to perform when the arena is closed: The example deallocates the memory allocated by malloc by invoking the C standard library function void free(void *ptr), which deallocates the memory referenced by a pointer returned by malloc. Note that this is an example of passing a pointer pointing to a zero-length memory segment to a foreign function.

Note:

MemorySegment::reinterpret is a restricted method, which, if used incorrectly, might crash the JVM or silently result in memory corruption. See Restricted Methods for more information.

The following example calls allocateMemory(long, Arena) to allocate a Java string with malloc:

```
String s = "My string!";
        try (Arena arena = Arena.ofConfined()) {
            // Allocate off-heap memory with malloc()
            var nativeText = allocateMemory(
                ValueLayout.JAVA CHAR.byteSize() * (s.length() + 1), arena);
            // Access off-heap memory
            for (int i = 0; i < s.length(); i++ ) {</pre>
                nativeText.setAtIndex(ValueLayout.JAVA CHAR, i, s.charAt(i));
            }
            // Add the string terminator at the end
            nativeText.setAtIndex(
                 ValueLayout.JAVA CHAR, s.length(), Character.MIN VALUE);
            // Print the string
            for (int i = 0; i < s.length(); i++ ) {</pre>
System.out.print((char)nativeText.getAtIndex(ValueLayout.JAVA CHAR, i));
            }
            System.out.println();
        } catch (Throwable t) {
            t.printStackTrace();
        }
```

See Zero-length memory segments in the java.lang.foreign.MemorySegment API specification and Functions returning pointers in the java.lang.foreign.Linker API specification for more information.



Memory Layouts and Structured Access

Accessing structured data using only basic operations can lead to hard-to-read code that's difficult to maintain. Instead, you can use *memory layouts* to more efficiently initialize and access more complicated native data types such as C structures.

For example, consider the following C declaration, which defines an array of Point structures, where each Point structure has two members, Point.x and Point.y:

```
struct Point {
    int x;
    int y;
} pts[10];
```

You can initialize and access such a native array as follows:

```
try (Arena arena = Arena.ofConfined()) {
            MemorySegment segment =
                arena.allocate((long)(2 * 4 * 10), 1);
            for (int i = 0; i < 10; i++) {
                int xValue = i;
                int yValue = i * 10;
                segment.setAtIndex(ValueLayout.JAVA INT, (i * 2),
                                                                       xValue);
                segment.setAtIndex(ValueLayout.JAVA INT, (i * 2) + 1, yValue);
            }
            for (int i = 0; i < 10; i++) {
                int xVal = segment.getAtIndex(ValueLayout.JAVA INT, (i * 2));
                int yVal = segment.getAtIndex(ValueLayout.JAVA INT, (i * 2) +
1);
                System.out.println("(" + xVal + ", " + yVal + ")");
            }
        }
```

The first argument in the call to the Arena::allocate method calculates the number of bytes required for the array. The arguments in the calls to the

MemorySegment::setAtIndex method calculate which memory address offsets to write into
each member of a Point structure. Similar arguments perform the same calculations for the
MemorySegment::getAtIndex method. To avoid these calculations, you can use a memory
layout.

To represent the array of Point structures, the following example uses a sequence memory layout:

```
try (Arena arena = Arena.ofConfined()) {
    SequenceLayout ptsLayout
    = MemoryLayout.sequenceLayout(10,
        MemoryLayout.structLayout(
            ValueLayout.JAVA_INT.withName("x"),
            ValueLayout.JAVA_INT.withName("y")));
```

```
VarHandle xHandle
    = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("x"));
VarHandle yHandle
    = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("y"));
MemorySegment segment = arena.allocate(ptsLayout);
for (int i = 0; i < ptsLayout.elementCount(); i++) {</pre>
    int xValue = i;
    int vValue = i * 10;
    xHandle.set(segment, OL, (long) i, xValue);
    yHandle.set(segment, OL, (long) i, yValue);
}
for (int i = 0; i < ptsLayout.elementCount(); i++) {</pre>
    int xVal = (int) xHandle.get(segment, OL, (long) i);
    int yVal = (int) yHandle.get(segment, OL, (long) i);
    System.out.println("(" + xVal + ", " + yVal + ")");
}
```

The first statement creates a sequence memory layout, which is represented by a SequenceLayout object. It contains a sequence of ten structure layouts, which are represented by StructLayout objects. The method MemoryLayout::structLayout returns a StructLayout object. Each structure layout contains two JAVA_INT value layouts named x and y:

}

```
SequenceLayout ptsLayout
   = MemoryLayout.sequenceLayout(10,
        MemoryLayout.structLayout(
            ValueLayout.JAVA_INT.withName("x"),
            ValueLayout.JAVA INT.withName("y")));
```

The predefined value ValueLayout.JAVA_INT contains information about how many bytes a Java int value requires.

The next statements create two *memory-access VarHandles* that obtain memory address offsets. A VarHandle is a dynamically strongly typed reference to a variable or to a parametrically-defined family of variables, including static fields, non-static fields, array elements, or components of an off-heap data structure.

The method PathElement.sequenceElement() retrieves a memory layout from a sequence layout. In this example, it retrieves one of the structure layouts from ptsLayout. The



method call PathElement.groupElement("x") retrieves a memory layout named x. You can create a memory layout with a name with the withName(String) method.

The for statements call VarHandle::set and VarHandle::get to access memory like MemorySegment::setAtIndex and MemorySegment::getAtIndex.

```
MemorySegment segment = arena.allocate(ptsLayout);
for (int i = 0; i < ptsLayout.elementCount(); i++) {
    int xValue = i;
    int yValue = i * 10;
    xHandle.set(segment, 0L, (long) i, xValue);
    yHandle.set(segment, 0L, (long) i, yValue);
}
for (int i = 0; i < ptsLayout.elementCount(); i++) {
    int xVal = (int) xHandle.get(segment, 0L, (long) i);
    int yVal = (int) yHandle.get(segment, 0L, (long) i);
    System.out.println("(" + xVal + ", " + yVal + ")");
}</pre>
```

In this example, the set method uses four arguments:

- 1. segment: the memory segment in which to set the value
- 2. OL: the base offset, which is a long coordinate that points to the start of the array
- 3. (long) i: a second long coordinate that indicates the array index in which to set the value
- 4. xValue and yValue: the actual value to set

The VarHandles xHandle and yHandle know the size of the Point structure (8 bytes) and the size of its int members (4 bytes). This means you don't have to calculate the number of bytes required for the array's elements or the memory address offsets like in the setAtIndex method.

🖓 Tip:

The base offset enables you to express complex access operations by injecting additional offset computation into the VarHandle. In particular, you can use memory segments and base offsets to model *variable-length arrays*. These are arrays whose size are not known statically and that cannot be represented using a sequence layout. You can access such memory segments with the MemoryLayout::arrayElementVarHandle method. See the section Working with variable-length arrays in the JavaDoc API documentation for the MemoryLayout interface for examples.

Checking for Native Errors Using errno

Some C standard library functions indicate errors by setting the value of the C standard library macro errno. You can access this value with a FFM API linker option.

The Linker::downcallHandle method contains a varargs parameter that enables you to specify additional linker options. These parameters are of type Linker.Option.



One linker option is Linker.Option.captureCallState(String...), which you use to save portions of the execution state immediately after calling a foreign function associated with a downcall method handle. You can use it to capture certain thread-local variables. When used with the "errno" string, it captures the errno value as defined by the C standard library. Specify this linker option (with the "errno" string) when creating a downcall handle for a native function that sets errno.

An example of a C standard library function that sets errno is log, which computes the natural (base e) logarithm of its argument. If this value is less than zero, then errno is set to the value 33, which represents a domain error. As most users won't recognize 33 as a domain error, you can invoke the C standard library function strerror, which returns a textual description of the errno value.

The following example invokes the log function then uses <code>captureCallState("errno")</code> to obtain error messages set by the log function:

```
static double invokeLog(double v) throws Throwable {
        double result = Double.NaN;
        // Setup handles
        Linker.Option ccs = Linker.Option.captureCallState("errno");
        StructLayout capturedStateLayout = Linker.Option.captureStateLayout();
        VarHandle errnoHandle =
capturedStateLayout.varHandle(PathElement.groupElement("errno"));
        // log C Standard Library function
        Linker linker = Linker.nativeLinker();
        SymbolLookup stdLib = linker.defaultLookup();
        MethodHandle log = linker.downcallHandle(
            stdLib.find("log").orElseThrow(),
            FunctionDescriptor.of(ValueLayout.JAVA DOUBLE,
ValueLayout.JAVA DOUBLE),
            ccs);
        // strerror C Standard Library function
        MethodHandle strerror = linker.downcallHandle(
            stdLib.find("strerror").orElseThrow(),
            FunctionDescriptor.of(ValueLayout.ADDRESS, ValueLayout.JAVA INT));
        // Actual invocation
        try (Arena arena = Arena.ofConfined()) {
            MemorySegment capturedState = arena.allocate(capturedStateLayout);
            result = (double) log.invokeExact(capturedState, v);
            if (Double.isNaN(result)) {
                \ensuremath{//} Indicates that an error occurred per the documentation of
                // the 'log' command.
                // Get more information by consulting the value of errno:
                int errno = (int) errnoHandle.get(capturedState);
                System.out.println("errno: " + errno); // 33
                // Convert errno code to a string message:
                String errrorString = ((MemorySegment))
```



In this example, the method captureStateLayout() returns a structure layout of the errno function. See Memory Layouts and Structured Access for more information.

Suppose that you call invokeLog(double) as follows:

```
System.out.println("log(2.718): " + invokeLog(2.718));
System.out.println("log(-1): " + invokeLog(-1));
```

The example then prints output similar to the following:

```
log(2.718): 0.999896315728952
errno: 33
errno string: Domain error
log(-1): NaN
```

💡 Tip:

Use the following code to obtain the names of the supported captured value layouts for the Linker.Option.captureCallState(String...) option for your operating system:

```
List<String> capturedNames = Linker.Option.captureStateLayout()
   .memberLayouts()
   .stream()
   .map(MemoryLayout::name)
   .flatMap(Optional::stream)
   .toList();
```

Slicing Allocators and Slicing Memory Segments

A *slicing allocator* returns a segment allocator that responds to allocation requests by returning consecutive contiguous regions of memory, or *slices*, obtained from an existing memory segment. You can also obtain a slice of a memory segment of any location within a memory segment with the method MethodSegment::asSlice.

