MicroEJ
KF-1.4

Trusted Execution Environment
Kernel & Features
Profile Specification
ESR0020

Reference: ESR-SPE-0020-KF
Version: 1.4
Rev: H
Authors: Fred Rivard, Frédéric Rivière, Jérôme Leroux

Copyright of The Software

DEFINITIONS

"ESR" means the Specification, including any modifications and upgrades, where these terms have been stated or referred to, and made available to You by MicroEJ, including without limitation, texts, drawing, codes,and examples.

"MicroEJ" means MicroEJ S.A. , operating under the brand name MicroEJ®, Société anonyme à conseil de surveillance et directoire which main offices are at Nantes, 11 rue du chemin rouge, 44373 Nantes, France, Registered under number 452870579, in France in accordance with the French law.

"You" means the legal entity or entities represented by the individual executing this Agreement.

READ ONLY RIGHTS

Subject to the terms and conditions contained herein, MicroEJ grants to You a non-exclusive, non-transferable, worldwide, and royalty-free license to view and read the ESR solely for purposes of Your internal evaluation. As a condition of the license grant, You shall not copy, modify, create derivative works of, publicly display, publicly perform, implement, disclose, distribute, or otherwise use the ESR, including without limitation, using the ESR to develop Software or Tool, similar or compatible with the software defined by the Specification.

INTELLECTUAL PROPERTY

The ESR is proprietary, protected under copyright law and patents. You have no right at any time to disclose, directly or indirectly, such material and/or information relating to the ESR, to any third party without MicroEJ's prior written approval.

GENERAL TERMS

THE ESR IS PROVIDED "AS IS", WITHOUT WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED.

THE READING OF THE ESR AND ALL CONSEQUENCES ARISING THEREOF IS YOUR SOLE RESPONSIBILITY. MICROEJ SHALL NOT BE LIABLE TO YOU FOR ANY LOSS OR DAMAGE CAUSED BY, ARISING FROM, DIRECTLY OR INDIRECTLY, OR IN CONNECTION WITH THE ESR.

MISCELLANEOUS

This Agreement shall be governed by, and interpreted in accordance with French Law. In no event shall this Agreement be construed against the drafter.

This Agreement contains the entire understanding between the parties concerning its subject matter and supersedes any other agreement or understanding, whether written or oral, which may exist or have existed between the parties on the subject matter hereof.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION.

MICROEJ MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN ANY ESR PUBLICATION AT ANY TIME.

Trademarks

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in cross-platform, networked environments. When it is used in this documentation

Contents

Tables

Illustrations

1 PREFACE TO KF PROFILE, ESR020

This document defines the KF profile, a Trusted Execution Environment (TEE) targeting virtual machines.

1.1 Who should use this specification?

This specification is targeted at the following audiences:

- Implementors of the KF specification.
- Application developers that target applications with the need of embedding third party software components that may be untrusted.
- Virtual machine providers.

1.2 Comments

Your comments about KF are welcome. Please send them by email to `contact@microej.com` with KF as subject.

1.3 Requirements

The term MUST indicates that the associated item is an absolute requirement.

The term MAY indicates that the associated item is optional.

The term SHOULD indicates that the associated item is highly recommended, but not required.

Although this specification defines minimal requirements, devices with more resources may also benefit from KF specification, especially when users are concerned with optimal resource usage.

The KF specification makes no hardware requirement for devices that run a Java virtual machine that implements this specification. Typical hardware for KF ranges from low-end 32-bit (such as Cortex-M0) to 64-bit multi-core cpu.

The KF profile specification makes minimal assumptions about the system software of the device. Although a Java virtual machine is required, the Kernel does not need to support an OS/RTOS while the virtual machine may be baremetal (i.e. the device boots directly in Java).

Compliant KF 1.4 implementations MUST include all packages, classes, and interfaces described in this specification, and implement the associated behavior.

1.4 Related Literature

JVM2: Tim Lindholm & Frank Yellin, The Java™ Virtual Machine Specification, Second Edition, 1999

JLS: James Gosling, Guy Steele, Bill Joy, Gilad Bracha, The Java™ Language Specification, Third Edition, 2005

OSGI: OSGi Alliance, https://osgi.org/download/r7/osgi.core-7.0.0.pdf, 2018

1.5 Document Conventions

In this document, references to methods of a Java class are written as `ClassName.methodName(args)`. This applies to both static and instance methods. Where the method is static this will be made clear in the accompanying text.

1.6 Implementation Notes

The KF specification does not include any implementation details. KF implementors are free to use whatever techniques they deem appropriate to implement the specification, with (or without) collaboration of any Java virtual machine provider. KF experts have taken great care not to mention any special virtual machines, nor any of their special features, in order to encourage fair competing implementations. Implementations are free to perform checks either at compile-time and/or at runtime.

2 INTRODUCTION

This specification defines a Trusted Execution Environment (TEE) for software modules called Features.

2.1 Basic Concepts

Kernel & Features semantic (KF) allows an application to be split into multiple parts:

- the main application, called the Kernel
- zero or more applications, called Features.

The Kernel is mandatory. It is assumed to be reliable, trusted and cannot be modified. If there is only one application (i.e. only one main entry point that the system starts with) then this application is called the Kernel.

A Feature is an application "extension" managed by the Kernel. A Feature is fully controlled by the Kernel: it can be installed (dynamically or statically pre-installed), started, stopped and uninstalled at any time independent of the system state (particularly, a Feature never depends on another Feature to be stopped). A Feature is optional, potentially not-trusted, maybe unreliable and can be executed without jeopardizing the safety of the Kernel execution and other Features.

Resources accesses (RAM, hardware peripherals, CPU time, …) are under control of the Kernel.

2.2 First Example

This simple example illustrates a log of a message called by a Kernel and a Feature. The `KernelExample` class is the main Kernel entry point. The `FeatureExample` class is a Feature entry point. The way these classes are assigned to contexts and how the Feature is installed is not described here. (the Feature is assumed to be installed before the Kernel main method starts).

2.2.1 Kernel class

```
package ej.kf.example.helloworld;

import ej.kf.Feature;
import ej.kf.Kernel;

/**
 * Defines a Kernel class. The Kernel entry point is the regular main method.
 */
public class KernelExample {

	public static void main(String[] args) throws Exception {
		log("Hello World !");
		for (Feature f : Kernel.getAllLoadedFeatures()) {
			f.start();
		}
	}

	/**
	 * Log a message, prefixed with the name of the caller
	 */
	public static void log(String message) {
		String name = Kernel.getContextOwner().getName();
		System.out.println('[' + name + "]: " + message);
	}

}
```

Illustration 2-1: Kernel Hello World Example

2.2.2 Feature class

```
package ej.kf.example.helloworld;

import ej.kf.FeatureEntryPoint;

/**
 * Defines a Feature class that implements {@link FeatureEntryPoint}
interface.
 */
public class FeatureExample implements FeatureEntryPoint {

	@Override
	public void start() {
		KernelExample.log("Hello World !");
	}

	@Override
	public void stop() {
	}

}
```

Illustration 2-2: Feature Hello World Example

2.2.3 Expected Output

```
[KERNEL]: Hello World !
[FEATURE]: Hello World !
```

3 OWNERSHIP RULES

At runtime, each type, object and thread execution context has an owner. This section defines ownership transmission and propagation rules.

3.1 Type

The owner of a type is fixed when such type is loaded and that owner cannot be modified after.

The owner of an array-of-type type is the owner of the type. Array of basetypes are lazily loaded. Those that are required by the Kernel are owned by the Kernel. Other arrays are loaded in any Feature that require them.

The owner of a type can be retrieved by calling `Kernel.getOwner(Object)` with the `Class` instance.

3.2 Object

When an object is created, it is assigned to the owner of the execution context owner.

The owner of an object can be retrieved by calling `Kernel.getOwner(Object)` with the given object.

3.3 Execution Context

When a thread is started, the first execution context is set to the owner of the thread object. When a method is called from Kernel mode (§3.4) and its receiver is owned by a Feature, the execution context is set to the owner of the receiver. In all other cases, the execution context of the method called is the execution context of the caller.

The owner of the current execution context can be retrieved by calling `Kernel.getContextOwner()`.

When a method returns, the execution context owner of the caller remains the one it was before the call was done.

The Kernel is the first application to run, and it is triggered by the system when it boots. The Kernel starts in Kernel mode, creating a first thread owned by the Kernel.

The Kernel can execute a dynamic piece of code (`java.lang.Runnable`) in a Feature context by calling `Kernel.runUnderContext()`.

3.4 Kernel Mode

An execution context is said to be in *Kernel mode* when the current execution context is owned by the Kernel. The method `Kernel.enter()` sets the current execution context owner to the Kernel. The method `Kernel.exit()` resets the current execution context owner to the one when the method `Kernel.enter()` was called.

4 EXECUTION RULES

Notes: this specification does not force all rules to be checked at runtime. When a rule is checked at runtime, a `java.lang.IllegalAccessError` must be thrown at the execution point where the rule is broken.

4.1 Type References

A type owned by the Kernel cannot refer to a type owned by a Feature.

A type owned by a Feature can refer to a type owned by the Kernel if and only if it has been exposed as an API type.

A type owned by a Feature cannot refer to a type owned by another Feature.

All the types of the KF library (package `ej.kf.*`) are owned by the Kernel. A type owned by a Feature cannot access any types of this library except the `ej.kf.FeatureEntryPoint` interface and the `ej.kf.Proxy` class.

4.2 Method References

A type owned by a Feature can refererence a method of type owned by the Kernel if and only if it has been exposed as an API method.

4.3 Field References

4.3.1 Instance Field References

A type owned by a Feature can refer to all instance fields of a type owned by the Kernel, if and only if the type has been exposed as an API type and the field is accessible according to [JLS] access control rules.

