Module java.base
Package java.lang

Class Class<T>

java.lang.Object
    java.lang.Class<T>

Type Parameters:
T - the type of the class modeled by this Class object. For example, the type of String.class is Class<String>. Use Class<?> if the class being modeled is unknown.

All Implemented Interfaces:
Serializable, Constable, TypeDescriptor, TypeDescriptor.OfField<Class<?>>, AnnotatedElement, GenericDeclaration, Type

```
public final class Class<T>
extends Object
implements Serializable, GenericDeclaration, Type, AnnotatedElement, TypeDescriptor.OfField<Class<?>>, Constable
```

Instances of the class Class represent classes and interfaces in a running Java application. An enum class and a record class are kinds of class; an annotation interface is a kind of interface. 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. The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword void are also represented as Class objects.

Class has no public constructor. Instead a Class object is constructed automatically by the Java Virtual Machine when a class is derived from the bytes of a class file through the invocation of one of the following methods:

- ClassLoader::defineClass
- java.lang.invoke.MethodHandles.Lookup::defineClass
- java.lang.invoke.MethodHandles.Lookup::defineHiddenClass

The methods of class Class expose many characteristics of a class or interface. Most characteristics are derived from the class file that the class loader passed to the Java Virtual Machine or from the class file passed to Lookup::defineClass or Lookup::defineHiddenClass. A few characteristics are determined by the class loading environment at run time, such as the module returned by getModule().

The following example uses a Class object to print the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The class of " + obj +
                       " is " + obj.getClass().getName());
}
```

It is also possible to get the Class object for a named class or interface (or for void) using a *class literal*. For example:

```
System.out.println("The name of class Foo is: "+Foo.class.getName());
```

Some methods of class Class expose whether the declaration of a class or interface in Java source code was *enclosed* within another declaration. Other methods describe how a class or interface is situated in a *nest*. A nest is a set of classes and interfaces, in the same run-time package, that allow mutual access to their private members. The classes and interfaces are known as *nestmates*. One nestmate acts as the *nest host*, and enumerates the other nestmates which belong to the nest; each of them in turn records it as the nest host. The classes and interfaces which belong to a nest, including its host, are determined when class files are generated, for example, a Java compiler will typically record a top-level class as the host of a nest where the other members are the classes and interfaces whose declarations are enclosed within the top-level class declaration.

Hidden Classes

A class or interface created by the invocation of Lookup::defineHiddenClass is a *hidden* class or interface. All kinds of class, including enum classes and record classes, may be hidden classes; all kinds of interface, including annotation interfaces, may be hidden interfaces. The name of a hidden class or interface is not a binary name, which means the following:

- A hidden class or interface cannot be referenced by the constant pools of other classes and interfaces.
- A hidden class or interface cannot be described in *nominal form* by Class::describeConstable, ClassDesc::of, or ClassDesc::ofDescriptor.
- A hidden class or interface cannot be discovered by Class::forName or ClassLoader::loadClass.

A hidden class or interface is never an array class, but may be the element type of an array. In all other respects, the fact that a class or interface is hidden has no bearing on the characteristics exposed by the methods of class Class.

Unnamed ClassesImplicit Classes

A class file representing an unnamed class[PREVIEW] is generated by a Java compiler from a source file for an unnamed class. The Class object representing an unnamed class is top-level, synthetic, and final. While an unnamed class does *not* have a name in its Java source form, several of the name-related methods of java.lang.Class do return non-null and non-empty results for the Class object representing an unnamed class.

Conventionally, a Java compiler, starting from a source file for an unnamed classimplicit class, say HelloWorld.java, creates a similarly-named class file, HelloWorld.class, where the class stored in that class file is named "HelloWorld", matching the base names of the source and class files. For the Class object of an unnamed classimplicit class HelloWorld, the methods to get the name and type name return results equal to "HelloWorld". The simple name of such an unnamed class is the empty string and the canonical name is null. implicit class is "HelloWorld" and the canonical name is "HelloWorld".

See *Java Language Specification*:
15.8.2 Class Literals

Since:
1.0

See Also:
ClassLoader.defineClass(byte[], int, int),
Serialized Form

Nested Class Summary

Nested classes/interfaces declared in interface java.lang.invoke.TypeDescriptor

TypeDescriptor.OfField<F extends TypeDescriptor.OfField<F>>, TypeDescriptor.OfMethod<F extends TypeDescriptor.OfField<F>,M extends TypeDescriptor.OfMethod<F,M>>

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method	Description
Set<AccessFlag>	accessFlags()	Returns an unmodifiable set of the access flags for this class, possibly empty.
Class<?>	arrayType()	Returns a Class for an array type whose component type is described by this Class.
<U> Class<? extends U>	asSubclass(Class<U> clazz)	Casts this Class object to represent a subclass of the class represented by the specified class object.
T	cast(Object obj)	Casts an object to the class or interface represented by this Class object.
Class<?>	componentType()	Returns the component type of this Class, if it describes an array type, or null otherwise.
Optional<ClassDesc>	describeConstable()	Returns a nominal descriptor for this instance, if one can be constructed, or an empty Optional if one cannot be.
String	descriptorString()	Returns the descriptor string of the entity (class, interface, array class, primitive type, or void) represented by this Class object.
boolean	desiredAssertionStatus()	Returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked.
static Class<?>	forName(Module module, String name)	Returns the Class with the given binary name in the given module.
static Class<?>	forName(String className)	Returns the Class object associated with the class or interface with the given string name.
static Class<?>	forName(String name, boolean initialize, ClassLoader loader)	Returns the Class object associated with the class or interface with the given string name, using the given class loader.
static Class<?>	forPrimitiveName(String primitiveName)	Returns the Class object associated with the primitive type of the given name.
AnnotatedType[]	getAnnotatedInterfaces()	Returns an array of AnnotatedType objects that represent the use of types to specify superinterfaces of the entity represented by this Class object.
AnnotatedType	getAnnotatedSuperclass()	Returns an AnnotatedType object that represents the use of a type to specify the superclass of the entity represented by this Class object.
<A extends Annotation> A	getAnnotation(Class<A> annotationClass)	Returns this element's annotation for the specified type if such an annotation is *present*, else null.
Annotation[]	getAnnotations()	Returns annotations that are *present* on this element.
<A extends Annotation> A[]	getAnnotationsByType(Class<A> annotationClass)	Returns annotations that are *associated* with this element.
String	getCanonicalName()	Returns the canonical name of the underlying class as defined by *The Java Language Specification*.
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.
ClassLoader	getClassLoader()	Returns the class loader for the class.
Class<?>	getComponentType()	Returns the Class representing the component type of an array.
Constructor<T>	getConstructor(Class<?>... parameterTypes)	Returns a Constructor object that reflects the specified public constructor of the class represented by this Class object.
Constructor<?>[]	getConstructors()	Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object.
<A extends Annotation> A	getDeclaredAnnotation(Class<A> annotationClass)	Returns this element's annotation for the specified type if such an annotation is *directly present*, else null.
Annotation[]	getDeclaredAnnotations()	Returns annotations that are *directly present* on this element.
<A extends Annotation> A[]	getDeclaredAnnotationsByType(Class<A> annotationClass)	Returns this element's annotation(s) for the specified type if such annotations are either *directly present* or *indirectly present*.
Class<?>[]	getDeclaredClasses()	Returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object.