Topics

- Slicing Allocators
- Slicing Memory Segments



Slicing Allocators

The following example allocates a memory segment named segment that can hold 60 Java int values. It then uses a slicing allocator by calling SegmentAllocator.slicingAllocator(MemorySgement) to obtain ten consecutive slices from segment. The example allocates an array of five integers in each slice. After, it prints the contents of each slice.

```
try (Arena arena = Arena.ofConfined()) {
        SequenceLayout SEQUENCE LAYOUT =
            MemoryLayout.sequenceLayout(60L, ValueLayout.JAVA INT);
        MemorySegment segment = arena.allocate(SEQUENCE LAYOUT);
        SegmentAllocator allocator =
SegmentAllocator.slicingAllocator(segment);
        MemorySegment s[] = new MemorySegment[10];
        for (int i = 0; i < 10; i++) {
            s[i] = allocator.allocateArray(
                ValueLayout.JAVA INT, 1, 2, 3, 4, 5);
        }
        for (int i = 0 ; i < 10 ; i++) {
            int[] intArray = s[i].toArray(ValueLayout.JAVA INT);
            System.out.println(Arrays.toString(intArray));
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
```

You can use segment allocators as building blocks to create arenas that support custom allocation strategies. For example, if a large number of native segments will share the same bounded lifetime, then a custom arena could use a slicing allocator to allocate the segments efficiently. This lets clients enjoy both scalable allocation (thanks to slicing) and deterministic deallocation (thanks to the arena).

The following example defines a *slicing arena* that behaves like a confined arena but internally uses a slicing allocator to respond to allocation requests. When the slicing arena is closed, the underlying confined arena is closed, invalidating all segments allocated in the slicing arena.

To keep this example short, it implements only a subset of the methods of Arena and SegmentAllocator (which is a superinterface of Arena).

```
public class SlicingArena implements Arena {
   final Arena arena = Arena.ofConfined();
   final SegmentAllocator slicingAllocator;
   SlicingArena(MemoryLayout m) {
      slicingAllocator =
   SegmentAllocator.slicingAllocator(arena.allocate(m));
   }
```



```
public MemorySegment allocateArray(
    ValueLayout.OfInt elementLayout, int... elements) {
    return slicingAllocator.allocateArray(elementLayout, elements);
}
public MemorySegment.Scope scope() {
    return arena.scope();
}
public void close() {
    arena.close();
}
```

With this slicing arena, you can rewrite the first example in this section more succinctly:

```
SequenceLayout SEQUENCE_LAYOUT =
   MemoryLayout.sequenceLayout(60L, ValueLayout.JAVA_INT);
try (Arena slicingArena = new SlicingArena(SEQUENCE_LAYOUT)) {
   MemorySegment s[] = new MemorySegment[10];
   for (int i = 0 ; i < 10 ; i++) {
      s[i] = slicingArena.allocateArray(
          ValueLayout.JAVA_INT, 1, 2, 3, 4, 5);
   }
   for (int i = 0 ; i < 10 ; i++) {
      int[] intArray = s[i].toArray(ValueLayout.JAVA_INT);
      System.out.println(Arrays.toString(intArray));
   }
} catch (Exception e) {
    e.printStackTrace();
}</pre>
```

Slicing Memory Segments

}

When a slicing allocator returns a slice, the slice's starting address is right after the end of the last slice that the slicing allocator returned. You can call MemorySegment.asSlice(long,long) to obtain a slice of a memory segment of any location within the memory segment and of any size, provided that slice's size stays within the spatial bounds of the original memory segment. The following example obtains a slice of a memory segment, then prints its contents:

```
String s = "abcdefghijklmnopqrstuvwxyz";
char c[] = s.toCharArray();
MemorySegment textSegment = MemorySegment.ofArray(c);
long b = ValueLayout.JAVA_CHAR.byteSize();
long firstLetter = 5;
long size = 6;
```

MemorySegment fghijk = textSegment.asSlice(firstLetter*b, size*b);

```
for (int i = 0; i < size; i++) {
    System.out.print((char)fghijk.get(ValueLayout.JAVA_CHAR, i*b));
}
System.out.println();</pre>
```

This example prints the following output:

fghijk

The method MemorySegment.elements (MemoryLayout) returns a stream of slices whose size matches that of the specified layout. Multiple threads could work in parallel to access these slices. To do this, however, the memory segment has to be accessible from multiple threads. You can do this by associating the memory segment with a shared arena, which you can create with Arena::ofShared.

The following example sums all int values in a memory segment in parallel.

```
void addRandomNumbers(int numElements) throws Throwable {
    int[] numbers = new Random().ints(numElements, 0, 1000).toArray();
    try (Arena arena = Arena.ofShared()) {
        SequenceLayout SEQUENCE_LAYOUT =
    MemoryLayout.sequenceLayout((long)numElements, ValueLayout.JAVA_INT);
        MemorySegment segment = arena.allocate(SEQUENCE_LAYOUT);
        MemorySegment.copy(numbers, 0, segment, ValueLayout.JAVA_INT, 0L,
        numElements);
        int sum = segment.elements(ValueLayout.JAVA_INT).parallel()
            .mapToInt(s -> s.get(ValueLayout.JAVA_INT, 0))
            .sum();
        System.out.println(sum);
    }
}
```

Restricted Methods

Some methods in the Foreign Function and Memory (FFM) API are unsafe and therefore *restricted*. If used incorrectly, restricted methods can crash the JVM and may silently result in memory corruption.

If you run an application that invokes one of the following restricted methods, the Java runtime will print a warning message. To enable code in a module *M* to use these restricted methods or any unsafe methods without warnings, specify the --enable-native-access=*M* command-line option. Specify multiple modules with a comma-separated list. To enable warning-free use for all code on the class path, specify the --enable-native-access=ALL-UNNAMED option. Alternatively, you can specify the JAR-file manifest attribute Enable-Native-Access: ALL-UNNAMED in an executable JAR to enable warning-free use by all code on the class path. Note that you cannot specify a module name as the value of this attribute.



Methods	Reasoning Behind Restricting the Methods
<pre>java.lang.ModuleLayer.Controller.e nableNativeAccess(Module)</pre>	The method enables native access for the specified module if the caller's module has native acces. This method is restricted because it propagates privileges to call restricted methods.
AddressLayout.withTargetLayout(Mem oryLayout)	Once you have an address layout with a given target layout, you can use it in a dereference operation (for example, MemorySegment.get (AddressLayout, long) to <i>resize</i> the segment being read, which is unsafe.
<pre>Linker.downcallhandle(FunctionDesc riptor, Linker.Option) Linker.downcallhandle(MemorySegmen t, FunctionDescriptor, Linker.Option)</pre>	Creating a downcall method handle is intrinsically unsafe. A linker has no way to verify that the provided function descriptor is compatible with the function being called. A symbol in a foreign library does not typically contain enough signature information, such as arity and the types of foreign function parameters, to enable the linker at runtime to validate linkage requests. When a client interacts with a downcall method handle obtained through an invalid linkage request, for example, by specifying a function descriptor featuring too many argument layouts, the result of such an interaction is unspecified and can lead to JVM crashes.
Linker.upcallStub(MethodHandle, FunctionDescriptor, Arena, Linker.Option)	As with creating downcall handles, the linker can't check whether the function pointer you are creating (like the qsort comparator in the example in Upcalls: Passing Java Code as a Function Pointer to a Foreign Function) is the correct one for the for the downcall you are passing it to (like the qsort method handle in the same example).
<pre>MemorySegment.reinterpret(long) MemorySegment.reinterpret(long, Arena, Consumer) MemorySegment.reinterpret(Arena, Consumer)</pre>	These methods allows you to change the size and lifetime of an existing segment by creating a new alias to the same region of memory. See Foreign Functions That Return Pointers for more information. The spatial or temporal bounds associated with the memory segment alias returned by these methods might be incorrect. For example, consider a region of memory that's 10 bytes long that's backing a zero-length memory segment. An application might overestimate the size of the region and use MemorySegment::reinterpret to obtain a segment that's 100 bytes long. Later, this might result in attempts to dereference memory outside the bounds of the region, which might cause a JVM crash or, even worse, result in silent memory corruption.
SymbolLookup.libraryLookup(String, Arena) SymbolLookup.libraryLookup(Path, Arena)	Loading a library can always cause execution of native code. For example, on Linux, they can be executed through dlopen hooks.

Table 12-1 Restricted Methods from the FFM API



Calling Native Functions with jextract

The jextract tool mechanically generates Java bindings from a native library header file. The bindings that this tool generates depend on the Foreign Function and Memory (FFM) API. With this tool, you don't have to create downcall and upcall handles for functions you want to invoke; the jextract tool generates code that does this for you.

Obtain the tool from the following site:

https://jdk.java.net/jextract/

Obtain the source code for jextract from the following site:

https://github.com/openjdk/jextract

This site also contains steps on how to compile and run jextract, additional documentation, and samples.

Topics

- Run a Python Script in a Java Application
- Call the qsort Function from a Java Application

Run a Python Script in a Java Application

The following steps show you how to generate Java bindings from the Python header file, Python.h, then use the generated code to run a Python script in a Java application. The Python script prints the length of a Java string.

1. Run the following command to generate Java bindings for Python.h:

```
jextract -1 :<absolute path of Python shared library> \
    --output <directory containing code generated by jextract> \
    -I <directory containing Python header files> \
    -t org.python <absolute path of Python.h>
```

For example:

```
jextract -l :/lib64/libpython3.6m.so.1.0 \
    --output gensrc \
    -I /usr/include/python3.6m \
    -t org.python /usr/include/python3.6m/Python.h
```

On Linux, to obtain the path of the Python shared library, you can run the following command:

```
ldconfig -p | grep libpython
```

Running this command prints output similar to the following:

```
libpython3.6m.so.1.0 (libc6,x86-64) => /lib64/libpython3.6m.so.1.0
libpython3.6m.so (libc6,x86-64) => /lib64/libpython3.6m.so
libpython3.so (libc6,x86-64) => /lib64/libpython3.so
```



```
libpython2.7.so.1.0 (libc6,x86-64) => /lib64/libpython2.7.so.1.0
libpython2.7.so (libc6,x86-64) => /lib64/libpython2.7.so
```

The value of the -1 option is a path or name of a shared library that the generated header class should load. If it starts with a colon (:), then the value is interpreted as a library path. Otherwise, it's a library name such as GL for libGL.so.

The jextract tool can resolve any library specifier known by the dynamic linker. Consequently, you can run this command as follows:

```
jextract -l :libpython3.6m.so.1.0 \
    --output gensrc \
    -I /usr/include/python3.6m \
    -t org.python /usr/include/python3.6m/Python.h
```

On Linux systems, if you can't find Python.h or the directory containing the Python header files, you might have to install the python-devel package.

2. In the same directory as classes, which should contain the Python Java bindings, create the following file, PythonMain.java:

```
import java.lang.foreign.Arena;
import java.lang.foreign.MemorySegment;
import static java.lang.foreign.MemorySegment.NULL;
import static org.python.Python h.*;
public class PythonMain {
    public static void main(String[] args) {
        String myString = "Hello world!";
        String script = """
                     string = "%s"
                     print(string, ': ', len(string), sep='')
                     """.formatted(myString).stripIndent();
        Py Initialize();
        try (Arena arena = Arena.ofConfined()) {
            MemorySegment nativeString = arena.allocateFrom(script);
            PyRun SimpleStringFlags(
                nativeString,
                NULL);
            Py Finalize();
        }
        Py Exit(0);
    }
}
```

3. Compile PythonMain.java with the following command:

javac -sourcepath gensrc PythonMain.java

4. Run PythonMain with the following command:

java -cp gensrc:. --enable-native-access=ALL-UNNAMED PythonMain

Call the qsort Function from a Java Application

As mentioned previously, <code>qsort</code> is a C library function that requires a pointer to a function that compares two elements. The following steps create Java bindings for the C standard library with <code>jextract</code>, create an upcall handle for the comparison function required by <code>qsort</code>, and then call the <code>qsort</code> function.