4.3.2 Static Field References

A type owned by a Feature can refer to a static field of a type owned by the Kernel if and only if it has been exposed as an API static field.

A static field of a type owned by a Feature cannot refer to an object owned by another Feature.

An object owned by a Feature can be assigned to a static field of a type owned by the Kernel if and only if the current execution context is in Kernel mode (§3.4), otherwise a `java.lang.IllegalAccessError` is thrown at runtime.

4.3.3 Context Local Static Field References

By default, a static field holding an object reference is stored in a single memory slot in the context of the owner of the type that defines the field.

The Kernel can declare a static field as a context local storage field in `kernel.intern` file (Section §9.6.1 for full format specification). A memory slot is then allocated for the Kernel and duplicated for each Feature. As it is a static field, it is initialized to `null`.

```
<kernel>
	<contextLocalStorage name="com.mycompany.MyType.MY_GLOBAL"/>
</kernel>
```

Illustration 4-1: Context Local Storage Declaration of a Static Field

The Kernel can declare an optional initialization method. This method is automatically invoked when the field is being read if its content is `null`. This gives a hook to lazily initialize the static field before its first read access. If the initialization method returns a `null` reference, a `java.lang.NullPointerException` is thrown.

```
<kernel>
	<contextLocalStorage
		name="com.mycompany.MyType.MY_GLOBAL"
		initMethod="com.mycompany.MyType.myInit()java.lang.Object"
	/>
</kernel>
```

Illustration 4-2: Context Local Storage Declaration of a Static Field with an Initialization Method

4.4 Object References

An object owned by a Feature cannot be assigned to an object owned by another Feature.

An object owned by a Feature can be assigned to an object owned by the Kernel if and only if the current execution context is in Kernel mode.

Note that all possible object assignments are included (field assignment, array assignment and array copies using `System.arraycopy()`).

4.5 Local References

An object owned by a Feature cannot be assigned into a local of an execution context owned by another Feature.

An object owned by a Feature can be assigned into a local of an execution context owned by the Kernel. When leaving Kernel mode explicitly with `Kernel.exit()`, all locals that refer to an object owned by another Feature are set to `null`.

4.6 Monitor Access

A method owned by a Feature cannot synchronize on an object owned by the Kernel.

4.7 Native Method Declaration

A class owned by a Feature cannot declare a `native` method.

4.8 Reflective Operations

4.8.1 `Class.forName`

Table 4-1 defines the semantic rules for `java.lang.Class.forName(String)` in addition to [JLS] specification. If it is not allowed by this specification, a `java.lang.ClassNotFoundException` is thrown as specified by [JLS].

Context Owner	Code Owner	Type Owner	`Class.forName(Type)` allowed
K	K	K	true
K	K	F	false
K	F	K	N/A
K	F	F	N/A
F	K	K	true
F_i	K	F_j	i==j
F	F	K	`true` if the type has been type has been exposed as an API type (§), `false` otherwise.
F_i	F_i	F_j	i==j

Table 4-1: `Class.forName(...)` access rules

4.8.2 `Class.newInstance`

Table 4-2 defines the semantic rules for `java.lang.Class.newInstance(Class)` in addition to [JLS] specification.

Context Owner	Code Owner	Class Owner	New instance ownwer
K	K	K	K
K	K	F	F
K	F	K	N/A
K	F	F	N/A
F	K	K	F
F	K	F	F
F	F	K	F
F	F	F	F

Table 4-2: `Class.newInstance(...)` access rules

4.8.3 `Class.getResourceAsStream`

Table 4-3 defines the semantic rules for `java.lang.Class.getResourceAsStream(String)` in addition to [JLS] specification. If it is not allowed by this specification, `null` is returned as specified by [JLS].

Context owner	Code owner	Resource owner	Class.getResourceAsStream(String) allowed
K	K	K	true
K	K	F	false
K	F	K	N/A
K	F	F	N/A
F	K	K	true
F_i	K	F_j	i==j If the same resource name is declared by both the Kernel and the Feature, the Feature resource takes precedence over the Kernel resource.
F	F	K	false
F_i	F_i	F_j	i==j

Table 4-3: `Class.getResourceAsStream(...)` access rules

4.8.4 `Thread.currentThread`

Threads and their execution contexts have owners. The `Thread.currentThread()` method relates to the thread's owner that is executing the current execution context only. There is no obligation that two execution contexts that are in a caller-callee relationship have the same (==) returned `java.lang.Thread` object when using `Thread.currentThread()` method.

If the Thread that initiated the execution has the same owner as the current execution context or if execution is in Kernel mode, then the thread that initiates the execution is returned, otherwise, a `java.lang.Thread` object owned by the Kernel is returned.

5 FEATURE LIFECYCLE

5.1 Entry point

Each Feature MUST define an implementation of the `ej.kf.FeatureEntryPoint`. `FeatureEntryPoint.start()` method is called when the Feature is started. It is considered to be the main method of the Feature application. `FeatureEntryPoint.stop()` method is called when the Feature is stopped. It gives a chance to the Feature to terminate properly.

5.2 States

A Feature is in one of the following states:

- **INSTALLED**: Feature has been successfully linked to the Kernel and is not running. There are no references from the Kernel to objects owned by this Feature.
- **STARTED**: Feature has been started and is running.

- **STOPPED**: Feature has been stopped and all its owned threads and execution contexts are terminated. The memory and resources are not yet reclaimed. See (§) for the complete stop sequence.
- **UNINSTALLED**: Feature has been unlinked from the Kernel.

Illustration 5-1 describes the Feature state diagram and the methods that changes Feature's state.

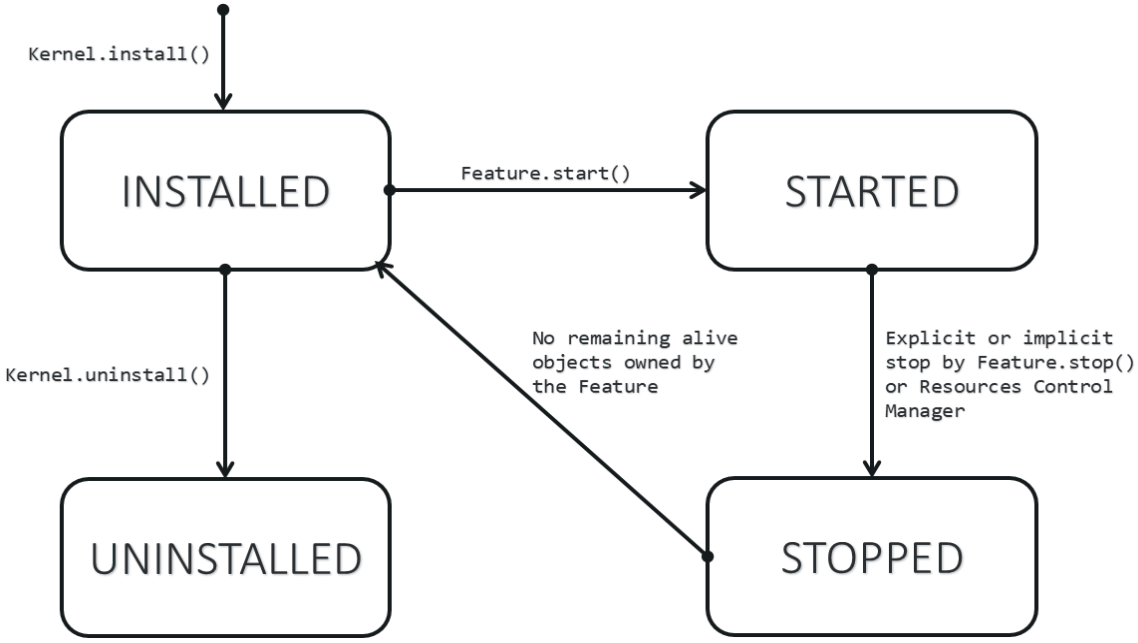

Illustration 5-1: Feature State Diagram

5.3 Installation

A Feature is installed by the Kernel using `Kernel.install(InputStream)`. The content of the Feature data to be loaded is implementation dependent. The Feature data is read and linked to the Kernel. If the Feature cannot be linked to the Kernel, an `ej.kf.IncompatibleFeatureException` is thrown. Otherwise, the Feature is added to the list of loaded Features and its state is set to INSTALLED.

5.4 Start

A Feature is started by the Kernel using `Feature.start()`. The Feature is switched in the STARTED state. A new thread owned by the Feature is created and started. Next steps are executed by the newly created thread:

- Feature clinits are executed
- Entrypoint is instanciated
- `FeatureEntryPoint.start()` is called

5.5 Stop

A Feature is stopped explicitly by the Kernel using `Feature.stop()`. Features may be stopped implicitly by the Resource Control Manager. Next steps are executed:

- On explicit `Feature.stop()` call, a new thread owned by the Feature is created and `FeatureEntryPoint.stop()` is executed within this new thread. Wait until this new thread is done, and timeout of a global timeout stop-time occurred[1].
- The Feature state is set to STOPPED.
- Marks all objects owned by the Feature as dead objects, which implies that a `ej.kf.DeadFeatureException` is thrown in threads that are running the stopped Feature code or in threads that want to call stopped Feature code, or threads that accesses to objects owned by the stopped Feature.
- All execution contexts, from any thread, owned by the Feature are cleared.
- All objects owned by the Feature have their references (to other objects) set to `null`.
- The alive[2] threads owned by the Feature are promoted to `java.lang.Thread` objects owned by the Kernel.
- Native resources (files, sockets, …) opened by the Feature[3] that remain opened after `FeatureEntryPoint.stop()` execution are closed abruptly.
- `FeatureStateListener.stateChanged(...)` is called for each registered listener.
- If there are no remaining alive objects[4]:
  - Feature state is set to INSTALLED,
  - `FeatureStateListener.stateChanged(...)` is called for each registered listener.

