Java Core Reflection

API and Specification

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

September, 1996

Please
Recycle

Java Core Reflection

Overview **9**
- Reflection Model 10
- Security Model 12
  - JDK Security Policy 13
- Data Conversions 13
  - Wrapping and Unwrapping Conversions 14
  - Widening Conversions 14
- Packaging 16

The class java.lang.Class **17**
- Methods 17
  - toString 18
  - forName 18
  - newInstance 18
  - isInstance 19
  - isAssignableFrom 19
  - isInterface 19
  - isArray 20
  - isPrimitive 20
  - getName 20
  - getClassLoader 20

getSuperclass 21
getInterfaces 21
getComponentType 21
getModifiers 21
getDeclaringClass 22
getClasses 22
getFields 22
getMethods 23
getConstructors 23
getField 23
getMethod 24
getConstructor 24
getDeclaredClasses 25
getDeclaredFields 25
getDeclaredMethods 25
getDeclaredConstructors 26

The interface java.lang.reflect.Member 27
Fields 27
PUBLIC 27
DECLARED 27
Methods 27
getDeclaringClass 27
getName 27
getModifiers 28

The class java.lang.reflect.Field 29
Methods 29
getDeclaringClass 29
getName 29
getModifiers 29
getType 30

equals 30
hashCode 30
toString 30
get 31
getBoolean 31
getByte 32
getChar 32
getShort 32
getInt 32
getLong 33
getFloat 33
getDouble 33
set 34
setBoolean 35
setByte 35
setChar 35
setShort 35
setInt 36
setLong 36
setFloat 36
setDouble 36

The class java.lang.reflect.Method 37

Methods 37
getDeclaringClass 37
getName 37
getModifiers 37
getReturnType 38
getParameterTypes 38
getExceptionTypes 38
equals 38
hashCode 38

toString 39
invoke. 39

The class java.lang.reflect.Constructor 41
Methods 41
getDeclaringClass. 41
getName. 41
getModifiers 41
getParameterTypes 42
getExceptionTypes 42
equals 42
hashCode 42
toString 43
newInstance. 43

The class java.lang.reflect.Array 45
Methods 45
newInstance. 45
getLength. 46
get 46
getBoolean 47
getByte 47
getChar 47
getShort 47
getInt 48
getLong 48
getFloat 48
getDouble 49
set 49
setBoolean 50
setByte 50
setChar. 50

setShort 50
setInt 51
setLong 51
setFloat 51
setDouble 51

The class java.lang.reflect.Modifier 52
Fields 52
PUBLIC 52
PRIVATE 52
PROTECTED 52
STATIC 52
FINAL 52
SYNCHRONIZED 53
VOLATILE 53
TRANSIENT 53
NATIVE 53
INTERFACE 53
ABSTRACT 53
Methods 54
isPublic 54
isPrivate 54
isProtected 54
isStatic 54
isFinal 54
isSynchronized 54
isVolatile 55
isTransient 55
isNative 55
isInterface 55
isAbstract 55
toString 55

The class java.lang.reflect.InvocationTargetException 56
Constructors 56
InvocationTargetException 56
InvocationTargetException 56
Methods 56
getTargetException 56

Acknowledgements 57

Java Core Reflection 1

Overview

The Java Core Reflection API provides a small, type-safe and secure API which 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, to access and modify fields, to invoke methods of classes and objects, and to access and modify elements of arrays.

The reflection API consists of:

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

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

- Two new classes—`Byte` and `Short`. These are subclasses of the class `Number`, similar to the class `Integer`, whose instances serve as object wrappers for primitive values of type `byte` and `short`,

- New objects, instances of the class `Class`, to represent the primitive Java types `void`, `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`, at runtime, and,
- A new uninstantiable placeholder class—`Void`—to hold a reference to the `Class` object representing the type `void`.

