Java Platform, Standard Edition

Core Libraries

Release 19
F55174-01
September 2022

Java Platform, Standard Edition Core Libraries, Release 19

F55174-01

Contents

3 Enhanced Deprecation

4 XML Catalog API

5 Creating Unmodifiable Lists, Sets, and Maps

6 Process API

13 Foreign Function and Memory API

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

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 19 Documentation.

Conventions

The following text conventions are used in this document:

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

1 Java Core Libraries

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

Topics in this Guide

- Serialization Filtering
- Enhanced Deprecation
- XML Catalog API
- Creating Unmodifiable Lists, Sets, and Maps
- Process API
- Preferences API
- Java Logging Overview
- Java NIO
- Java Networking
- Pseudorandom Number Generators
- Virtual Threads
- Foreign Function and Memory API

Other Core Libraries Guides

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

Security Related Topics

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

2

Serialization Filtering

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

Topics:

Addressing Deserialization Vulnerabilities

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

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 `java.io.Serializable` interface or the

`java.io.Externalizable` subinterface. In the JDK, serialization is used in many areas, including Remote Method Invocation (RMI), custom RMI for interprocess communication (IPC) protocols (such as the Spring HTTP invoker), and Java Management Extensions (JMX).

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

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 `ObjectInputFilter` class.

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 context-appropriate set of classes.
- Provide metrics to the filter for graph size and complexity during deserialization to validate normal graph behaviors.
- Allow RMI-exported objects to validate the classes expected in invocations.

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 `ObjectInputFilter` class.

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:

- 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.Ap
plication
```

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 `$JAVA_HOME` by specifying it as a Security Property. (Note that a system property supersedes a Security Property value.) Edit the file `$JAVA_HOME/conf/security/java.security` and add the pattern-based filter to the `jdk.serialFilter` Security Property.

Define a Class Filter

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

In the following example, the filter rejects one class 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

You can set a custom filter on one `ObjectInputStream`, or, to apply the same filter to every stream, set a JVM-wide filter. If an `ObjectInputStream` 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.

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

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 `ObjectInputFilter.Status.UNDECIDED`. Deserialization continues, and any remaining filters are run until the object is accepted or rejected. If there are no other filters, the object is accepted.

Setting a Custom Filter as a Method

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

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

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

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

```
        static ObjectInputFilter.Status
baseFilter(ObjectInputFilter.FilterInfo info) {
```

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

Creating a Filter with ObjectInputFilter Methods

The `ObjectInputFilter` 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 `ObjectInputFilter.Status.ALLOWED` 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:

```
ObjectInputFilter intFilter = ObjectInputFilter.allowFilter(
    cl -> cl.equals(Integer.class),
ObjectInputFilter.Status.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:

```
ObjectInputFilter f = ObjectInputFilter.rejectFilter(cl ->
    cl.getClassLoader() == ClassLoader.getSystemClassLoader(),
Status.UNDECIDED);
```

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

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 `ObjectInputFilter` parameters every time its `apply` 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$$Lambda$8/0x0000000800c00c60@76ed5528,
            ifTrue: ALLOWED, ifFalse: UNDECIDED),
        predicate(
            SimpleFilterFactory$$Lambda$9/0x0000000800c01800@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 `$JAVA_HOME` by specifying it in a Security Property. Note that a system property supersedes a Security Property value. Edit the file `$JAVA_HOME/conf/security/java.security` and specify the filter factory's class name in the `jdk.serialFilterFactory` 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 `sun.rmi.registry.registryFilter` 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 `jdk.serialFilter` global system property to increase protection for other serialization users that do not have their own custom filters.

The RMI Registry has a built-in 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 `jmx.remote.rmi.server.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. That filter is disabled by default. To enable the filter, define the `jmx.remote.rmi.server.serial.filter.pattern` management property with a pattern.

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

```
com.sun.management.jmxremote.serial.filter.pattern=java.lang.*;java.math.BigI
nteger;java.math.BigDecimal;java.util.*;javax.management.openmbean.*;javax.ma
nagement.ObjectName;java.rmi.MarshalledObject;!*
```

Logging Filter Actions

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

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 `@Deprecated`. 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 `@Deprecated` annotation from the `java.lang.Thread` class:

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

Semantics of Deprecation

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

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

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

Notifications and Warnings

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

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

Compiler Deprecation Warnings

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

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

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

This is an example of an ordinary deprecation warning.

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

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

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

Here is an example of a removal warning.

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

Suppressing Deprecation Warnings

The `javac -Xlint` options control warnings for all files compiled in a particular run of `javac`. You may have identified specific locations in source code that generate

warnings that you no longer want to see. You can use the `@SuppressWarnings` annotation to suppress warnings whenever that code is compiled. Place the `@SuppressWarnings` annotation at the declaration of the class, method, field, or local variable that uses a deprecated API.

The `@SuppressWarnings` options are:

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

Here's an example of suppressing a warning.

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

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

Running jdeprscan

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

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

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

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

4

XML Catalog API

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

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

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

Learning More About Creating Catalogs

To learn about creating catalogs, see 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">
    <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:

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

With this catalog 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 `jaxp.properties` file with a line:

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

javax.xml.catalog.files

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

Catalog URI

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

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

Using system or uri Entries

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

Use Catalog with XML Processors

Use the XML Catalog API with various Java XML processors.