The method `Feature.stop()` can be called several times, until the Feature is INSTALLED.

5.6 Deinstallation

A Feature is uninstalled by the Kernel using `Kernel.uninstall()`. The Feature code is unlinked from the Kernel and reclaimed. The Feature is removed from the list of loaded Features and its state is set to UNINSTALLED. The Feature does not exist anymore in the system.

1 A decent global timeout stop-time is 2,000ms.
2 An alive thread is a thread in which at least one execution context is alive.
3 The Kernel MUST track (native) resources that the Kernel granted access for the Feature. See Native resources control section.
4 If there are remaining alive Feature objects after the Kernel listeners are called, the Feature stays in the `STOPPED` state forever (the Kernel has an issue)

6 CLASS SPACES

6.1 Overview

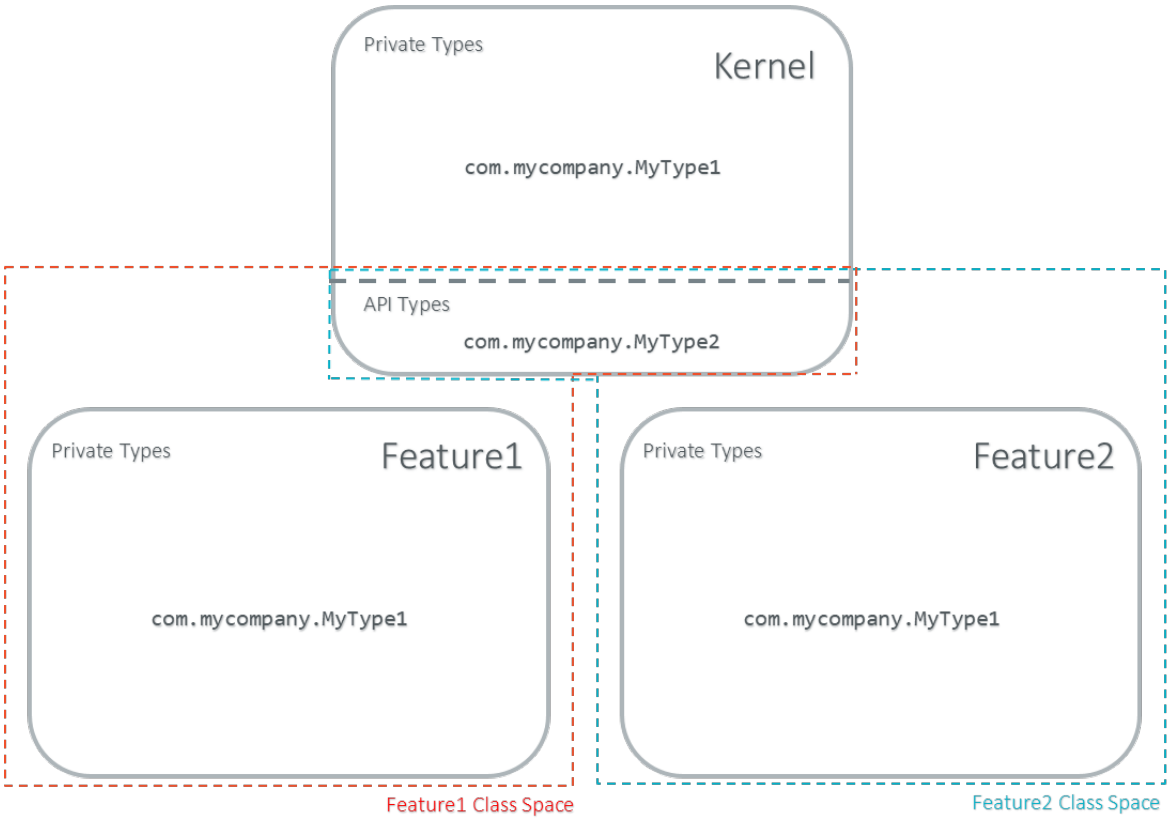

Illustration 6-1: Kernel & Features Class Spaces Overview

6.2 Private Types

The Kernel and the Features define their own private name space. Internal types are only accessible from within the Kernel or Features that define these types. The Kernel or a Feature can have only one type for a specific fully qualified name, insuring there are not two types in the Kernel or in a Feature sharing the same fully qualified name.

6.3 Kernel API Types

The Kernel can expose some of its types, methods and static fields as API to Features. A file describes the list of the types, the methods and the static fields that Features can refer to.

Here is an example for exposing `System.out.println(String)` to a Feature:

```
<require>
      <field name="java.lang.System.out"/>
      <method name="java.io.PrintStream.println(java.lang.String)void"/>
</require>
```

Illustration 6-2: Kernel API Example for exposing `System.out.println`

Section 9.2 describes the Kernel API file format.

6.4 Precedence Rules

APIs exposed by the Kernel are publicly available for all Features: they form the global name space.

A Kernel API type (from the global name space) always takes precedence over a Feature type with the same fully qualified name when a Feature is loaded[5].

7 RESOURCE CONTROL MANAGER

7.1 CPU Control: Quotas

A Kernel can assign an execution quota to a Feature using `Feature.setExecutionQuota()`. The quota is expressed in execution units.

Quotas account to the running current context owner.

When a Feature has reached its execution quota, its execution is suspended until all other Features have reached their execution quota. When there are no threads owned by Features eligible to be scheduled, the execution counter of all Features is reset.

Setting a Feature execution quota to zero causes the Feature to be suspended (the Feature is paused).

7.2 RAM Control: Feature Criticality

Each Feature has a criticality level between `Feature.MIN_CRITICALITY` and `Feature.MAX_CRITICALITY`. When an execution context cannot allocate new objects because a memory limit has been reached, Features shall be stopped following next semantic:

- Select the Feature with the lowest criticality.
- If the selected Feature has a criticality lower than the current execution context owner criticality, then stop the selected Feature and all the Features with the same criticality.
- If no memory is available, repeat these two previous steps in sequence until there are no more Features to stop.

If no memory is reclaimed, then an `OutOfMemoryException` is thrown.

7.3 Time-out Control: Watchdog

All method calls that are done from a Kernel mode to a Feature mode are automatically executed under the control of a watchdog.

5 An exposed type from the Kernel cannot be overloaded by a Feature.

The watchdog timeout is set according to the following rules:

- use the watchdog timeout of the current execution context if it has been set,
- else use the watchdog timeout of the current thread if it has been set,
- else use the global system watchdog timeout.

The global system watchdog timeout value is set to `Long.MAX_VALUE` at system startup.

When the watchdog timeout occurs the offending Feature is stopped.

7.4 Native Resource Control: Security Manager

The Kernel is responsible for holding all the native calls. The Kernel shall provide methods (API) that systematically check, using the standard security manager, that the access to a native call is granted to the specific Feature.

When an object owned by a Feature is not allowed to access a native resource, a specific exception shall be thrown.

Any native resource opened by a Feature must be registered by the Kernel and closed when the Feature is stopped.

8 COMMUNICATION BETWEEN FEATURES

8.1 Introduction

A Feature can communicate with another Feature, through a remote method invocation mechanism based on pure Java interfaces.

A Feature can call a method owned by another Feature, provided:

- Both Features own an interface in their class space with the same fully qualified name
- Both Features have declared such interface as a shared interface
- The source Feature has declared a Proxy class for its shared interface
- The target Feature has registered to the Kernel an instance of a class implementing its shared interface
- The source Feature has requested from the Kernel an instance of a class implementing its interface
- The Kernel has bound the source interface to the target instance and returned an instance to the source Feature, implementing its shared interface
- The source Feature calls a method declared in the shared interface using this instance as receiver
- A method with the exact descriptor exists in the target Feature interface
- The arguments given by the source Feature can be transferred to the target Feature
- The value returned by the target Feature can be transferred to the source Feature (if the method does not return `void`)

8.2 Shared Interface Declaration

To declare an interface as a shared interface, it must be registered in a shared interfaces file, as following:

```
<sharedInterfaces>
      <sharedInterface name="mypackage.MyInterface"/>
</sharedInterfaces>
```

Illustration 8-1: Shared Interface Declaration Example

Section 9.4 describes the Shared Interface file format specification.

An interface declared as Shared Interface can extends Feature interfaces (which are not declared as Shared Interfaces) or Kernel interfaces.

A Shared Interface is composed of all methods declared by itself and its super types. Each method must comply with the following:

- types declared for parameters and optional return value must be transferable types (see section 8.5)
- exceptions thrown must be owned by the Kernel

8.3 Proxy Class

In addition to the Shared Interface declaration, a Proxy class must be implemented, with the following specification:

- its fully qualified name is the shared interface fully qualified name append with `Proxy`.
- it extends `ej.kf.Proxy`
- it implements the Shared Interface
- it provides an implementation of all interface methods

As the Proxy is implemented by the Feature that will use the Shared Interface, it is free to implement the desired behavior and ensure its own robustness. Although it is not part of this specification, it is strongly encouraged that Proxy methods implementation comply with the expected behavior, even when the remote Feature returns an unexpected behavior (such as `ej.kf.DeadFeatureException` if the remote Feature is killed).

Usually, the following template is applied:

```
try {
    return invokeXXX();
} catch (Throwable e) {
    // Implement a behavior that complies with the method specification.
    // i.e. return a valid error code or throw a documented exception.
    // Logging traces for debug can also be added here.
}
```

Illustration 8-2: Proxy Method Implementation Template

The `ej.kf.Proxy.invokeXXX()` method invokes the target method corresponding to the enclosing proxy method. There is one `invokeXXX` method for each returned type (`invokeBoolean`, `invokeByte`, `invokeChar`, `invokeShort`, `invokeInt`, `invokeLong`, `invokeFloat`, `invokeDouble`, `invokeRef`) and each Proxy method should use the right one that matches its return type.