The reflection API accomodates two categories of applications:

- Standard applications that need to discover and use the `public` members of a target object based on its class. These applications require runtime access to the `public` fields and methods of an object. Examples in this category are services like Java Beans, and lightweight tools like object inspectors. These applications can use the instances of the classes `Field`, `Method`, and `Constructor` obtained from the methods `getField()`, `getMethod()`, `getFields()`, `getMethods()`, and `getConstructors()` of class `Class`.
- Sophisticated applications that need to discover and use the members declared by a given class. These applications need runtime 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 runtime services such as Object Serialization. These applications can use the instances of the classes `Field`, `Method`, and `Constructor` obtained from the methods `getDeclaredFields()`, `getDeclaredMethods()`, and `getDeclaredConstructors()` of class `Class`.

Reflection Model

The `Field`, `Method`, and `Constructor` classes are `final`, and are only created by the Java Virtual Machine. Objects which are instances of these classes are used to manipulate the underlying objects, setting and getting field values, invoking methods on objects or classes, creating new instances of classes, and getting and setting the elements of arrays.

The reflection package models the Java Virtual Machine for field access and method invocation. Resolution of overloaded methods based on the numbers and types of actual parameters is not provided, and inherited fields must be accessed relative to the class in which they are declared.

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. This consists of the class or interface declaring

the member, the name of the member, and the Java language modifiers (such as `public`, `protected`, `abstract`, `synchronized`, etc.) for the member.

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

A `Method` object represents a reflected method. The underlying method may be an abstract method, an instance method, or a static method. Methods on a `Method` return the formal parameter types, the return type, and the checked exception types of the underlying method. Methods of class Method can also be used to invoke the underlying method on objects. Instance method invocation uses dynamic method resolution based on the object's runtime type 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 on the declaring class.

A `Constructor` object represents a reflected constructor. Methods on a `Constructor` return the formal parameter types and the checked exception types of the underlying constructor. Methods of class `Constructor` can also be used to create and initialize a new instance of the class that declares the constructor, provided the class is instantiable.

The `Array` class exports static methods to dynamically create arrays with primitive or class component types, and to dynamically get and set array component values.

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

Finally, there are nine new `Class` objects (not classes) that are used to represent primitive Java types at runtime. These are created by the Java Virtual Machine, and have the same names as the primitive types that they represent. These `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
```

Security Model

Access to the reflection API is controlled on a class-by-class basis by the system's security manager. The security manager may use the standard Java class loader tagging mechanism and/or the Java security infrastructure to make its policy decision.

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 are the only source for instances of `Field`, `Method`, and `Constructor`. These methods first delegate security checking to the system security manager, which throws a `SecurityException` should the request be denied.
- If initial reflective access to a member or a set of members is granted, standard Java language access control checks for protected, package-access and private members will normally occur when the individual reflected members are used to operate on the underlying members of objects, such as to get or set field values, or to invoke methods and constructors. However, unrestricted access may be granted to privileged code: we are currently designing a (security-checked) interface that will allow the default language access control for a member to be overridden.

The policy decision is centralized in a new method of class `SecurityManager`:

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

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

If the requested access to the set of members is denied, the method should throw a `SecurityException`. If the requested access to the set is granted, standard Java language access control will be enforced when using a reflected member from this set to access the underlying member on objects. This is when a `Field` is used to get or set a field value, when a `Method` is used to invoke a method, or when a `Constructor` used to create and initialize a new

instance of a class. If access is denied at that point, the member will throw an `IllegalAccessException`.

JDK Security Policy

In the JDK (but not part of the language specification itself), class `AppletSecurity` implements the following policy:

- Untrusted (applet) code may discover only the public members of classes tagged with the same class loader as the applet class,
- Trusted code, defined as code loaded from the local host or code signed by a trusted entity, may discover all members of all classes,
- Any code that gains access to a member may use it with standard Java language access control,
- There is no notion of privileged code as of this draft.

This policy is conservative with respect to untrusted code. Applets can perform fewer legal operations via reflection than they can via source code. For example, an applet class cannot by itself, gain reflective access to the public members of public builtin classes such as `java.awt.Frame`*, although it may link to such members. However, trusted code may gain access to such members, and hand these to applet code, which can use them with standard Java language access control.

This policy also preserves standard Java access control rules for application code.

The JDK security policy is expected to evolve with Java's security framework.

Data Conversions

Some of the methods in the reflection package need to perform automatic data conversions between primitive and object types. These are the generic methods for getting and setting field and array component values, and the methods for method and constructor invocation. *Wrapping conversions* are used to convert from primitive types to class types. *Unwrapping conversions* are

* This restriction is motivated by the need to preserve certain invariants for implementations of the security manager that rely on checking the "distance" in terms of frames on the Java runtime stack between a security-checked service and a potentially untrusted client. However, JavaSoft is in the process of designing a more robust implementation of the security manager. Among other benefits, this will permit a policy whereby applet code may be safely granted reflective access to public members of public builtin classes.

used to convert from class types to primitive types. The rules for these are defined below.

Additionally, field access and method invocation permit *widening conversions* on primitive and reference types. These are detailed below.

Wrapping and Unwrapping Conversions

A primitive value is automatically wrapped in an object when it is retrieved via `Field.get()` or `Array.get()`, as is a primitive value 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. The 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 conversions are used between primitive and class 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 return type of void is converted to the special reference `null`.

Widening Conversions

The widening conversions permitted at runtime by the reflection package are those permitted at compile time in method invocation contexts as defined in *The Java Language Specification*, section 5.3. These conversions are attempted after possible unwrapping conversions, when assigning a value to a field or

when matching an actual method or constructor parameter to a formal parameter.

Widening conversions are performed by the following operations:

- `Field.set()` and its primitive variants,
- `Array.set()` and its primitive variants,
- `Method.invoke()`, and,
- `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*
- From any interface type to type `Object`
- From any array type to type `Object`
- From any array type to type `Cloneable`
- From an array type *S[]* to an array type *T[]*, provided *S* and *T* are reference types and there is a widening conversion from *S* to *T*
- From the null type to any reference type

---

* See *The Java Language Specification*, section 5.1.2.

** See *The Java Language Specification*, section 5.1.4.

Packaging

The reflection API is in a new subpackage of `java.lang` called `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
```