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

Use Catalog with DOM

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

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

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

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

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

Use Catalog with SAX

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

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

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

Given an XML file `XI_simple.xml`:

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

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

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

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

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

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

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

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

Use Catalog with StAX

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

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

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

Note that unlike the `DocumentBuilderFactory` class that has both `setFeature` and `setAttribute` methods, the `XMLInputFactory` class defines only a `setProperty` method. The XML Catalog API features including `XMLConstants.USE_CATALOG` are all set through this `setProperty` method. For example, to disable `USE_CATALOG` on a `XMLStreamReader` object, you can do the following:

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

Use Catalog with Schema Validation

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

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

The XMLSchema schema document contains references to external DTD:

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

And to `xsd` import:

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

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

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

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

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

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

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

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

- A `system` entry:

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

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

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

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

Use Catalog with Transform

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

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

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

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

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

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

Calling Order for Resolvers

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

Custom Resolver Preferred to Catalog Resolver

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

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

Detecting Errors

Detect configuration issues by isolating the problem.

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

Dectecting Configuration Issues

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

Table 4-1 RESOLVE Settings

`RESOLVE` Value	`CatalogResolver` Behavior	Description
`strict` (default)	Throws a `CatalogException` if no match is found with a specified reference	An unmatched reference may indicate a possible error in the catalog or in setting the catalog.
`continue`	Returns quietly	This is useful in a production environment where you want the XML processors to continue resolving any external references not covered by the catalog.
`ignore`	Returns quietly	For processors such as SAX, that allow skipping the external references, the `ignore` value instructs the `CatalogResolver` object to return an empty `InputSource` object, thus skipping the external reference.

5

Creating Unmodifiable Lists, Sets, and Maps

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

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

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

Topics

Use Cases

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

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

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

For a collection that holds data that is modified throughout the course of the program, a modifiable collection is the best choice. Modifications are performed in-place, so that incremental additions or deletions of data elements are quite inexpensive. If this were done with an unmodifiable collection, a complete copy would have to be made to add or remove a single element, which usually has unacceptable overhead.

Syntax

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

Topics

Unmodifiable List Static Factory Methods

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

A list is an ordered collection 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:

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

Example 5-2 Examples

In JDK 8:

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

In JDK 9 and later:

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

See Unmodifiable Sets.

Unmodifiable Map Static Factory Methods

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

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

The syntax of these methods is:

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

Example 5-3 Examples

In JDK 8:

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

In JDK 9 and later:

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

Example 5-4 Map with Arbitrary Number of Pairs

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

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

See Unmodifiable Maps.

Creating Unmodifiable Copies of Collections

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

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

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

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

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

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

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

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

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

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

Creating Unmodifiable Collections from Streams

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

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

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

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

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

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

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

Randomized Iteration Order

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

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

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

jshell> /exit
|  Goodbye

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

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

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

About Unmodifiable Collections

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Unmodifiable Collections vs. Unmodifiable Views

The unmodifiable collections behave in the same way as the unmodifiable views returned by the `Collections.unmodifiable...` methods. (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.

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

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

Topics

- ProcessHandle.Info Interface

ProcessBuilder Class

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

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

Table 6-1 ProcessBuilder Class Attributes and Related Methods

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

Process Class

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

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

Table 6-2 Process Class Methods

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

ProcessHandle Interface

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

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

Table 6-3 ProcessHandle Interface Methods

Method Type	Related Methods
Retrieve all operating system processes.	• `allProcesses()`
Retrieve process handle.	• `current()` • `of(long pid)` • `parent()`

Table 6-3 (Cont.) ProcessHandle Interface Methods

Method Type	Related Methods
Retrieve information about the process.	• `isAlive()` • `pid()` • `info()`
Retrieve stream of direct and indirect child processes.	• `children()` • `descendants()`
Destroy process.	• `destroy()` • `destroyForcibly()`
Return a `CompletableFuture` instance that will be completed when the process exits. See Handling Processes When They Terminate with the onExit Method for an example.	• `onExit()`

ProcessHandle.Info Interface

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

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

Table 6-4 ProcessHandle.Info Interface Methods

Method	Description
`arguments()`	Returns the arguments of the process as a `String` array.
`command()`	Returns the executable path name of the process.
`commandLine()`	Returns the command line of the process.
`startInstant()`	Returns the start time of the process.
`totalCpuDuration()`	Returns the 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).redirectError(Redirect.INHERIT).`

Filtering Processes with Streams

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

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

```
public class ProcessTest {

  // ...

  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 `onExit().thenAccept(onExitMethod)` on each of the processes; `onExitMethod` prints the process ID (PID), exit status, and output of the process.

```
public class ProcessTest {

  // ...