8.4 Object Binding

The Kernel can bind an object owned by a Feature to an object owned by another Feature using the method `ej.kf.Kernel.bind()`.

- When the target type is owned by the Kernel, the object is converted using the most accurate Kernel type converter.
- When the target type is owned by the Feature, it must be a shared interface. In this case, a Proxy instance is returned. Object identity is preserved across Features: calling `ej.kf.Kernel.bind()` multiple times with the same parameters returns the same object.

8.5 Arguments Transfer

A base type argument is directly passed without conversion (by copy).

A reference argument is subject to conversion rules, according to Table 8-1.

Type	Owner	Instance Owner	Transfer Rule
Any Class, Array or Interface	Kernel	Kernel	Direct reference is passed to the target Feature.
Any Class, Array or Interface	Kernel	Feature	Converted to the target Feature if Kernel has registered a Kernel type converter, otherwise Forbidden. See section 8.6.
Array of base types	Any	Feature	A new array is allocated in the target Feature and elements are copied into.
Array of references	Any	Feature	A new array is allocated in the target Feature and for each element is applied these conversion rules. (recursively).
Shared Interface	Feature	Feature	A Proxy to the original object is created and passed to the receiving Feature. • If argument is already a Proxy and the target owner is the same than the target Shared Interface owner, the original object is passed. (unwrapping) • Otherwise a new Proxy wrapping on the original object is passed.
Any Class, Array or Interface	Feature	Feature	Forbidden.

Table 8-1 Shared Interface Argument Conversion Rules

8.6 Kernel Type Converters

By default, Feature instances of types owned by the Kernel cannot be be passed across a Shared Interface method invocation.

The Kernel can register a converter for each allowed type, using `Kernel.addConverter()`. The converter must implement `ej.kf.Converter and` can implement one of the following behaviors:

- by wrapper: manually allocating a Proxy reference by calling `Kernel.newProxy()`
- by copy: with the help of `Kernel.clone()`

9 CONFIGURATION FILES

9.1 Kernel and Features Declaration

A Kernel must provide a declaration file named `kernel.kf`. A Feature must provide a declaration file named `[name].kf`.

KF Declaration file is a Properties file. It must appear at the root of any application classpath (directory or JAR file). Keys are described hereafter:

Key	Usage	Description
entryPoint	Mandatory for Feature only.	The fully qualified name of the class that implements `ej.kf.FeatureEntryPoint`
name	Optional	`KERNEL` by default for the Kernel, or the name of the file without the `.kf` extension for Features.
version	Mandatory	String version, that can retrieved using `ej.kf.Module.getVersion()`

Illustration 9-1: KF Definition File Properties Specification

9.2 Kernel API Definition

By default, when building a Kernel, no types are exposed as API for Features, except `ej.kf.FeatureEntryPoint`. Kernel types, methods and static fields allowed to be accessed by Features must be declared in one or more `kernel.api` files. They must appear at the root of any application classpath (directory or JAR file). Kernel API file is an XML file, with the following schema:

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
    <xs:element name='require'>
        <xs:complexType>
            <xs:choice minOccurs='0' maxOccurs='unbounded'>
                <xs:element ref='type'/>
                <xs:element ref='field'/>
                <xs:element ref='method'/>
            </xs:choice>
        </xs:complexType>
    </xs:element>

    <xs:element name='type'>
        <xs:complexType>
            <xs:attribute name='name' type='xs:string' use='required'/>
        </xs:complexType>
    </xs:element>

    <xs:element name='field'>
        <xs:complexType>
            <xs:attribute name='name' type='xs:string' use='required'/>
        </xs:complexType>
    </xs:element>

    <xs:element name='method'>
        <xs:complexType>
            <xs:attribute name='name' type='xs:string' use='required'/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Illustration 9-2: Kernel API XML Schema

Tag	Attributes	Description
require		The root element
field		Static field declaration. Declaring a field as a Kernel API automatically declares its type as a Kernel API.
name	Fully qualified name on the form `[type].[fieldName]`	
method		Method or constructor declaration. Declaring a method or a constructor as a Kernel API automatically declares its type as a Kernel API
name	Fully qualified name on the form `[type].[methodName]([typeArg1,...,typeArgN)typeReturned`. Types are fully qualified names or one of a base type as described by the Java language (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`) When declaring a constructor, `methodName` is the single type name. When declaring a void method or a constructor, `typeReturned` is `void`	
type		Type declaration. Declaring a type as Kernel API automatically declares all its super types (classes and interfaces) and the default constructor (if any) as Kernel API.
name	Fully qualified name on the form `[package].[package].[typeName]`	

Illustration 9-3: Kernel API Tags Specification

9.3 Identification

Kernel and Features require an X509[6] certificate for identification. The 6 first fields defined by RFC 2253[7] can be read by calling `ej.kf.Module.getProvider().getValue(...)`.

The certificate file must be configured as following:

- placed beside the related `[name].kf` file
- named `[name].cert`
- `DER`-encoded and may be supplied in binary or printable (Base64) encoding. If the certificate is provided in Base64 encoding, it must be bounded at the beginning by `-----BEGIN CERTIFICATE-----`, and must be bounded at the end by `-----END CERTIFICATE-----`.

6 https://tools.ietf.org/html/rfc5280

7:`CN (commonName), L (localityName), ST (stateOrProvinceName), O (organizationName), OU (organizationalUnitName), C (countryName).`

9.4 Shared Interface Declaration

A Shared Interface file is an XML file ending with the `.si` suffix with the following format:

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

 <xs:element name='sharedInterfaces'>
  <xs:complexType>
   <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element ref='sharedInterface'/>
   </xs:choice>
  </xs:complexType>
 </xs:element>

 <xs:element name='sharedInterface'>
  <xs:complexType>
   <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
 </xs:element>

</xs:schema>
```

Illustration 9-4: Shared Interface XML Schema Specification

Shared interface files must appear at the root of any application classpath (directory or JAR file).

9.5 Kernel Advanced Configuration

`The kernel.intern` files is for Kernel advanced configurations such as declaring context local storage static fields (§4.3.3). It must appear at the root of any application classpath (directory or JAR file).

```
 <!--
      Root Element
  -->
 <xs:element name='kernel'>
  <xs:complexType>
   <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element ref='contextLocalStorage'/>
    <xs:element ref='property'/>
   </xs:choice>
  </xs:complexType>
 </xs:element>
```

Illustration 9-5: Kernel Intern Root XML Schema Specification

9.6 Context Local Storage Static Field Configuration

9.6.1 XML Schema & Format

