Java™ Core Reflection

API and Specification

JavaSoft
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043 U.S.A
408-343-1400

January 1997

Copyright Information

© 1996, 1997, Sun Microsystems, Inc. All rights reserved.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

This document is protected by copyright. No part of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

The information described in this document may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, JavaBeans, JDK, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.
Adobe® is a registered trademark of Adobe Systems, Inc.
Netscape Navigator™ is a trademark of Netscape Communications Corporation.
All other product names mentioned herein are the trademarks of their respective owners.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT 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 DOCUMENT. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Please
Recycle

Java™ Core Reflection

Java™ Core Reflection

Overview

The Java™ Core Reflection API provides a small, type-safe, and secure API that supports introspection about the classes and objects in the current Java Virtual Machine. If permitted by security policy, the API can be used to:

- construct new class instances and new arrays
- access and modify fields of objects and classes
- invoke methods on objects and classes
- access and modify elements of arrays

The Core Reflection API defines new classes and methods, as follows:

- Three new classes—`Field`, `Method`, and `Constructor`—that reflect class and interface members and constructors. These classes provide:
  - reflective information about the underlying member or constructor
  - a type-safe means to use the member or constructor to operate on Java objects
- New methods of class `Class` that provide for the construction of new instances of the `Field`, `Method`, and `Constructor` classes.
- One new class—`Array`—that provides methods to dynamically construct and access Java arrays.
- One new utility class—`Modifier`—that helps decode Java language modifier information about classes and their members.

There are also some additions to the `java.lang` package that support reflection. These additions are:

- Two new classes—`Byte` and `Short`. These new classes are subclasses of the class `Number`, and are similar to the class `Integer`. Instances of these new classes serve as object wrappers for primitive values of type `byte` and `short`, respectively.
- New objects, instances of the class `Class`, to represent the primitive Java types `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`, and the keyword `void`, at run-time.
- A new, uninstantiable placeholder class—`Void`—to hold a reference to the `Class` object representing the keyword `void`.

Applications

The Core Reflection API accommodates two categories of applications.

One category is comprised of applications that need to discover and use all of the `public` members of a target object based on its run-time class. These applications require run-time access to all the `public` fields, methods, and constructors of an object. Examples in this category are services such as *Java™ Beans[1]*, and lightweight tools, such as object inspectors. These applications use the instances of the classes `Field`, `Method`, and `Constructor` obtained through the methods `getField`, `getMethod`, `getConstructor`, `getFields`, `getMethods`, and `getConstructors` of class `Class`.

The second category consists of sophisticated applications that need to discover and use the members declared by a given class. These applications need run-time access to the implementation of a class at the level provided by a `class` file. Examples in this category are development tools, such as debuggers, interpreters, inspectors, and class browsers, and run-time services, such as *Java™ Object Serialization[2].* These applications use instances of the classes `Field`, `Method`, and `Constructor` obtained through the methods `getDeclaredField`, `getDeclaredMethod`, `getDeclaredConstructor`, `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` of class `Class`.

Reflection Model

The three classes `Field`, `Method`, and `Constructor` are `final`. Only the Java Virtual Machine may create instances of these classes; these objects are used to manipulate the underlying objects; that is, to:

- get reflective information about the underlying member or constructor
- get and set field values
- invoke methods on objects or classes
- create new instances of classes

The `final` uninstantiable class `Array` provides `static` methods that permit creating new arrays, and getting and setting the elements of arrays.

Member Interface

The classes `Field`, `Method` and `Constructor` implement the `Member` interface. The methods of `Member` are used to query a reflected member for basic identifying information. Identifying information consists of the class or interface that declared the member, the name of the member itself, and the Java language modifiers (such as `public`, `protected`, `abstract`, `synchronized`, and so on) for the member.

Field Objects

A `Field` object represents a reflected field. The underlying field may be a class variable (a `static` field) or an instance variable (a non-`static` field). Methods of class `Field` are used to obtain the type of the underlying field, and to get and set the underlying field's value on objects.

Method Objects

A `Method` object represents a reflected method. The underlying method may be an abstract method, an instance method, or a class (`static`) method.

Methods of class `Method` are used to obtain the formal parameter types, the return type, and the checked exception types of the underlying method. In addition, the `invoke` method of class `Method` is used to invoke the underlying method on target objects. Instance and abstract method invocation uses dynamic method resolution based on the target object's run-time class and the reflected method's declaring class, name, and formal parameter types. (Thus, it is permissible to invoke a reflected interface method on an object that is an instance of a class that implements the interface.) Static method invocation uses the underlying static method of the method's declaring class.

Constructor Objects

A `Constructor` object represents a reflected constructor. Methods of class `Constructor` are used to obtain the formal parameter types and the checked exception types of the underlying constructor. In addition, the `newInstance` method of class `Constructor` is used to create and initialize a new instance of the class that declares the constructor, provided the class is instantiable.

Array and Modifier Classes

The `Array` class is an uninstantiable class that exports class methods to create Java arrays with primitive or class component types. Methods of class `Array` are also used to get and set array component values.

The `Modifier` class is an uninstantiable class that exports class methods to decode Java language modifiers for classes and members. The language modifiers are encoded in an integer, and use the encoding constants defined by *The Java Virtual Machine Specification*.

Representation of Primitive Java Types

Finally, there are nine new `Class` objects that are used to represent the eight primitive Java types and `void` at run-time. (Note that these are `Class` *objects*, not classes.) The Core Reflection API uses these objects to identify the following:

- primitive field types
- primitive method and constructor parameter types
- primitive method return types

The Java Virtual Machine creates these nine `Class` objects. They have the same names as the types that they represent. The `Class` objects may only be referenced via the following `public final static` variables:

```
java.lang.Boolean.TYPE
java.lang.Character.TYPE
java.lang.Byte.TYPE
java.lang.Short.TYPE
java.lang.Integer.TYPE
```

```
java.lang.Long.TYPE
java.lang.Float.TYPE
java.lang.Double.TYPE
java.lang.Void.TYPE
```

In particular, these `Class` objects are not accessible via the `forName` method of class `Class`.

Security Model

The Java security manager controls access to the Core Reflection API on a class-by-class basis. There are two levels of checks to enforce security and safety, as follows:

- The new methods of class `Class` that give reflective access to a member or a set of members of a class are the only source for instances of `Field`, `Method`, and `Constructor`. These methods first delegate security checking to the system security manager (if installed), which throws a `SecurityException` should the request for reflective access be denied.
- Once the system security manager grants initial reflective access to a member, any code may query the reflected member for its identifying information. However, standard Java language access control checks—for `protected`, default (package) access, and `private` classes and members—will normally occur when the individual reflected members are used to operate on the underlying members of objects,that is, to get or set field values, to invoke methods, or to create and initialize new objects. Unrestricted access, which overrides standard language access control rules, may be granted to privileged code (such as debugger code)—a future version of this specification will define the interface by which this may be accomplished.

