Bundesamt
für Sicherheit in der
Informationstechnik

Deutschland
Digital•Sicher•BSI•

BSI TR-03151-2 Appendix Java

Technical Guideline BSI TR-03151 Secure Element API (SE API)

Part 2: Interface Mapping
Appendix Java - Mapping of API definitions to Java

Version 1.1.0
2023-02-13

Version history

Version	Date	Description
1.1.0	2023-02-13	Initial version

Table 1 Version history

Federal Office for Information Security
Poost Box 20 03 63
D-53133 Bonn

E-Mail: registrierkassen@bsi.bund.de
Internet: https://www.bsi.bund.de

Table of Contents

1 Introduction

This document is an appendix to Technical Guideline [BSI TR-03151-2] which describes the programming language specific mapping of an API definition to Java. The API definition is not part of this appendix.

Within [BSI TR-03151-2] certain concepts and constructs of the API are defined. Here, these constructs are mapped to Java. Regarding the data types the approach is based on the OMG IDL mappings to Java in [OMGx] and [OMG2017a], but has been modified as they focus on a translation into CORBA constructs (which is not the objective for the interface mapping used in [BSI TR-03151-2]).

2 Mapping of the API definition to Java

The following sub-chapters contain the mapping of API constructs to Java, including the mapping of data types, exceptions/error conditions, and function parameters.

2.1 Mapping of data types

Data types are used to store values and are usually manipulated during the execution of functions. There are basic types, that usually only contain one value at a time and more complex ones which may contain multiple values. This sub-chapter introduces the mapping of OMG IDL data types to Java.

2.1.1 Mapping of basic types

Table 2 contains the mapping of basic OMG IDL data types to Java data types. This table has been specified under consideration of the following aspects:

- OMG IDL standard definition for the syntax of basic types in [OMG2017a], Chap. 7.4.1.4.4.1.1, p. 25 f.
- The value ranges for the integer types in OMG IDL as defined in [OMG2017a], Chap. 7.4.1.4.4.1.1.1, p. 26.
- The mapping of the OMG IDL integer types to the corresponding Java types as defined in [OMGx], Table 4.1, p. 6. The definition of the value ranges regarding the relevant Java types occurs in [ORACLE2017] (see Chap. 4.2.1, p. 43).

OMG IDL	Java	Comment
short (-2^{15} ... $2^{15}-1$)	short (-2^{15} ... $2^{15}-1$)	
long (-2^{31} ... $2^{31}-1$)	int (-2^{31} ... $2^{31}-1$)	
long long (-2^{63} ... $2^{63}-1$)	long (-2^{63} ... $2^{63}-1$)	
unsigned short (0 ... $2^{16}-1$)	int (-2^{31} ... $2^{31}-1$)	The range of the corresponding Java type does not match because Java does not support unsigned types[1]. The Java type int SHALL be used. The developers SHALL examine that relevant parameter values belong to the correct range.

[1] Since Java 8, there are special methods found in the classes Integer and Long to represent unsigned data types. As there is no type annotation to differentiate an unsigned from a signed value and to avoid interoperability issues, values SHALL always be interpreted as signed values.

OMG IDL	Java	Comment
unsigned long (0 ... 2^{32}-1)	long (-2^{63} ... 2^{63}- 1)	The range of the corresponding Java type does not match because Java does not support unsigned types[1]. Therefore, the Java type long SHALL be used. Here, the developers SHALL examine that relevant parameter values belong to the correct range.
unsigned long long (0 ... 2^{64} -1)	long (-2^{63} ... 2^{63}- 1)	The range of the corresponding Java type does not match because Java does not support unsigned types[1]. The Java type long SHALL be used for the mapping. In this context, the value range from 0 to 2^{63}- 1 SHALL be relevant for the mapping. Accordingly, the maximal value of the used Java type long is smaller than the maximal value of the corresponding OMG IDL type.
octet (see [OMG2017a], Chap. 7.4.1.4.4.1.1.6, p. 27)	byte (see [ORACLE2017], Chap. 4.2.1, p. 43)	
boolean (see [OMG2017a], Chap. 7.4.1.4.4.1.1.5, p. 27)	boolean (see [ORACLE2017], Chap. 4.2, p. 43)	
Native type DateTime	Represented using the Java class java.time.ZonedDataTime	An OMG IDL native type allows a mapping to a type of a specific programming language. The date/time SHALL be represented in UTC.