Instances of the the class `Class` represent Java types in a way that allows them to be manipulated by a running Java program. Java classes and interfaces are represented at runtime by `Class` objects. 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 nine primitive Java types are also represented as `Class` objects.

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

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,
- 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 Java language access modifiers for the represented class or interface,
- get the component type for a represented array type.

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

---

* The class `Class` is logically part of the `java.lang.reflect` subpackage, but remains in `java.lang` for backwards compatibility.

toString

```
public String toString()
```

If this `Class` object represents a class (which may be a declared class or an array class), returns a `String` consisting of the word `class`, a space, and the fully-qualified name of the class. If this `Class` object represents an interface, 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, returns the name of the primitive type.

This method overrides the `toString()` method in 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 class. If it succeeds, returns the `Class` object representing the class. If it fails, the method throws a `ClassNotFoundException`.

Note that `Class` objects that represent primitive types cannot be obtained via this method.

newInstance

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

This method attempts to create and initialize a new instance of the class represented by this `Class` object, provided it represents an instantiable class (whether a declared class or an array 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 call to `newInstance()` will complete abruptly for the same reason. Otherwise, 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` on an attempt to instantiate an abstract class or an interface, or a class that represents a primitive type.

isInstance

```
public boolean isInstance(Object obj)
```

If this `Class` object represents a class, returns `true` if the specified `Object` argument is an instance of the represented class; `false` otherwise. If this `Class` object represents an array type, returns `true` if the specified object can be converted to an object of the array type by an identity conversion or by a widening reference conversion; `false` otherwise. If this `Class` object represents an interface, returns `true` if the class (or any superclass) of the object implements this interface; `false` otherwise. If this `Class` object represents a primitive type, returns `false`.

isAssignableFrom

```
public boolean isAssignableFrom(Class cls) throws
NullPointerException
```

Informally, this method tests whether the class or interface represented by this `Class` object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified `Class` parameter. 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.