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

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

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

  private static String output(InputStream inputStream) {
    String s = "";
    try (BufferedReader br = new BufferedReader(new
InputStreamReader(inputStream))) {
      s =
br.lines().collect(Collectors.joining(System.getProperty("line.separator")));
    } catch (IOException e) {
      System.err.println("Caught IOException");
      e.printStackTrace();
    }
    return s;
  }

  // ...
}
```

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

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

Press enter to continue ...

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

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

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

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

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

```
  static void startSeveralProcesses (
    List<ProcessBuilder> pBList, Consumer<Process> onExitMethod)
    throws InterruptedException {
    System.out.println("Number of processes: " + pBList.size());
```

```
    pBList.stream().forEach(
      pb -> {
        try {
          Process p = pb.start();
          System.out.printf("Start %d, %s%n",
            p.pid(), p.info().commandLine().orElse("<na>"));
          p.onExit().get();
          printGrepResults(p);
        } catch (IOException|InterruptedException|ExecutionException
e ) {
          System.err.println("Exception caught");
          e.printStackTrace();
        }
      }
    );
  }
```

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

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

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

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

Controlling Access to Sensitive Process Information

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

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

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

> **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

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

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

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

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

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

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

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

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

        ...
    }
```

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

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

Obtain Preferences Objects for a Static Method

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

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

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

    ...
}
```

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

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

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

Atomic Updates

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

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

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

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

```
static int X_DEFAULT = 50, Y_DEFAULT = 25;
void parsePrefs() {
    String position = prefs.get(POSITION, X_DEFAULT + "," + Y_DEFAULT);
    int x, y;
    try {
        int i = position.indexOf(',');
```

```
            x = Integer.parseInt(coordinates.substring(0, i));
            y = Integer.parseInt(position.substring(i + 1));
        } catch(Exception e) {
            // Value was corrupt, just use defaults
            x = X_DEFAULT;
            y = Y_DEFAULT;
        }
        ...
}
```

Determine Backing Store Status

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

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

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

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

Design FAQ

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

Topics:

How does this Preferences API relate to the Properties API?

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

How does the Preferences API relate to JNDI?

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

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

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

How was it decided which methods should throw BackingStoreException?

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

Why doesn't this API provide stronger guarantees concerning concurrent access by multiple VMs? Similarly, why doesn't the API allow multiple Preferences updates to be combined into a single "transaction", with all or nothing semantics?

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

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

In the Java programming universe, case-sensitive String keys are ubiquitous. In particular, they are provided by the Properties class, which this API is intended to replace. It is not

uncommon for people to use Properties in a fashion that demands case-sensitivity. For example, Java package names (which are case-sensitive) are sometimes used as keys. It is recognized that this design decision complicates the life of the systems programmer who implements Preferences atop a backing store with case-insensitive keys, but this is considered an acceptable price to pay, as far more programmers will use the Preferences API than will implement it.

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

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

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

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

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

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

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

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

Why is Preferences an abstract class rather than an interface?

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

Where is the default backing store?

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

On Linux systems, the system preferences are typically stored at *java-home*`/.systemPrefs` in a network installation, or `/etc/.java/.systemPrefs` in a local installation. If both are present, `/etc/.java/.systemPrefs` takes

precedence. The system preferences location can be overridden by setting the system property `java.util.prefs.systemRoot`. The user preferences are typically stored at *`user-home`*`/.java/.userPrefs`. The user preferences location can be overridden by setting the system property `java.util.prefs.userRoot`.

8

Java Logging Overview

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

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

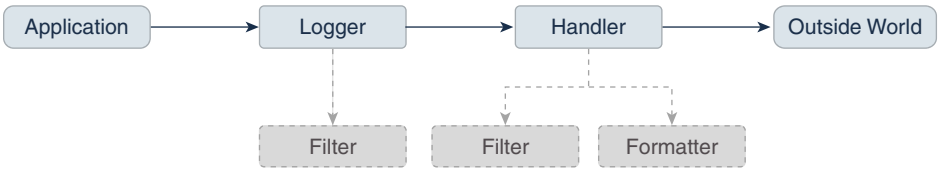
Topics

Overview of Control Flow

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

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

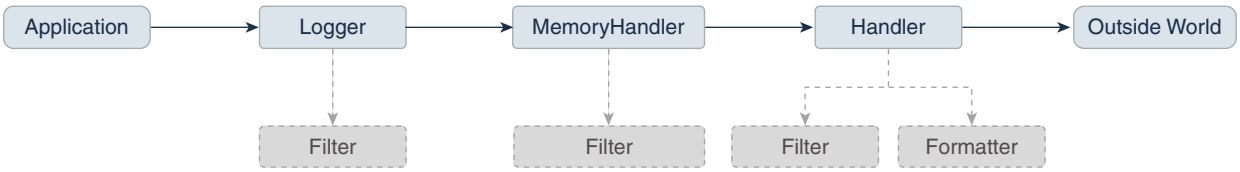
Figure 8-1 Java Logging Control Flow



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

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

Figure 8-2 Java Logging Control Flow with MemoryHandler



The APIs are structured so that calls on the Logger APIs can be cheap when logging is disabled. If logging is disabled for a given log level, then the Logger can make a cheap comparison test and return. If logging is enabled for a given log level, the Logger is still careful to minimize costs before passing the `LogRecord` 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 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.

Table 9-2 Charset Classes

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.

Table 9-3 Channel Interfaces and Classes

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 *scattering* 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 *gathering* 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 *position* 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.

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

Channel Interface or Class	Description
`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.
`MulticastChannel`	A network channel that supports Internet Protocol (IP) multicasting. IP multicasting is the transmission of IP datagrams to members of a *group* 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.

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

Channel Interface or Class	Description
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.
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++) {
                File f = new File(args[i]);
                try {
                    grep(f);
                } catch (IOException x) {
                    System.err.println(f + ": " + x);
                }
            }
        }
}
```

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;
        }
        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++) {
            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++) {
                    locals.add(parseAddress(args[i]));
                }
                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);
                        }
                    }
                } 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++) {
                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
  - `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 class Chmod {

    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++) {
                    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 (g) {
                if (r) toRemove.add(GROUP_READ);
                if (w) toRemove.add(GROUP_WRITE);
                if (x) toRemove.add(GROUP_EXECUTE);
            }
            if (o) {
                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) {
            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) {
            String arg = args[argi];
            if (!arg.startsWith("-"))
                break;
            if (arg.length() < 2)
                usage();
            for (int i=1; i<arg.length(); i++) {
                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++) {
            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.cla
ss)) {
            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 `$JAVA_HOME/conf/net.properties` 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 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.

