Module java.base
Package java.util

Class ServiceLoader<S>

java.lang.Object
    java.util.ServiceLoader<S>

Type Parameters:

S - The type of the service to be loaded by this loader

All Implemented Interfaces:

Iterable<S>

```
public final class ServiceLoader<S>
extends Object
implements Iterable<S>
```

A facility to load implementations of a service.

A *service* is a well-known interface or class for which zero, one, or many service providers exist. A *service provider* (or just *provider*) is a class that implements or subclasses the well-known interface or class. A `ServiceLoader` is an object that locates and loads service providers deployed in the run time environment at a time of an application's choosing. Application code refers only to the service, not to service providers, and is assumed to be capable of choosing between multiple service providers (based on the functionality they expose through the service), and handling the possibility that no service providers are located.

Obtaining a service loader

An application obtains a service loader for a given service by invoking one of the static `load` methods of `ServiceLoader`. If the application is a module, then its module declaration must have a *uses* directive that specifies the service; this helps to locate providers and ensure they will execute reliably. In addition, if the application module does not contain the service, then its module declaration must have a *requires* directive that specifies the module which exports the service. It is strongly recommended that the application module does **not** require modules which contain providers of the service.

A service loader can be used to locate and instantiate providers of the service by means of the `iterator` method. `ServiceLoader` also defines the `stream` method to obtain a stream of providers that can be inspected and filtered without instantiating them.

As an example, suppose the service is `com.example.CodecFactory`, an interface that defines methods for producing encoders and decoders:

```
     package com.example;
     public interface CodecFactory {
         Encoder getEncoder(String encodingName);
         Decoder getDecoder(String encodingName);
     }
```

The following code obtains a service loader for the `CodecFactory` service, then uses its iterator (created automatically by the enhanced-for loop) to yield instances of the service providers that are located:

```
     ServiceLoader<CodecFactory> loader = ServiceLoader.load(CodecFactory.class);
     for (CodecFactory factory : loader) {
         Encoder enc = factory.getEncoder("PNG");
         if (enc != null)
             ... use enc to encode a PNG file
             break;
         }
```

If this code resides in a module, then in order to refer to the `com.example.CodecFactory` interface, the module declaration would require the module which exports the interface. The module declaration would also specify use of `com.example.CodecFactory`:

```
     requires com.example.codec.core;
     uses com.example.CodecFactory;
```

Sometimes an application may wish to inspect a service provider before instantiating it, in order to determine if an instance of that service provider would be useful. For example, a service provider for `CodecFactory` that is capable of producing a "PNG" encoder may be annotated with `@PNG`. The following code uses service loader's `stream` method to yield instances of `Provider<CodecFactory>` in contrast to how the iterator yields instances of `CodecFactory`:

```
     ServiceLoader<CodecFactory> loader = ServiceLoader.load(CodecFactory.class);
     Set<CodecFactory> pngFactories = loader
            .stream()                                              // Note a below
            .filter(p -> p.type().isAnnotationPresent(PNG.class))  // Note b
            .map(Provider::get)                                    // Note c
            .collect(Collectors.toSet());
```

a. A stream of `Provider<CodecFactory>` objects
b. `p.type()` yields a `Class<CodecFactory>`
c. `get()` yields an instance of `CodecFactory`

Designing services

A service is a single type, usually an interface or abstract class. A concrete class can be used, but this is not recommended. The type may have any accessibility. The methods of a service are highly domain-specific, so this API specification cannot give concrete advice about their form or function. However, there are two general guidelines:

1. A service should declare as many methods as needed to allow service providers to communicate their domain-specific properties and other quality-of-implementation factors. An application which obtains a service loader for the service may then invoke these methods on each instance of a service provider, in order to choose the best provider for the application.

2. A service should express whether its service providers are intended to be direct implementations of the service or to be an indirection mechanism such as a "proxy" or a "factory". Service providers tend to be indirection mechanisms when domain-specific objects are relatively expensive to instantiate; in this case, the service should be designed so that service providers are abstractions which create the "real" implementation on demand. For example, the `CodecFactory` service expresses through its name that its service providers are factories for codecs, rather than codecs themselves, because it may be expensive or complicated to produce certain codecs.

Developing service providers

A service provider is a single type, usually a concrete class. An interface or abstract class is permitted because it may declare a static provider method, discussed later. The type must be public and must not be an inner class.