Table 2 Mapping of data types

As the comment column of Table 2 shows, there are several data types where the Java equivalent offers a different range of possible values than the OMG IDL type.

2.1.2 Mapping of complex types

2.1.2.1 Mapping of strings

In Java OMG IDL strings are mapped to the data type java.lang.String (see OMGx, Chap. 4.4.4).

In the mapping context, it is possible to encounter empty strings. An empty string in Java is different from a null value. Instead, an empty string shall be represented by a java.lang.String where the number of characters is zero.

Text 1 shows how OMG IDL strings are mapped to Java.

OMG IDL

```
string<100> textWithBounds
string textWithoutBounds
string emptyText = ""
```

Corresponding Java Code

```
String textWithBounds
String textWithoutBounds
String emptyText = ""
```

Text 1: Example for mapping of OMG IDL strings to Java

2.1.2.2 Mapping of enumerations

OMG IDL enumerations are represented by enum types in Java (see [ORACLE2017], Chap. 8.9).

Text 2 shows an example for the mapping of an OMG IDL enumeration to Java.

OMG IDL

```
enum Color {red, green, blue, orange};
enum Fruit {apple, orange};
```

Corresponding Java code

```
enum Color{red, green, blue, orange}
enum Fruit{apple, orange};
```

Text 2: Example for mapping of OMG IDL enumeration to Java

2.1.2.3 Mapping of arrays

OMG IDL arrays are represented by arrays in Java (see [ORACLE2017], Chap. 10, p. 347). The data type that is used for the array definition is based on the mapping in chapter 2.1.1.

Text 3 below shows how an octet array as defined in OMG IDL is mapped to Java.

OMG IDL as used in the Technical Guideline

```
octet variableExample[]
```

Corresponding Java Code

```
byte[] variableLengthArray
```

Text 3: Example of an array in the API definition and Java

2.1.2.4 Definition context

Java uses the construct ‘package ‘(cf. [ORACLE2017], Chap. 7.4, p. 181) to structure source code. All source code files containing the API functions are placed in an API specific package named in reverse-DNS style package naming e.g. “de.bsi.seapi”.

All functions are declared using the help of the Java construct 'interface' (see [ORACLE2017], Chap. 9, p. 293 ff.). The interface named in a specific way e.g. "SEAPI" is declared within a Java-file of the same name. It contains the definition of all functions as public Java methods to be used by an implementing class.

Text 4 below shows an example of how the API functions are mapped to Java language constructs.

Java Code excerpt (in file SEAPI.java)

```
package de.bsi.seapi


public interface SEAPI {
        // function definition of SE API functions
}
```

Text 4: Example of the definition context in Java

2.2 Mapping of Exceptions

Java is a programming language which supports the concept of exceptions as part of the language itself.

Java distinguishes between checked and unchecked exceptions. Checked exceptions must be handled by the developer during implementation, this is enforced at compile time. Unchecked exceptions, sometimes referred to as runtime exceptions, are not checked at compile time.

Exceptions as declared in the API definition are mapped to checked Java exceptions (see [ORACLE2017], Chap. 11.1, p 360). An exception is implemented by a Java class of the same name that extends the class java.lang.Exception.

Java exceptions are assigned to an interface function by the key word 'throws'.

The following example shows the mapping of the exceptions ErrorIllegalDayValue and ErrorIllegalMonthValue to Java[2].

Properties of a function to be implemented

```
Function name: saveTheDate
Input parameter 1: short day
Input parameter 2: short month
Possible exception 1: ErrorIllegalDayValue
Possible exception 2: ErrorIllegalMonthValue
```

Corresponding Java code

```
void saveTheDate(short day, short month)
        throws ErrorIllegalDayValue, ErrorIllegalMonthValue;
```

Text 5 Example for mapping of exceptions to Java

In the API mapping context, all Java exceptions are subclasses of a global exception class (e.g. SeapiException) which itself extends java.lang.Exception.

[2] Note that the exceptions are definied within their own Java files with a filename corresponding to the exception name.

2.2.1 Mapping exceptions with additional information

In some cases additional information has to be conveyed alongside an occurred exception. This is stated in the detailed description of a function. Such information is stored within an additional parameter of the exception on creation (constructor) and passed to the caller. The caller might retrieve the additional information from the occurred exception using a “getter”-method.