```
<xs:element name='contextLocalStorage'>
 <xs:complexType>
  <!--
     Static Field Simple Name.
  -->
  <xs:attribute name='name' type='xs:string' use='required'/>
  <!--
     Optional Initialization Method descriptor, as specified by Kernel API
method descriptor.
   -->
  <xs:attribute name='initMethod' type='xs:string' use='optional'/>
 </xs:complexType>
</xs:element>
```

Table 9-1: Context Local Storage XML Schema Specification

9.6.2 Typical Example

The following illustration describes the definition of a context local storage static field (`I`), which is duplicated in each context (Kernel and Features):

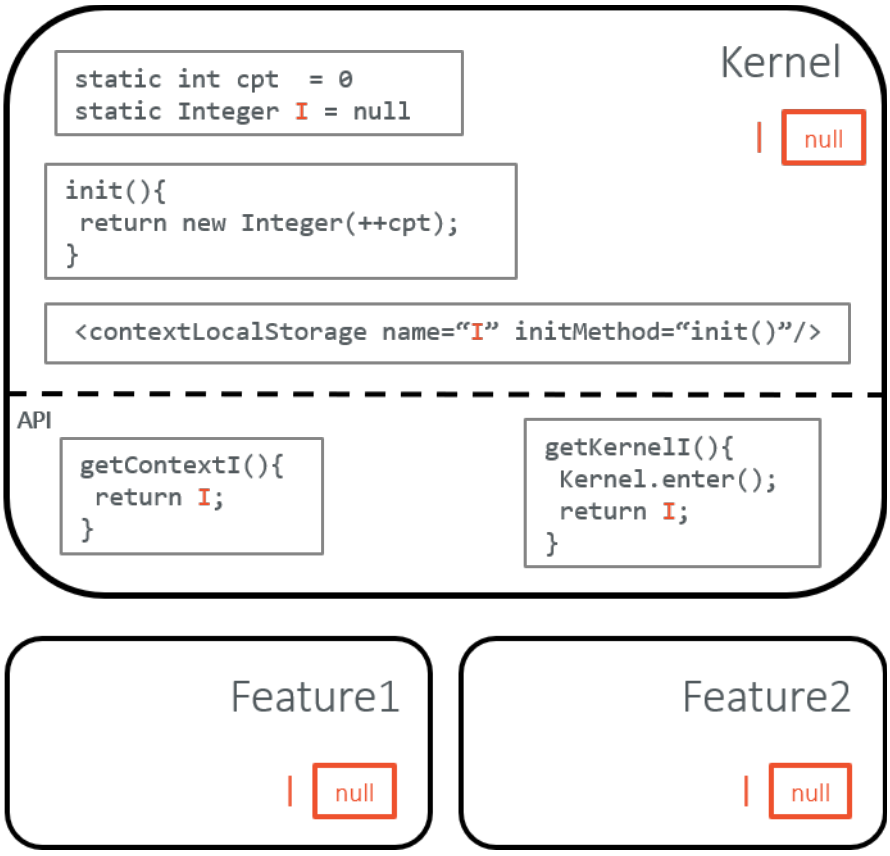

Illustration 9-6: Context Local Storage of Static Field Example

The following illustration describes a detailed sequence of method calls with the expected behavior.

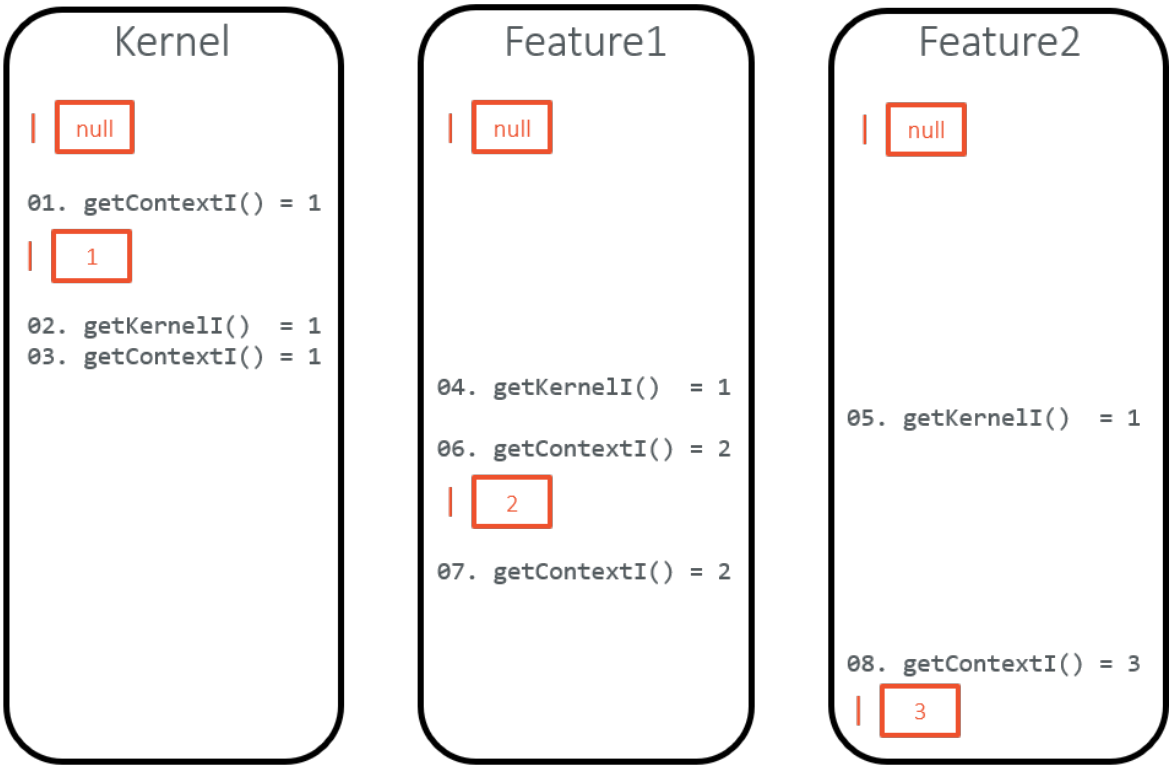

Illustration 9-7: Context Local Storage Example of Initialization Sequence

10 JAVA SPECIFICATION

Package ej.kf

Contains KF classes.

See:
Description

Interface Summary		Page
Converter<T>	A `Converter` is able to give a representation of an object owned by a Feature to an other Feature.	25
FeatureEntryPoint	Each Feature shall define one entry point that implements this interface.	32
FeatureStateListener	The listener interface for receiving notifications when the state of a Feature has changed.	33
Principal	This interface represents and identifies the Kernel or a Feature.	49
UncaughtExceptionHandler	**Deprecated.** *Use*	56

Class Summary		Page
Feature	A Feature represents an optional part of an application that adds new features and services.	27
Kernel	The Kernel represents the atomic part of an application.	36
Module	A `Module` is either Kernel or a Feature.	46
Proxy<T>	The superclass of proxy classes.	52

Enum Summary		Page
Feature.State	A Feature state.	30

Exception Summary		Page
AlreadyLoadedFeatureException	This exception is thrown if a Feature being loaded has already been loaded.	24
DeadFeatureException	This exception is thrown by the system when a Feature code has been stopped because it is being uninstalled.	26
IncompatibleFeatureException	This exception is thrown if a Feature being loaded is not compatible with the current Kernel.	34
InvalidFormatException	This exception is thrown when a Feature data being loaded has an invalid format.	35
UnknownFeatureException	This exception is thrown if a Feature being unloaded is unknown.	57

Package ej.kf Description

Contains KF classes. (ESR020).

Class AlreadyLoadedFeatureException

ej.kf

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └ej.kf.AlreadyLoadedFeatureException
```

All Implemented Interfaces:
Serializable

```
public class AlreadyLoadedFeatureException
extends Exception
```

This exception is thrown if a Feature being loaded has already been loaded.

See Also:
`Kernel.load(java.io.InputStream)`

Constructor Summary	Page
AlreadyLoadedFeatureException(Feature feature) Creates an AlreadyLoadedFeatureException with the previously loaded Feature.	24

Method Summary		Page
Feature	**getFeature**() Returns the previously loaded Feature.	24

Constructor Detail

AlreadyLoadedFeatureException

```
public AlreadyLoadedFeatureException(Feature feature)
```

Creates an AlreadyLoadedFeatureException with the previously loaded Feature.

Parameters:
feature - the previously loaded Feature

Method Detail

getFeature

```
public Feature getFeature()
```

Returns the previously loaded Feature.

Returns:
the previously loaded Feature.

Interface Converter<T>

ej.kf

Type Parameters:

`T` - the Kernel type managed by this `Converter`.

```
public interface Converter<T>
```

A `Converter` is able to give a representation of an object owned by a Feature to an other Feature. A `Converter` is able to convert instances of one and only one Kernel type.

A `Converter` is free to decide the kind of conversion to apply, depending on the managed type. For example, a `Converter` for immutables instances of types such as `String` will most likely return a copy (clone), wheras a `Converter` for instances of types such as `InputStream` will most likely return a wrapper on the original object.

See Also:

`Kernel.bind(Object, Class, Feature)`, `Kernel.clone(Object, Module)`

Method Summary		Page
`T`	**convert**`(T source, Feature targetOwner)` Converts an object owned by a Feature to an other Feature.	25
`Class<T>`	**getType**`()` Gets the Kernel type managed by this `Converter`.	25

Method Detail

convert

```
T convert(T source,
          Feature targetOwner)
```

Converts an object owned by a Feature to an other Feature.

Parameters:

`source` - the source object to be converted
`targetOwner` - the owner of the converted object

Returns:

the converted object, owned by the target owner

---

getType

```
Class<T> getType()
```

Gets the Kernel type managed by this `Converter`.

Returns:

the Kernel type managed by this `Converter`.

Class DeadFeatureException

ej.kf

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └ej.kf.DeadFeatureException
```

All Implemented Interfaces:

Serializable

```
public class DeadFeatureException
extends RuntimeException
```

This exception is thrown by the system when a Feature code has been stopped because it is being uninstalled. Only kernel code can receive this exception: it never occurs in Feature code. Each call from kernel to a Feature shall catch this exception and handle the fact the Feature is not able to execute the desired "service".

Constructor Summary	Page
DeadFeatureException()	26

Constructor Detail

DeadFeatureException

```
public DeadFeatureException()
```

Class Feature

ej.kf

```
java.lang.Object
  └ ej.kf.Module
      └ ej.kf.Feature
```

```
public class Feature
extends Module
```

A Feature represents an optional part of an application that adds new features and services.

A Feature only depends on Kernel code. A Feature is considered as unreliable code from Kernel point of view.

Instances of this class are owned by the Kernel.

Nested Class Summary		Page
static enum	**Feature.State** A Feature state.	30

Method Summary		Page
Thread[]	**getAllAliveThreads**() Gets a snapshot of all alive threads owned by this Feature (some threads included in the returned array may have been terminated when this method returns, some new threads may have been created when this method returns).	27
Feature.State	**getState**() Returns the current Feature state.	28
void	**start**() Causes this Feature to start.	28
void	**stop**() Causes this Feature to stop.	28

Methods inherited from class ej.kf.Module
getExecutionCounter, getExecutionQuota, getName, getProvider, getUID, getVersion, setExecutionQuota

Method Detail

getAllAliveThreads

```
public Thread[] getAllAliveThreads()
```

Gets a snapshot of all alive threads owned by this Feature (some threads included in the returned array may have been terminated when this method returns, some new threads may have been created when this method returns).

Returns:

the threads owned by this Feature

Throws:
`IllegalStateException` - if the Feature is not in the `Feature.State.STARTED` state

---

getState

```
public Feature.State getState()
```

Returns the current Feature state.

Returns:
this Feature's state.
See Also:
`Feature.State`

---

start

```
public void start()
```

Causes this Feature to start. A new thread owned by the Feature is created and started. The Feature is switched in the `Feature.State.STARTED` state and this method returns. Next steps are executed asynchronously within the new thread context:

- Feature clinits are executed
- Entry point is instantiated
- `FeatureEntryPoint.start()` is called

Throws:
`IllegalStateException` - if the Feature state is not in `Feature.State.INSTALLED` state

---

stop

```
public void stop()
```

Causes this Feature to stop. The following steps are executed:

- A new thread owned by the Feature is created and `FeatureEntryPoint.stop()` is executed.
- Wait until this thread is normally terminated or timeout occurred.
- `ej.lang.Resource.reclaim()` is called for each resource that remains open by the Feature.
- A `DeadFeatureException` is thrown in threads that are running Feature code or in threads that want to call Feature code.
- Wait until all threads owned by this Feature are terminated.
- Feature state is set to `Feature.State.STOPPED`.
- Objects owned by the Feature are reclaimed. If there are no remaining alive objects, the Feature state is set to `Feature.State.INSTALLED`.

When the new Feature state is `Feature.State.INSTALLED`, the Feature runtime has been fully reclaimed (threads and objects). Otherwise, the new Feature state is `Feature.State.STOPPED` and there are some remaining Feature objects references from Kernel.

This method can be called multiple times by the Kernel to reclaim objects again and thus to try to switch the Feature in the `Feature.State.INSTALLED` state.

When Feature state is set to `Feature.State.STOPPED`, Kernel application shall remove all its references to objects owned by this Feature, through the calls of `FeatureStateListener.stateChanged(Feature, State)`.

Throws:

`IllegalStateException` - if the Feature is in the `Feature.State.INSTALLED` or `Feature.State.UNINSTALLED` state

Enum Feature.State

ej.kf