The initial policy decision is centralized in a new method of class `SecurityManager`, the `checkMemberAccess` method

```
void checkMemberAccess(Class,int) throws SecurityException
```

The `Class` parameter identifies the class or interface whose members need to be accessed. The `int` parameter identifies the set of members to be accessed—either `Member.PUBLIC` or `Member.DECLARED`.

If the requested access to the specified set of members of the specified class is denied, the method should throw a `SecurityException`. If the requested access to the set is granted, the method should return.

As stated earlier, standard Java language access control will be enforced when a reflected member from this set is used to operate on an underlying object, that is, when:

- a `Field` is used to get or set a field value
- a `Method` is used to invoke a method
- a `Constructor` is used to create and initialize a new instance of a class

If access is denied at that point, the reflected member will throw an `IllegalAccessException`.

Java Language Policy

The Java language security policy for applications is that any code may gain reflective access to all the members and constructors (including non-`public` members and constructors) of any class it may link against. Application code that gains reflective access to a member or constructor may only *use* the reflected member or constructor with standard Java language access control.

JDK 1.1 Security Policy

Sun's Java Development Kit 1.1 (JDK1.1) implements its own security policy that is *not* part of the language specification. In Sun's JDK1.1, the class `AppletSecurity` implements the following policy:

- Untrusted (applet) code is granted access to:
  - All `public` members of all `public` classes loaded by the same class loader as the untrusted code
  - All `public` members of `public` system classes
  - All declared (including non-`public`) members of all classes loaded by the same class loader as the untrusted code
- Trusted (applet) code, defined as code signed by a trusted entity, is additionally granted access to all members of system classes.
- System code, defined as code loaded from `CLASSPATH`, is additionally granted access to all classes loaded by all class loaders.

Any code that gains reflective access to a member may only use it with standard Java language access control. There is no notion of privileged code, and no means to override the standard language access control checks.

This policy is conservative with respect to untrusted code—it is more restrictive than the linker for the Java Virtual Machine. For example, an untrusted class cannot, by itself, access a `protected` member of a system superclass via reflection, although it can via the linker. (However, system code may access such members and pass them to untrusted code.)

The JDK security policy is expected to evolve with the security framework for Java.

Data Conversions

Certain methods in the reflection package perform automatic data conversions between values of primitive types and objects of class types. These are the generic methods for getting and setting field and array component values, and the methods for method and constructor invocation.

There are two types of automatic data conversions. *Wrapping conversions* convert from values of primitive types to objects of class types. *Unwrapping conversions* convert objects of class types to values of primitive types. The rules for these conversions are defined in “Wrapping and Unwrapping Conversions.”

Additionally, field access and method invocation permit *widening conversions* on primitive and reference types. These conversions are documented in *The Java Language Specification*, section 5, and are detailed in “Widening Conversions.”

Wrapping and Unwrapping Conversions

A primitive value is automatically wrapped in an object when it is retrieved via `Field.get` or `Array.get`, or when it is returned by a method invoked via `Method.invoke`.

Similarly, an object value is automatically unwrapped when supplied as a parameter in a context that requires a value of a primitive type. These contexts are:

- `Field.set`, where the underlying field has a primitive type
- `Array.set`, where the underlying array has a primitive element type
- `Method.invoke` or `Constructor.newInstance`, where the corresponding formal parameter of the underlying method or constructor has a primitive type

The following table shows the correspondences between primitive types and class (wrapper) types:

`boolean`	`java.lang.Boolean`
`char`	`java.lang.Character`
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

A method that is declared `void` returns the special reference `null` when it is invoked via `Method.invoke`.

Widening Conversions

The reflection package permits the same widening conversions at run-time as permitted in method invocation contexts at compile time. These conversions are defined in *The Java Language Specification*, section 5.3.

Widening conversions are performed at run-time:

- when a value is retrieved from a field or an array via the methods of `Field` and `Array`
- when a value is stored into a field or an array via the methods of `Field` and `Array`
- when an unwrapped actual parameter value is converted to the type of its corresponding formal parameter during method or constructor invocation via `Method.invoke` or `Constructor.newInstance`

The permitted *widening primitive conversions* are:

- From `byte` to `short`, `int`, `long`, `float`, or `double`
- From `short` to `int`, `long`, `float`, or `double`
- From `char` to `int`, `long`, `float`, or `double`
- From `int` to `long`, `float`, or `double`
- From `long` to `float` or `double`
- From `float` to `double`.

The permitted *widening reference conversions* are:

- From a class type *S* to a class type *T*, provided that *S* is a subclass of *T*
- From a class type *S* to an interface type *K*, provided that *S* implements *K*
- From an interface type *J* to an interface type *K*, provided that *J* is a subinterface of *K*

Packaging

The Core Reflection API is in a new subpackage of `java.lang` named `java.lang.reflect`. This avoids compatibility problems caused by Java's default package importation rules.

The class java.lang.Class

```
package java.lang;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;
public final class Class extends Object
```

Instances of the class `Class` represent Java types in a way that allows them to be manipulated by a running Java program. Every array also belongs to a class that is reflected as a `Class` object that is shared by all arrays with the same element type and number of dimensions. Finally, the eight primitive Java types and `void` are also represented by unique `Class` objects.

There are no public constructors for class `Class`. The Java Virtual Machine automatically constructs `Class` objects when new classes are loaded; such objects cannot be created by user programs.

While the class `Class` properly belongs in the `java.lang.reflect` package, it remains in `java.lang` for backwards compatibility.

Methods

The class `Class` is augmented with new methods to:

- determine if a `Class` object represents an array type
- determine if a `Class` object represents a primitive type
- determine the Java language modifiers of the represented class type
- reflect the members and constructors of the represented type
- determine if an object is an instance of the represented class, or implements the represented interface
- determine if a class or interface is a superclass or superinterface of a class or interface
- get the component type for a represented array type

The existing and new methods of class `Class` are described below.

toString

```
public String toString()
```

Returns a `String` consisting of the word `class`, a space, and the fully-qualified name of the class if this `Class` object represents a class (either a declared class or an array class). If this `Class` object represents an interface, then this method returns a `String` consisting of the word `interface`, followed by a space, followed by the fully-qualified name of the interface. If this `Class` object represents a primitive type, then this method returns the name of the primitive type. If this `Class` object represents `void`, returns the String "`void`".

This method overrides the `toString` method of class `Object`.

forName

```
public static Class forName(String className)
    throws ClassNotFoundException
```

Given the fully-qualified name for a class, this method attempts to locate, load, and link the specified class. If it succeeds, returns the `Class` object representing the class. If it fails, the method throws a `ClassNotFoundException`.