Every defined API exception provides the possibility of passing the cause of the exception or a detailed message to the caller. In cases where the Technical Guideline defining the API does not define the contents of this additional parameter, it MAY be used by the developer to pass the cause of an exception to the caller. It SHALL NOT be used for other purposes. Note that additional information as requested by the detailed description of a function SHALL always be conveyed using the additional parameter and the “getter”-method.

Text 6 below shows ErrorIncorrectPin as an example for an exception with an additional parameter.

Java code excerpt for the exception ErrorIncorrectPin

```
public final class ErrorIncorrectPin extends SeapiException {

    /**
    * Constructs a new ErrorIncorrectPin exception with null as the value for its
    * detail message and an information about the PIN retry counter
    *
    * @param remainingRetries number of remaining retries.
    */
    public ErrorIncorrectPin(short remainingRetries) {
            super();
            this.remainingRetries = remainingRetries;
    }

    // ...

    public short getRemainingRetries() {
            return remainingRetries;
    }
```

Text 6: Code excerpt from the declaration of ErrorIncorrectPin which contains the number of remaining retries as parameter.

2.3 Mapping of function parameters

2.3.1 Mapping function input parameters

When mapping input parameters of a primitive type to Java, the OMG IDL types are directly replaced by the corresponding Java types.

The following text shows an example of this translation.

Properties of a function to be implemented

Function name: addKeyValue

Input parameter 1: short key

Input parameter 2: long value

Corresponding Java code

```
void addKeyValue(short key, int value);
```

Text 7 Example for input parameters in Java

Input parameters in form of OMG IDL arrays are mapped to the particular array constructs in Java, following the declaration of a Java array.

The following text shows an example for the mapping of input parameters with the types of a byte array and a string respectively.

Properties of a function to be implemented

Function name: saveData

Input parameter 1: octet inputData[]

Input parameter 2: string comment

Corresponding Java code

```
void saveData(byte inputData[], String comment);
```

Text 8 Example for input parameters in form of arrays and strings

2.3.2 Mapping of function output parameters

While some languages allow for the separate definition of in- and output definitions, Java handles function output in the form of return values.

The data type of a return value for a Java function is denoted by the type of the function. In the context of the API mapping, there are 3 possibilities for return values:

- No output parameters: no return values
- One output parameter: one return value
- Multiple output parameters: one complex return value

In the following, each of the three possibilities will be discussed.

2.3.2.1 No output parameter

Functions without output parameters are mapped to functions of return type 'void' in Java. This means no return value is expected. The successful execution of such a function can be determined by its execution without raising an exception.

Text 9 below shows an example for a function with no output parameters.

Properties of a function to be implemented:

Function name: noOutputParameter

No output parameter defined

Corresponding Java Code

```
void noOutputParameter();
```

Text 9: Example for the handling of functions with no output parameter

2.3.2.2 One output parameter

Functions with one output parameter are mapped to functions with a return type of the corresponding Java data type. Such a data type may be primitive e.g. in the case of returning an integer of type int but may also be reference types e.g. in the case of returning strings or arrays.

The exact return type of the function is based on the type mapping described in chapter 2.1 applied to the OMG IDL type of the output parameter. The name of the output parameter is omitted.

In the case of functions with one output parameter, a lack of raising exceptions denotes the successful execution of a function.

Text 10 below shows an example for a function with one output parameter.

Properties of a function to be implemented:

Function name: oneOutputParameter

Output parameter: octet myData[]

Corresponding Java Code

```
byte[] oneOutputParameter();
```

Text 10: Example for the handling of functions with one output parameter

2.3.2.3 Multiple output parameters

Functions which return more than one output parameter require an output data type object. Such an object is a wrapper or container for several variables of different data types. The caller of such a function has to use the returned object to access its content.

Data types used as a container to return multiple parameters are denoted by the suffix "Result" and a prefix based on the name of the function. They are declared in a separate file and contain parameters which correspond to the declared name and may be accessed by using "getter"-methods. The type of the parameters is mapped according to chapter 2.1.

Text 11 below shows an example for a function with two output parameter.

Properties of a function to be implemented:

Function name: twoOutputParameters

Output parameter 1: string firstParam

Output parameter 2: long secondParam

Corresponding Java Code

```
class TwoOutputParametersResult {
        private String firstParam;
        private int secondParam;

        public TwoOutputParametersResult(String firstParam, int secondParam) {
                this.firstParam = firstParam;
                this.secondParam = secondParam;
        }

        public String getFirstParam() { return firstParam; }
        public String getSecondParam() { return secondParam; }
}

TwoOutputParametersResult twoOutputParameters();
```

Text 11: Example for the handling of functions with two output parameter

In the case of multiple output parameter functions, a successful execution is denoted by the lack of exceptions raised.