1. Run the following command to create Java bindings for stdlib.h, which is the header file for the C standard library:

```
jextract --output <directory containing code generated by jextract> \
    -t org.unix <absolute path to stdlib.h>
```

For example:

jextract -- output gensrc -t org.unix /usr/include/stdlib.h

The generated Java bindings for stdlib.h include a Java class named stdlib_h, which includes a Java method named <code>qsort(MemorySegment, long, long, MemorySegment)</code>, and a Java interface named <code>__compar_fn_t</code>, which includes a method named <code>allocate</code> that creates a function pointer for the comparison function required by the <code>qsort</code> function.

2. In the same directory where you generated the Java bindings for stdlib.h, create the following Java source file, <code>QsortMain.java</code>:

```
import static org.unix.stdlib h.*;
import java.lang.foreign.Arena;
import java.lang.foreign.MemorySegment;
import java.lang.foreign.ValueLayout;
public class QsortMain {
    public static void main(String[] args) {
        int[] unsortedArray = new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 };
        try (Arena a = Arena.ofConfined()) {
            // Allocate off-heap memory and store unsorted
Array in it
            11
            // stdlib h.C INT is a constant generated by jextract
            MemorySegment array = a.allocateFrom(C INT,
unsortedArray);
            // Create upcall stub for the comparison function
            11
            // MemorySegment
org.unix. compar fn t.allocate( compar fn t, Arena)
            // is from compar fn t.java, generated by jextract
            MemorySegment comparFunc = org.unix. compar fn t.allocate(
```

```
(addr1, addr2) ->
                    Integer.compare(
                        addr1.get(C INT, 0),
                        addr2.get(C INT, 0)),
                    a);
            // Call gsort
            qsort(array, (long) unsortedArray.length, C INT.byteSize(),
comparFunc);
            // Copy off-heap memory into new int[]
            int[] sortedArray = array.toArray(C INT);
            for (int num : sortedArray) {
                System.out.print(num + " ");
            1
            System.out.println();
        }
    }
}
```

The following statement creates an upcall, comparFunc, from a lambda expression:

```
// Create upcall for comparison function
//
// MemorySegment
org.unix.__compar_fn_t.allocate(__compar_fn_t, SegmentScope)
// is from __compar_fn-t.java, generated by jextract
MemorySegment comparFunc = org.unix.__compar_fn_t.allocate(
        (addr1, addr2) ->
        Integer.compare(
            addr1.get(C_INT, 0),
            addr2.get(C_INT, 0)),
            a);
```

Consequently, you don't have to create a method handle for the comparison function as described in Upcalls: Passing Java Code as a Function Pointer to a Foreign Function .

3. Compile QsortMain.java with the following command:

javac -sourcepath gensrc QsortMain.java

4. Run QsortMain with the following command:

java -cp gensrc:. --enable-native-access=ALL-UNNAMED QsortMain



13 Scoped Values

A *scoped value* is a value that may be safely and efficiently shared to methods without using method parameters.

See the ScopedValue class in the Java SE API specification for more information.

Note:

This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Language and VM Features.

For background information about scoped values, see JEP 481.



14 Concurrency

Java SE's concurrency APIs provide a powerful, extensible framework of high-performance threading utilities such as thread pools and blocking queues. This package frees the programmer from the need to craft these utilities by hand, in much the same manner the collections framework did for data structures. Additionally, these packages provide low-level primitives for advanced concurrent programming.

The concurrency APIs, which are contained in the package java.util.concurrent, are classes that are designed to be used as building blocks in building concurrent classes or applications. Just as the collections framework simplified the organization and manipulation of in-memory data by providing implementations of commonly used data structures, the concurrency utilities simplify the development of concurrent classes by providing implementations of building blocks commonly used in concurrent designs. The concurrency utilities include a high-performance, flexible thread pool; a framework for asynchronous execution of tasks; a host of collection classes optimized for concurrent access; synchronization utilities such as counting semaphores; atomic variables; locks; and condition variables.

Using the concurrency utilities, instead of developing components such as thread pools yourself, offers a number of advantages:

- **Reduced programming effort.** It is easier to use a standard class than to develop it yourself.
- **Increased performance.** The implementations in the concurrency utilities were developed and peer-reviewed by concurrency and performance experts; these implementations are likely to be faster and more scalable than a typical implementation, even by a skilled developer.
- Increased reliability. Developing concurrent classes is difficult. The low-level concurrency primitives provided by the Java language (synchronized, volatile, wait(), notify(), and notifyAll()) are difficult to use correctly, and errors using these facilities can be difficult to detect and debug. By using standardized, extensively tested concurrency building blocks, many potential sources of threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated. The concurrency utilities were carefully audited for deadlock, starvation, and race conditions.
- **Improved maintainability.** Programs that use standard library classes are easier to understand and maintain than those that rely on complicated, homegrown classes.
- **Increased productivity.** Developers are likely to already understand the standard library classes, so there is no need to learn the API and behavior of ad hoc concurrent components. Additionally, concurrent applications are simpler to debug when they are built on reliable, well-tested components.

In short, using the concurrency APIs to implement a concurrent application can help your program be clearer, shorter, faster, more reliable, more scalable, easier to write, easier to read, and easier to maintain.

The concurrency APIs include the following:



API	Description
Virtual threads	Virtual Threads are lightweight threads that reduce the effort of writing, maintaining, and debugging high-throughput concurrent applications.
Structured concurrency	Structured Concurrency treats groups of related tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.
Task scheduling framework	The Executor interface standardizes invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies. Implementations are provided that enable tasks to be executed within the submitting thread, in a single background thread (see Executors::newSingleThreadExecutor
	, as with events in Swing, in a newly created thread, or in a thread pool (see Executors::newFixedThreadPool), and developers can create customized implementations of Executor that support arbitrary execution policies. The built-in implementations offer configurable policies such as queue length limits and saturation policy (see RejectedExecutionHandler) that can improve the stability of applications by preventing runaway resource use.
Fork/join framework	Based on the ForkJoinPool class, this framework is an implementation of Executor. It is designed to efficiently run a large number of tasks using a pool of worker threads. A <i>work-</i> <i>stealing</i> technique is used to keep all the worker threads busy, to take full advantage of multiple processors.
Concurrent collections	Concurrent collections include Queue, BlockingQueue and BlockingDeque.
Atomic variables	Utility classes are provided that atomically manipulate single variables (primitive types or references), providing high-performance atomic arithmetic and compare-and-set methods. The atomic variable implementations in the java.util.concurrent.atomic package offer higher performance than would be available by using synchronization (on most platforms), making them useful for implementing high-performance concurrent algorithms and conveniently implementing counters and sequence number generators.
Synchronizers	Synchronizers are general purpose synchronization classes that facilitate coordination between threads. These include Semaphore, CyclicBarrier, CountdownLatch, Phaser, and Exchanger.

Table 14-1 Concurrency APIs



API	Description
Locks	While locking is built into the Java language through the synchronized keyword, there are a number of limitations to built-in monitor locks. The java.util.concurrent.locks package provides a high-performance lock implementation with the same memory semantics as synchronization, and it also supports specifying a timeout when attempting to acquire a lock, multiple condition variables per lock, nonnested ("hand- over-hand") holding of multiple locks, and support for interrupting threads that are waiting to acquire a lock
Nanosecond-granularity timing	The System.nanoTime method enables access to a nanosecond-granularity time source for making relative time measurements and methods that accept timeouts (such as the BlockingQueue.offer, BlockingQueue.poll, Lock.tryLock, Condition.await, and Thread.sleep) can take timeout values in nanoseconds. The actual precision of the System.nanoTime method is platform-dependent.
Thread-local variables	Thread-Local Variables are variables of type ThreadLocal. Unlike "regular" variables, each thread that access a thread-local variable has its own, independently initialized copy of the variable.

Table 14-1 (Cont.) Concurrency APIs

Virtual Threads

Virtual threads are lightweight threads that reduce the effort of writing, maintaining, and debugging high-throughput concurrent applications.

For background information about virtual threads, see JEP 444.

A *thread* is the smallest unit of processing that can be scheduled. It runs concurrently with and largely independently of—other such units. It's an instance of java.lang.Thread. There are two kinds of threads, platform threads and virtual threads.

Topics

- What is a Platform Thread?
- What is a Virtual Thread?
- Why Use Virtual Threads?
- Creating and Running a Virtual Thread
- Scheduling Virtual Threads and Pinned Virtual Threads
- Debugging Virtual Threads
- Virtual Threads: An Adoption Guide



What is a Platform Thread?

A *platform thread* is implemented as a thin wrapper around an operating system (OS) thread. A platform thread runs Java code on its underlying OS thread, and the platform thread captures its OS thread for the platform thread's entire lifetime. Consequently, the number of available platform threads is limited to the number of OS threads.

Platform threads typically have a large thread stack and other resources that are maintained by the operating system. They are suitable for running all types of tasks but may be a limited resource.

What is a Virtual Thread?

Like a platform thread, a *virtual thread* is also an instance of java.lang.Thread. However, a virtual thread isn't tied to a specific OS thread. A virtual thread still runs code on an OS thread. However, when code running in a virtual thread calls a blocking I/O operation, the Java runtime suspends the virtual thread until it can be resumed. The OS thread associated with the suspended virtual thread is now free to perform operations for other virtual threads.

Virtual threads are implemented in a similar way to virtual memory. To simulate a lot of memory, an operating system maps a large virtual address space to a limited amount of RAM. Similarly, to simulate a lot of threads, the Java runtime maps a large number of virtual threads to a small number of OS threads.

Unlike platform threads, virtual threads typically have a shallow call stack, performing as few as a single HTTP client call or a single JDBC query. Although virtual threads support thread-local variables and inheritable thread-local variables, you should carefully consider using them because a single JVM might support millions of virtual threads.

Virtual threads are suitable for running tasks that spend most of the time blocked, often waiting for I/O operations to complete. However, they aren't intended for long-running CPU-intensive operations.

Why Use Virtual Threads?

Use virtual threads in high-throughput concurrent applications, especially those that consist of a great number of concurrent tasks that spend much of their time waiting. Server applications are examples of high-throughput applications because they typically handle many client requests that perform blocking I/O operations such as fetching resources.

Virtual threads are not faster threads; they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency).

Creating and Running a Virtual Thread

The Thread and Thread.Builder APIs provide ways to create both platform and virtual threads. The java.util.concurrent.Executors class also defines methods to create an ExecutorService that starts a new virtual thread for each task.