Constructor<T>	getDeclaredConstructor(Class<?>... parameterTypes)	Returns a Constructor object that reflects the specified constructor of the class represented by this Class object.
Constructor<?>[]	getDeclaredConstructors()	Returns an array of Constructor objects reflecting all the constructors implicitly or explicitly declared by the class represented by this Class object.
Field	getDeclaredField(String name)	Returns a Field object that reflects the specified declared field of the class or interface represented by this Class object.
Field[]	getDeclaredFields()	Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
Method	getDeclaredMethod(String name, Class<?>... parameterTypes)	Returns a Method object that reflects the specified declared method of the class or interface represented by this Class object.
Method[]	getDeclaredMethods()	Returns an array containing Method objects reflecting all the declared methods of the class or interface represented by this Class object, including public, protected, default (package) access, and private methods, but excluding inherited methods.
Class<?>	getDeclaringClass()	If the class or interface represented by this Class object is a member of another class, returns the Class object representing the class in which it was declared.
Class<?>	getEnclosingClass()	Returns the immediately enclosing class of the underlying class.
Constructor<?>	getEnclosingConstructor()	If this Class object represents a local or anonymous class within a constructor, returns a Constructor object representing the immediately enclosing constructor of the underlying class.
Method	getEnclosingMethod()	If this Class object represents a local or anonymous class within a method, returns a Method object representing the immediately enclosing method of the underlying class.
T[]	getEnumConstants()	Returns the elements of this enum class or null if this Class object does not represent an enum class.
Field	getField(String name)	Returns a Field object that reflects the specified public member field of the class or interface represented by this Class object.
Field[]	getFields()	Returns an array containing Field objects reflecting all the accessible public fields of the class or interface represented by this Class object.
Type[]	getGenericInterfaces()	Returns the Types representing the interfaces directly implemented by the class or interface represented by this Class object.
Type	getGenericSuperclass()	Returns the Type representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class object.
Class<?>[]	getInterfaces()	Returns the interfaces directly implemented by the class or interface represented by this Class object.
Method	getMethod(String name, Class<?>... parameterTypes)	Returns a Method object that reflects the specified public member method of the class or interface represented by this Class object.
Method[]	getMethods()	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 those inherited from superclasses and superinterfaces.
int	getModifiers()	Returns the Java language modifiers for this class or interface, encoded in an integer.
Module	getModule()	Returns the module that this class or interface is a member of.
String	getName()	Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object.
Class<?>	getNestHost()	Returns the nest host of the nest to which the class or interface represented by this Class object belongs.
Class<?>[]	getNestMembers()	Returns an array containing Class objects representing all the classes and interfaces that are members of the nest to which the class or interface represented by this Class object belongs.
Package	getPackage()	Gets the package of this class.
String	getPackageName()	Returns the fully qualified package name.
Class<?>[]	getPermittedSubclasses()	Returns an array containing Class objects representing the direct subinterfaces or subclasses permitted to extend or implement this class or interface if it is sealed.
ProtectionDomain	getProtectionDomain()	Returns the ProtectionDomain of this class.
RecordComponent[]	getRecordComponents()	Returns an array of RecordComponent objects representing all the record components of this record class, or null if this class is not a record class.
URL	getResource(String name)	Finds a resource with a given name.
InputStream	getResourceAsStream(String name)	Finds a resource with a given name.
Object[]	getSigners()	Gets the signers of this class.
String	getSimpleName()	Returns the simple name of the underlying class as given in the source code.
Class<? super T>	getSuperclass()	Returns the Class representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class.
String	getTypeName()	Return an informative string for the name of this class or interface.
TypeVariable<Class<T>>[]	getTypeParameters()	Returns an array of TypeVariable objects that represent the type variables declared by the generic declaration represented by this GenericDeclaration object, in declaration order.
boolean	isAnnotation()	Returns true if this Class object represents an annotation interface.
boolean	isAnnotationPresent(Class<? extends Annotation> annotationClass)	Returns true if an annotation for the specified type is *present* on this element, else false.
boolean	isAnonymousClass()	Returns true if and only if the underlying class is an anonymous class.
boolean	isArray()	Determines if this Class object represents an array class.
boolean	isAssignableFrom(Class<?> cls)	Determines if 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.
boolean	isEnum()	Returns true if and only if this class was declared as an enum in the source code.
boolean	isHidden()	Returns true if and only if the underlying class is a hidden class.
boolean	isInstance(Object obj)	Determines if the specified Object is assignment-compatible with the object represented by this Class.
boolean	isInterface()	Determines if this Class object represents an interface type.
boolean	isLocalClass()	Returns true if and only if the underlying class is a local class.
boolean	isMemberClass()	Returns true if and only if the underlying class is a member class.
boolean	isNestmateOf(Class<?> c)	Determines if the given Class is a nestmate of the class or interface represented by this Class object.
boolean	isPrimitive()	Determines if this Class object represents a primitive type or void.
boolean	isRecord()	Returns true if and only if this class is a record class.
boolean	isSealed()	Returns true if and only if this Class object represents a sealed class or interface.
boolean	isSynthetic()	Returns true if and only if this class has the synthetic modifier bit set.
boolean	isUnnamedClass()	**Preview.** Returns true if and only if the underlying class is an unnamed class.
T	newInstance()	**Deprecated.** This method propagates any exception thrown by the nullary constructor, including a checked exception.
String	toGenericString()	Returns a string describing this Class, including information about modifiers and type parameters.
String	toString()	Converts the object to a string.

Methods declared in class java.lang.Object

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

Method Details

toString