`Class` objects for array types may be obtained via this method. These Class objects are automatically constructed by the Java Virtual Machine.

`Class` objects that represent the primitive Java types or `void` cannot be obtained via this method.

newInstance

```
public Object newInstance()
    throws InstantiationException, IllegalAccessException
```

Creates and initializes a new instance of the class represented by this `Class` object, provided it represents an instantiable class. This is done exactly as if by an instance creation expression with an empty argument list. If evaluation of such an instance creation expression would complete abruptly, then the invocation of `newInstance` will complete abruptly for the same reason. Otherwise, it returns the newly created and initialized instance.

The method throws an `IllegalAccessException` if the class or initializer is not accessible to the calling class. The method throws an `InstantiationException` if it attempts to instantiate an abstract class or an interface, or if it is invoked on a `Class` object that represents a primitive type or `void`.

isInstance

`public boolean isInstance(Object obj)`

This method is the dynamic equivalent of the Java language `instanceof` operator. The method returns `true` if the specified `Object` argument is non-`null` and can be cast to the reference type represented by this `Class` object without raising a `ClassCastException`. It returns `false` otherwise.

If this `Class` object represents a primitive type or `void`, returns `false`.

See *The Java Language Specification*, section 15.19.2.

isAssignableFrom

`public boolean isAssignableFrom(Class fromClass)`

This method tests whether the type represented by the specified `Class` parameter can be converted to the type represented by this `Class` object via an identity conversion or via a widening reference conversion. It returns `true` if so, `false` otherwise.

If this `Class` object represents a primitive type, returns `true` if the specified `Class` parameter is exactly this `Class` object, `false` otherwise.

This method throws a `NullPointerException` if the specified `Class` parameter is `null`.

See *The Java Language Specification*, sections 5.1.1, 5.1.4 and 5.2.

isInterface

`public boolean isInterface()`

If this `Class` object represents an interface type, returns `true`, otherwise returns `false`.

isArray

`public boolean isArray()`

If this `Class` object represents an array type, returns `true`; otherwise returns `false`.

isPrimitive

`public boolean isPrimitive()`

If this `Class` object represents a primitive Java type, returns `true`; otherwise returns `false`.

There are nine predefined `Class` objects that represent theprimitive Java types and `void`. These are created by the Java Virtual Machine, and have the same names as the primitive types that they represent. These objects may only be accessed via the following `public static final` variables:

```
java.lang.Boolean.TYPE
java.lang.Character.TYPE
java.lang.Byte.TYPE
java.lang.Short.TYPE
java.lang.Integer.TYPE
java.lang.Long.TYPE
java.lang.Float.TYPE
java.lang.Double.TYPE
java.lang.Void.TYPE
```

These are the only `Class` objects for which this method returns `true`.

getName

`public String getName()`

Returns the fully-qualified name of the class (declared or array), interface, primitive type or `void` represented by this `Class` object, as a `String`.

getModifiers

`public int getModifiers()`

Returns the Java language modifiers for this class or interface, encoded in an integer. The modifiers consist of the Java Virtual Machine's constants for `public`, `protected`, `private`, `final`, and `interface`; they should be decoded using the methods of class `Modifier`.

The modifier encodings are defined in *The Java Virtual Machine Specification*, table 4.1.

getClassLoader

`public ClassLoader getClassLoader()`

Returns the class loader object that loaded this `Class`. Returns `null` if this `Class` was not loaded by a class loader.

getSuperclass

`public Class getSuperclass()`

If this `Class` object represents a class other than `Object`, returns the `Class` that represents the superclass of the class. Returns `null` if this `Class` represents the class `Object`, or if it represents an interface type or a primitive type.

getInterfaces

`public Class[] getInterfaces()`

Returns an array of `Class` objects representing the interfaces of the class or interface represented by this `Class` object. If this `Class` object represents a class, returns an array containing objects representing the interfaces directly implemented by the class. If this `Class` object represents an interface, returns an array containing objects representing the direct superinterfaces of the interface. Returns an array of length `0` if this `Class` implements no interfaces or if it represents a primitive type.

getComponentType

`public Class getComponentType()`

If this `Class` object represents an array type, returns the `Class` object representing the component type of the array; otherwise returns `null`.

getDeclaringClass

`public Class getDeclaringClass()`

If this class or interface is a member of another class, returns the `Class` object representing the class of which it is a member (its *declaring class*). Returns a `null` reference if this class or interface is not a member of any other class.

getClasses

```
public Class[] getClasses()
```

Returns an array containing `Class` objects representing all the `public` classes and interfaces that are members of the class represented by this `Class` object. This includes `public` class and interface members inherited from superclasses and `public` class and interface members declared by the class. Returns an array of length `0` if the class has no public member classes or interfaces, or if this `Class` object represents a primitive type.

getFields

```
public Field[] getFields()
    throws SecurityException
```

Returns an array containing `Field` objects reflecting all the `public` *accessible* fields of the class or interface represented by this `Class` object, including those declared by the class or interface and those declared by superclasses and superinterfaces. (Thus, the array includes the `public` *member* fields of the class as well as any additional `public` *hidden* fields.) Returns an array of length `0` if the class or interface has no `public` accessible fields.

Note that the implicit `length` field for array types is not reflected by this method. User code should use the methods of class `Array` to manipulate arrays.

The method throws a `SecurityException` if access to this information is denied.

See *The Java Language Specification*, sections 8.2 and 8.3.

getMethods

```
public Method[] getMethods()
    throws SecurityException
```

Returns an array containing `Method` objects reflecting all the `public` *member* methods of the class or interface represented by this `Class` object, including those declared by the class or interface and and those inherited from superclasses and superinterfaces. Returns an array of length `0` if the class or interface has no `public` member methods.

The method throws a `SecurityException` if access to this information is denied.

See *The Java Language Specification*, sections 8.2 and 8.4.

getConstructors

```
public Constructor[] getConstructors()
    throws SecurityException
```

Returns an array containing `Constructor` objects that reflect all the `public` constructors of the class represented by this `Class` object. Returns an array of length `0` if the class has no `public` constructors, or if this `Class` object represents an interface or a primitive type.

The method throws a `SecurityException` if access to this information is denied.

getField

```
public Field getField(String name)
    throws NoSuchFieldException, SecurityException
```

Returns a `Field` object that reflects the specified `public` *accessible* field of the class or interface represented by this `Class` object. The `name` parameter is a `String` specifying the simple name of the desired field.

The field to be reflected is located by searching all the accessible fields of the class or interface represented by this `Class` object for a `public` field with the specified name.

The method throws a `NoSuchFieldException` if a matching field is not found.

The method throws a `SecurityException` if access to the underlying field is denied.