Topics

- Creating a Virtual Thread with the Thread Class and the Thread.Builder Interface
- Creating and Running a Virtual Thread with the Executors.newVirtualThreadPerTaskExecutor() Method



Multithreaded Client Server Example

Creating a Virtual Thread with the Thread Class and the Thread.Builder Interface

Call the Thread.ofVirtual() method to create an instance of Thread.Builder for creating virtual threads.

The following example creates and starts a virtual thread that prints a message. It calls the join method to wait for the virtual thread to terminate. (This enables you to see the printed message before the main thread terminates.)

```
Thread thread = Thread.ofVirtual().start(() -> System.out.println("Hello"));
thread.join();
```

The Thread.Builder interface lets you create threads with common Thread properties such as the thread's name. The Thread.Builder.OfPlatform subinterface creates platform threads while Thread.Builder.OfVirtual creates virtual threads.

The following example creates a virtual thread named MyThread with the Thread.Builder interface:

```
Thread.Builder builder = Thread.ofVirtual().name("MyThread");
Runnable task = () -> {
    System.out.println("Running thread");
};
Thread t = builder.start(task);
System.out.println("Thread t name: " + t.getName());
t.join();
```

The following example creates and starts two virtual threads with Thread.Builder:

```
Thread.Builder builder = Thread.ofVirtual().name("worker-", 0);
Runnable task = () -> {
    System.out.println("Thread ID: " + Thread.currentThread().threadId());
};
// name "worker-0"
Thread t1 = builder.start(task);
t1.join();
System.out.println(t1.getName() + " terminated");
// name "worker-1"
Thread t2 = builder.start(task);
t2.join();
System.out.println(t2.getName() + " terminated");
```

This example prints output similar to the following:

```
Thread ID: 21
worker-0 terminated
Thread ID: 24
worker-1 terminated
```



Creating and Running a Virtual Thread with the Executors.newVirtualThreadPerTaskExecutor() Method

Executors let you to separate thread management and creation from the rest of your application.

The following example creates an ExecutorService with the Executors.newVirtualThreadPerTaskExecutor() method. Whenever ExecutorService.submit(Runnable) is called, a new virtual thread is created and started to run the task. This method returns an instance of Future. Note that the method Future.get() waits for the thread's task to complete. Consequently, this example prints a message once the virtual thread's task is complete.

```
try (ExecutorService myExecutor =
Executors.newVirtualThreadPerTaskExecutor()) {
   Future<?> future = myExecutor.submit(() -> System.out.println("Running
thread"));
   future.get();
   System.out.println("Task completed");
   // ...
```

Multithreaded Client Server Example

The following example consists of two classes. EchoServer is a server program that listens on a port and starts a new virtual thread for each connection. EchoClient is a client program that connects to the server and sends messages entered on the command line.

EchoClient creates a socket, thereby getting a connection to EchoServer. It reads input from the user on the standard input stream, and then forwards that text to EchoServer by writing the text to the socket. EchoServer echoes the input back through the socket to the EchoClient. EchoClient reads and displays the data passed back to it from the server. EchoServer can service multiple clients simultaneously through virtual threads, one thread per each client connection.

```
public class EchoServer {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port>");
            System.exit(1);
        }
        int portNumber = Integer.parseInt(args[0]);
        try (
            ServerSocket serverSocket =
                new ServerSocket(Integer.parseInt(args[0]));
        ) {
            while (true) {
                Socket clientSocket = serverSocket.accept();
                // Accept incoming connections
                // Start a service thread
                Thread.ofVirtual().start(() -> {
```

```
try (
                        PrintWriter out =
                            new PrintWriter(clientSocket.getOutputStream(),
true);
                        BufferedReader in = new BufferedReader(
                            new
InputStreamReader(clientSocket.getInputStream()));
                    ) {
                        String inputLine;
                        while ((inputLine = in.readLine()) != null) {
                            System.out.println(inputLine);
                            out.println(inputLine);
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                });
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on
port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
public class EchoClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.err.println(
                "Usage: java EchoClient <hostname> <port>");
            System.exit(1);
        }
        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);
        try (
            Socket echoSocket = new Socket(hostName, portNumber);
            PrintWriter out =
                new PrintWriter(echoSocket.getOutputStream(), true);
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(echoSocket.getInputStream()));
        ) {
            BufferedReader stdIn =
                new BufferedReader(
                    new InputStreamReader(System.in));
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
                if (userInput.equals("bye")) break;
            1
        } catch (UnknownHostException e) {
```

```
System.err.println("Don't know about host " + hostName);
System.exit(1);
} catch (IOException e) {
System.err.println("Couldn't get I/O for the connection to " +
hostName);
System.exit(1);
}
```

Scheduling Virtual Threads and Pinned Virtual Threads

The operating system schedules when a platform thread is run. However, the Java runtime schedules when a virtual thread is run. When the Java runtime schedules a virtual thread, it assigns or *mounts* the virtual thread on a platform thread, then the operating system schedules that platform thread as usual. This platform thread is called a *carrier*. After running some code, the virtual thread can *unmount* from its carrier. This usually happens when the virtual thread performs a blocking I/O operation. After a virtual thread unmounts from its carrier is free, which means that the Java runtime scheduler can mount a different virtual thread on it.

A virtual thread cannot be unmounted during blocking operations when it is *pinned* to its carrier. A virtual thread is pinned in the following situations:

- The virtual thread runs code inside a synchronized block or method
- The virtual thread runs a native method or a foreign function (see Foreign Function and Memory API)

Pinning does not make an application incorrect, but it might hinder its scalability. Try avoiding frequent and long-lived pinning by revising synchronized blocks or methods that run frequently and guarding potentially long I/O operations with java.util.concurrent.locks.ReentrantLock.

Debugging Virtual Threads

}

Virtual threads are still threads; debuggers can step through them like platform threads. JDK Flight Recorder and the j cmd tool have additional features to help you observe virtual threads in your applications.

Topics

- JDK Flight Recorder Events for Virtual Threads
- Viewing Virtual Threads in jcmd Thread Dumps

JDK Flight Recorder Events for Virtual Threads

JDK Flight Recorder (JFR) can emit these events related to virtual threads:

- jdk.VirtualThreadStart and jdk.VirtualThreadEnd indicate when a virtual thread starts and ends. These events are disabled by default.
- jdk.VirtualThreadPinned indicates that a virtual thread was pinned (and its carrier thread wasn't freed) for longer than the threshold duration. This event is enabled by default with a threshold of 20 ms.
- jdk.VirtualThreadSubmitFailed indicates that starting or unparking a virtual thread failed, probably due to a resource issue. *Parking* a virtual thread releases the underlying



carrier thread to do other work, and *unparking* a virtual thread schedules it to continue. This event is enabled by default.

Enable the events jdk.VirtualThreadStart and jdk.VirtualThreadEnd through JDK Mission Control or with a custom JFR configuration as described in Flight Recorder Configurations in Java Platform, Standard Edition Flight Recorder API Programmer's Guide.

To print these events, run the following command, where recording.jfr is the file name of your recording:

```
jfr print --events
jdk.VirtualThreadStart,jdk.VirtualThreadEnd,jdk.VirtualThreadPinned,jdk.VirtualTh
readSubmitFailed recording.jfr
```

Viewing Virtual Threads in jcmd Thread Dumps

You can create a thread dump in plain text was well as JSON format:

```
jcmd <PID> Thread.dump_to_file -format=text <file>
jcmd <PID> Thread.dump to file -format=json <file>
```

The JSON format is ideal for debugging tools that accept this format.

The jcmd thread dump lists virtual threads that are blocked in network I/O operations and virtual threads that are created by the ExecutorService interface. It does not include object addresses, locks, JNI statistics, heap statistics, and other information that appears in traditional thread dumps.

Virtual Threads: An Adoption Guide

Virtual threads are Java threads that are implemented by the Java runtime rather than the OS. The main difference between virtual threads and the traditional threads—which we've come to call *platform threads*—is that we can easily have a great many active virtual threads, even millions, running in the same Java process. It is their high number that gives virtual threads their power: they can run server applications written in the thread-per-request style more efficiently by allowing the server to process many more requests concurrently, leading to higher throughput and less waste of hardware.

Because virtual threads are an implementation of java.lang.Thread and conform to the same rules that specified java.lang.Thread since Java SE 1.0, developers don't need to learn new concepts to use them. However, the inability to spawn very many platform threads—the only implementation of threads available in Java for many years—has bred practices designed to cope with their high cost. These practices are counterproductive when applied to virtual threads, and must be unlearned. Moreover, the vast difference in cost informs a new way of thinking about threads that may be foreign at first.

This guide is not intended to be comprehensive and cover every important detail of virtual threads. It is meant but to provide an introductory set of guidelines to help those who wish to start using virtual threads make the best of them.

Write Simple, Synchronous Code Employing Blocking I/O APIs in the Thread-Per-Request Style

Virtual threads can significantly improve the throughput—*not* the latency—of servers written in the thread-per-request style. In this style, the server dedicates a thread to processing each incoming request for its entire duration. It dedicates *at least* one thread because, when

processing a single request, you may want to employ more threads to carry some tasks concurrently.

Blocking a platform thread is expensive because it holds on to the thread—a relatively scarce resource—while it is not doing much meaningful work. Because virtual threads can be plentiful, blocking them is cheap and encouraged. Therefore, you should write code in the straightforward synchronous style and use blocking I/O APIs.

For example, the following code, written in the non-blocking, asynchronous style, won't benefit much from virtual threads.

```
CompletableFuture.supplyAsync(info::getUrl, pool)
   .thenCompose(url -> getBodyAsync(url,
HttpResponse.BodyHandlers.ofString()))
   .thenApply(info::findImage)
   .thenCompose(url -> getBodyAsync(url,
HttpResponse.BodyHandlers.ofByteArray()))
   .thenApply(info::setImageData)
   .thenAccept(this::process)
   .exceptionally(t -> { t.printStackTrace(); return null; });
```

On the other hand, the following code, written in the synchronous style and using simple blocking IO, will benefit greatly:

```
try {
   String page = getBody(info.getUrl(), HttpResponse.BodyHandlers.ofString());
   String imageUrl = info.findImage(page);
   byte[] data = getBody(imageUrl,
HttpResponse.BodyHandlers.ofByteArray());
   info.setImageData(data);
   process(info);
} catch (Exception ex) {
   t.printStackTrace();
}
```

Such code is also easier to debug in a debugger, profile in a profiler, or observe with threaddumps. To observe virtual threads, create a thread dump with the jcmd command:

jcmd <pid> Thread.dump to file -format=json <file>

The more of the stack that's written in this style, the better virtual threads will be for both performance and observability. Programs or frameworks written in other styles that don't dedicate a thread per task should not expect to see a significant benefit from virtual threads. Avoid mixing synchronous, blocking code with asynchronous frameworks.

Represent Every Concurrent Task as a Virtual Thread; Never Pool Virtual Threads

The hardest thing to internalize about virtual threads is that, while they have the same behavior as platform threads they should not represent the same program concept.