A service provider and its supporting code may be developed in a module, which is then deployed on the application module path or in a modular image. Alternatively, a service provider and its supporting code may be packaged as a JAR file and deployed on the application class path. The advantage of developing a service provider in a module is that the provider can be fully encapsulated to hide all details of its implementation.

An application that obtains a service loader for a given service is indifferent to whether providers of the service are deployed in modules or packaged as JAR files. The application instantiates service providers via the service loader's iterator, or via `Provider` objects in the service loader's stream, without knowledge of the service providers' locations.

Deploying service providers as modules

A service provider that is developed in a module must be specified in a *provides* directive in the module declaration. The provides directive specifies both the service and the service provider; this helps to locate the provider when another module, with a *uses* directive for the service, obtains a service loader for the service. It is strongly recommended that the module does not export the package containing the service provider. There is no support for a module specifying, in a *provides* directive, a service provider in another module.

A service provider that is developed in a module has no control over when it is instantiated, since that occurs at the behest of the application, but it does have control over how it is instantiated:

- If the service provider declares a provider method, then the service loader invokes that method to obtain an instance of the service provider. A provider method is a public static method named "provider" with no formal parameters and a return type that is assignable to the service's interface or class.

  In this case, the service provider itself need not be assignable to the service's interface or class.

- If the service provider does not declare a provider method, then the service provider is instantiated directly, via its provider constructor. A provider constructor is a public constructor with no formal parameters.

  In this case, the service provider must be assignable to the service's interface or class

A service provider that is deployed as an automatic module on the application module path must have a provider constructor. There is no support for a provider method in this case.

As an example, suppose a module specifies the following directive:

```
    provides com.example.CodecFactory with com.example.impl.StandardCodecs,
             com.example.impl.ExtendedCodecsFactory;
```

where

- `com.example.CodecFactory` is the two-method service from earlier.
- `com.example.impl.StandardCodecs` is a public class that implements `CodecFactory` and has a public no-args constructor.
- `com.example.impl.ExtendedCodecsFactory` is a public class that does not implement CodecFactory, but it declares a public static no-args method named "provider" with a return type of `CodecFactory`.

A service loader will instantiate `StandardCodecs` via its constructor, and will instantiate `ExtendedCodecsFactory` by invoking its `provider` method. The requirement that the provider constructor or provider method is public helps to document the intent that the class (that is, the service provider) will be instantiated by an entity (that is, a service loader) which is outside the class's package.

Deploying service providers on the class path

A service provider that is packaged as a JAR file for the class path is identified by placing a *provider-configuration file* in the resource directory `META-INF/services`. The name of the provider-configuration file is the fully qualified binary name of the service. The provider-configuration file contains a list of fully qualified binary names of service providers, one per line.

For example, suppose the service provider `com.example.impl.StandardCodecs` is packaged in a JAR file for the class path. The JAR file will contain a provider-configuration file named:

```
META-INF/services/com.example.CodecFactory
```

that contains the line:

```
com.example.impl.StandardCodecs # Standard codecs
```

The provider-configuration file must be encoded in UTF-8. Space and tab characters surrounding each service provider's name, as well as blank lines, are ignored. The comment character is `'#'` (`U+0023` NUMBER SIGN); on each line all characters following the first comment character are ignored. If a service provider class name is listed more than once in a provider-configuration file then the duplicate is ignored. If a service provider class is named in more than one configuration file then the duplicate is ignored.

A service provider that is mentioned in a provider-configuration file may be located in the same JAR file as the provider-configuration file or in a different JAR file. The service provider must be visible from the class loader that is initially queried to locate the provider-configuration file; this is not necessarily the class loader which ultimately locates the provider-configuration file.

Timing of provider discovery

Service providers are loaded and instantiated lazily, that is, on demand. A service loader maintains a cache of the providers that have been loaded so far. Each invocation of the `iterator` method returns an `Iterator` that first yields all of the elements cached from previous iteration, in instantiation order, and then lazily locates and instantiates any remaining providers, adding each one to the cache in turn. Similarly, each invocation of the stream method returns a `Stream` that first processes all providers loaded by previous stream operations, in load order, and then lazily locates any remaining providers. Caches are cleared via the `reload` method.

Errors

When using the service loader's `iterator`, the `hasNext` and `next` methods will fail with `ServiceConfigurationError` if an error occurs locating, loading or instantiating a service provider. When processing the service loader's stream then `ServiceConfigurationError` may be thrown by any method that causes a service provider to be located or loaded.