Table 10-1 HTTP Client Properties

Property	Default Value	Description
`jdk.httpclient.allowRestrictedHeaders`	*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, the following request headers are not allowed to be set by user code: `connection`, `content-length`, `expect`, `host`, and `upgrade`. You can override this behavior with this property. The names 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.bufsize`	16384 (16 kB)	The size to use for internal allocated buffers in bytes.
`jdk.httpclient.connectionPoolSize`	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.connectionWindowSize`	2^26	The HTTP/2 client connection window size in bytes. The maximum size is 2^31-1. This value cannot be smaller than the stream window size.
`jdk.httpclient.disableRetryConnect`	false	Whether automatic retry of connection failures is disabled. If false, then retries are attempted (subject to the retry limit).
`jdk.httpclient.enableAllMethodRetry`	false	Whether it is permitted to automatically retry non-idempotent HTTP requests.

Table 10-1 (Cont.) HTTP Client Properties

Property	Default Value	Description
`jdk.httpclient.enablepush`	1	Whether HTTP/2 push promise is enabled. A value of 1 enables push promise; a value of 0 disables it.
`jdk.httpclient.hpack.maxheadertablesize`	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-separated list of any of the following items: • `errors` • `requests` • `headers` • `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, if you specify the following value, then the Java Logging API logs all possible HTTP Client events: `errors,requests,headers,frames:control:data:window,ssl,trace,channel` Note that you can replace `control:data:window` with `all`. The name of the logger is `jdk.httpclient.HttpClient`, and all logging is at level `INFO`.

Table 10-1 (Cont.) HTTP Client Properties

Property	Default Value	Description
`jdk.httpclient.keepalive.timeout`	1200	The number of seconds to keep idle HTTP/1.1 connections alive in the keep alive cache.
`jdk.httpclient.maxframesize`	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.maxstreams`	100	The maximum number of HTTP/2 streams per connection.
`jdk.httpclient.receiveBufferSize`	*The operating system's default value*	The HTTP client socket receive buffer size in bytes.
`jdk.httpclient.redirects.retrylimit`	5	The maximum number of attempts to send a HTTP request when redirected or any failure occurs for any reason.
`jdk.httpclient.websocket.writeBufferSize`	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.

IPv4 and IPv6 Protocol Properties

These two properties are checked only once, at startup.

Table 10-2 IPv4 and IPv6 Protocol Properties

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.

Table 10-2 (Cont.) IPv4 and IPv6 Protocol Properties

Property	Default Value	Description
`java.net.preferIPv6Addresses`	`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 over 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.

HTTP Proxy Properties

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

Table 10-3 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.

Table 10-3 (Cont.) HTTP Proxy Properties

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 (`|`) 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: `-Dhttp.nonProxyHosts="*.example.com|localhost"` The default value excludes all common variations of the loopback address.

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 10-4 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.

Table 10-5 FTP Proxy Properties

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. The default value excludes all common variations of the loopback address.

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 10-6 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.
`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.

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

Table 10-7 Other Proxy-Related Properties

Property	Default Value	Description
`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.
`jdk.http.auth.tunneling.disabledSchemes`	`Basic`	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 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.

Table 10-7 (Cont.) Other Proxy-Related Properties

Property	Default Value	Description
`jdk.http.auth.proxying.disabledSchemes`	*No default value*	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.

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.

Ues these properties to control the selection of this directory:

Table 10-8 UNIX Doman Sockets Properties

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.

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

1. 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:

1. 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 10-9 Other HTTP URL Stream Protocol Handler Properties

Property	Default Value	Description
`http.agent`	`Java/<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).
`http.auth.digest.cnonceRepeat`	`5`	See System Properties That Modify the Behavior of HTTP Digest Authentication Mechanism.
`http.auth.digest.validateProxy`	`false`	See System Properties That Modify the Behavior of HTTP Digest Authentication Mechanism.

Table 10-9 (Cont.) Other HTTP URL Stream Protocol Handler Properties

Property	Default Value	Description
`http.auth.digest.validateServer`	`false`	See System Properties That Modify the Behavior of HTTP Digest Authentication Mechanism.
`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: 1. Do not specify it. In some environments, the domain is not actually required and the application does not have to specify it. 2. The domain name can be encoded within the user name by prefixing the domain name followed by a backslash (\) before the user name. With this method, existing applications that use 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.

Table 10-9 (Cont.) Other HTTP URL Stream Protocol Handler Properties

Property	Default Value	Description
`http.keepAlive`	`true`	Indicates if persistent (keep-alive) connections should be supported. 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.queuedConnections`	`10`	The maximum number of keep-alive connections to be on the queue for clean up.
`http.KeepAlive.remainingData`	`512`	The maximum amount of data in kilobytes that will be cleaned off the underlying socket so that it can be reused.
`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.