Platform threads are scarce, and are therefore a precious *resource*. Precious resources need to be managed, and the most common way to manage platform threads is with thread pools. A question that you then need to answer is, how many threads should we have in the pool?



But virtual threads are plentiful, and so each should represent not some shared, pooled, resource but a task. From a managed resource threads turn into *application domain objects*. The question of how many virtual threads we should have becomes obvious, just as the question of how many strings we should use to store a set of user names in memory is obvious: The number of virtual threads is always equal to the number of concurrent tasks in your application.

Converting *n* platform threads to *n* virtual threads would yield little benefit; rather, it's *tasks* that need to be converted.

To represent every application task as a thread, don't use a shared thread pool executor like in the following example:

```
Future<ResultA> f1 = sharedThreadPoolExecutor.submit(task1);
Future<ResultB> f2 = sharedThreadPoolExecutor.submit(task2);
// ... use futures
```

Instead, use a virtual thread executor like in the following example:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
  Future<ResultA> f1 = executor.submit(task1);
  Future<ResultB> f2 = executor.submit(task2);
  // ... use futures
}
```

The code still uses an ExecutorService, but the one returned from Executors.newVirtualThreadPerTaskExecutor() doesn't employ a thread pool. Rather, it creates a new virtual thread for each submitted tasks.

Furthermore, that ExecutorService itself is lightweight, and we can create a new one just as we would with any simple object. That allows us to rely on the newly added ExecutorService.close() method and the try-with-resources construct. The close method, that is implicitly called at the end of the try block will automatically wait for all tasks submitted to the ExecutorService—that is, all virtual threads spawned by the ExecutorService—to terminate.

This is a particularly useful pattern for fanout scenarios, where you wish to concurrently perform multiple outgoing calls to different services like in the following example:

```
void handle(Request request, Response response) {
  var url1 = ...
  var url2 = ...
  try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    var future1 = executor.submit(() -> fetchURL(url1));
    var future2 = executor.submit(() -> fetchURL(url2));
    response.send(future1.get() + future2.get());
    } catch (ExecutionException | InterruptedException e) {
        response.fail(e);
    }
}
String fetchURL(URL url) throws IOException {
    try (var in = url.openStream()) {
        return new String(in.readAllBytes(), StandardCharsets.UTF 8);
    }
```



}

}

You should create a new virtual thread, as shown above, for even small, short-lived concurrent tasks.

For even more help writing the fanout pattern and other common concurrency patterns, with better observability, use structured concurrency.

As a rule of thumb, if your application never has 10,000 virtual threads or more, it is unlikely to benefit from virtual threads. Either it experiences too light a load to need better throughput, or you have not represented sufficiently many tasks to virtual threads.

Use Semaphores to Limit Concurrency

Sometimes there is a need to limit the concurrency of a certain operation. For example, some external service may not be able to handle more than ten concurrent requests. Because platform threads are a precious resource that is usually managed in a pool, thread pools have become so ubiquitious that they're used for this purpose of restricting concurrency, like in the following example:

```
ExecutorService es = Executors.newFixedThreadPool(10);
...
Result foo() {
   try {
      var fut = es.submit(() -> callLimitedService());
      return f.get();
   } catch (...) { ... }
}
```

This example ensures that there are at most ten concurrent requests to the limited service.

But restricting concurrency is only a side-effect of thread pools' operation. Pools are designed to share scarce resources, and virtual threads aren't scarce and therefore should never be pooled!

When using virtual threads, if you want to limit the concurrency of accessing some service, you should use a construct designed specifically for that purpose: the <u>Semaphore</u> class. The following example demonstrates this class:

```
Semaphore sem = new Semaphore(10);
...
Result foo() {
    sem.acquire();
    try {
        return callLimitedService();
    } finally {
        sem.release();
    }
}
```

Threads that happen to call foo will be throttled, that is, blocked, so that only ten of them can make progress at a time, while others will go about their business unencumbered.

Simply blocking some virtual threads with a semaphore may appear to be substantially different from submitting tasks to a fixed thread pool, but it isn't. Submitting tasks to a thread



pool queues them up for later execution, but the semaphore internally (or any other blocking synchronization construct for that matter) creates a queue of threads that are blocked on it that mirrors the queue of tasks waiting for a pooled thread to execute them. Because virtual threads *are* tasks, the resulting structure is equivalent:

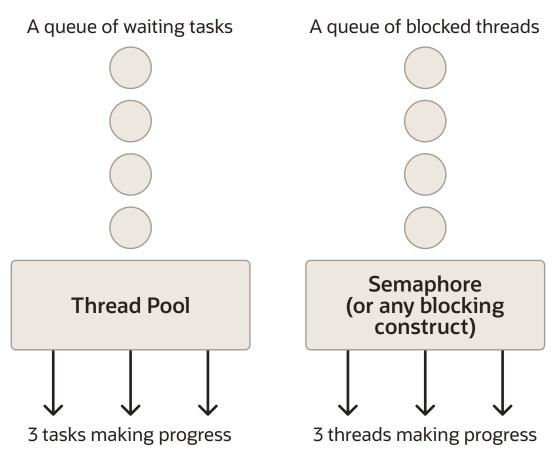


Figure 14-1 Comparing a Thread Pool with a Semaphore

Even though you can think of a pool of platform threads as workers processing tasks that they pull from a queue and of virtual threads as the tasks themselves, blocked until they may continue, the underlying representation in the computer is virtually identical. Recognizing the equivalence between queued tasks and blocked threads will help you make the most of virtual threads.

Database connection pools themselves serve as a semaphore. A connection pool limited to ten connections would block the eleventh thread attempting to acquire a connection. There is no need to add an additional semaphore on top of the connection pool.

Don't Cache Expensive Reusable Objects in Thread-Local Variables

Virtual threads support thread-local variables just as platform threads do. See Thread-Local Variables for more information. Usually, thread-local variables are used to associate some context-specific information with the currently running code, such as the current transaction and user ID. This use of thread-local variables is perfectly reasonable with virtual threads. However, consider using the safer and more efficient scoped values. See Scoped Values for more information.

There is another use of thread-local variables which is fundamentally at odds with virtual threads: caching reusable objects. These objects are typically expensive to create (and consume a significant amount of memory), are mutable, and not thread-safe. They are cached in a thread-local variable to reduce the number of times they are instantiated and their number of instances in memory, but they are reused by the multiple tasks that run on the thread at differerent times.

For example, an instance of SimpleDateFormat is expensive to create and isn't thread-safe. A pattern that emerged is to cache such an instance in a ThreadLocal like in the following example:

```
static final ThreadLocal<SimpleDateFormat> cachedFormatter =
    ThreadLocal.withInitial(SimpleDateFormat::new);
void foo() {
    ...
    cachedFormatter.get().format(...);
    ...
}
```

This kind of caching is helpful only when the thread—and therefore the expensive object cached in the thread local—is shared and reused by multiple tasks, as would be the case when platform threads are pooled. Many tasks may call f_{00} when running in the thread pool, but because the pool only contains a few threads, the object will only be instantiated a few times—once per pool thread—cached, and reused.

However, virtual threads are never pooled and never reused by unrelated tasks. Because every task has its own virtual threads, every call to foo from a different task would trigger the instantiation of a new SimpleDateFormat. Moreover, because there may be a great many virtual threads running concurrently, the expensive object may consume quite a lot of memory. These outcomes are the very opposite of what caching in thread locals intends to achieve.

There is no single general alternative to offer, but in the case of SimpleDateFormat, you should replace it with DateTimeFormatter. DateTimeFormatter is immutable, and so a single instance can be shared by all threads:

```
static final DateTimeFormatter formatter = DateTimeFormatter...;
void foo() {
    ...
    formatter.format(...);
    ...
}
```

Note that using thread-local variables to cache shared expensive objects is sometimes done behind the scenes by asynchronous frameworks, under their implicit assumption that they are used by a very small number of pooled threads. This is one reason why mixing virtual threads and asynchronous frameworks is not a good idea: a call to a method may result in instantiating costly objects in thread-local variables that were intended to be cached and shared.

Avoid Lengthy and Frequent Pinning

A current limitation of the implementation of virtual threads is that performing a blocking operation while inside a synchronized block or method causes the JDK's virtual thread scheduler to block a precious OS thread, whereas it wouldn't if the blocking operation were

done outside of a synchronized block or method. We call that situation "pinning". Pinning may adversely affect the throughput of the server if the blocking operation is both long-lived and frequent. Guarding short-lived operations, such as in-memory operations, or infrequent ones with synchronized blocks or methods should have no adverse effect.

To detect the instances of pinning that might be harmful, (JDK Flight Recorder (JFR) emits the jdk.VirtualThreadPinned thread when a blocking operation is pinned; by default this event is enabled when the operation takes longer than 20ms.

Alternatively, you can use the the system property jdk.tracePinnedThreads to emit a stack trace when a thread blocks while pinned. Running with the option –

Djdk.tracePinnedThreads=full prints a complete stack trace when a thread blocks while pinned, highlighting native frames and frames holding monitors. Running with the option – Djdk.tracePinnedThreads=short limits the output to just the problematic frames.

If these mechanisms detect places where pinning is both long-lived and frequent, replace the use of synchronized with ReentrantLock in those particular places (again, there is no need to replace synchronized where it guards a short lived or infrequent operations). The following is an example of long-lived and frequent use of a synchronized block.