```
public String toString()
```

Converts the object to a string. The string representation is the string "class" or "interface", followed by a space, and then by the name of the class in the format returned by getName. If this Class object represents a primitive type, this method returns the name of the primitive type. If this Class object represents void this method returns "void". If this Class object represents an array type, this method returns "class " followed by getName.

Overrides:
toString in class Object

Returns:
a string representation of this Class object.

toGenericString

```
public String toGenericString()
```

Returns a string describing this Class, including information about modifiers and type parameters. The string is formatted as a list of type modifiers, if any, followed by the kind of type (empty string for primitive types and class, enum, interface, @interface, or record as appropriate), followed by the type's name, followed by an angle-bracketed comma-separated list of the type's type parameters, if any, including informative bounds on the type parameters, if any. A space is used to separate modifiers from one another and to separate any modifiers from the kind of type. The modifiers occur in canonical order. If there are no type parameters, the type parameter list is elided. For an array type, the string starts with the type name, followed by an angle-bracketed comma-separated list of the type's type parameters, if any, followed by a sequence of [] characters, one set of brackets per dimension of the array.

Note that since information about the runtime representation of a type is being generated, modifiers not present on the originating source code or illegal on the originating source code may be present.

Returns:
a string describing this Class, including information about modifiers and type parameters

Since:
1.8

forName

```
public static Class<?> forName(String className)
                        throws ClassNotFoundException
```

Returns the Class object associated with the class or interface with the given string name. Invoking this method is equivalent to:

```
Class.forName(className, true, currentLoader)
```

where currentLoader denotes the defining class loader of the current class.

For example, the following code fragment returns the runtime Class object for the class named java.lang.Thread:

```
Class<?> t = Class.forName("java.lang.Thread");
```

A call to forName("X") causes the class named X to be initialized.

In cases where this method is called from a context where there is no caller frame on the stack (e.g. when called directly from a JNI attached thread), the system class loader is used.

Parameters:
className - the binary name of the class or the string representing an array type

Returns:
the Class object for the class with the specified name.

Throws:
LinkageError - if the linkage fails
ExceptionInInitializerError - if the initialization provoked by this method fails
ClassNotFoundException - if the class cannot be located

See *Java Language Specification*:
12.2 Loading of Classes and Interfaces
12.3 Linking of Classes and Interfaces
12.4 Initialization of Classes and Interfaces

forName

```
public static Class<?> forName(String name,
                               boolean initialize,
                               ClassLoader loader)
                        throws ClassNotFoundException
```

Returns the Class object associated with the class or interface with the given string name, using the given class loader. Given the binary name for a class or interface, this method attempts to locate and load the class or interface. The specified class loader is used to load the class or interface. If the parameter loader is null, the class is loaded through the bootstrap class loader. The class is initialized only if the initialize parameter is true and if it has not been initialized earlier.

This method cannot be used to obtain any of the Class objects representing primitive types or void, hidden classes or interfaces, or array classes whose element type is a hidden class or interface. If name denotes a primitive type or void, for example I, an attempt will be made to locate a user-defined class in the unnamed package whose name is I instead. To obtain a Class object for a named primitive type such as int or long use forPrimitiveName(String).

To obtain the Class object associated with an array class, the name consists of one or more '[' representing the depth of the array nesting, followed by the element type as encoded in the table specified in Class.getName().

Examples:

```
Class<?> threadClass = Class.forName("java.lang.Thread", false, currentLoader);
Class<?> stringArrayClass = Class.forName("[Ljava.lang.String;", false, currentLoader);
Class<?> intArrayClass = Class.forName("[[[I", false, currentLoader);   // Class of int[][][]
Class<?> nestedClass = Class.forName("java.lang.Character$UnicodeBlock", false, currentLoader);
Class<?> fooClass = Class.forName("Foo", true, currentLoader);
```

A call to getName() on the Class object returned from forName(*N*) returns *N*.

A call to forName("[L*N*;") causes the element type named *N* to be loaded but not initialized regardless of the value of the initialize parameter.

API Note:
This method throws errors related to loading, linking or initializing as specified in Sections 12.2, 12.3, and 12.4 of *The Java Language Specification*. In addition, this method does not check whether the requested class is accessible to its caller.

Parameters:
name - the binary name of the class or the string representing an array class
initialize - if true the class will be initialized (which implies linking). See Section 12.4 of *The Java Language Specification*.
loader - class loader from which the class must be loaded

Returns:
class object representing the desired class

Throws:
LinkageError - if the linkage fails
ExceptionInInitializerError - if the initialization provoked by this method fails
ClassNotFoundException - if the class cannot be located by the specified class loader
SecurityException - if a security manager is present, and the loader is null, and the caller's class loader is not null, and the caller does not have the RuntimePermission("getClassLoader")

See *Java Language Specification*:
12.2 Loading of Classes and Interfaces
12.3 Linking of Classes and Interfaces
12.4 Initialization of Classes and Interfaces
13.1 The Form of a Binary

Since:
1.2

See Also:
forName(String), ClassLoader

forName