```
java.lang.Object
  └java.lang.Enum<Feature.State>
      └ej.kf.Feature.State
```

All Implemented Interfaces:

Comparable<Feature.State>, Serializable

Enclosing class:

Feature

```
public static enum Feature.State
extends Enum<Feature.State>
```

A Feature state.

See Also:

Feature.getState()

Enum Constant Summary	Page
INSTALLED A Feature in the INSTALLED state has been successfully linked to the Kernel and is not running.	30
STARTED A Feature in the STARTED state has been started and is running.	31
STOPPED A Feature in the STOPPED state has been stopped and all its threads are terminated.	31
UNINSTALLED A Feature in the UNINSTALLED state has been unlinked from the Kernel.	31

Method Summary		Page
static Feature.State	**valueOf**(String name)	31
static Feature.State[]	**values**()	31

Enum Constant Detail

INSTALLED

```
public static final Feature.State INSTALLED
```

A Feature in the INSTALLED state has been successfully linked to the Kernel and is not running. There are no references from the Kernel to objects owned by this Feature.

STARTED

```
public static final Feature.State STARTED
```

A Feature in the `STARTED` state has been started and is running.

STOPPED

```
public static final Feature.State STOPPED
```

A Feature in the `STOPPED` state has been stopped and all its threads are terminated. There are remaining references from the Kernel to objects owned by this Feature.

UNINSTALLED

```
public static final Feature.State UNINSTALLED
```

A Feature in the `UNINSTALLED` state has been unlinked from the Kernel. All Feature methods except `Feature.getState()` throw an `IllegalStateException`

Method Detail

values

```
public static Feature.State[] values()
```

valueOf

```
public static Feature.State valueOf(String name)
```

Interface FeatureEntryPoint

ej.kf

```
public interface FeatureEntryPoint
```

Each Feature shall define one entry point that implements this interface. Methods are called by the Kernel.

Method Summary		Page
void	**start**() This method is called once by the Kernel when a Feature has been newly started.	32
void	**stop**() This method is called once by the Kernel when a Feature is going to be unloaded.	32

Method Detail

start

```
void start()
```

This method is called once by the Kernel when a Feature has been newly started. It is executed in a dedicated thread owned by the Feature, so it may consider it as its "main" thread. This allows a Feature to connect to the application (by adding new Feature points, services, ...)

---

stop

```
void stop()
```

This method is called once by the Kernel when a Feature is going to be unloaded. It is executed in a dedicated thread owned by the Feature. Feature is responsible to do its best effort to properly stop threads and close resources it has created as soon as possible.

Interface FeatureStateListener

ej.kf

```
public interface FeatureStateListener
```

The listener interface for receiving notifications when the state of a Feature has changed. Object instances of classes that implement this interface must be added to the Kernel listener list using `Kernel.addFeatureStateListener(FeatureStateListener)`

Method Summary		Page
void	**stateChanged**(Feature feature, Feature.State previousState) Called when the state of a Feature has changed.	33

Method Detail

stateChanged

```
void stateChanged(Feature feature,
                  Feature.State previousState)
```

Called when the state of a Feature has changed.

Parameters:

`feature` - the Feature which state has changed

`previousState` - the previous state, `null` if Feature state is `Feature.State.INSTALLED`

Class IncompatibleFeatureException

ej.kf

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └ej.kf.IncompatibleFeatureException
```

All Implemented Interfaces:
Serializable

```
public class IncompatibleFeatureException
extends Exception
```

This exception is thrown if a Feature being loaded is not compatible with the current Kernel.

See Also:
Kernel.load(java.io.InputStream)

Constructor Summary	Page
IncompatibleFeatureException()	34

Method Summary		Page
String	getExpectedKernelVersion() Get the expected version of the Kernel on which this Feature can be installed	34

Constructor Detail

IncompatibleFeatureException

```
public IncompatibleFeatureException()
```

Method Detail

getExpectedKernelVersion

```
public String getExpectedKernelVersion()
```

Get the expected version of the Kernel on which this Feature can be installed

Returns:
the expected Kernel version
See Also:
Module.getVersion()

Class InvalidFormatException

ej.kf

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └ej.kf.InvalidFormatException
```

All Implemented Interfaces:
Serializable

```
public class InvalidFormatException
extends Exception
```

This exception is thrown when a Feature data being loaded has an invalid format.

See Also:
Kernel.load(java.io.InputStream)

Constructor Summary	Page
InvalidFormatException()	35

Constructor Detail

InvalidFormatException

```
public InvalidFormatException()
```

Class Kernel

ej.kf

```
java.lang.Object
  └ej.kf.Module
      └ej.kf.Kernel
```

```
public class Kernel
extends Module
```

The Kernel represents the atomic part of an application. Kernel code is assumed to be reliable. The `Kernel` class provides core methods to manage Features. It is intended to be used only by the Kernel code, and not viewed from the Feature.

Method Summary		Page
static void	**addConverter**(Converter<?> converter) Adds the `Converter` to the list of converters.	42
static void	**addFeatureStateListener**(FeatureStateListener listener) Adds the `FeatureStateListener` to the list of listeners that are notified when the state of a Feature has changed.	41
static boolean	**areEquivalentSharedInterfaces**(Class<?> si1, Class<?> si2) Tells whether the given classes are equivalent shared interfaces.	45
static <T> T	**bind**(T o, Class<T> targetType, Feature targetOwner) Binds an `Object` owned by a Feature to an other Feature.	43
static <T> T	**clone**(T from, Module toOwner) Creates and returns a copy of the given object, so that the newly created object is owned by the given `Module`.	42
static void	**enter**() Enters in Kernel mode: the current thread context is switched to be owned by the Kernel.	39
static void	**exit**() Exits from Kernel mode: the current thread context is restored to the owner of the caller of the method (which can remain the Kernel).	40
static FeatureStateListener[]	**getAllFeatureStateListeners**() Returns an array containing all the `FeatureStateListener` that are notified when the state of a Feature has changed.	41
static Feature[]	**getAllLoadedFeatures**() Returns the set of Features currently loaded.	39
static Module	**getContextOwner**() Returns the owner of the current thread context.	40
static Class<?>	**getEquivalentSharedInterface**(Class<?> si, Feature target) Gets the equivalent shared interface in the given target Feature.	44

static Class<?>	**getImplementedSharedInterface**(Class<?> fromClass, Class<?> topInterface) Gets the first shared interface implemented by the given class under the hierarchy of topInterface.	45
static Kernel	**getInstance**() Returns the singleton instance representing the Kernel.	37
String	**getName**() Gets the name of this module.	41
static Module	**getOwner**(Object o) Returns the owner of the given Object.	40
static Class<?>	**getSharedInterface**(Class<?> si, Class<?> topInterface, Feature target) From a shared interface, gets the closest shared interface in the given target Feature.	44
static Feature	**install**(InputStream is) Installs a Feature from an InputStream.	38
static boolean	**isAPI**(Class<?> c) Tells whether the given class is a Kernel API.	44
static boolean	**isInKernelMode**() Tells whether the current thread context is currently in Kernel mode.	40
static boolean	**isSharedInterface**(Class<?> c) Tells whether the given class is a shared interface (i.e.	43
static Feature	**load**(InputStream is) Installs a Feature from an InputStream and starts it.	38
static <T> Proxy<T>	**newProxy**(T ref, Module owner) Allocates a new Proxy and sets its reference to the given object.	44
static void	**removeConverter**(Converter<?> converter) Removes the Converter to the list of converters.	43
static void	**removeFeatureStateListener**(FeatureStateListener listener) Removes the FeatureStateListener to the list of listeners that are notified when the state of a Feature has changed.	41
static void	**runUnderContext**(Module contextOwner, Runnable runnable) Calls the Runnable.run() method with current context set to the given Module.	42
static void	**setUncaughtExceptionHandler**(UncaughtExceptionHandler handler) **Deprecated.** *Use Thread.setUncaughtExceptionHandler(java.lang.Thread.UncaughtExceptionHandler)*	40
static void	**uninstall**(Feature f) Uninstalls a Feature.	38
static boolean	**unload**(Feature f) Stops a Feature and uninstalls it if its state is Feature.State.INSTALLED after Feature.stop().	39

Methods inherited from class ej.kf.Module
getExecutionCounter, getExecutionQuota, getProvider, getUID, getVersion, setExecutionQuota

Method Detail

getInstance

public static Kernel **getInstance**()

Returns the singleton instance representing the Kernel.

Returns:
the singleton instance representing the Kernel

---

load

```
public static Feature load(InputStream is)
                    throws IOException,
                           InvalidFormatException,
                           IncompatibleFeatureException,
                           AlreadyLoadedFeatureException
```

Installs a Feature from an `InputStream` and starts it.

Parameters:
`is` - the input stream from where the Feature data is loaded.
Returns:
the loaded Feature, in the `Feature.State.STARTED` state.
Throws:
`IOException` - if something occurs when reading `InputStream`
`InvalidFormatException` - if Feature content is invalid
`IncompatibleFeatureException` - if Feature is not compatible with the current Kernel
`AlreadyLoadedFeatureException` - if Feature is already loaded
See Also:
`install(InputStream)`, `Feature.start()`

---

install