Table 10-9 (Cont.) Other HTTP URL Stream Protocol Handler Properties

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 host or hosts for which a CBT is sent. Domains can be: – Single hosts like `example` or `example.com` – 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.

Table 10-9 (Cont.) Other HTTP URL Stream Protocol Handler Properties

Property	Default Value	Description
`jdk.http.ntlm.transparentAuth`	*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`: Transparent 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.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.

Table 10-10 Address Cache Properties

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

Table 10-11 Enhanced Exception Messages Property

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).

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

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 2^{64}. A leapable PRNG can jump even further, typically 2^{128} 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 2^{48} 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 2^{256} or larger, such as L64X256MixRandom, L64X1024MixRandom, L128X256MixRandom, or L128X1024MixRandom.

12

Virtual Threads

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

> **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 virtual threads, see JEP 425.

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?

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. Platform threads support thread-local variables.

Platforms threads 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, 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

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:

```
        try {
            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();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

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

```
public class CreateNamedThreadsWithBuilders {

    public static void main(String[] args) {

        try {
            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");

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

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");
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
```

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;
            }
        } 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, the 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. Java Flight Recorder and the `jcmd` tool have additional features to help you observe virtual threads in your applications.

Topics

Java Flight Recorder Events for Virtual Threads

Java 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). 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.VirtualT
hreadSubmitFailed 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.

Practical Advice for Virtual Threads

Because virtual threads are inexpensive and plentiful, many programming techniques you would usually use because platform threads are expensive and heavyweight are no longer applicable or recommended.

Topics

Don't Pool Virtual Threads

A *thread pool* is a group of preconstructed platform threads that are reused when they become available. Some thread pools have a fixed number of threads while others create new threads as needed.

Don't pool virtual threads. Create one for every application task. Virtual threads are short-lived and have shallow call stacks. They don't need the additional overhead or the functionality of thread pools.

Use Semaphores for Limited Resources

A *semaphore* restricts the number of threads that can access a physical or logical resource. Use semaphores (instead of thread pools) if you need to limit concurrency, for example, if you make sure only a specified number of threads can access a limited resource, such as requests to a database.

Avoid Pinning

As mentioned in Scheduling Virtual Threads and Pinned Virtual Threads, pinning might hinder an application's scalability. Avoid frequent and long-lived pinning by revising `synchronized` blocks and guard potentially long I/O operations with `java.util.concurrent.locks.ReentrantLock`, which controls access to a shared resource by multiple threads in a similar way as implicit locks in synchronized code.

Review Usage of Thread-Local Variables

Carefully consider the use of thread-local variables because there could be many virtual threads in your application.

13

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.

> **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 the Foreign Function and Memory API, see JEP 424.

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

Topics

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

Consider 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. See Allocating Off-Heap Memory.
2. Store the Java string in the off-heap memory that you allocated. See Dereferencing Off-Heap Memory.

> **Note:**
>
> Some methods enable you to perform these two steps with one method call; see Methods That Allocate and Populate Off-Heap Memory.

3. Build and then call a method handle that points to the `strlen` function. See Linking and Calling a C Function.

The following example calls `strlen` with the Foreign Function and Memory (FFM) API:

```
    static long invokeStrlen(String s) throws Throwable {

        try (MemorySession session = MemorySession.openConfined()) {

            // 1. Allocate off-heap memory, and
            // 2. Dereference off-heap memory
            MemorySegment nativeString = session.allocateUtf8String(s);

            // 3. Link and call C function

            // 3a. Obtain an instance of the native linker
            Linker linker = Linker.nativeLinker();

            // 3b. Locate the address of the C function
            SymbolLookup stdLib = linker.defaultLookup();
            MemorySegment strlen_addr = stdLib.lookup("strlen").get();

            // 3c. Create a description of the C function signature
            FunctionDescriptor strlen_sig =
                FunctionDescriptor.of(ValueLayout.JAVA_LONG,
ValueLayout.ADDRESS);

            // 3d. Create a downcall handle for the C function
            MethodHandle strlen = linker.downcallHandle(strlen_addr,
strlen_sig);

            // 3e. Call the C function directly from Java
            return (long)strlen.invoke(nativeString);
        }
    }
```

Allocating Off-Heap Memory

Off-heap data is data stored in memory outside the Java runtime and therefore not subject to garbage collection. Off-heap data is stored in *off-heap memory*, which is represented by a `MemorySegment` object. To invoke a C function from a Java application, its arguments must be in off-heap memory.

The following example create a memory segment off-heap to store the contents of a Java string:

```
String s = ...
MemorySegment nativeString =
    MemorySegment.allocateNative(s.length()+1, MemorySession.global());
```

This example allocates a block of off-heap memory of size `s.length() + 1`, which is large enough to hold the contents of the Java string, plus the trailing terminator character.

All memory segments are associated with a *memory session*, which determines when the memory segment is valid or can be accessed. A memory session is represented by a `MemorySession` object. In the previous example, the memory segment is associated with the global memory session, which is a memory session that cannot be closed. As a result, the off-heap memory will only be deallocated when the JVM process ends. If you want a closeable memory session, then you can use a confined memory session:

```
String s = ...
try (MemorySession session = MemorySession.openConfined()) {
    MemorySegment.allocateNative(s.length()+1, session);
    // ...
} // nativeString is deallocated here
```

A *confined memory session* is a session that can only be accessed by the current thread. This example creates a confined memory session in a `try`-with-resources statement. At the end of the `try`-with-resources block, the confined memory session is closed and all the off-heap memory associated with it is released.

Dereferencing Off-Heap Memory

After creating an empty memory segment, `nativeString`, the next step is to copy the characters of the Java string into the segment:

```
for(int i = 0; i < s.length(); i++) {
    nativeString.set(ValueLayout.JAVA_BYTE, i, (byte)s.charAt(i));
}

// Add the string terminator at the end
nativeString.set(ValueLayout.JAVA_BYTE, s.length(),(byte)0);
```

The first argument of the `MemorySegment::set` method is of type `ValueLayout.OfByte`. A *value layout* models the memory layout associated with values of basic data types such as primitives. A value layout encodes the size, the endianness, the alignment of the piece of memory to be dereferenced, and the Java type to be used for the dereference operation. In this example, `ValueLayout.JAVA_BYTE` has a size of one byte and native endianness, although this is irrelevant as the example is only reading a single byte.

When using this layout, the API expects clients to encode dereferenced values in Java as the primitive type `byte`. The second argument of the `MemoryLayout::set` method is the index, relative to the address of the memory segment, where to write the `byte` value.

You can allocate and populate more complex data types in off-heap memory. See Memory Layouts and Structured Access.

Methods That Allocate and Populate Off-Heap Memory

The `SegmentAllocator` interface contains methods that both allocate off-heap memory and copy Java data into it. The following `invokeStrlen(String s)` example uses the class `MemorySession`, which implements the `SegmentAllocator` class. The example calls the method `SegmentAllocator::allocateUtf8String`, which converts a string into a UTF-8 encoded, null-terminated C string, then stores the result into a memory segment.

```
static long invokeStrlen(String s) throws Throwable {

    try (MemorySession session = MemorySession.openConfined()) {

        // 1. Allocate off-heap memory, and
        // 2. Dereference off-heap memory
        MemorySegment nativeString = session.allocateUtf8String(s);
        // ...
    }
}
```

Linking and Calling a C Function

Calling a native function, such as the `strlen` C standard library function, involves the following steps:

1. Obtaining an Instance of the Native Linker
2. Locating the Address of the C Function
3. Creating the Description of the C Function Signature
4. Creating the Downcall Handle for the C Function
5. Calling the C Function Directly from Java

Obtaining an Instance of the Native Linker

A *linker* provides access to foreign functions from Java code, and access to Java code from foreign functions. The *native linker* 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

address of the native function. The following statements obtain the address of the `strlen` function:

```
            Linker linker = Linker.nativeLinker();
            SymbolLookup libc = SymbolLookup.libraryLookup("libc.so.6",
session);
            MemorySegment strlen_addr = libc.lookup("strlen").get();
```

`SymbolLookup::libraryLookup(String, MemorySession)` creates a library lookup, which locates all the symbols in a user-specified native library. It loads the native library and associates it with a `MemorySession` object. In this example, `libc.so.6` is the file name of the C standard library for many Linux systems.

Because `strlen` is part of the C standard library, you can use instead the native linker's *default lookup*. This is a symbol lookup for symbols in a set of commonly used libraries (including the C standard library). This means that you don't have to specify a system-dependent library file name:

```
            // 3a. Obtain an instance of the native linker
            Linker linker = Linker.nativeLinker();

            // 3b. Locate the address of the C function
            SymbolLookup stdLib = linker.defaultLookup();
            MemorySegment strlen_addr = stdLib.lookup("strlen").get();
```

Creating the Description of 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 arguments and its return value (if any). The following creates a function descriptor for the `strlen` function:

```
            // 3c. 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 the native primitive type. 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.

The subsequent arguments of `FunctionDescriptor::of` are the layouts of the native function's arguments. In this example, it's `ValueLayout.ADDRESS`; all pointer types are modeled with this value layout. Note that `struct` types, which aren't shown here, are modeled with struct layouts. See Memory Layouts and Structured Access.

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.

```
            // 3d. Create a downcall handle for the C function
            MethodHandle strlen = linker.downcallHandle(strlen_addr,
strlen_sig);
```

A downcall method handle has a `MethodType` associated with it, which represents the arguments and return type accepted and returned by a method handle. In this example, the `MethodType` associated with `strlen` has one parameter, `Addressable`, and its return type is `long`. Call `MethodHandle::type` to obtain a method handle's `MethodType`. See Downcall method handles in the `Linker` interface JavaDoc API specification for information about how a method's type is derived.

Calling the C Function Directly from Java

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

```
            // 3e. Call the C function directly from Java
            return (long)strlen.invoke(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. More specifically, it 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:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

It takes four arguments:

- `base`: Pointer to 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 function that compares two elements

The following example calls the `qsort` function to sort an `int` array. However, this method requires a pointer to a function that compares two array elements. The example defines a comparison method (`Qsort::qsortCompare`), creates a method

handle to represent this comparison method, and then creates a function pointer from this method handle.