See *The Java Language Specification*, sections 8.2 and 8.3.

getMethod

```
public Method getMethod(String name, Class[] parameterTypes)
    throws NoSuchMethodException, SecurityException
```

Returns a `Method` object that reflects the specified public *member* method of the class or interface represented by this `Class` object. The `name` parameter is a `String` specifying the simple name the desired method, and the `parameterTypes` parameter is an array of `Class` objects that identify the method's formal parameter types, in declared order.

The method to reflect is located by searching all the member methods of the class or interface represented by this `Class` object for a `public` method with the specified name and exactly the same formal parameter types.

The method throws a `NoSuchMethodException` a matching method is not found.

The method throws a `SecurityException` if access to the underlying method is denied.

See *The Java Language Specification*, sections 8.2 and 8.4.

getConstructor

```
public Constructor getConstructor(Class[] parameterTypes)
    throws NoSuchMethodException, SecurityException
```

Returns a `Constructor` object that reflects the specified `public` constructor of the class represented by this `Class` object. The `parameterTypes` parameter is an array of `Class` objects that identify the constructor's formal parameter types, in declared order.

The constructor to reflect is located by searching all the constructors of the class represented by this `Class` object for a `public` constructor with the exactly the same formal parameter types.

The method throws a `NoSuchMethodException` if a matching constructor is not found.

The method throws a `SecurityException` if access to the underlying constructor is denied.

getDeclaredClasses

```
public Class[] getDeclaredClasses()
    throws SecurityException
```

Returns an array of `Class` objects reflecting all the classes and interfaces declared as members of the class represented by this `Class` object. This includes `public`, `protected`, default (package) access, and `private` classes and interfaces declared by the class, but excludes inherited classes and interfaces. Returns an array of length `0` if the class declares no classes or interfaces as members, or if this `Class` object represents a primitive type.

The method throws a `SecurityException` if access to this information is denied.

getDeclaredFields

```
public Field[] getDeclaredFields()
    throws SecurityException
```

Returns an array of `Field` objects reflecting all the fields declared by the class or interface represented by this `Class` object. This includes `public`, `protected`, default (package) access, and `private` fields, but excludes inherited fields. Returns an array of length `0` if the class or interface declares no fields, or if this `Class` object represents a primitive type.

The method throws a `SecurityException` if access to this information is denied.

See *The Java Language Specification*, sections 8.2 and 8.3.

getDeclaredMethods

```
public Method[] getDeclaredMethods()
    throws SecurityException
```

Returns an array of `Method` objects reflecting all the methods declared by the class or interface represented by this `Class` object. This includes `public`, `protected`, default (package) access, and `private` methods, but excludes inherited methods. Returns an array of length `0` if the class or interface declares no methods, or if this `Class` object represents a primitive type.

The method throws a `SecurityException` if access to this information is denied.

See *The Java Language Specification*, sections 8.2 and 8.4.

getDeclaredConstructors

```
public Constructor[] getDeclaredConstructors()
    throws SecurityException
```

Returns an array of `Constructor` objects reflecting all the constructors declared by the class represented by this `Class` object. These are `public`, `protected`, default (package) access, and `private` constructors. Returns an array of length `0` if this `Class` object represents an interface or a primitive type.

The method throws a `SecurityException` if access to this information is denied.

getDeclaredField

```
public Field getDeclaredField(String name)
    throws NoSuchFieldException, SecurityException
```

Returns a `Field` object that reflects the specified declared field of the class or interface represented by this `Class` object. The `name` parameter is a `String` that specifies the simple name of the desired field.

The method throws a `NoSuchFieldException` if a field with the specified name is not found.

The method throws a `SecurityException` if access to the underlying field is denied.

See *The Java Language Specification*, sections 8.2 and 8.3.

getDeclaredMethod

```
public Method getDeclaredMethod(String name,
                                Class[] parameterTypes)
    throws NoSuchMethodException, SecurityException
```

Returns a `Method` object that reflects the specified declared method of the class or interface represented by this `Class` object. The `name` parameter is a `String` that specifies the simple name of the desired method, and the `parameterTypes` parameter is an array of `Class` objects that identify the method's formal parameter types, in declared order.

The method throws a `NoSuchMethodException` if a matching method is not found.

The method throws a `SecurityException` if access to the underlying method is denied.

See *The Java Language Specification*, sections 8.2 and 8.4.

getDeclaredConstructor

```
public Constructor getDeclaredConstructor(Class[] parameterTypes)
    throws NoSuchMethodException, SecurityException
```

Returns a `Constructor` object that reflects the specified constructor of the class or interface represented by this `Class` object. The `parameterTypes` parameter is an array of `Class` objects that identify the constructor's formal parameter types, in declared order.

The method throws a `NoSuchMethodException` if a matching constructor is not found.

The method throws a `SecurityException` if access to the underlying constructor is denied.

The interface java.lang.reflect.Member

```
package java.lang.reflect;
public interface Member
```

The `Member` interface reflects identifying information about a class or interface member or constructor.

Fields

PUBLIC

```
public final static int PUBLIC
```

Identifies the public members of a class or interface.

This is used by class `SecurityManager` when determining security policy.

DECLARED

```
public final static int DECLARED
```

Identifies the declared members of a class or interface.

This is used by class `SecurityManager` when determining security policy.

Methods

getDeclaringClass

```
public abstract Class getDeclaringClass()
```

Returns the `Class` object representing the class or interface that declares this member or constructor.

getName

```
public abstract String getName()
```

Returns the name of this member or constructor, as a `String`. The member or constructor name does not include the name of its declaring class or interface.

getModifiers

```
public abstract int getModifiers()
```

Returns the Java language modifiers for this member or constructor, encoded in an integer. The `Modifier` class should be used to decode the modifiers.

The modifier encodings are defined in *The Java Virtual Machine Specification*, Tables 4.1, 4.3, and 4.4.

The class java.lang.reflect.Field

```
package java.lang.reflect;
public final class Field extends Object implements Member
```

A `Field` provides information about, and access to, a single field of a class or interface. The reflected field may be a class variable (`static` field) or an instance variable (instance field).

Only the Java Virtual Machine may create `Field` objects; user code obtains `Field` references via the methods `getField`, `getFields`, `getDeclaredField`, and `getDeclaredFields` of class `Class`.

`A Field` permits widening conversions to occur during `get` or `set` operations, but throws an `IllegalArgumentException` if a narrowing conversion would occur.

Methods

getDeclaringClass

```
public Class getDeclaringClass()
```

Returns the `Class` object representing the class or interface that declares the field represented by this `Field` object.

getName

```
public String getName()
```

Returns the simple name of the field represented by this `Field` object.

getModifiers

```
public int getModifiers()
```

Returns the Java language modifiers for the field represented by this `Field` object, encoded in an integer. The `Modifier` class should be used to decode the modifiers.