```
public static Feature install(InputStream is)
                     throws IOException,
                            InvalidFormatException,
                            IncompatibleFeatureException,
                            AlreadyLoadedFeatureException
```

Installs a Feature from an `InputStream`. Feature shall have been generated against the current Kernel classes. Feature data is read and linked to the Kernel. The Feature is added to the list of loaded features and its state is set to `Feature.State.INSTALLED`. The given input stream is let open.

Parameters:
`is` - the input stream from where the Feature data is loaded.
Returns:
the loaded Feature, in the `Feature.State.INSTALLED` state.
Throws:
`IOException` - if something occurs when reading `InputStream`
`InvalidFormatException` - if Feature content is invalid
`IncompatibleFeatureException` - if Feature is not compatible with the current Kernel
`AlreadyLoadedFeatureException` - if Feature is already loaded
See Also:
`getAllLoadedFeatures()`

---

uninstall

```
public static void uninstall(Feature f)
```

Uninstalls a Feature. When this method returns, the Feature code has been unlinked from the Kernel and reclaimed. The Feature is removed from the list of loaded features and its state is set to `Feature.State.UNINSTALLED`.

Parameters:
`f` - the feature to be uninstalled.
Throws:
`IllegalStateException` - if Feature state is not `Feature.State.INSTALLED`
See Also:
`getAllLoadedFeatures()`

---

unload

```
public static boolean unload(Feature f)
                     throws UnknownFeatureException
```

Stops a Feature and uninstalls it if its state is `Feature.State.INSTALLED` after `Feature.stop()`.

Parameters:
`f` - the feature to be unloaded.
Returns:
`true` if Feature state is `Feature.State.UNINSTALLED`, `false` otherwise.
Throws:
`UnknownFeatureException` - if the given Feature is unknown.
`IllegalStateException` - if Feature state is `Feature.State.UNINSTALLED`
See Also:
`uninstall(Feature)`, `Feature.stop()`

---

getAllLoadedFeatures

```
public static Feature[] getAllLoadedFeatures()
```

Returns the set of Features currently loaded.

Returns:
all Features that are not in the state `Feature.State.UNINSTALLED`.

---

enter

```
public static void enter()
```

Enters in Kernel mode: the current thread context is switched to be owned by the Kernel. If the current context was already in Kernel mode, this method does nothing.

The context owner is automatically restored when returning from the method (equivalent to calling `exit()` before returning).

See Also:
`exit()`

---

exit

```
public static void exit()
```

Exits from Kernel mode: the current thread context is restored to the owner of the caller of the method (which can remain the Kernel). If the restored context is owned by a Feature, all locals that refer to an object owned by an other Feature are reset to `null`.

See Also:
`enter()`

---

isInKernelMode

```
public static boolean isInKernelMode()
```

Tells whether the current thread context is currently in Kernel mode.

Returns:
the result of `Kernel.getContextOwner() == Kernel.getInstance()`

---

getOwner

```
public static Module getOwner(Object o)
```

Returns the owner of the given `Object`.

Parameters:
`o` - the object.
Returns:
the owner of the object.

---

getContextOwner

```
public static Module getContextOwner()
```

Returns the owner of the current thread context.

Returns:
the context owner.

---

setUncaughtExceptionHandler

```
@Deprecated
public static void setUncaughtExceptionHandler(UncaughtExceptionHandler handler)
```

Deprecated. *Use `Thread.setUncaughtExceptionHandler(java.lang.Thread.UncaughtExceptionHandler)`*

Sets the handler invoked when a Feature thread abruptly terminates due to an uncaught exception.

Parameters:
`handler` - the handler to register, or `null` if no explicit handler.

---

getName

```
public String getName()
```

Gets the name of this module.

Overrides:
`getName` in class `Module`
Returns:
the name of this module, or `"KERNEL"` String if not set

---

addFeatureStateListener

```
public static void addFeatureStateListener(FeatureStateListener listener)
```

Adds the `FeatureStateListener` to the list of listeners that are notified when the state of a Feature has changed.

Parameters:
`listener` - the new listener to add
Throws:
`NullPointerException` - if listener is `null`

---

removeFeatureStateListener

```
public static void removeFeatureStateListener(FeatureStateListener listener)
```

Removes the `FeatureStateListener` to the list of listeners that are notified when the state of a Feature has changed.

Does nothing if the listener is not registered or `null`.

Parameters:
`listener` - the listener to be removed

---

getAllFeatureStateListeners

```
public static FeatureStateListener[] getAllFeatureStateListeners()
```

Returns an array containing all the `FeatureStateListener` that are notified when the state of a Feature has changed.

Returns:
an array of `FeatureStateListener[]` with all the listeners

---

runUnderContext

```
public static void runUnderContext(Module contextOwner,
                                   Runnable runnable)
```

Calls the `Runnable.run()` method with current context set to the given `Module`.

Parameters:
`contextOwner` - the context owner that will execute the method
`runnable` - the `Runnable` instance to run under the given `Feature` context.
Throws:
`IllegalAccessError` - if the `Runnable` instance is not accessible to the context owner, or if the `Runnable` is owned by a Feature and must run in Kernel context.

clone

```
public static <T> T clone(T from,
                          Module toOwner)
                   throws CloneNotSupportedException
```

Creates and returns a copy of the given object, so that the newly created object is owned by the given `Module`. The source object class must be `String` or must implement `Cloneable`. Otherwise, a `CloneNotSupportedException` is thrown. If the source object owner and the target owner are the same, this method is equivalent to `Object.clone()` method applied on the source object. Otherwise, the object can be cloned if the source object class is owned by the Kernel and all its object references are accessible to the new owner. In all other cases, an `IllegalAccessError` is thrown.

Type Parameters:
`T` - the Kernel type of the object to clone
Parameters:
`from` - the object to clone
`toOwner` - the owner of the cloned object
Returns:
the cloned object
Throws:
`CloneNotSupportedException` - if the source object cannot be cloned
`IllegalAccessError` - if the creation of the new object would break access rules
`NullPointerException` - if one of the arguments is `null`

addConverter

```
public static void addConverter(Converter<?> converter)
```

Adds the `Converter` to the list of converters. Registered converters are used by `bind(Object, Class, Feature)`.

Parameters:
`converter` - the new converter to add
Throws:
`NullPointerException` - if converter is `null`
`IllegalArgumentException` - if a converter managing the same type is already registered
See Also:
`Converter.getType()`

removeConverter

```
public static void removeConverter(Converter<?> converter)
```

Removes the `Converter` to the list of converters.

Does nothing if the converter is not registered or `null`.

Parameters:
`converter` - the converter to be removed

bind

```
public static <T> T bind(T o,
                         Class<T> targetType,
                         Feature targetOwner)
```

Binds an `Object` owned by a Feature to an other Feature.

When the target type is owned by the Kernel, the object is converted using the most accurate registered converter.

When the target type is owned by the Feature, it must be a shared interface. In this case, a `Proxy` instance is returned. `Object` identity is preserved across Features: calling multiple times this method with the same parameters returns the same object.

Type Parameters:
`T` - the Kernel type of the object to bind
Parameters:
`o` - the object to be converted
`targetType` - the type of the converted object
`targetOwner` - the owner of the converted object
Returns:
an object owned by the target owner, or null if o is null or is a `Proxy` that refers to a dead object
Throws:
`IllegalAccessError` - if the given object cannot be bounded to the given type
`IllegalArgumentException` - if the given type is not a shared interface

isSharedInterface

```
public static boolean isSharedInterface(Class<?> c)
```

Tells whether the given class is a shared interface (i.e. an interface owned by a Feature and defined as shared).

Parameters:
`c` - the class to test
Returns:
`true` if the class is a shared interface, `false` otherwise

newProxy

```
public static <T> Proxy<T> newProxy(T ref,
                                    Module owner)
```

Allocates a new `Proxy` and sets its reference to the given object.

Parameters:
`ref` - the `Proxy` reference
`owner` - the owner of the `Proxy` instance
Returns:
the new `Proxy` instance initialized with the given reference

---

isAPI

```
public static boolean isAPI(Class<?> c)
```

Tells whether the given class is a Kernel API.

Parameters:
`c` - the class to test
Returns:
`true` if the class is a Kernel API, `false` otherwise

---

getSharedInterface

```
public static Class<?> getSharedInterface(Class<?> si,
                                          Class<?> topInterface,
                                          Feature target)
```

From a shared interface, gets the closest shared interface in the given target Feature.

The closest shared interface is computed by returning the first equivalent shared interface in the target Feature, starting from the interface `si` to the interface `topInterface` (included).

If the owner of the given shared interface is the target, the same shared interface is returned.

Parameters:
`si` - a shared interface that extends `topInterface` interface
`topInterface` - a shared interface or an interface owned by the Kernel, which is assignable from `si`.
`target` - the target Feature where to find the closest equivalent shared interface of `si`
Returns:
the closest shared interface as described, null if not found.
Throws:
`IllegalArgumentException` - if `si` is not a shared interface or if `topInterface` is not an interface or if `topInterface` is not assignable from `fromClass`
See Also:
`getEquivalentSharedInterface(Class, Feature)`

---

getEquivalentSharedInterface

```
public static Class<?> getEquivalentSharedInterface(Class<?> si,
                                                    Feature target)
```

Gets the equivalent shared interface in the given target Feature.

The equivalent shared interface is the interface owned by the target Feature such as `areEquivalentSharedInterfaces(Class, Class)` is `true`.

Parameters:
`si` - a shared interface
`target` - the target Feature where to find the equivalent shared interface of `si`
Returns:
the equivalent shared interface, null if not found
Throws:
`IllegalArgumentException` - if `si` is not a shared interface

---

areEquivalentSharedInterfaces