```
public static Class<?> forName(Module module,
                               String name)
```

Returns the Class with the given binary name in the given module.

This method attempts to locate and load the class or interface. It does not link the class, and does not run the class initializer. If the class is not found, this method returns null.

If the class loader of the given module defines other modules and the given name is a class defined in a different module, this method returns null after the class is loaded.

This method does not check whether the requested class is accessible to its caller.

API Note:
This method does not support loading of array types, unlike forName(String, boolean, ClassLoader). The class name must be a binary name. This method returns null on failure rather than throwing a ClassNotFoundException, as is done by the forName(String, boolean, ClassLoader) method. The security check is a stack-based permission check if the caller loads a class in another module.

Parameters:
module - A module
name - The binary name of the class

Returns:
Class object of the given name defined in the given module; null if not found.

Throws:
NullPointerException - if the given module or name is null
LinkageError - if the linkage fails
SecurityException -
- if the caller is not the specified module and RuntimePermission("getClassLoader") permission is denied; or
- access to the module content is denied. For example, permission check will be performed when a class loader calls ModuleReader.open(String) to read the bytes of a class file in a module.

See *Java Language Specification*:
12.2 Loading of Classes and Interfaces
12.3 Linking of Classes and Interfaces

Since:
9

forPrimitiveName

```
public static Class<?> forPrimitiveName(String primitiveName)
```

Returns the Class object associated with the primitive type of the given name. If the argument is not the name of a primitive type, null is returned.

Parameters:
primitiveName - the name of the primitive type to find

Returns:
the Class object associated with the primitive type of the given name

Throws:
NullPointerException - if the argument is null

See *Java Language Specification*:
4.2 Primitive Types and Values
15.8.2 Class Literals

Since:
22

newInstance

```
@Deprecated(since="9")
public T newInstance()
             throws InstantiationException,
                    IllegalAccessException
```

Deprecated.
This method propagates any exception thrown by the nullary constructor, including a checked exception. Use of this method effectively bypasses the compile-time exception checking that would otherwise be performed by the compiler. The Constructor.newInstance method avoids this problem by wrapping any exception thrown by the constructor in a (checked) InvocationTargetException.

The call

```
clazz.newInstance()
```

can be replaced by

```
clazz.getDeclaredConstructor().newInstance()
```

The latter sequence of calls is inferred to be able to throw the additional exception types InvocationTargetException and NoSuchMethodException. Both of these exception types are subclasses of ReflectiveOperationException.

Creates a new instance of the class represented by this Class object. The class is instantiated as if by a new expression with an empty argument list. The class is initialized if it has not already been initialized.

Returns:
a newly allocated instance of the class represented by this object.

Throws:
IllegalAccessException - if the class or its nullary constructor is not accessible.
InstantiationException - if this Class represents an abstract class, an interface, an array class, a primitive type, or void; or if the class has no nullary constructor; or if the instantiation fails for some other reason.
ExceptionInInitializerError - if the initialization provoked by this method fails.
SecurityException - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of s.checkPackageAccess() denies access to the package of this class.

isInstance

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

Determines if the specified Object is assignment-compatible with the object represented by this Class. 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.

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

Parameters:
obj - the object to check

Returns:
true if obj is an instance of this class

Since:
1.1

isAssignableFrom

```
public boolean isAssignableFrom(Class<?> cls)
```

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

Specifically, 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.

Parameters:
cls - the Class object to be checked

Returns:
the boolean value indicating whether objects of the type cls can be assigned to objects of this class

Throws:
NullPointerException - if the specified Class parameter is null.

Since:
1.1

isInterface