The modifier encodings are defined in *The Java Virtual Machine Specification*, Table 4.3.

getType

`public Class getType()`

Returns a `Class` object that identifies the declared type for the field represented by this `Field` object.

equals

`public boolean equals(Object obj)`

Compares this `Field` against the specified `Object`. Returns `true` if they are the same; `false` otherwise. Two `Field` objects are the same if they have the same declaring class and the same name.

This method overrides the `equals` method of class `Object`.

hashCode

`public int hashCode()`

Returns a hashcode for this `Field` object. This is computed as the exclusive-or of the hashcodes for the `Field`'s declaring class name and its simple name.

This method overrides the `hashCode` method of class `Object`.

toString

`public String toString()`

Returns a `String` object describing this `Field`. The format is the Java language modifiers for the represented field, if any, followed by the field type, followed by a space, followed by the fully-qualified name of the class declaring the field, followed by a period, followed by the name of the field. For example:

```
public static final int java.lang.Thread.MIN_PRIORITY
private int java.io.FileDescriptor.fd
```

The modifiers are placed in canonical order as specified in *The Java Language Specification*. This is `public`, `protected`, or `private` first, and then other modifiers in the following order: `static`, `final`, `transient`, and `volatile`.

This method overrides the `toString` method of class `Object`.

get

```
public Object get(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field`, on the specified object. The value is automatically wrapped in an object if it has a primitive type.

The underlying field's value is obtained as follows:

- If the underlying field is a `static` field, the object argument is ignored; it may be `null`.
- Otherwise, the underlying field is an instance field. If the specified object argument is `null`, the method throws a `NullPointerException`. If the specified object is not an instance of the class or interface declaring the underlying field, the method throws an `IllegalArgumentException`.
- If this `Field` object enforces Java language access control, and the underlying field is inaccessible, the method throws an `IllegalAccessException`.
- Otherwise, the value is retrieved from the underlying instance or static field. If the field has a primitive type, the value is wrapped in an object before being returned, otherwise it is returned as is.

Primitive variants of `Field.get` are also provided for efficiency; these avoid the final wrapping conversion. They are described below.

getBoolean

```
public boolean getBoolean(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as a `boolean`. See `Field.get` for the detailed procedure.

If the underlying field is not of type `boolean`, the method throws an `IllegalArgumentException`.

getByte

```
public byte getByte(Object obj)
    throws NullPointerException, IllegalArgumentException,
```

```
    IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as a `byte`. See `Field.get` for the detailed procedure.

If the underlying field is not of type `byte`, the method throws an `IllegalArgumentException`.

getChar

```
public char getChar(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as a `char`. See `Field.get` for the detailed procedure.

If the underlying field's value is not of type `char`, the method throws an `IllegalArgumentException`.

getShort

```
public short getShort(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as a `short`. See `Field.get` for the detailed procedure.

If the underlying field's value cannot be converted to a `short` by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

getInt

```
public int getInt(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as an `int`. See `Field.get` for the detailed procedure.

If the underlying field's value cannot be converted to an `int` by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

getLong

```
public long getLong(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as a `long`. See `Field.get` for the detailed procedure.

If the underlying field's value cannot be converted to a `long` by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

getFloat

```
public float getFloat(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as a `float`. See `Field.get` for the detailed procedure.

If the underlying field's value cannot be converted to a `float` by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

getDouble

```
public double getDouble(Object obj)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Returns the value of the field represented by this `Field` object on the specified object, as a `double`. See `Field.get` for the detailed procedure.

If the underlying field's value cannot be converted to a `double` by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

set

```
public void set(Object obj, Object value)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the field represented by this `Field` object on the specified object argument to the specified new value. The new value is automatically unwrapped if the underlying field has a primitive type.

The operation proceeds as follows:

- If the underlying field is a `static` field, the object argument is ignored; it may be `null`.
- Otherwise the underlying field is an instance field. If the specified object argument is `null`, the method throws a `NullPointerException`. If the specified object argument is not an instance of the class or interface declaring the underlying field, the method throws an `IllegalArgumentException`.
- If this `Field` object enforces Java language access control, and the underlying field is inaccessible, the method throws an `IllegalAccessException`.
- If the underlying field is a `final` field, the method throws an `IllegalAccessException`.
- If the underlying field is of a primitive type, an unwrapping conversion is attempted to convert the new object value to a value of a primitive type. If the new object value is `null`, the conversion fails by throwing a `NullPointerException`. If the new object value is not an instance of a standard Java wrapper class, the conversion fails by throwing an `IllegalArgumentException`.
- If, after possible unwrapping, the new value cannot be converted to the type of the underlying field by an identity or a widening conversion, the method throws an `IllegalArgumentException`.
- The field is set to the possibly unwrapped and widened new value.

Primitive variants of `Field.set` are also provided for efficiency; these let application code avoid having to wrap the new field value. They are described below.

setBoolean

```
public void setBoolean(Object obj, boolean z)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `boolean` value. See `Field.set` for the detailed procedure.

If the underlying field is not of type `boolean`, the method throws an `IllegalArgumentException`.

setByte

```
public void setByte(Object obj, byte b)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `byte` value. See `Field.set` for the detailed procedure.

If the new value cannot be converted to the type of the underlying field by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

setChar

```
public void setChar(Object obj, char c)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `char` value. See `Field.set` for the detailed procedure.

If the new value cannot be converted to the type of the underlying field by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

setShort

```
public void setShort(Object obj, short s)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `short` value. See `Field.set` for the detailed procedure.

If the new value cannot be converted to the type of the underlying field by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

setInt

```
public void setInt(Object obj, int i)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `int` value. See `Field.set` for the detailed procedure.

If the new value cannot be converted to the type of the underlying field by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

setLong

```
public void setLong(Object obj, long l)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `long` value. See `Field.set` for the detailed procedure.

If the new value cannot be converted to the type of the underlying field by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

setFloat

```
public void setFloat(Object obj, float f)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `float` value. See `Field.set` for the detailed procedure.

If the new value cannot be converted to the type of the underlying field by an identity or a widening conversion, the method throws an `IllegalArgumentException`.

setDouble

```
public void setDouble(Object obj, double d)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException
```

Sets the value of the field represented by this `Field` object on the specified object argument to the specified `double` value. See `Field.set` for the detailed procedure.

If the underlying field is not of type `double`, the method throws an `IllegalArgumentException`.

The class java.lang.reflect.Method

```
package java.lang.reflect;
public final class Method extends Object implements Member
```

A `Method` provides information about, and access to, a single method of a class or an interface. The reflected method may be an `abstract` method, a class (`static`) method, or an instance method.

Only the Java Virtual Machine may create `Method` objects; user code obtains `Method` references via the methods `getMethod`, `getMethods`, `getDeclaredMethod`, and `getDeclaredMethods` of class `Class`.

A `Method` permits widening conversions to occur when matching the actual parameters to `invoke` with the underlying method's formal parameters, but it throws an `IllegalArgumentException` if a narrowing conversion would occur.

Methods

getDeclaringClass