```
    class Qsort {
        static int qsortCompare(MemoryAddress addr1, MemoryAddress addr2) {
            return Integer.compare(
                addr1.get(ValueLayout.JAVA_INT, 0),
                addr2.get(ValueLayout.JAVA_INT, 0));
        }
    }

    static int[] qsortTest (int[] unsortedArray) throws Throwable {

        int[] sortedArray;

        try (MemorySession session = MemorySession.openConfined()) {

            // Allocate off-heap memory and store unsortedArray in it
            MemorySegment array = session.allocateArray(
                ValueLayout.JAVA_INT,
                unsortedArray);

            // Obtain instance of native linker
            Linker linker = Linker.nativeLinker();

            // Create downcall handle for qsort
            MethodHandle qsort = linker.downcallHandle(
                linker.defaultLookup().lookup("qsort").get(),
                FunctionDescriptor.ofVoid(
                    ValueLayout.ADDRESS,
                    ValueLayout.JAVA_LONG,
                    ValueLayout.JAVA_LONG,
                    ValueLayout.ADDRESS)
            );

            // Create method handle for qsortCompare
            FunctionDescriptor compareFunc_sig = FunctionDescriptor.of(
                ValueLayout.JAVA_INT,
                ValueLayout.ADDRESS,
                ValueLayout.ADDRESS);

            MethodHandle comparHandle = MethodHandles.lookup().findStatic(
                Qsort.class, "qsortCompare",
Linker.upcallType(compareFunc_sig));

            // Create function pointer for qsortCompare
            MemorySegment comparFunc = linker.upcallStub(comparHandle,
                compareFunc_sig, session);

            // Call qsort
            qsort.invoke(array, (long) unsortedArray.length, 4L, comparFunc);

            // Dereference off-heap memory
            sortedArray = array.toArray(ValueLayout.JAVA_INT);
        }
```

```
        return sortedArray;
    }
```

The following statement allocates off-heap memory, then stores the `int` array to be sorted in it:

```
            // Allocate off-heap memory and store unsortedArray in it
            MemorySegment array = session.allocateArray(
                ValueLayout.JAVA_INT,
                unsortedArray);
```

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

```
            // Obtain instance of native linker
            Linker linker = Linker.nativeLinker();

            // Create downcall handle for qsort
            MethodHandle qsort = linker.downcallHandle(
                linker.defaultLookup().lookup("qsort").get(),
                FunctionDescriptor.ofVoid(
                    ValueLayout.ADDRESS,
                    ValueLayout.JAVA_LONG,
                    ValueLayout.JAVA_LONG,
                    ValueLayout.ADDRESS)
            );
```

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

```
    class Qsort {
        static int qsortCompare(MemoryAddress addr1, MemoryAddress
addr2) {
            return Integer.compare(
                addr1.get(ValueLayout.JAVA_INT, 0),
                addr2.get(ValueLayout.JAVA_INT, 0));
        }
    }
```

In this method, the `int` values are represented by `MemoryAddress` objects. A *memory address* models a reference to a memory location. To obtain a value from a memory address, 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 addresses in this example store only one value.

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

```
            // Create method handle for qsortCompare
            FunctionDescriptor compareFunc_sig = FunctionDescriptor.of(
                ValueLayout.JAVA_INT,
                ValueLayout.ADDRESS,
```

```
            ValueLayout.ADDRESS);

        MethodHandle comparHandle = MethodHandles.lookup().findStatic(
            Qsort.class, "qsortCompare",
Linker.upcallType(compareFunc_sig));
```

The `MethodHandles.Lookup::findStatic` 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.

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

```
// Create function pointer for qsortCompare
MemorySegment comparFunc = linker.upcallStub(comparHandle,
    compareFunc_sig, session);
```

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::qsortCompare`
- The memory session to associate with the function pointer (which is a memory segment)

The following statement calls the `qsort` function:

```
// Call qsort
qsort.invoke(array, (long) unsortedArray.length, 4L, comparFunc);
```

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

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

```
// Dereference off-heap memory
sortedArray = array.toArray(ValueLayout.JAVA_INT);
```

Memory Layouts and Structured Access

Accessing structured data using only basic dereferencing 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 such a native array as follows:

```
        try (MemorySession session = MemorySession.openConfined()) {

            MemorySegment segment =
                MemorySegment.allocateNative(2 * 4 * 10, session);

            for (int i = 0; i < 10; i++) {
                segment.setAtIndex(ValueLayout.JAVA_INT, (i * 2),
i); // x
                segment.setAtIndex(ValueLayout.JAVA_INT, (i * 2) + 1,
i); // y
            }
            // ...
        }
```

The `MemorySegment::allocateNative` method calculates the number of bytes required for the array. The `MemorySegment::setAtIndex` method calculates which memory address offsets to write into each member of a `Point` structure. 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 (MemorySession session = MemorySession.openConfined()) {

            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 =
                MemorySegment.allocateNative(ptsLayout, session);

            for (int i = 0; i < ptsLayout.elementCount(); i++) {
                xHandle.set(segment, (long) i, i);
                yHandle.set(segment, (long) i, i);
```

```
        }
        // ...
    }
```

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 `GroupLayout` objects. The method `MemoryLayout::structLayout` returns a `GroupLayout` 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.

```
VarHandle xHandle
    = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("x"));
VarHandle yHandle
    = ptsLayout.varHandle(PathElement.sequenceElement(),
        PathElement.groupElement("y"));
```

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` statement calls `VarHandle::set` to dereference memory like `MemorySegment::setAtIndex`. In this example, it sets a value (the second argument) at an index (the third argument) in a memory segment (the first argument). 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.

```
MemorySegment segment =
    MemorySegment.allocateNative(ptsLayout, session);

for (int i = 0; i < ptsLayout.elementCount(); i++) {
    xHandle.set(segment, (long) i, i);
    yHandle.set(segment, (long) i, i);
}
```

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 suppress this message, add the `--enable-native-access=ALL-UNNAMED` command-line option.