More formally, 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. See *The Java Language Specification*, sections 5.1.1 and 5.1.4 , for details.

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

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 to represent primitive Java types. 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 or interface represented by this `Class` object, as a `String`.

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 or a primitive type.

getInterfaces

```
public Class[] getInterfaces()
```

Returns an array of classes representing the interfaces of this class or interface. If this `Class` object represents a class, returns an array containing objects representing the interfaces directly implemented by this class. If this `Class` object represents an interface, returns an array containing the direct superinterfaces of this 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 represents an array type, returns the `Class` object representing the component type of the array; otherwise returns `null`.

getModifiers

```
public int getModifiers()
```

Returns the access modifiers for this class or interface as an integer. Class access modifiers consist of the constants for `public`, `protected`, `private`, `final`, and `interface`; they should be decoded using the methods of class `Modifier`.

See *The Java Virtual Machine Specification*, Table 4.1.

getDeclaringClass

```
public Class getDeclaringClass()
```

(Java2) 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.

See *Java2: Overview and Specification*.

getClasses

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

(Java2) Returns an array containing `Class` objects representing all the public classes and interfaces that are members of this class. This includes public class and interface members inherited from superclasses and public class or interface members declared by this class. Returns an array of length 0 if the class has no public member classes or interfaces.

See *Java2: Overview and Specification*.

getFields

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

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

Note that the implicit `length` field for array types is not reflected by this method. Use the `Array` class to manipulate arrays.

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

See *The Java Language Specification*, section 8.2.

getMethods

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

Returns an array containing `Method` objects reflecting all the public 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*, section 8.2.

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.

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

getField

```
public Field getField(String name, Class type) throws
IllegalArgumentException, SecurityException
```

NOTE: There needs to be a java.lang.NoSuchFieldException class.

Returns a `Field` object that reflects the specified public field of the class or interface represented by this `Class` object. The `name` parameter is a `String` that names the underlying field, and the `type` parameter is a `Class` object that identifies the field type.

The `Field` is conceptually located by searching the array of public fields returned by `Class.getFields()` for a `Field` with the same name and type. If a matching `Field` cannot be found, the method throws an `IllegalArgumentException`.

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

getMethod

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

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

The `Method` is conceptually located by searching the array of public methods returned by `Class.getMethods()` for a `Method` with the same name and type. If a matching `Method` cannot be found, the method throws a `NoSuchMethodException`.

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

getConstructor

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

Returns a `Constructor` object that reflects the specified public 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 `Constructor` is conceptually located by searching the array of public constructors returned by `Class.getConstructors()` for a constructor with the same parameter types. If a matching `Constructor` cannot be found, the method throws a `NoSuchMethodException`.

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

getDeclaredClasses

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

(Java2) Returns an array of `Class` objects reflecting all the classes and interfaces declared as members of the class represented by this `Class` object. These include public, protected, default (package) access, and private classes and interfaces. Inherited classes and interfaces are not included. Returns an array of length 0 if the class declares no classes or interfaces as members.

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

See *Java2: Overview and Specification.*

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. These include public, protected, default (package) access, and private fields. Inherited fields are not included. Returns an array of length 0 if the class or interface declares no fields.

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

See *The Java Language Specification*, section 8.2.

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. These include public, protected, default (package) access, and private methods. Inherited methods are not included. Returns an array of length 0 if the class or interface declares no methods.

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

See *The Java Language Specification*, section 8.2.

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.

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

See *The Java Language Specification*, section 8.2.

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.

DECLARED

```
public final static int DECLARED
```

Identifies the declared members of a class or interface.

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 access modifiers for this member or constructor, as an integer. The `Modifier` class should be used to decode the modifiers.