```
synchronized(lockObj) {
    frequentIO();
}
```

You can replace it with the following:

```
lock.lock();
try {
    frequentIO();
} finally {
    lock.unlock();
}
```

Structured Concurrency

Structured concurrency treats groups of related tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.

The principal class of the structured concurrency API is StructuredTaskScope in the java.util.concurrent package. This class enables you coordinate a group of concurrent subtasks as a unit. With a StructuredTaskScope instance, you *fork* each subtask, which runs them in their own individual thread. After, you *join* them as a unit. As a result, the StructuredTaskScope ensures that the subtasks are completed before the main task continues. Alternatively, you can specify that the application continues when one subtask succeeds.



Note:

This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Language and VM Features.

For background information about structured concurrency, see JEP 480.

Basic Usage of the StructuredTaskScope Class

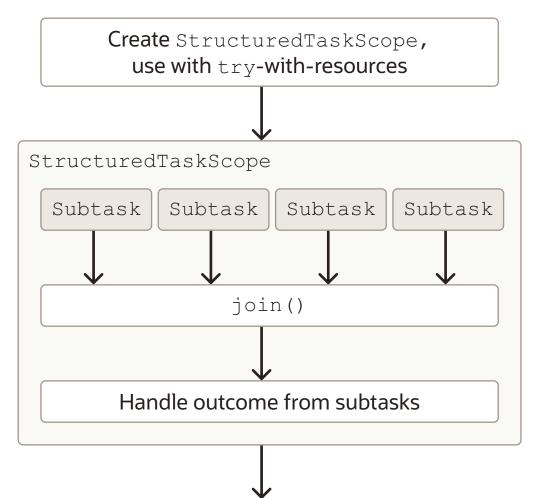
To use the StructuredTaskScope class, you follow these general steps:

- 1. Create a StructuredTaskScope; use it with a try-with-resources statement.
- 2. Define your subtasks as instances of Callable.
- 3. Within the try block, fork each subtask in its own thread with StructuredTaskScope::fork.
- 4. Call StructuredTaskScope::join.
- 5. Handle the outcome from the subtasks.
- 6. Ensure that the StructuredTaskScope is shut down.

The following figure illustrates these steps. Notice that the task scope must wait for all subtasks to finish execution because of the join() method. Afterward, it can handle the results of the subtask.







In general, code that use the StructuredTaskScope class has the following structure:

```
Callable<String> task1 = ...
Callable<Integer> task2 = ...
try (var scope = new StructuredTaskScope<Object>()) {
    Subtask<String> subtask1 = scope.fork(task1);
    Subtask<Integer> subtask2 = scope.fork(task2);
    scope.join();
    ... process results/exceptions ...
} // close
```

Because the StructuredTaskScope was defined in a try-with-resources statement, at the end of the try block, the StructuredTaskScope is shut down, and the task scope waits for threads running any unfinished subtasks to complete.



The StructuredTaskScope class defines the shutdown method to shut down a task scope without closing it. This method cancels all unfinished subtasks by interrupting the threads. In addition, the shutdown method enables subclasses of StructuredTaskScope to implement a *policy* that doesn't require all subtasks to finish. The section Common Shutdown Policies: ShutdownOnSuccess and ShutdownOnFailure describe two subclasses of StructuredTaskScope, ShutdownOnSuccess and ShutdownOnFailure. The first implements a policy that shuts down a task scope as soon as a subtask completes successfully while the second shuts down a task scope as soon as a subtask throws an exception.

Common Shutdown Policies: ShutdownOnSuccess and ShutdownOnFailure

The StructuredTaskScope class contains two subclasses, ShutdownOnFailure and ShutdownOnSuccess. These subclasses implement two common shutdown policies. ShutdownOnFailure cancels all subtasks if one of them fails, while ShutdownOnSuccess cancels all remaining subtasks if one of them succeeds. These shutdown policies are examples of *short-circuiting patterns*. A short-circuiting pattern encourages subtasks to complete quickly by enabling the main task to interrupt and cancel subtasks whose outcomes are no longer needed.

The following example demonstrates the <code>StructuredTaskScope.ShutdownOnFailure</code> and <code>StructuredTaskScope.ShutdownOnSuccess</code> classes. Each task scope forks five subtasks that sleep for a random duration of time. However, if the duration is greater than a specified threshold, the subtask throws a <code>TooSlowException</code>. The

handleShutDownOnFailure() method prints the total duration of all subtasks if none of them threw an exception. The handleShutDownOnSuccess() method prints the duration of the subtask that is completed first:

Figure 14-3 SCRandomTasks.java

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.StructuredTaskScope.*;
import java.util.concurrent.StructuredTaskScope.Subtask;
import java.util.function.*;
import java.util.stream.*;
public class SCRandomTasks {
    class TooSlowException extends Exception {
        public TooSlowException(String s) {
            super(s);
        }
    }
    public Integer randomTask(int maxDuration, int threshold) throws
InterruptedException, TooSlowException {
        int t = new Random().nextInt(maxDuration);
        System.out.println("Duration: " + t);
        if (t > threshold) {
            throw new TooSlowException ("Duration " + t + " greater than
threshold " + threshold);
        }
        Thread.sleep(t);
        return Integer.valueOf(t);
```

```
}
   void handleShutdownOnFailure() throws ExecutionException,
InterruptedException {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            // var t = new SCRandomTasks();
            var subtasks = IntStream.range(0, 5)
                                     .mapToObj(i -> scope.fork(() ->
randomTask(1000, 850)))
                                     .toList();
            scope.join()
                 .throwIfFailed();
            var totalDuration = subtasks.stream()
                                        .map(t -> t.get())
                                        .reduce(0, Integer::sum);
            System.out.println("Total duration: " + totalDuration);
        }
    }
    void handleShutdownOnSuccess() throws ExecutionException,
InterruptedException {
        try (var scope = new StructuredTaskScope.ShutdownOnSuccess()) {
            IntStream.range(0, 5)
                     .mapToObj(i -> scope.fork(() -> randomTask(1000, 850)))
                     .toList();
            scope.join();
            System.out.println("First task to finish: " + scope.result());
        }
    }
    public static void main(String[] args) {
        var myApp = new SCRandomTasks();
        try {
            System.out.println("Running handleShutdownOnFailure...");
            myApp.handleShutdownOnFailure();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        try {
            System.out.println("Running handleShutdownOnSuccess...");
            myApp.handleShutdownOnSuccess();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

It prints output similar to the following:

```
Running handleShutdownOnFailure...
Duration: 359
Duration: 676
Duration: 322
Duration: 591
```



Duration: 315 Total duration: 2263 Running handleShutdownOnSuccess... Duration: 480 Duration: 40 Duration: 868 Duration: 526 Duration: 532 First task to finish: 40

The StructuredTaskScope.ShutdownOnFailure class captures the first exception thrown by one of its subtasks, then invokes the shutdown method. This prevents any new subtasks from starting, interrupts all unfinished threads running other subtasks, and enables the application to continue running. To access the captured exception, call the

ShutdownOnFailure::exception method. If you want to rethrow the exception instead, call the ShutdownOnFailure::throwIfFailed method, which this example demonstrates:

```
scope.join()
    .throwIfFailed();
```

The StructuredTaskScope.ShutdownOnSuccess class captures the result of the first subtask to be completed successfully, and like ShutdownOnFailure, invokes the shutdown method. To access the result of the subtask that completed successfully, call the ShutdownOnSuccess::result method, which this example demonstrates:

```
System.out.println("First task to finish: " + scope.result());
```

Implement Your Own StructuredTaskScope Policies

You can implement your own StructuredTaskScope policies that handle subtasks differently than ShutdownOnFailure and ShutdownOnSuccess. Do this by extending the StructuredTaskScope class.

The following example, CollectingScope, contains two methods that return two streams of subtasks of the same type: successfulTasks() returns a stream of subtasks, and failedTasks() returns a stream of subtasks that threw an exception:

Figure 14-4 CollectingScope.java

```
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
public class CollectingScope<T> extends StructuredTaskScope<T> {
    private final Queue<Subtask<? extends T>> successSubtasks = new
LinkedTransferQueue<>();
    private final Queue<Subtask<? extends T>> failedSubtasks = new
LinkedTransferQueue<>();
    @Override
    protected void handleComplete(Subtask<? extends T> subtask) {
        if (subtask.state() == Subtask.State.SUCCESS) {
            successSubtasks.add(subtask);
        }
}
```



```
} else if (subtask.state() == Subtask.State.FAILED) {
        failedSubtasks.add(subtask);
    }
}
@Override
public CollectingScope<T> join() throws InterruptedException {
   super.join();
   return this;
}
public Stream<Subtask<? extends T>> successfulTasks() {
    super.ensureOwnerAndJoined();
   return successSubtasks.stream();
}
public Stream<Subtask<? extends T>> failedTasks() {
   super.ensureOwnerAndJoined();
   return failedSubtasks.stream();
}
```

To use this class in the example SCRanndomTasks as described in Common Shutdown Policies: ShutdownOnSuccess and ShutdownOnFailure, add the following method. It prints the total duration of the subtasks that completed successfully and the exceptions of the subtasks that threw exceptions.

```
void handleBoth() throws InterruptedException {
        try (var scope = new CollectingScope()) {
            // var t = new SCRandomTasks();
            var subtasks = IntStream.range(0, 5)
                                    .mapToObj(i -> scope.fork(() ->
randomTask(1000, 500)))
                                     .toList();
            scope.join();
            var totalDuration = scope.successfulTasks()
                                      .mapToInt(st -> (Integer)
((Subtask)st).get())
                                      .reduce(0, Integer::sum);
            System.out.println("Total duration: " + totalDuration);
            scope.failedTasks()
                 .forEach(ft ->
                    System.out.println(((Exception)
((Subtask)ft).exception()).getMessage()));
       }
    }
```

It prints output similar to the following:

Duration: 501 Duration: 211 Duration: 661

1



```
Duration: 903
Duration: 839
Total duration: 211
Duration 501 greater than threshold 500
Duration 661 greater than threshold 500
Duration 903 greater than threshold 500
Duration 839 greater than threshold 500
```

In the example CollectingScope, before successfulTasks() and failedTasks() return streams for successSubtasks and failedSubtasks, respectively, it calls StructuredTaskScope::ensureOwnerAndJoined. This ensures that the example can only access successSubtasks and failedSubtasks provided that the current thread is the owner of the task scope, and the task scope has joined the subtasks after they have been forked.

Debugging StructuredTaskScope and Its Forked Subtasks with the jcmd Command

The jcmd tool can emit a thread dump in JSON format. This thread dump displays the threads running the forked subtasks of a StructuredTaskScope in an array, along with their stack traces.

Consider the following example that forks three subtasks. These subtasks repeatedly alternate between printing a message and sleeping for one second.

Figure 14-5 SCObervable.java

```
import java.util.*;
import java.util.function.*;
import java.util.concurrent.*;
import java.util.stream.*;
public class SCObservable {
    static Long sleepOneSecond(String s) throws InterruptedException {
        long pid = ProcessHandle.current().pid();
        for (int i = 0; i<60; i++)
            System.out.println("[" + pid + ", " + s + "]" + " Sleeping for
1s...");
            Thread.sleep(1000);
        }
        return Long.valueOf(pid);
    }
    void handle() throws ExecutionException, InterruptedException {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            Supplier<Long> task1 = scope.fork(() -> sleepOneSecond("task1"));
            Supplier<Long> task2 = scope.fork(() -> sleepOneSecond("task2"));
            Supplier<Long> task3 = scope.fork(() -> sleepOneSecond("task3"));
            scope.join()
                 .throwIfFailed();
        }
    }
```

```
public static void main(String[] args) {
    try {
        var myApp = new SCObservable();
        myApp.handle();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

It prints output similar to the following:

}

```
[10852, task1] Sleeping for 1s...
[10852, task2] Sleeping for 1s...
[10852, task3] Sleeping for 1s...
[10852, task1] Sleeping for 1s...
...
```