- `static Linker nativeLinker()`, `SymbolLookup.libraryLookup(String, MemorySession)` and `SymbolLookup.libraryLookup(Path, MemorySession)`: These methods are required to create a downcall method handle, which is intrinsically unsafe. 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.

  JVM crashes might occur with upcalls because they are typically invoked in the context of a downcall method handle invocation.
- All dereferencing methods of `MemoryAddress`: Because a memory address does not feature temporal or spatial bounds, the runtime has no way to check the correctness of the memory dereference operation.
- `MemorySegment::ofAddress` and `VaList::ofAddress`: Sometimes it's necessary to turn a memory address obtained from native code into a memory segment with full spatial, temporal and confinement bounds. To do this, clients can obtain a native segment *unsafely* from a memory address by providing the segment size as well as the segment session. This is a restricted operation because, for instance, an incorrect segment size could result in a JVM crash when attempting to dereference the memory segment.

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 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 qsort Function from 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 -l <absolute path of Python shared library> \
  --output classes \
  -I <directory containing Python header files> \
  -t org.python <absolute path of Python.h>
```

> **Note:**
>
> - On Linux systems, to obtain the file name of the Python shared library, run the following command. This example assumes that you have Python 3 installed in your system.
>
>   ```
>   ldd $(which python3)
>   ```
>
> - 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.
> - If you want to examine the classes and methods that the `jextract` tool creates, run the command with the `--source` option. For example, the following command generates the source files of the Java bindings for `Python.h`:
>
>   ```
>   jextract --source \
>     --output src \
>     -I <directory containing Python header files> \
>     -t org.python <absolute path of Python.h>
>   ```

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

```
import java.lang.foreign.MemoryAddress;
import java.lang.foreign.MemorySegment;
import java.lang.foreign.MemorySession;
import static org.python.Python_h.*;
import org.python.*;

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 (MemorySession session = MemorySession.openConfined()) {
                MemorySegment nativeString =
    session.allocateUtf8String(script);
                PyRun_SimpleStringFlags(
                    nativeString,
                    MemoryAddress.NULL);
                Py_Finalize();
            }
            Py_Exit(0);
        }
    }
```

3. Compile `PythonMain.java` with the following command:

```
javac --enable-preview -source 19 \
  -classpath classes \
  PythonMain.java
```

4. Run `PythonMain` with the following command:

```
java -cp classes:. -Djava.library.path=<location of Python shared
library> PythonMain
```

Call qsort Function from Java Application

As mentioned previously, `qsort` 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 `jextract`, create an upcall handle for the comparison function required by `qsort`, and then call the `qsort` 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 classes -t org.unix <absolute path to stdlib.h>
```

   The generated Java bindings for `stdlib.h` include a Java class named `stdlib_h`, which includes a Java method named `qsort(Addressable, long, long, Addressable)`, and a Java interface named `__compar_fn_t`, which includes a method named `allocate` that creates a function pointer for the comparison function required by the `qsort` function. To examine the source code of the Java bindings that `jextract` generates, run the tool with the `-source` option:

```
jextract --source --output src -t org.unix <absolute path to
stdlib.h>
```

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

```
import static org.unix.__compar_fn_t.*;
import static org.unix.stdlib_h.*;
import java.lang.foreign.*;
import java.lang.invoke.*;

public class QsortMain {

    public static void main(String[] args) {

        int[] unsortedArray = new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 };

        try (MemorySession session = MemorySession.openConfined()) {

            // Allocate off-heap memory and store unsortedArray in it
            MemorySegment array = session.allocateArray(
                ValueLayout.JAVA_INT,
                unsortedArray);

            // Create upcall for comparison function
            MemorySegment comparFunc = allocate(
                (addr1, addr2) ->
                    Integer.compare(
                        addr1.get(ValueLayout.JAVA_INT, 0),
                        addr2.get(ValueLayout.JAVA_INT, 0)),
                    session);

            // Call qsort
            qsort(array, (long) unsortedArray.length, 4L,
comparFunc);

            // Dereference off-heap memory
            int[] sortedArray = array.toArray(ValueLayout.JAVA_INT);

            for (int num : sortedArray) {
                System.out.print(num + " ");
            }
            System.out.println();
        }
    }
}
```

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

```
            // Create upcall for comparison function
            MemorySegment comparFunc = allocate(
                (addr1, addr2) ->
                    Integer.compare(
                        addr1.get(ValueLayout.JAVA_INT, 0),
                        addr2.get(ValueLayout.JAVA_INT, 0)),
                    session);
```

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 --enable-preview -source 19 -cp classes QsortMain.java
```

4. Run `QsortMain` with the following command:

```
java --enable-preview -cp classes:. QsortMain
```