2.3.3 Mapping of optional and conditional function parameters

Java does not have optional parameters as a built-in functionality. Since an API definition may require some parameters to be optional or conditional, this has to be adapted to Java by overloading functions and offering appropriate documentation.[3]

In case of optional parameters overloaded functions with and without the optional parameter are offered. Text 12 below shows an example for a function with an optional input parameter.

[3] Overloading is the practice of defining multiple functions with the same name but different function signatures.

Properties of a function to be implemented

Function name: exampleFunc

Input parameter 1: string requiredParam

Input parameter 2 (optional): boolean optionalParam

Corresponding Java Code

```
void exampleFunc(String requiredParam);
void exampleFunc(String requiredParam, boolean optionalParam);
```

Text 12: Example for the handling of functions with an optional input parameter

In case of conditional parameters the value "null" is used when calling the function to indicate a parameter is not set. Text 13 below shows an example for a function with two conditional input parameters.

Properties of a function to be implemented

Function name: doesUserExist

Input parameter 1 (conditional): string firstname

Input parameter 2 (conditional): string lastname

Output parameter: boolean exists

Condition: at least one of firstname or lastname is present

Corresponding Java Code

```
boolean doesUserExist(String firstname, String lastname);
```

Possible Java calls of the method

```
doesUserExist("John", "Doe");
doesUserExist("John", null);
doesUserExist(null, "Doe");
```

Illegal Java calls of the method

```
doesUserExist(null, null);
```

Text 13: Example for the handling of functions with two conditional input parameters

The specific implementation shall therefore consider that a passed value might be null and appropriate checks should be applied.

Note that "null" is not a valid value to be used for input parameters of a primitive data type. Nevertheless as there are currently no functions with conditional parameters of a primitive type no further definition for that case is made.

Furthermore some input parameters might have a specific value which also indicates optionality. For example the value "0" for an input parameter of type integer may have the same semantics as not having the parameter at all. Please consider the description of the parameter in the referring guideline for further informations. In such cases not every possible combination of function signatures with and without that optional parameter might be represented. This is in contrast to the previous definition of optional parameters and aims to keep the number of overloads for a single function to a reasonable level.

2.3.4 Mapping of large data function parameters

It might be necessary for functions to return a large amount of data in form of a byte array. This especially applies to functions which export data from a device. This might pose a challenge for devices with limited memory. Such functions therefore do not return a byte array directly but instead an instance of java.io.InputStream.

The InputStream allows the caller fine grained control over reading the export data byte-by-byte or in chunks. This way data might be processed and e.g. be written to file without the necessity to keep the complete exported data in memory. The implementation of the necessary subclass of InputStream is left to the developer of the specific API implementation, which incorporates the interface class. Nevertheless the implementation shall adhere to the interface definition of java.io.InputStream.

Text 14 below shows an example for a function which utilises an InputStream[4].

Properties of a function to be implemented

Function name: exportData

Output parameter 1: octet data[]

Output parameter 2: string filename

Corresponding Java Code

```
ExportDataResult exportData();
```

Possible Java call of the method

```
ExportDataResult result = exportData();
File targetFile = new File(result.getFilename());
InputStream inputStream = result.getInputStream();
OutputStream outputStream = new FileOutputStream(targetFile);

byte[] buffer = new byte[1024];
int readBytes;
while ((readBytes = inputStream.read(buffer)) != -1) {
        outputStream.write(buffer, 0, readBytes);
}

inputStream.close();
outputStream.close();
```

Text 14: Example for the handling of functions using an InputStream

[4] The implementation of the InputStream is not part of the example. In addition to the InputStream this example features a second output parameter for the filename which shall be used when storing the data. Therefore, the mechanism described in chapter 2.3.2.3 is also shown here. Please note this example is no implementation advice and only serves explanatory purpose. Any error handling is omitted here.

References

BSI TR-03151-2	BSI: Technical Guideline BSI TR-03151-2: Secure Element API (SE API) – Part 2: Interface Mapping
OMGx	OMG: IDL to Java Language Mapping, Version 1.3, 2008
OMG2017a	OMG: Interface Definition Language, Version 4.1, 2017
ORACLE2017	James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith: The Java® LanguageSpecificationJava SE, 9 Edition, 2017