See *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 implements Member
```

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

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

`Field` permits widening conversions to occur during a `get()` or `set()` operation, 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 name of the field represented by this `Field` object.

getModifiers

```
public int getModifiers()
```

Returns the access modifiers for the field represented by this `Field` object, as an integer. The `Modifier` class should be used to decode the modifiers.

See *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 `Fields` are the same if they were declared by the same class and have the same name and type.

This method overrides the `equals()` method in 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 underlying field's declaring class name and its name.

This method overrides the `hashCode()` method in class `Object`.

toString

```
public String toString()
```

Returns a `String` object describing this `Field`. The format is the access 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`, `volatile`.

This method overrides the `toString()` method in 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`.

---

* See `Class.isInstance()` for the rules to determine whether an object is an instance of a class or interface.

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 cannot be converted to a `char` by an identity or widening conversion, 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 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 widening conversion, 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 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 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 widening conversion, the method throws an `IllegalArgumentException`.

set

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

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 static, 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 final, the method throws an `IllegalAccessException`.
- If the underlying field is of a primitive type, an unwrapping conversion is attempted to convert the new value to a value of a primitive type. If this attempt fails, the method throws an `IllegalArgumentException`.
- If, after possible unwrapping, the new value cannot be converted to the type of the underlying field by an identity or 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 avoid the unwrapping conversion for the new field value. They are described below.

setBoolean

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

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.

setByte

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

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.

setChar

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

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.

setShort

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

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.

setInt

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

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.

setLong

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

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.

setFloat

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

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.

setDouble

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

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.

The class java.lang.reflect.Method

```
package java.lang.reflect;

public final class Method implements Member
```

`Method` provides information about, and access to, a single method on a class or interface. The reflected method may be a class, instance, or abstract method.

Only the Java Virtual Machine may create `Method` objects; user code obtains `Method` references via `Class.getMethod()`, `Class.getMethods()`, and `Class.getDeclaredFields()`.

`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 name of the method represented by this `Method` object, as a `String`.

getModifiers

```
public int getModifiers()
```

Returns the access modifiers for the method represented by this `Method` object, as an integer. The `Modifier` class should be used to decode the modifiers.

See *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 checked exceptions thrown by the underlying method represented by this `Method` object. Returns an array of length 0 if the 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 `Methods` are the same if they were declared by the same class and have the same name and formal parameter types.

This method overrides the `equals()` method in class `Object`.

hashCode

```
public int hashCode()
```

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

This method overrides the `hashCode()` method in class `Object`.

toString

```
public String toString()
```

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

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

The access 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`, `native`.

This method overrides the `toString()` method in 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 static, then the specified object argument is ignored. It may be `null`.

- Otherwise, the method is 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`.
- 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 value to a value of a primitive type. If this attempt fails, the invocation throws an `IllegalArgumentException`.
  - If, after possible unwrapping, the parameter value cannot be converted to the corresponding formal parameter type by an identity or widening conversion, the invocation throws an `IllegalArgumentException`.
- If the underlying method is an instance 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 runtime type of the target object will occur.
- If the underlying method is static, it is invoked as exactly the method on the declaring class.
- Control transfers to the underlying method. If the method completes abruptly by throwing an exception, the exception is placed in an `InvocationTargetException` and thrown in turn to the caller of `invoke()`.
- If the method completes normally, the value it returns is returned to the caller of `invoke()`; if the value has a primitive type, it is first appropriately wrapped in an object. If the underlying method return type is `void`, the invocation returns `null`.

The class java.lang.reflect.Constructor

```
package java.lang.reflect;

public final class Constructor implements Member
```

`Constructor` provides information about, and access to, a single constructor of a 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 `Class.getConstructor()`, `Class.getConstructors()`, and `Class.getDeclaredConstructors()`.

`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, as a `String`. This is the same as the fully-qualified name of its declaring class.

getModifiers

```
public int getModifiers()
```

Returns the access modifiers for the constructor represented by this `Constructor` object, as an integer. The `Modifier` class should be used to decode the access modifiers.

See *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 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 `Constructors` are the same if they were declared by the same class and have the same formal parameter types.

This method overrides the `equals()` method in class `Object`.

hashCode

```
public int hashCode()
```

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

This method overrides the `hashCode()` method in class `Object`.

toString