```
public static boolean areEquivalentSharedInterfaces(Class<?> si1,
                                                    Class<?> si2)
```

Tells whether the given classes are equivalent shared interfaces.

Two classes are equivalent shared interfaces if they are share interfaces and have the same fully qualified name.

Parameters:
`si1` - a class to test
`si2` - a class to test
Returns:
`true` if the given classes are equivalent shared interfaces, `false` otherwise

---

getImplementedSharedInterface

```
public static Class<?> getImplementedSharedInterface(Class<?> fromClass,
                                                     Class<?> topInterface)
```

Gets the first shared interface implemented by the given class under the hierarchy of `topInterface`.

If `fromClass` is a shared interface it is directly returned.

Parameters:
`fromClass` - a class or an interface owned by a `Feature` that implements `topInterface`
`topInterface` - an interface implemented by `fromClass`
Returns:
the shared interface type as described or `null` if no shared interface found
Throws:
`IllegalArgumentException` - if the given class is an array or is owned by the Kernel or if `topInterface` is not an interface or if `topInterface` is not assignable from `fromClass`

Class Module

ej.kf

```
java.lang.Object
  └ ej.kf.Module
```

Direct Known Subclasses:
Feature, Kernel

```
public class Module
extends Object
```

A `Module` is either Kernel or a Feature. It owns a set of classes, objects, threads and stack contexts.

Constructor Summary	Page
Module()	46

Method Summary	Page
long **getExecutionCounter**() Gets the current execution counter, since the last reset.	48
int **getExecutionQuota**() Gets the execution quota.	48
String **getName**() Gets the name of this module.	46
Principal **getProvider**() Gets the identification of this module provider.	47
byte[] **getUID**() Gets a byte sequence that uniquely identifies the current module.	47
String **getVersion**() Gets a `String` that represents this module version.	47
void **setExecutionQuota**(int quota) Sets the execution quota allocated to the threads owned by this `Module`.	47

Constructor Detail

Module

```
public Module()
```

Method Detail

getName

```
public String getName()
```

Gets the name of this module.

Returns:
the internal name

---

getProvider

```
public Principal getProvider()
```

Gets the identification of this module provider.

Returns:
the module identification.

---

getVersion

```
public String getVersion()
```

Gets a `String` that represents this module version.

Returns:
the module version
See Also:
`IncompatibleFeatureException.getExpectedKernelVersion()`

---

getUID

```
public byte[] getUID()
```

Gets a byte sequence that uniquely identifies the current module.

Returns:
this module UID

---

setExecutionQuota

```
public void setExecutionQuota(int quota)
```

Sets the execution quota allocated to the threads owned by this `Module`. This quota is expressed in execution units.

A Thread owned by a `Module` which execution quota set to `0` will never be scheduled.

A Thread owned by a `Module` which execution quota set to `-1` is always eligible to scheduling.

Calling this method induces a global reset of the quantum of all the Modules.

A `Module` is created with an execution quota set to `-1`. When the quota of all `Module` is set to `-1`, the execution counting is disabled. When the quota of a `Module` is set to a value other than `-1`, the execution counting is enabled.

Parameters:
`quota` - the execution quota to set to this `Module` in execution units

---

Throws:

`IllegalArgumentException` - if the given quota is lower than `-1`.

---

getExecutionQuota

```
public int getExecutionQuota()
```

Gets the execution quota.

Returns:

the quota of this `Module` in execution units.

---

getExecutionCounter

```
public long getExecutionCounter()
```

Gets the current execution counter, since the last reset. The execution counters are reset each time `setExecutionQuota(int)` is called on a `Module`.

Returns:

the total amount of execution units that has been consumed by threads owned by this `Module`, or `0` if execution counting is disabled.

Interface Principal

ej.kf

```
public interface Principal
```

This interface represents and identifies the Kernel or a Feature. Identification uses the 6 well-known fields defined in RFC 2253.

```
CN       commonName
L        localityName
ST       stateOrProvinceName
O        organizationName
OU       organizationalUnitName
C        countryName
```

Field Summary		Page
int	**FIELD_C** C: Country	50
int	**FIELD_CN** CN: Common name	49
int	**FIELD_L** L: Locality	49
int	**FIELD_O** O: Organization	50
int	**FIELD_OU** OU: Organizational Unit	50
int	**FIELD_ST** ST: State or Province	50

Method Summary		Page
String	**getValue**(int field) Gets the value of one of the fields.	50
String	**toString**() Gets a string representation of the X.500 distinguished name using the format defined in RFC 2253.	50

Field Detail

FIELD_CN

```
public static final int FIELD_CN
```

CN: Common name

FIELD_L

```
public static final int FIELD_L
```

L: Locality

FIELD_ST

```
public static final int FIELD_ST
```

ST: State or Province

FIELD_O

```
public static final int FIELD_O
```

O: Organization

FIELD_OU

```
public static final int FIELD_OU
```

OU: Organizational Unit

FIELD_C

```
public static final int FIELD_C
```

C: Country

Method Detail

getValue

```
String getValue(int field)
```

Gets the value of one of the fields.

Parameters:
`field` - One of the 6 well-known fields.
Returns:
the value of the required field.
Throws:
`IndexOutOfBoundsException` - if field is out of bounds.

---

toString

```
String toString()
```

Gets a string representation of the X.500 distinguished name using the format defined in RFC 2253.

Overrides:

`toString` in class `Object`

Class Proxy<T>

ej.kf

```
java.lang.Object
  └ ej.kf.Proxy<T>
```

Type Parameters:

T - the type managed by this Proxy

```
public class Proxy<T>
extends Object
```

The superclass of proxy classes. A proxy class is a class that directly extends this class and directly implements a shared interface. Each method of the implemented interface must be defined according to the following pattern:

```
public class MyProxy extends Proxy implements MySharedInterface{
      public void foo(){
            try{
                  invoke();
            }
            catch(Throwable e){
                  // return or throw default case (deny of service)
            }
      }
```

A Proxy instance has a link to a reference owned by an other Feature. The reference may be automatically removed by the garbage collector.

Constructor Summary	Page
Proxy() The default constructor.	53

Method Summary		Page
T	**getReference**() Returns the reference managed by this Proxy.	55
protected void	**invoke**() This method has for effect to invoke the same method on the reference.	53
protected boolean	**invokeBoolean**() This method has for effect to invoke the same method on the reference.	53
protected byte	**invokeByte**() This method has for effect to invoke the same method on the reference.	53
protected char	**invokeChar**() This method has for effect to invoke the same method on the reference.	54
protected double	**invokeDouble**() This method has for effect to invoke the same method on the reference.	55
protected float	**invokeFloat**() This method has for effect to invoke the same method on the reference.	55
protected int	**invokeInt**() This method has for effect to invoke the same method on the reference.	54
protected long	**invokeLong**() This method has for effect to invoke the same method on the reference.	54

protected Object	**invokeRef**() This method has for effect to invoke the same method on the reference.	55
protected short	**invokeShort**() This method has for effect to invoke the same method on the reference.	54

Constructor Detail

Proxy

```
public Proxy()
```

The default constructor.

Method Detail

invoke

```
protected final native void invoke()
                            throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Throws:

`Throwable` - any kind of exceptions must be catched by the caller

---

invokeBoolean

```
protected final native boolean invokeBoolean()
                                       throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:

the boolean result of the call of the target method

Throws:

`Throwable` - any kind of exceptions must be catched by the caller

---

invokeByte

```
protected final native byte invokeByte()
                                  throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:

the byte result of the call of the target method

Throws:

`Throwable` - any kind of exceptions must be catched by the caller

---

invokeChar

```
protected final native char invokeChar()
                                throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:
the char result of the call of the target method
Throws:
`Throwable` - any kind of exceptions must be catched by the caller

invokeShort

```
protected final native short invokeShort()
                                 throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:
the short result of the call of the target method
Throws:
`Throwable` - any kind of exceptions must be catched by the caller

invokeInt

```
protected final native int invokeInt()
                              throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:
the int result of the call of the target method
Throws:
`Throwable` - any kind of exceptions must be catched by the caller

invokeLong

```
protected final native long invokeLong()
                                throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:
the long result of the call of the target method
Throws:
`Throwable` - any kind of exceptions must be catched by the caller

invokeFloat

```
protected final native float invokeFloat()
                                    throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:
the float result of the call of the target method
Throws:
`Throwable` - any kind of exceptions must be catched by the caller

---

invokeDouble

```
protected final native double invokeDouble()
                                     throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:
the double result of the call of the target method
Throws:
`Throwable` - any kind of exceptions must be catched by the caller

---

invokeRef

```
protected final native Object invokeRef()
                                   throws Throwable
```

This method has for effect to invoke the same method on the reference. Each argument of the current method is converted and passed to the underlying method.

Returns:
the `Object` result of the call of the target method
Throws:
`Throwable` - any kind of exceptions must be catched by the caller

---

getReference

```
public T getReference()
```

Returns the reference managed by this `Proxy`.

Returns:
the reference or null if the reference has been reclaimed.

Interface UncaughtExceptionHandler

ej.kf

```
public interface UncaughtExceptionHandler
```

Deprecated. *Use*

Handler for uncaught exceptions thrown in a Feature.

Method Summary		Page
void	**uncaughtException**(Feature f, Thread t, Throwable e)	56

Method Detail

uncaughtException

```
void uncaughtException(Feature f,
                       Thread t,
                       Throwable e)
```

Class UnknownFeatureException

ej.kf

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └ej.kf.UnknownFeatureException
```

All Implemented Interfaces:
Serializable

```
public class UnknownFeatureException
extends Exception
```

This exception is thrown if a Feature being unloaded is unknown.

See Also:
Kernel.unload(Feature)

Constructor Summary	Page
UnknownFeatureException()	57

Constructor Detail

UnknownFeatureException

```
public UnknownFeatureException()
```