While this example is running, you can create a thread dump by running the following command in a different console, where < pid > is the process ID of the running Java process:

jcmd <pid> Thread.dump_to_file -format=json <output_file>

The following excerpt from a sample thread dump output file shows the StructuredTaskScope with the threads of its forked subtasks in an array. The thread dump also shows the reference to the parent of the StructuredTaskScope so that the structure of the program can be reconstituted from the thread dump:

```
{
  "threadDump": {
    "processId": "10852",
    "time": "2023-06-22T13:59:05.156805300Z",
    "runtimeVersion": "21-ea+27-LTS-2343",
    "threadContainers": [
      {
        "container": "<root>",
        "parent": null,
        "owner": null,
        "threads": [
         {
           "tid": "1",
           "name": "main",
           "stack": [
              "java.base\/jdk.internal.misc.Unsafe.park(Native Method)",
              "java.base\/
java.util.concurrent.locks.LockSupport.park(LockSupport.java:371)",
              "java.base\/
jdk.internal.misc.ThreadFlock.awaitAll(ThreadFlock.java:315)",
              "java.base\/
java.util.concurrent.StructuredTaskScope.implJoin(StructuredTaskScope.java:621
)",
              "java.base\/
java.util.concurrent.StructuredTaskScope.join(StructuredTaskScope.java:647)",
```

```
"java.base\/
java.util.concurrent.StructuredTaskScope$ShutdownOnFailure.join(StructuredTask
Scope.java:1200)",
              "SCObservable.handle(SCObservable.java:22)",
              "SCObservable.main(SCObservable.java:30)"
           1
         },
         . . .
        ],
        "threadCount": "7"
      },
      . . .
        "container":
"java.util.concurrent.StructuredTaskScope$ShutdownOnFailure@5674cd4d",
        "parent": "<root>",
        "owner": "1",
        "threads": [
         {
           "tid": "21",
           "name": "",
           "stack": [
              "java.base\/
java.lang.VirtualThread.parkNanos(VirtualThread.java:631)",
              "java.base\/
java.lang.VirtualThread.sleepNanos(VirtualThread.java:803)",
              "java.base\/java.lang.Thread.sleep(Thread.java:507)",
              "SCObservable.sleepOneSecond(SCObservable.java:12)",
              "SCObservable.lambda$handle$0(SCObservable.java:19)",
              "java.base\/
java.util.concurrent.StructuredTaskScope$SubtaskImpl.run(StructuredTaskScope.j
ava:883)",
              "java.base\/java.lang.VirtualThread.run(VirtualThread.java:311)"
           ]
         },
         {
           "tid": "23",
           "name": "",
           "stack": [
              . . .
              "SCObservable.sleepOneSecond(SCObservable.java:12)",
              "SCObservable.lambda$handle$1(SCObservable.java:20)",
              . . .
           ]
         },
         {
           "tid": "24",
           "name": "",
           "stack": [
              . . .
              "SCObservable.sleepOneSecond(SCObservable.java:12)",
              "SCObservable.lambda$handle$2(SCObservable.java:21)",
```

```
...
]
],
"threadCount": "3"
}
]
}
```

Thread-Local Variables

A *thread-local variable* is a variable of type ThreadLocal. Each thread that access a threadlocal variable has its own, independently initialized copy of the variable. To write or read a thread-local variable's value, call its set or get method, respectively. Typically, a thread-local variable is declared as a final static field so that many components can reach it easily.

In the following example, the class TLDBConn represents a database connection. The TLBDBConn::open method prints a string and the user's name. The class TLServer represents the database itself. It contains one method, TLServer::fetchOrder, which returns a string containing the user's name. The class TLApplication creates several TLDBConn objects, each created with a different user and each in its own thread. The TLApplication::testConnection randomly varies the duration of the thread so that the threads have a chance to run concurrently.

Figure 14-6 User.java

```
public class User {
    public String name;
    public User(String n) {
        name = n;
    }
}
```

Figure 14-7 TLDBConn.java

```
public class TLDBConn {
    final static ThreadLocal<User> TLUSER = new ThreadLocal<>();
    public static String open(String info) {
        System.out.println(info + ": " + TLUSER.get().name);
        return info + ": " + TLUSER.get().name;
    }
}
```



Figure 14-8 TLServer.java

```
public class TLServer {
    public static String fetchOrder() {
        return "Fetching order for " + TLDBConn.TLUSER.get().name;
    }
}
```

Figure 14-9 TLApplication.java

```
import java.util.*;
public class TLApplication {
    public void testConnection(User u) {
        Runnable r = () \rightarrow \{
            TLDBConn.TLUSER.set(u);
            TLDBConn.open("Thread " + Thread.currentThread().getName() + ",
testConnection");
            System.out.println(TLServer.fetchOrder());
            try {
                Thread.sleep(new Random().nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            TLDBConn.TLUSER.set(new User(u.name + " renamed"));
            TLDBConn.open("Thread " + Thread.currentThread().getName() + ",
testConnection");
        };
        Thread t = new Thread(r, u.name);
        t.start();
    }
    public static void main(String[] args) {
        TLApplication myApp = new TLApplication();
        for(int i=0 ; i<5; i++) {</pre>
            myApp.testConnection(new User("user" + i));
            try {
                Thread.sleep(new Random().nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



TLApplication prints output similar to the following:

```
Thread user0, testConnection: user0
Fetching order for user0
Thread user1, testConnection: user1
Fetching order for user1
Thread user2, testConnection: user2
Fetching order for user2
Thread user0, testConnection: user0 renamed
Thread user1, testConnection: user1 renamed
Thread user2, testConnection: user2 renamed
Thread user3, testConnection: user3
Fetching order for user3
Thread user3, testConnection: user3 renamed
Thread user4, testConnection: user4
Fetching order for user4
Thread user4, testConnection: user4 renamed
```

Note that even though the member variable TLDBConn.USER is declared as final static, its value is unique for each thread created by TLApplication.

Also, note that the TLServer::fetchOrder method has no parameters. It doesn't require code, in particular, TLApplication::testConnection to pass it a User parameter. TLServer::fetchOrder can directly access the TLDBConn.USER thread-local variable that corresponds to the thread in which it's running:

```
return "Fetching order for " + TLDBConn.TLUSER.get().name;
```

Consequently, thread-local variables enable you to hide method arguments.

Inheriting Thread-Local Variables

When a parent thread starts a child thread, none of the values of the parent thread's threadlocal variables are inherited by the child thread. However, if you want a child thread to inherit the values of its parent's thread-local values, then create a thread-local variable with the InheritableThreadLocal class instead.

The following example includes a InheritableThreadLocal variabled named TLADMIN in addition to the ThreadLocal named TLUSER:

Figure 14-10 TLDBConn.java

```
public class TLDBConn {
    final static ThreadLocal<User> TLUSER = new ThreadLocal<>();
    final static InheritableThreadLocal<User> TLADMIN = new
InheritableThreadLocal<>();
    public static String open(String info) {
        System.out.println(info + ": " + TLUSER.get().name);
        return info + ": " + TLUSER.get().name;
    }
}
```



The following method starts a thread named childThread within a thread. The thread childThread retrieves the value of the InheritableThreadLocal variable named TLADMIN and attempts to retrieve the value of the ThreadLocal variable named TLUSER:

```
public void testConnectionWithInheritableTL(User u) {
        Runnable r = () \rightarrow \{
            TLDBConn.TLUSER.set(u);
            TLDBConn.TLADMIN.set(new User("Admin"));
            TLDBConn.open("Thread " + Thread.currentThread().getName() + ",
testConnection");
            System.out.println(TLServer.fetchOrder());
            try {
                Thread.sleep(new Random().nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            Thread childThread = new Thread(
                 () -> {
                    System.out.println("Child thread");
                    System.out.println("TLADMIN: " +
TLDBConn.TLADMIN.get().name);
                    try {
                         System.out.println("TLUSER: " +
TLDBConn.TLUSER.get().name);
                    } catch (NullPointerException e) {
                         System.out.println("NullPointerException: TLUSER
hasn't beet set");
                     }
            );
            childThread.start();
            TLDBConn.TLUSER.set(new User(u.name + " renamed"));
            TLDBConn.open("Thread " + Thread.currentThread().getName() + ",
testConnection");
        };
        Thread t = new Thread(r, u.name);
        t.start();
    }
```

When you call this method, the following statement in the instantiation of childThread throws a NullPointerException:

```
System.out.println("TLUSER: " + TLDBConn.TLUSER.get().name);
```

The value of the ThreadLocal variable TLUSER hasn't been inherited by childThread. However, the value of the InheritableThreadLocal variable TLADMIN has been inherited by childThread. When childThread is started, it prints the following output:

```
Child thread
TLADMIN: Admin
NullPointerException: TLUSER hasn't beet set
```

Issues with Thread-Local Variables

Unfortunately, thread-local variables have some design flaws.

Note: Scoped Values can address these issues with thread-local variables.

Unconstrained Mutability

Every thread-local variable is mutable. This might make it difficult to discern in your application's code which components update the shared state and in what order. In the example described in Thread-Local Variables, TLApplication::testConnection reassigns TLDBConn.TLUSER with a new value:

```
Runnable r = () -> {
    TLDBConn.TLUSER.set(u);
    TLDBConn.open("Thread " + Thread.currentThread().getName() + ",
testConnection");
    System.out.println(TLServer.fetchOrder());
    // ...
    TLDBConn.TLUSER.set(new User(u.name + " renamed"));
    TLDBConn.open("Thread " + Thread.currentThread().getName() + ",
testConnection");
    };
```

Unbounded Lifetime

The Java runtime retains a thread's incarnation of a thread-local variable for the lifetime of the thread or until code in the thread calls the thread local variable's remove method. If you omit calling this method, then the Java runtime might retain thread data longer than necessary. If you're using a thread pool, then a value in a thread-local variable set in one task might leak into another task. If you have set the value of a thread-local variable multiple times in a thread, then there might not be a clear point when it's safe for the thread to call the remove method, which may cause a long-term memory leak.

Expensive Inheritance

The overhead of thread-local variables may be worse when using large numbers of threads because thread-local variables of a parent thread can be inherited by child threads.



15 Stream Gatherers

Stream gatherers enable you to create custom intermediate operations, which enables stream pipelines to transform data in ways that aren't easily achievable with exisiting built-in intermediate operations.

Note:

This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Language and VM Features.

For background information about stream gatherers, see JEP 473.

Topics

- What Is a Gatherer?
- Creating a Gatherer
- Creating Gatherers with Factory Methods
- Built-In Gatherers
- Composing Gatherers

What Is a Gatherer?

A *gatherer* is an intermediate operation that transforms a stream of input elements into a stream of output elements, optionally applying a final action when it reaches the end of the stream of input elements.

Remember that an intermediate operation, such as Stream.map(Function), produces a new stream, while a terminal operation, such as Stream.forEach(Consumer), produces a non-stream result. A non-stream result could be a primitive value (like a double value), a collection, or in the case of forEach, no value at all.

Gatherers can do the following:

- Transform elements in a one-to-one, one-to-many, many-to-one, or many-to-many fashion
- Track previously seen elements to influence the transformation of later elements
- Short-circuit, or stop processing input elements to transform infinite streams to finite ones
- Process a stream in parallel



Note:

A gather will process a stream in parallel only if you specify a combiner function when you create the gatherer. See The Combiner Function in Creating a Gatherer. A gatherer's default combiner turns parallelization off even if you call parallel().

Examples of gathering operations include the following:

- Grouping elements into batches
- Deduplicating consecutively similar elements
- Incremental accumulation functions
- Incremental reordering functions

Creating a Gatherer

To create a gatherer, implement the Gatherer interface.

The following example creates a gatherer that returns the largest integer from a stream of integers. However, if the gatherer encounters an integer equal or larger to its argument limit, then it returns that integer and stops processing the stream's integers.