```
public Class getDeclaringClass()
```

Returns the `Class` object for the class or interface that declares the method represented by this `Method` object.

getName

```
public String getName()
```

Returns the simple name of the method represented by this `Method` object.

getModifiers

```
public int getModifiers()
```

Returns the Java language modifiers for the method represented by this `Method` object, encoded in an integer. The `Modifier` class should be used to decode the modifiers.

The modifier encodings are defined in *The Java Virtual Machine Specification*, Table 4.4.

getReturnType

```
public Class getReturnType()
```

Returns a `Class` object that represents the formal return type of the method represented by this `Method` object.

getParameterTypes

```
public Class[] getParameterTypes()
```

Returns an array of `Class` objects that represent the formal parameter types, in declaration order, of the method represented by this `Method` object. Returns an array of length `0` if the underlying method takes no parameters.

getExceptionTypes

```
public Class[] getExceptionTypes()
```

Returns an array of `Class` objects that represent the types of the checked exceptions thrown by the underlying method represented by this `Method` object. Returns an array of length `0` if the underlying method throws no checked exceptions.

equals

```
public boolean equals(Object obj)
```

Compares this `Method` against the specified object. Returns `true` if the objects are the same; `false` otherwise. Two `Method` objects are the same if they have the same declaring class, the same name, and the same formal parameter types.

This method overrides the `equals` method of class `Object`.

hashCode

```
public int hashCode()
```

Returns a hashcode for this `Method`. The hashcode is computed as the exclusive-or of the hashcodes of the `Method`'s declaring class name and its simple name.

This method overrides the `hashCode` method of class `Object`.

toString

```
public String toString()
```

Returns a string describing the underlying method represented by this `Method` object. The string is formatted as the underlying method's Java language modifiers, if any, followed by the fully-qualified name of the method return type or `void`, followed by a space, followed by the fully-qualified name of class declaring the method, followed by a period, followed by the simple method name, followed by a parenthesized, comma-separated list of the fully-qualified names of the underlying method's formal parameter types. If the underlying method throws checked exceptions, the parameter list is followed by a space, followed by the word `throws` followed by a comma-separated list of the fully-qualified names of the thrown exception types. For example:

```
    public boolean java.lang.Object.equals(java.lang.Object)
    public final java.lang.String java.lang.Thread.getName()
```

The method modifiers are placed in canonical order as specified by *The Java Language Specification*. This is `public`, `protected` or `private` first, and then other modifiers in the following order: `abstract`, `static`, `final`, `synchronized`, and `native`.

This method overrides the `toString` method of class `Object`.

invoke

```
public Object invoke(Object obj, Object[] args)
    throws NullPointerException, IllegalArgumentException,
        IllegalAccessException, InvocationTargetException
```

Invokes the underlying method represented by this `Method` object on the specified object with the specified parameters. Individual parameters are automatically unwrapped to match primitive formal parameters, and both primitive and reference parameters are subject to widening conversions as necessary. The value returned by the underlying method is automatically wrapped in an object if it has a primitive type.

Method invocation proceeds with the following steps, in order:

- If the underlying method is a class (`static`) method, then the specified object argument is ignored. It may be `null`.
- Otherwise, the underlying method is an abstract method or an instance method. If the specified object argument is `null`, the invocation throws a

`NullPointerException`. Otherwise, if the specified object argument is not an instance of the class or interface declaring the underlying method, the invocation throws an `IllegalArgumentException`.

- If this `Method` object enforces Java language access control and the underlying method is inaccessible, the invocation throws an `IllegalAccessException`.
- If the number of actual parameters supplied via `args` is different from the number of formal parameters required by the underlying method, the invocation throws an `IllegalArgumentException`. If the underlying method takes no parameters, `args` may be `null`.
- For each actual parameter in the supplied `args` array:
  - If the corresponding formal parameter has a primitive type, an unwrapping conversion is attempted to convert the object parameter value to a value of a primitive type. If the object parameter is `null`, the conversion fails by throwing a `NullPointerException`. If the object parameter is not an instance of a standard Java wrapper class, the conversion fails by throwing an `IllegalArgumentException`.
  - If, after possible unwrapping, the parameter value cannot be converted to the corresponding formal parameter type by an identity conversion or a widening conversion, the invocation throws an `IllegalArgumentException`.
- If the underlying method is a class (`static`) method, it is invoked as exactly the underlying method of the relevant declaring class.
- If the underlying method is not a class (`static`) method, it is invoked using dynamic method lookup as documented in *The Java Language Specification*, section 15.11.4.4; in particular, overriding based on the class of the target object will occur.
- Control transfers to the underlying method.
- If the underlying method completes abruptly by throwing an exception, the exception is caught and placed in a new exception object of class `InvocationTargetException`; `invoke` then completes abruptly by throwing this new exception.
- If the underlying method completes normally, the value it returns is returned by `invoke`; if the value has a primitive type, it is first appropriately wrapped in an object. If the underlying method is declared `void`, `invoke` returns `null`.

The class java.lang.reflect.Constructor

```
package java.lang.reflect;
public final class Constructor extends Object implements Member
```

A `Constructor` provides information about, and access to, a single constructor of a declared class. A `Constructor` may be used to create and initialize a new instance of the class that declares the reflected constructor, provided the class is instantiable.

Only the Java Virtual Machine may create `Constructor` objects; user code obtains `Constructor` references via the methods `getConstructor`, `getConstructors`, `getDeclaredConstructor`, and `getDeclaredConstructors` of class `Class`.

A `Constructor` permits widening conversions to occur when matching the actual parameters to `newInstance` with the underlying constructor's formal parameters, but it throws an `IllegalArgumentException` if a narrowing conversion would occur.

Methods

getDeclaringClass

```
public Class getDeclaringClass()
```

Returns the `Class` object for the class that declares the constructor represented by this `Constructor` object.

getName

```
public String getName()
```

Returns the name of the constructor represented by this `Constructor` object. This is the same as the fully-qualified name of its declaring class.

getModifiers

```
public int getModifiers()
```

Returns the Java language modifiers for the constructor represented by this `Constructor` object, encoded in an integer. The `Modifier` class should be used to decode the modifiers.

The modifier encodings are defined in *The Java Virtual Machine Specification*, Table 4.4.

getParameterTypes

```
public Class[] getParameterTypes()
```

Returns an array of `Class` objects that represent the formal parameter types, in declaration order, of the constructor represented by this `Constructor` object. Returns an array of length `0` if the underlying constructor takes no parameters.

getExceptionTypes

```
public Class[] getExceptionTypes()
```

Returns an array of `Class` objects that represent the classes of the checked exceptions thrown by the underlying constructor represented by this `Constructor` object. Returns an array of length `0` if the constructor throws no checked exceptions.

equals