When loading or instantiating a service provider in a module, `ServiceConfigurationError` can be thrown for the following reasons:

- The service provider cannot be loaded.
- The service provider does not declare a provider method, and either it is not assignable to the service's interface/class or does not have a provider constructor.
- The service provider declares a public static no-args method named "provider" with a return type that is not assignable to the service's interface or class.
- The service provider class file has more than one public static no-args method named "`provider`".
- The service provider declares a provider method and it fails by returning `null` or throwing an exception.
- The service provider does not declare a provider method, and its provider constructor fails by throwing an exception.

When reading a provider-configuration file, or loading or instantiating a provider class named in a provider-configuration file, then `ServiceConfigurationError` can be thrown for the following reasons:

- The format of the provider-configuration file violates the format specified above;
- An `IOException` occurs while reading the provider-configuration file;
- A service provider cannot be loaded;
- A service provider is not assignable to the service's interface or class, or does not define a provider constructor, or cannot be instantiated.

Security

Service loaders always execute in the security context of the caller of the iterator or stream methods and may also be restricted by the security context of the caller that created the service loader. Trusted system code should typically invoke the methods in this class, and the methods of the iterators which they return, from within a privileged security context.

Concurrency

Instances of this class are not safe for use by multiple concurrent threads.

Null handling

Unless otherwise specified, passing a `null` argument to any method in this class will cause a `NullPointerException` to be thrown.

Since:

1.6

Nested Class Summary

Nested Classes

Modifier and Type	Class	Description

static interface	`ServiceLoader.Provider<S>`	Represents a service provider located by `ServiceLoader`.

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`Optional<S>`	`findFirst()`	Load the first available service provider of this loader's service.
`Iterator<S>`	`iterator()`	Returns an iterator to lazily load and instantiate the available providers of this loader's service.
`static <S> ServiceLoader<S>`	`load(Class<S> service)`	Creates a new service loader for the given service type, using the current thread's context class loader.
`static <S> ServiceLoader<S>`	`load(Class<S> service, ClassLoader loader)`	Creates a new service loader for the given service.
`static <S> ServiceLoader<S>`	`load(ModuleLayer layer, Class<S> service)`	Creates a new service loader for the given service type to load service providers from modules in the given module layer and its ancestors.
`static <S> ServiceLoader<S>`	`loadInstalled(Class<S> service)`	Creates a new service loader for the given service type, using the platform class loader.
`void`	`reload()`	Clear this loader's provider cache so that all providers will be reloaded.
`Stream<ServiceLoader.Provider<S>`	`stream()`	Returns a stream to lazily load available providers of this loader's service.
`String`	`toString()`	Returns a string describing this service.

Methods declared in class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Methods declared in interface java.lang.Iterable

`forEach`, `spliterator`

Method Details

iterator

```
public Iterator<S> iterator()
```

Returns an iterator to lazily load and instantiate the available providers of this loader's service.

To achieve laziness the actual work of locating and instantiating providers is done by the iterator itself. Its `hasNext` and `next` methods can therefore throw a `ServiceConfigurationError` for any of the reasons specified in the Errors section above. To write robust code it is only necessary to catch `ServiceConfigurationError` when using the iterator. If an error is thrown then subsequent invocations of the iterator will make a best effort to locate and instantiate the next available provider, but in general such recovery cannot be guaranteed.

Caching: The iterator returned by this method first yields all of the elements of the provider cache, in the order that they were loaded. It then lazily loads and instantiates any remaining service providers, adding each one to the cache in turn. If this loader's provider caches are cleared by invoking the `reload` method then existing iterators for this service loader should be discarded. The `hasNext` and `next` methods of the iterator throw `ConcurrentModificationException` if used after the provider cache has been cleared.

The iterator returned by this method does not support removal. Invoking its `remove` method will cause an `UnsupportedOperationException` to be thrown.

Specified by:

`iterator` in interface `Iterable<S>`

API Note:

Throwing an error in these cases may seem extreme. The rationale for this behavior is that a malformed provider-configuration file, like a malformed class file, indicates a serious problem with the way the Java virtual machine is configured or is being used. As such it is preferable to throw an error rather than try to recover or, even worse, fail silently.