```
public boolean isInterface()
```

Determines if this Class object represents an interface type.

Returns:
true if this Class object represents an interface; false otherwise.

isArray

```
public boolean isArray()
```

Determines if this Class object represents an array class.

Specified by:
isArray in interface TypeDescriptor.OfField<T>

Returns:
true if this Class object represents an array class; false otherwise.

Since:
1.1

isPrimitive

```
public boolean isPrimitive()
```

Determines if this Class object represents a primitive type or void.

There are nine predefined Class objects to represent the eight primitive types and void. These are created by the Java Virtual Machine, and have the same names as the primitive types that they represent, namely boolean, byte, char, short, int, long, float, and double.

No other class objects are considered primitive.

Specified by:
isPrimitive in interface TypeDescriptor.OfField<T>

API Note:
A Class object represented by a primitive type can be accessed via the TYPE public static final variables defined in the primitive wrapper classes such as Integer.TYPE. In the Java programming language, the objects may be referred to by a class literal expression such as int.class. The Class object for void can be expressed as void.class or Void.TYPE.

Returns:
true if and only if this class represents a primitive type

See *Java Language Specification*:
15.8.2 Class Literals

Since:
1.1

See Also:
Boolean.TYPE, Character.TYPE, Byte.TYPE, Short.TYPE, Integer.TYPE, Long.TYPE, Float.TYPE, Double.TYPE, Void.TYPE

isAnnotation

```
public boolean isAnnotation()
```

Returns true if this Class object represents an annotation interface. Note that if this method returns true, isInterface() would also return true, as all annotation interfaces are also interfaces.

Returns:
true if this Class object represents an annotation interface; false otherwise

Since:
1.5

isSynthetic

```
public boolean isSynthetic()
```

Returns true if and only if this class has the synthetic modifier bit set.

Returns:
true if and only if this class has the synthetic modifier bit set

See *Java Language Specification*:
13.1 The Form of a Binary

See *Java Virtual Machine Specification*:
4.1 The ClassFile Structure

Since:
1.5

See Also:
Java programming language and JVM modeling in core reflection

getName

```
public String getName()
```

Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object.

If this Class object represents a class or interface, not an array class, then:

- If the class or interface is not hidden, then the binary name of the class or interface is returned.
- If the class or interface is hidden, then the result is a string of the form: N + '/' + <suffix> where N is the binary name indicated by the class file passed to Lookup::defineHiddenClass, and <suffix> is an unqualified name.

If this Class object represents an array class, then the result is a string consisting of one or more '[' characters representing the depth of the array nesting, followed by the element type as encoded using the following table:

Element Type	Encoding
boolean	Z
byte	B
char	C
class or interface with binary name *N*	L*N*;
double	D
float	F
int	I
long	J
short	S

If this Class object represents a primitive type or void, then the result is a string with the same spelling as the Java language keyword which corresponds to the primitive type or void.

Examples:

```
String.class.getName()
    returns "java.lang.String"
Character.UnicodeBlock.class.getName()
    returns "java.lang.Character$UnicodeBlock"
byte.class.getName()
    returns "byte"
(new Object[3]).getClass().getName()
    returns "[Ljava.lang.Object;"
(new int[3][4][5][6][7][8][9]).getClass().getName()
    returns "[[[[[[[I"
```

API Note:
Distinct class objects can have the same name but different class loaders.

Returns:
the name of the class, interface, or other entity represented by this Class object.

See *Java Language Specification*:
13.1 The Form of a Binary

getClassLoader

```
public ClassLoader getClassLoader()
```

Returns the class loader for the class. Some implementations may use null to represent the bootstrap class loader. This method will return null in such implementations if this class was loaded by the bootstrap class loader.

If this Class object represents a primitive type or void, null is returned.

Returns:
the class loader that loaded the class or interface represented by this Class object.

Throws:
SecurityException - if a security manager is present, and the caller's class loader is not null and is not the same as or an ancestor of the class loader for the class whose class loader is requested, and the caller does not have the RuntimePermission("getClassLoader")

See Also:
ClassLoader,
SecurityManager.checkPermission(java.security.Permission),
RuntimePermission

getModule

```
public Module getModule()
```

Returns the module that this class or interface is a member of. If this class represents an array type then this method returns the Module for the element type. If this class represents a primitive type or void, then the Module object for the java.base module is returned. If this class is in an unnamed module then the unnamed Module of the class loader for this class is returned.

Returns:
the module that this class or interface is a member of

Since:
9

getTypeParameters