```
public boolean equals(Object obj)
```

Compares this `Constructor` against the specified object. Returns `true` if the objects are the same; `false` otherwise. Two `Constructor` objects are the same if they have the same declaring class and the same formal parameter types.

This method overrides the `equals` method of class `Object`.

hashCode

```
public int hashCode()
```

Returns a hashcode for this `Constructor`. The hashcode is the same as the hashcode for the `Constructor`'s declaring class name.

This method overrides the `hashCode` method of class `Object`.

toString

```
public String toString()
```

Returns a string describing the underlying constructor represented by this `Constructor` object. The string is formatted as the Java language modifiers for the underlying constructor, if any, followed by the fully-qualified name of class declaring the underlying constructor, followed by a parenthesized, comma-separated list of the fully-qualified names of the constructor's formal parameter types. If the constructor throws checked exceptions, the parameter list is followed by a space, followed by the word `throws` followed by a comma-separated list of the fully-qualified names of the thrown exception types. For example:

```
    public java.util.Hashtable(int,float)
```

The only possible modifiers for a constructor are one of `public`, `protected` or `private`, or none if the constructor has default (package) access.

This method overrides the `toString` method of class `Object`.

newInstance

```
public Object newInstance(Object initargs[])
    throws InstantiationException, IllegalArgumentException,
        IllegalAccessException, InvocationTargetException
```

Uses the constructor represented by this `Constructor` object to create and initialize a new instance of the constructor's declaring class, with the specified initialization parameters. Individual parameters are automatically unwrapped to match primitive formal parameters, and both primitive and reference parameters are subject to widening conversions as necessary. Returns the newly created and initialized object.

Creation proceeds with the following steps, in order:

- If the class that declares the underlying constructor represents an abstract class, the creation throws an `InstantiationException`.
- If this `Constructor` object enforces Java language access control and the underlying constructor is inaccessible, the creation throws an `IllegalAccessException`.
- If the number of actual parameters supplied via `initargs` is different from the number of formal parameters required by the underlying constructor,

the creation throws an `IllegalArgumentException`. If the underlying constructor takes no parameters, `initargs` may be `null`.

- A new instance of the constructor's declaring class is created, and its fields are initialized to their default initial values.
- For each actual parameter in the supplied `initargs` array:
  - If the corresponding formal parameter has a primitive type, an unwrapping conversion is attempted to convert the object parameter value to a value of a primitive type. If the object parameter is `null`, the conversion fails by throwing a `NullPointerException`. If the object parameter is not an instance of a standard Java wrapper class, the conversion fails by throwing an `IllegalArgumentException`.
  - If, after possible unwrapping, the parameter value cannot be converted to the corresponding formal parameter type by an identity conversion or a widening conversion, the invocation throws an `IllegalArgumentException`.
- Control transfers to the underlying constructor to initialize the new instance.
- If the underlying constructor completes abruptly by throwing an exception, the exception is caught and placed in a new exception object of class `InvocationTargetException`; `newInstance` then completes abruptly by throwing this new exception.
- If the underlying constructor completes normally, the newly created and initialized instance is returned by `newInstance`.

The class java.lang.reflect.Array

```
package java.lang.reflect;
public final class Array extends Object
```

The `Array` class is an uninstantiable class that exports static methods to create Java arrays with primitive or class component types, and to get and set array component values.

Methods

newInstance

```
public static Object newInstance(Class componentType, int length)
    throws NullPointerException, NegativeArraySizeException
```

Returns a new array with the specified component type and length. The array is created as if by the equivalent array creation expression, namely:

```
    new componentType[length]
```

The method throws a `NullPointerException` if the specified `componentType` argument is `null`.

The method throws a `NegativeArraySizeException` if the specified `length` argument is negative.

newInstance

```
public static Object newInstance(Class componentType,
                                 int[] dimensions)
    throws NullPointerException, IllegalArgumentException,
        NegativeArraySizeException
```

Returns a new array with the specified component type and dimensions. The array is created as if by the equivalent array creation expression, namely:

```
    new componentType[dimensions[0]][dimensions[1]]...
```

The method throws a `NullPointerException` if either the `componentType` argument or the `dimensions` argument is `null`.

The method throws an `IllegalArgumentException` if the specified `dimensions` argument is a zero-dimensional array, or if the number of requested dimensions exceeds the limit on the number of array dimensions supported by the implementation (typically `255`).

The method throws a `NegativeArraySizeException` if any of the elements of the specified `dimensions` array is negative.

getLength

```
public static int getLength(Object array)
    throws NullPointerException, IllegalArgumentException
```

Returns the length of the specified array.

Throws a `NullPointerException` if the specified object argument is `null`.

Throws an `IllegalArgumentException` if the specified object argument is not an array.

get

```
public static Object get(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed component of the specified array object. The value is automatically wrapped in an object if it has a primitive type.

The operation proceeds as follows:

- If the specified object is `null`, the method throws a `NullPointerException`.
- If the specified object is not an array, the method throws an `IllegalArgumentException`.
- If the specified `index` argument is negative, or if it is greater than or equal to the length of the specified array, the method throws an `ArrayIndexOutOfBoundsException`.
- The value of the indexed component is fetched. If the array's component type is primitive, the value is wrapped in an object. The possibly wrapped value is returned.

Primitive variants of `Array.get` are also provided for efficiency; these avoid the final wrapping conversion. They are described below.

getBoolean

```
public static boolean getBoolean(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as a `boolean`. See `Array.get` for the detailed procedure.

If the indexed value is not of type `boolean`, the method throws an `IllegalArgumentException`.

getByte

```
public static byte getByte(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as a `byte`. See `Array.get` for the detailed procedure.

If the indexed value is not of type `byte`, the method throws an `IllegalArgumentException`.

getChar

```
public static char getChar(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as a `char`. See `Array.get` for the detailed procedure.

If the indexed value is not of type `char`, the method throws an `IllegalArgumentException`.

getShort

```
public static short getShort(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as a `short`. See `Array.get` for the detailed procedure.

If the indexed value cannot be converted to a `short` by an identity or widening conversion, the method throws an `IllegalArgumentException`.

getInt

```
public static int getInt(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as an `int`. See `Array.get` for the detailed procedure.

If the indexed value cannot be converted to an `int` by an identity conversion or a widening conversion, the method throws an `IllegalArgumentException`.

getLong

```
public static long getLong(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as a `long`. See `Array.get` for the detailed procedure.

If the indexed value cannot be converted to a `long` by an identity conversion or a widening conversion, the method throws an `IllegalArgumentException`.

getFloat

```
public static float getFloat(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as a `float`. See `Array.get` for the detailed procedure.

If the indexed value cannot be converted to a `float` by an identity conversion or a widening conversion, the method throws an `IllegalArgumentException`.

getDouble

```
public static double getDouble(Object array, int index)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Returns the value of the indexed element in the specified array object, as a `double`. See `Array.get` for the detailed procedure.