```
record BiggestInt(int limit) implements Gatherer<Integer, List<Integer>,
Integer> {
        // The initializer creates a new private ArrayList to keep track of
the
        // largest integer across elements.
        @Override
        public Supplier<List<Integer>> initializer() {
            return () -> new ArrayList<Integer>(1);
        }
        // The integrator
        @Override
        public Integrator<List<Integer>, Integer, Integer> integrator() {
            return Integrator.of(
                (max, element, downstream) -> {
                    // Save the integer if it's the largest so far.
                    if (max.isEmpty()) max.addFirst(element);
                    else if (element > max.getFirst()) max.set(0, element);
                    // If the integer is equal or greater to the limit,
                    // "short-circuit": emit the current integer downstream
                    // and return false to stop processing stream elements
                    if (element >= limit) {
                        downstream.push(element);
                        return false;
                    }
```

```
// Return true to continue processing stream elements
                    return true;
                }
            );
        }
        // The combiner, which is used during parallel evaluation
        @Override
        public BinaryOperator<List<Integer>> combiner() {
            return (leftMax, rightMax) -> {
                // If either the "left" or "right" ArrayLists contain
                // no value, then return the other
                if (leftMax.isEmpty()) return rightMax;
                if (rightMax.isEmpty()) return leftMax;
                // Return the ArrayList that contains the larger integer
                int leftVal = leftMax.getFirst();
                int rightVal = rightMax.getFirst();
                if (leftVal > rightVal) return leftMax;
                else return rightMax;
            };
        }
        @Override
        public BiConsumer<List<Integer>, Downstream<? super Integer>>
finisher() {
            // Emit the largest integer, if there is one, downstream
            return (max, downstream) -> {
                if (!max.isEmpty()) {
                    downstream.push(max.getFirst());
            };
        }
    }
```

You can use this gather as follows:

```
System.out.println(Stream.of(5,4,2,1,6,12,8,9)
         .gather(new BiggestInt(11))
         .findFirst()
         .get());
```

It prints the following output:

12

You can also use this gatherer in parallel:



.parallel()
.findFirst()
.get());

The Gatherer<T, A, R> interface has three type parameters:

- T: The type of input elements to the gather operation. This example process a stream of Integer elements.
- A: The type of the gatherer's private state object, which the gatherer can use to track previously seen elements to influence the transformation of later elements. This example uses a List<Integer> to store the largest Integer it has encountered so far in the input stream.
- R: The type of output elements from the gatherer operation. This example returns an Integer value.

You create a gatherer by defining four functions that work together that process input elements. Some of these functions are optional depending on your gatherer's operation:

- initializer (): Creates the gatherer's private state object
- integrator(): Integrates a new element from the input stream, possibly inspects the private state object, and possibly emits elements to the output stream
- combiner(): Combines two private state objects into one when the gatherer is
 processing the stream in parallel
- finisher(): Optionally performs an action after the gatherer has processed all input elements; it could inspect the private state object or emit additional output elements

The Initializer Function

The optional initializer function creates the gatherer's private state object. This example creates an empty ArrayList with a capacity of only one Integer as its meant to store the largest Integer the gatherer has encountered so far.

```
@Override
public Supplier<List<Integer>> initializer() {
    return () -> new ArrayList<Integer>(1);
}
```

The Integrator Function

Every gatherer requires an integrator function. To create an integrator function, call either Gatherer.Integrator.of(Gatherer.Integrator) Or Gatherer.Integrator.ofGreedy(Gatherer.Integrator). These methods take as an argument a lambda expression that contains three parameters. This example uses the following lamda expression:

```
(max, element, downstream) -> {
    // Save the integer if it's the largest so far.
    if (max.isEmpty()) max.addFirst(element);
    else if (element > max.getFirst()) max.set(0, element);
    // If the integer is equal or greater to the limit,
    // "short-circuit": emit the current integer downstream
```

```
// and return false to stop processing stream elements
if (element >= limit) {
    downstream.push(element);
    return false;
}
// Return true to continue processing stream elements
return true;
```

The parameter max is the private state object.

ļ

The parameter element is the input element that the integrator function is currently processing.

The parameter downstream is a Gatherer. Downstream object. When you call its push method, it passes its argument to the next stage in the pipeline.

An integrator function returns a boolean value. If it returns true, then it will process the next element of the input stream. if it returns false, then it will short-circuit and stop processing input elements.

Tip:

The Downstream::push method returns true if the downstream is willing to push additional elements, so your integrator function can return its return value if you want to continue processing stream elements.

In this example, if element is equal or greather than limit, the integrator function passes element to the next stage in the pipeline, then returns false. The integrator won't process any more input elements, and the Downstream object can no longer push values.

Note:

If you don't expect your integrator function to short-circuit and you want it to process all elements of your input stream, use Integrator::ofGreedy instead of Integrator::of.

The Combiner Function

The optional combiner function is called only if you're running the gatherer in parallel. The combiner function is a lambda expression that contains two parameters, which represent two private state objects.



```
// Return the ArrayList that contains the larger integer
int leftVal = leftMax.getFirst();
int rightVal = rightMax.getFirst();
if (leftVal > rightVal) return leftMax;
else return rightMax;
};
```

This example returns the private state object (an ArrayList) that contains the largest integer.

The Finisher Function

The optional finisher function is a lambda expression that contains two parameters:

```
@Override
public BiConsumer<List<Integer>, Downstream<? super Integer>>
finisher() {
    // Emit the largest integer, if there is one, downstream
    return (max, downstream) -> {
        if (!max.isEmpty()) {
            downstream.push(max.getFirst());
            }
        };
    };
}
```

The parameter max is the private state object and downstream is a Gatherer. Downstream object.

In this example, the finisher function pushes the value contained in the private state object. Note that this value won't be pushed if the integrator function returned false. You can check whether a Downstream object is no longer processing input elements by calling the method Gatherer.Downstream::isRejecting. If it returns true, it's no longer processing input elements.

Note:

If the finisher function pushes a value downstream, then that value is contained in an Optional object.

Creating Gatherers with Factory Methods

Instead of implementing the Gatherer interface, you can call one of the factory methods in the Gatherer interface to create a gatherer.

The following example is the same one as described in Creating a Gatherer except it calls the Gatherer::of method:

```
static Gatherer<Integer, List<Integer>, Integer> biggestInt(int limit) {
```

```
return Gatherer.of(
```



```
// Supplier
        () -> { return new ArrayList<Integer>(1); },
        // Integrator
        Gatherer.Integrator.of(
            (max, element, downstream) -> {
                System.out.println("Processing " + element);
                if (max.isEmpty()) max.addFirst(element);
                else if (element > max.getFirst()) max.set(0, element);
                if (element >= limit) {
                    downstream.push(element);
                    return false;
                }
                return true;
            }
        ),
        // Combiner
        (leftMax, rightMax) -> {
            if (leftMax.isEmpty()) return rightMax;
            if (rightMax.isEmpty()) return leftMax;
            int leftVal = leftMax.getFirst();
            int rightVal = rightMax.getFirst();
            if (leftVal > rightVal) return leftMax;
            else return rightMax;
        },
        // Finisher
        (max, downstream) -> {
            if (!max.isEmpty()) {
                downstream.push(max.getFirst());
            }
        }
   );
}
```

You can call this gatherer as follows:

```
System.out.println(Stream.of(5,4,2,1,6,12,8,9)
    .gather(biggestInt(11))
    .parallel()
    .findFirst()
    .get());
```

Built-In Gatherers

The Gatherers class contains the following built-in gatherers:

- fold (Supplier initial, BiFunction folder): This is an many-to-one gatherer that constructs an aggregate incrementally until no more input elements exist. It has two parameters:
 - initial: This is the identity value or the value that the gatherer emits if the input stream contains no elements.
 - folder: This is a lambda expression that contains two parameters: the first is the aggregate the gatherer is constructing and the second is the element that's currently being processed.

The following example uses the fold gatherer to convert a stream of numbers to a semicolon-separated string:

System.out.println(semicolonSeparated);

It prints the following:

```
1;2;3;4;5;6;7;8;9
```

- mapConcurrent (int maxConcurrency, Function mapper): This is a one-to-one gatherer that invokes mapper for each input element in the stream concurrently, up to the limit specified by maxConcurrency. You can use this limit for the following:
 - As a rate-limiting construct to prevent the gatherer from issuing too many concurrent requests to things like an external service or a database
 - As a performance-enhancer to enable multiple, separate operations to be performed concurrenty while avoiding converting the entire stream into a parallel stream

This gatherer preserves the ordering of the stream.

• scan (Supplier initial, BiFunction scanner): This is a one-to-one gatherer that performs a *prefix scan*, which is an incremental accumulation. Starting with an initial value obtained from the parameter initial, it obtains subsequent values by applying scanner to the current value and the next input element. The gatherer then emits the value downstream. The following example demonstrates this gatherer:



It prints the following output:

- 101 103 106 110 115 121 128 136 145
- windowFixed (int windowSize): This is a many-to-many gatherer that gathers elements in *windows*, which are encounter-ordered groups of elements. The parameter windowSize specifies the size of the windows. The following example demonstrates this gatherer:

```
List<List<Integer>> windows =
Stream.of(1,2,3,4,5,6,7,8).gather(Gatherers.windowFixed(3)).toList();
windows.forEach(System.out::println);
```

It prints the following output:

- [1, 2, 3] [4, 5, 6] [7, 8]
- windowSliding (int windowSize): Similar to windowFixed, this is a many-to-many gatherer that gathers elements in windows. However, each subsequent window includes all elements of the previous window except for its first element, and adds the next element in the stream. The following example demonstrates this gatherer:

```
List<List<Integer>> moreWindows =
Stream.of(1,2,3,4,5,6,7,8).gather(Gatherers.windowSliding(3)).toList();
moreWindows.forEach(System.out::println);
```

It prints the following output:

[1, 2, 3] [2, 3, 4] [3, 4, 5] [4, 5, 6] [5, 6, 7] [6, 7, 8]

Composing Gatherers

You can compose two or more gatherers into a single gatherer with the Gatherer.andThen(Gatherer).



The following example composes a new gatherer with the scan and fold gatherers examples as described in the previous section Built-In Gatherers:

This example prints the following output:

```
101;103;106;110;115;121;128;136;145
```

Note that the following statement to generate the value of ${\tt t}$ is the same as the previous example:

```
var t = Stream.of(1,2,3,4,5,6,7,8,9)
    .gather(sc)
    .gather(fo)
    .findFirst()
    .get();
```

Successively calling the gather method is the same as calling andThen(Gatherer). The following two statements are equivalent, where a and b are gatherers:

```
stream.gather(a).gather(b);
stream.gather(a.andThen(b));
```