Returns:

An iterator that lazily loads providers for this loader's service

stream

```
public Stream<ServiceLoader.Provider<S>> stream()
```

Returns a stream to lazily load available providers of this loader's service. The stream elements are of type `Provider`, the `Provider`'s `get` method must be invoked to get or instantiate the provider.

To achieve laziness the actual work of locating providers is done when processing the stream. If a service provider cannot be loaded for any of the reasons specified in the Errors section above then `ServiceConfigurationError` is thrown by whatever method caused the service provider to be loaded.

Caching: When processing the stream then providers that were previously loaded by stream operations are processed first, in load order. It then lazily loads any remaining service providers. If this loader's provider caches are cleared by invoking the `reload` method then existing streams for this service loader should be discarded. The returned stream's source `spliterator` is *fail-fast* and will throw `ConcurrentModificationException` if the provider cache has been cleared.

The following examples demonstrate usage. The first example creates a stream of `CodecFactory` objects, the second example is the same except that it sorts the providers by provider class name (and so locate all providers).

```
    Stream<CodecFactory> providers = ServiceLoader.load(CodecFactory.class)
            .stream()
            .map(Provider::get);

    Stream<CodecFactory> providers = ServiceLoader.load(CodecFactory.class)
            .stream()
            .sorted(Comparator.comparing(p -> p.type().getName()))
            .map(Provider::get);
```

Returns:

A stream that lazily loads providers for this loader's service

Since:

9

load

```
public static <S> ServiceLoader<S> load(Class<S> service,
                                        ClassLoader loader)
```

Creates a new service loader for the given service. The service loader uses the given class loader as the starting point to locate service providers for the service. The service loader's `iterator` and `stream` locate providers in both named and unnamed modules, as follows:

- Step 1: Locate providers in named modules.

  Service providers are located in all named modules of the class loader or to any class loader reachable via parent delegation.

  In addition, if the class loader is not the bootstrap or platform class loader, then service providers may be located in the named modules of other class loaders. Specifically, if the class loader, or any class loader reachable via parent delegation, has a module in a module layer, then service providers in all modules in the module layer are located.

  For example, suppose there is a module layer where each module is in its own class loader (see `defineModulesWithManyLoaders`). If this `ServiceLoader.load` method is invoked to locate providers using any of the class loaders created for the module layer, then it will locate all of the providers in the module layer, irrespective of their defining class loader.

  Ordering: The service loader will first locate any service providers in modules defined to the class loader, then its parent class loader, its parent parent, and so on to the bootstrap class loader. If a class loader has modules in a module layer then all providers in that module layer are located (irrespective of their class loader) before the providers in the parent class loader are located. The ordering of modules in same class loader, or the ordering of modules in a module layer, is not defined.

  If a module declares more than one provider then the providers are located in the order that its module descriptor lists the providers. Providers added dynamically by instrumentation agents (see `redefineModule`) are always located after providers declared by the module.

- Step 2: Locate providers in unnamed modules.

  Service providers in unnamed modules are located if their class names are listed in provider-configuration files located by the class loader's `getResources` method.

  The ordering is based on the order that the class loader's `getResources` method finds the service configuration files and within that, the order that the class names are listed in the file.

In a provider-configuration file, any mention of a service provider that is deployed in a named module is ignored. This is to avoid duplicates that would otherwise arise when a named module has both a *provides* directive and a provider-configuration file that mention the same service provider.

The provider class must be visible to the class loader.

API Note:

If the class path of the class loader includes remote network URLs then those URLs may be dereferenced in the process of searching for provider-configuration files.

This activity is normal, although it may cause puzzling entries to be created in web-server logs. If a web server is not configured correctly, however, then this activity may cause the provider-loading algorithm to fail spuriously.

A web server should return an HTTP 404 (Not Found) response when a requested resource does not exist. Sometimes, however, web servers are erroneously configured to return an HTTP 200 (OK) response along with a helpful HTML error page in such cases. This will cause a `ServiceConfigurationError` to be thrown when this class attempts to parse the HTML page as a provider-configuration file. The best solution to this problem is to fix the misconfigured web server to return the correct response code (HTTP 404) along with the HTML error page.

Type Parameters:

`S` - the class of the service type

Parameters:

`service` - The interface or abstract class representing the service

`loader` - The class loader to be used to load provider-configuration files and provider classes, or `null` if the system class loader (or, failing that, the bootstrap class loader) is to be used

Returns:

A new service loader

Throws:

`ServiceConfigurationError` - if the service type is not accessible to the caller or the caller is in an explicit module and its module descriptor does not declare that it uses `service`

load

```
public static <S> ServiceLoader<S> load(Class<S> service)
```

Creates a new service loader for the given service type, using the current thread's context class loader.