If the indexed value cannot be converted to a `double` by an identity conversion or a widening conversion, the method throws an `IllegalArgumentException`.

set

```
public static void set(Object array, int index, Object value)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed component of the specified array object to the specified new value. The new value is first automatically unwrapped if the array has a primitive component type.

The operation proceeds as follows:

- If the specified object argument is `null`, the method throws a `NullPointerException`.
- If the specified object argument is not an array, the method throws an `IllegalArgumentException`.
- If the specified `index` argument is negative, or if it is greater than or equal to the length of the specified array, the method throws an `ArrayIndexOutOfBoundsException`.
- If the array has a primitive component type, an unwrapping conversion is attempted to convert the new object value to a value of a primitive type. If the object parameter is `null`, the conversion fails by throwing a `NullPointerException`. If the object parameter is not an instance of a standard Java wrapper class, the conversion fails by throwing an `IllegalArgumentException`.
- If, after possible unwrapping, the new value cannot be converted to a value of the array component type by an identity conversion or a widening conversion, the method throws an `IllegalArgumentException`.
- The indexed array component is set to the possibly unwrapped and widened new value.

Primitive variants of `Array.set` are also provided for efficiency; these let user code avoid having to wrap the new value. They are described below.

setBoolean

```
public static void setBoolean(Object array, int index, boolean z)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed element of the specified array object to the specified `boolean` value. See `Array.set` for the detailed procedure.

setByte

```
public static void setByte(Object array, int index, byte b)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed element of the specified array object to the specified `byte` value. See `Array.set` for the detailed procedure.

setChar

```
public static void setChar(Object array, int index, char c)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed element of the specified array object to the specified `char` value. See `Array.set` for the detailed procedure.

setShort

```
public static void setShort(Object array, int index, short s)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed component of the specified array object to the specified `short` value. See `Array.set` for the detailed procedure.

setInt

```
public static void setInt(Object array, int index, int i)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed component of the specified array object to the specified `int` value. See `Array.set` for the detailed procedure.

setLong

```
public static void setLong(Object array, int index, long l)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed component of the specified array object to the specified `long` value. See `Array.set` for the detailed procedure.

setFloat

```
public static void setFloat(Object array, int index, float f)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed component of the specified array object to the specified `float` value. See `Array.set` for the detailed procedure.

setDouble

```
public static void setDouble(Object array, int index, double d)
    throws NullPointerException, IllegalArgumentException,
        ArrayIndexOutOfBoundsException
```

Sets the indexed component of the specified array object to the specified `double` value. See `Array.set` for the detailed procedure.

The class java.lang.reflect.Modifier

```
package java.lang.reflect;
public final class Modifier extends Object
```

The `Modifier` class is an uninstantiable class that exports static methods and constants that are used to decode the Java language modifiers for classes and members, which are encoded in an integer.

Fields

NOTE: The values for the integer modifier constants below are those defined in *The Java Virtual Machine Specification*, chapter 4.

PUBLIC

```
public final static int PUBLIC
```

The integer constant for the `public` access modifier.

PRIVATE

```
public final static int PRIVATE
```

The integer constant for the `private` access modifier.

PROTECTED

```
public final static int PROTECTED
```

The integer constant for the `protected` access modifier.

STATIC

```
public final static int STATIC
```

The integer constant for the `static` modifier.

FINAL

```
public final static int FINAL
```

The integer constant for the `final` modifier.

SYNCHRONIZED

```
public final static int SYNCHRONIZED
```

The integer constant for the `synchronized` modifier.

VOLATILE

```
public final static int VOLATILE
```

The integer constant for the `volatile` modifier.

TRANSIENT

```
public final static int TRANSIENT
```

The integer constant for the `transient` modifier.

NATIVE

```
public final static int NATIVE
```

The integer constant for the `native` modifier.

INTERFACE

```
public final static int INTERFACE
```

The integer constant for the `interface` modifier.

ABSTRACT

```
public final static int ABSTRACT
```

The integer constant for the `abstract` modifier.

Methods

isPublic

`public static boolean isPublic(int mod)`

Returns `true` if the specified integer includes the `public` modifier.

isPrivate

`public static boolean isPrivate(int mod)`

Returns `true` if the specified integer includes the `private` modifier.

isProtected

`public static boolean isProtected(int mod)`

Returns `true` if the specified integer includes the `protected` modifier.

isStatic

`public static boolean isStatic(int mod)`

Returns `true` if the specified integer includes the `static` modifier.

isFinal

`public static boolean isFinal(int mod)`

Returns `true` if the specified integer includes the `final` modifier.

isSynchronized

`public static boolean isSynchronized(int mod)`

Returns `true` if the specified integer includes the `synchronized` modifier.

isVolatile

`public static boolean isVolatile(int mod)`

Returns `true` if the specified integer includes the `volatile` modifier..

isTransient

`public static boolean isTransient(int mod)`

Returns `true` if the specified integer includes the `transient` modifier..

isNative

`public static boolean isNative(int mod)`

Returns `true` if the specified integer includes the `native` modifier..

isInterface

`public static boolean isInterface(int mod)`

Returns true if the specified integer includes the `interface` modifier..

isAbstract

`public static boolean isAbstract(int mod)`

Returns true if the specified integer includes the `abstract` modifier..

toString

`public static String toString(int mod)`

Returns a string containing a space-separated list of the names of the modifiers included in the specified integer. For example:

```
public final synchronized
private transient volatile
```

The modifier names are returned in canonical order as specified by *The Java Language Specification*.

The class java.lang.reflect.InvocationTargetException

```
package java.lang.reflect;
public class InvocationTargetException extends Exception
```

`InvocationTargetException` is a checked exception that wraps an exception thrown by an invoked method or constructor.

Constructors

InvocationTargetException

```
public InvocationTargetException(Throwable target)
```

Constructs an `InvocationTargetException` with the specified `target` exception.

InvocationTargetException

```
public InvocationTargetException(Throwable target, String detail)
```

Constructs an `InvocationTargetException` with the specified `target` exception and a `detail` message `String` describing the exception.

Methods

getTargetException

```
public Throwable getTargetException()
```

Returns the the underlying target exception wrapped by this `InvocationTargetException`.

Acknowledgements

The reflection API was designed by Nakul Saraiya and Bill Joy, and benefited from substantial input from John Rose. Thanks also to our colleagues at JavaSoft and our external reviewers for their valuable comments.

References

[1] *Java™ Beans Specification*, JavaSoft

[2] *Java™ Object Serialization Specification*, JavaSoft

JavaSoft
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
408-343-1400

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000