```
public String toString()
```

Returns a `String` object describing the underlying constructor represented by this `Constructor` object. The string is formatted as the constructor access modifiers, if any, followed by the fully-qualified name of class declaring the constructor, followed by a parenthesized, comma-separated list 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 thrown exception types. For example:

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

The only possible access 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 in 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`.
- 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 value to a value of the primitive type. If this attempt fails, the creation throws an `IllegalArgumentException`.
  - If, after possible unwrapping, the parameter value cannot be converted to the corresponding formal parameter type by an identity or widening conversion, the creation throws an `IllegalArgumentException`.
- Control transfers to the underlying constructor to initialize the new instance. If the constructor completes abruptly by throwing an exception, the exception is placed in an `InvocationTargetException` and thrown in turn to the caller of `newInstance()`.
- If the constructor completes normally, returns the newly created and initialized instance.

The class java.lang.reflect.Array

```
package java.lang.reflect;

public final class Array
```

The `Array` class provides static methods to dynamically create and access Java arrays.

Methods

newInstance

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

Creates a new array with the specified component type and length, as if by the equivalent array creation expression, namely:

```
new componentType[length]
```

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

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

newInstance

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

Creates a new array with the specified component type and dimensions, as if by the equivalent array creation expression, namely:

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

The method throws a `NullPointerException` if the specified `componentType` 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 components in the specified `dimension` argument is negative.

getLength

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

Returns the length of the specified array object, as an `int`. If the object argument is not an array, the method throws an `IllegalArgumentException`.

get

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

Returns the value of the indexed component of 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. Returns the possibly wrapped value.

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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

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

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

getByte

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

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

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

getChar

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

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

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

getShort

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

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

If the 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
IllegalArgumentException, ArrayIndexOutOfBoundsException,
NullPointerException
```

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

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

getLong

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

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

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

getFloat

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

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

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

getDouble

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

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

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

set

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

Sets the value of 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 component type is primitive, an unwrapping conversion is attempted to convert the new value to a primitive value. If this attempt fails, the method throws an `IllegalArgumentException`.
- If, after possible unwrapping, the new value cannot be converted to the array component type by an identity or 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 avoid the unwrapping conversion for the new value. They are described below.

setBoolean

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

Sets the value of the indexed component 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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Sets the value of the indexed component 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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Sets the value of the indexed component 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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Sets the value of 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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Sets the value of 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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Sets the value of 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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Sets the value of 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 IllegalArgumentException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Sets the value of 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
```

The `Modifier` class provides static methods and constants to decode class and member access modifier flags encoded in an integer.

Fields

NOTE: The values for the integer access 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 modifier `mod` includes the `public` specifier

isPrivate

```
public static boolean isPrivate(int mod)
```

Returns `true` if the modifier `mod` includes the `private` specifier.

isProtected

```
public static boolean isProtected(int mod)
```

Returns `true` if the modifier `mod` includes the `protected` flag.

isStatic

```
public static boolean isStatic(int mod)
```

Returns `true` if the modifier `mod` includes the `static` specifier.

isFinal

```
public static boolean isFinal(int mod)
```

Returns `true` if the modifier `mod` includes the `final` specifier.

isSynchronized

```
public static boolean isSynchronized(int mod)
```

Returns `true` if the modifier `mod` includes the `synchronized` specifier.

isVolatile

```
public static boolean isVolatile(int mod)
```

Returns `true` if the modifier `mod` includes the `volatile` specifier.

isTransient

```
public static boolean isTransient(int mod)
```

Returns `true` if the modifier `mod` includes the `transient` specifier.

isNative

```
public static boolean isNative(int mod)
```

Returns `true` if the modifier `mod` includes the `native` specifier.

isInterface

```
public static boolean isInterface(int mod)
```

Returns true if the modifier `mod` includes the `interface` specifier.

isAbstract

```
public static boolean isAbstract(int mod)
```

Returns true if the modifier `mod` includes the `abstract` specifier.

toString

```
public static String toString(int mod)
```

Returns a `String` naming the access modifiers in the specified integer argument. For example:

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

The modifier names are return 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 Roger Riggs, Frank Yellin and Benjamin Renaud for their comments.

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