An invocation of this convenience method of the form

```
    ServiceLoader.load(service)
```

is equivalent to

```
    ServiceLoader.load(service, Thread.currentThread().getContextClassLoader())
```

API Note:

Service loader objects obtained with this method should not be cached VM-wide. For example, different applications in the same VM may have different thread context class loaders. A lookup by one application may locate a service provider that is only visible via its thread context class loader and so is not suitable to be located by the other application. Memory leaks can also arise. A thread local may be suited to some applications.

Type Parameters:

`S` - the class of the service type

Parameters:

`service` - The interface or abstract class representing the service

Returns:

A new service loader

Throws:

`ServiceConfigurationError` - if the service type is not accessible to the caller or the caller is in an explicit module and its module descriptor does not declare that it uses `service`

loadInstalled

```
public static <S> ServiceLoader<S> loadInstalled(Class<S> service)
```

Creates a new service loader for the given service type, using the platform class loader.

This convenience method is equivalent to:

```
    ServiceLoader.load(service, ClassLoader.getPlatformClassLoader())
```

This method is intended for use when only installed providers are desired. The resulting service will only find and load providers that have been installed into the current Java virtual machine; providers on the application's module path or class path will be ignored.

Type Parameters:

`S` - the class of the service type

Parameters:

`service` - The interface or abstract class representing the service

Returns:

A new service loader

Throws:

`ServiceConfigurationError` - if the service type is not accessible to the caller or the caller is in an explicit module and its module descriptor does not declare that it uses `service`

load

```
public static <S> ServiceLoader<S> load(ModuleLayer layer,
                                        Class<S> service)
```

Creates a new service loader for the given service type to load service providers from modules in the given module layer and its ancestors. It does not locate providers in unnamed modules. The ordering that the service loader's `iterator` and `stream` locate providers and yield elements is as follows:

- Providers are located in a module layer before locating providers in parent layers. Traversal of parent layers is depth-first with each layer visited at most once. For example, suppose L0 is the boot layer, L1 and L2 are modules layers with L0 as their parent. Now suppose that L3 is created with L1 and L2 as the parents (in that order). Using a service loader to locate providers with L3 as the context will locate providers in the following order: L3, L1, L0, L2.
- If a module declares more than one provider then the providers are located in the order that its module descriptor lists the providers. Providers added dynamically by instrumentation agents are always located after providers declared by the module.
- The ordering of modules in a module layer is not defined.

API Note:

Unlike the other load methods defined here, the service type is the second parameter. The reason for this is to avoid source compatibility issues for code that uses `load(S, null)`.

Type Parameters:

`S` - the class of the service type

Parameters:

`layer` - The module layer

`service` - The interface or abstract class representing the service

Returns:

A new service loader

Throws:

`ServiceConfigurationError` - if the service type is not accessible to the caller or the caller is in an explicit module and its module descriptor does not declare that it uses `service`

Since:

9

findFirst

```
public Optional<S> findFirst()
```

Load the first available service provider of this loader's service. This convenience method is equivalent to invoking the `iterator()` method and obtaining the first element. It therefore returns the first element from the provider cache if possible, it otherwise attempts to load and instantiate the first provider.

The following example loads the first available service provider. If no service providers are located then it uses a default implementation.

```
    CodecFactory factory = ServiceLoader.load(CodecFactory.class)
                                        .findFirst()
                                        .orElse(DEFAULT_CODECSET_FACTORY);
```

Returns:

The first service provider or empty `Optional` if no service providers are located

Throws:

`ServiceConfigurationError` - If a provider class cannot be loaded for any of the reasons specified in the Errors section above.

Since:

9

reload

```
public void reload()
```

Clear this loader's provider cache so that all providers will be reloaded.

After invoking this method, subsequent invocations of the `iterator` or `stream` methods will lazily locate providers (and instantiate in the case of `iterator`) from scratch, just as is done by a newly-created service loader.

This method is intended for use in situations in which new service providers can be installed into a running Java virtual machine.

toString

```
public String toString()
```

Returns a string describing this service.

Overrides:

`toString` in class `Object`

Returns:

A descriptive string