```
public TypeVariable<Class<T>>[] getTypeParameters()
```

Returns an array of TypeVariable objects that represent the type variables declared by the generic declaration represented by this GenericDeclaration object, in declaration order. Returns an array of length 0 if the underlying generic declaration declares no type variables.

Specified by:
getTypeParameters in interface GenericDeclaration

Returns:
an array of TypeVariable objects that represent the type variables declared by this generic declaration

Throws:
GenericSignatureFormatError - if the generic signature of this generic declaration does not conform to the format specified in section 4.7.9 of *The Java Virtual Machine Specification*

Since:
1.5

getSuperclass

```
public Class<? super T> getSuperclass()
```

Returns the Class representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class. If this Class represents either the Object class, an interface, a primitive type, or void, then null is returned. If this Class object represents an array class then the Class object representing the Object class is returned.

Returns:
the direct superclass of the class represented by this Class object

getGenericSuperclass

```
public Type getGenericSuperclass()
```

Returns the Type representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class object.

If the superclass is a parameterized type, the Type object returned must accurately reflect the actual type arguments used in the source code. The parameterized type representing the superclass is created if it had not been created before. See the declaration of ParameterizedType for the semantics of the creation process for parameterized types. If this Class object represents either the Object class, an interface, a primitive type, or void, then null is returned. If this Class object represents an array class then the Class object representing the Object class is returned.

Returns:
the direct superclass of the class represented by this Class object

Throws:
GenericSignatureFormatError - if the generic class signature does not conform to the format specified in section 4.7.9 of *The Java Virtual Machine Specification*
TypeNotPresentException - if the generic superclass refers to a non-existent type declaration
MalformedParameterizedTypeException - if the generic superclass refers to a parameterized type that cannot be instantiated for any reason

Since:
1.5

getPackage

```
public Package getPackage()
```

Gets the package of this class.

If this class represents an array type, a primitive type or void, this method returns null.

Returns:
the package of this class.

getPackageName

```
public String getPackageName()
```

Returns the fully qualified package name.

If this class is a top level class, then this method returns the fully qualified name of the package that the class is a member of, or the empty string if the class is in an unnamed package.

If this class is a member class, then this method is equivalent to invoking `getPackageName()` on the enclosing class.

If this class is a local class or an anonymous class, then this method is equivalent to invoking `getPackageName()` on the declaring class of the enclosing method or enclosing constructor.

If this class represents an array type then this method returns the package name of the element type. If this class represents a primitive type or void then the package name "`java.lang`" is returned.

Returns:
the fully qualified package name

See *Java Language Specification*:
6.7 Fully Qualified Names

Since:
9

getInterfaces

```
public Class<?>[] getInterfaces()
```

Returns the interfaces directly implemented by the class or interface represented by this `Class` object.

If this `Class` object represents a class, the return value is an array containing objects representing all interfaces directly implemented by the class. The order of the interface objects in the array corresponds to the order of the interface names in the `implements` clause of the declaration of the class represented by this `Class` object. For example, given the declaration:

```
class Shimmer implements FloorWax, DessertTopping { ... }
```

suppose the value of `s` is an instance of `Shimmer`; the value of the expression:

```
s.getClass().getInterfaces()[0]
```

is the `Class` object that represents interface `FloorWax`; and the value of:

```
s.getClass().getInterfaces()[1]
```

is the `Class` object that represents interface `DessertTopping`.

If this `Class` object represents an interface, the array contains objects representing all interfaces directly extended by the interface. The order of the interface objects in the array corresponds to the order of the interface names in the `extends` clause of the declaration of the interface represented by this `Class` object.

If this `Class` object represents a class or interface that implements no interfaces, the method returns an array of length 0.

If this `Class` object represents a primitive type or void, the method returns an array of length 0.

If this `Class` object represents an array type, the interfaces `Cloneable` and `java.io.Serializable` are returned in that order.

Returns:
an array of interfaces directly implemented by this class

getGenericInterfaces

```
public Type[] getGenericInterfaces()
```

Returns the `Type`s representing the interfaces directly implemented by the class or interface represented by this `Class` object.

If a superinterface is a parameterized type, the `Type` object returned for it must accurately reflect the actual type arguments used in the source code. The parameterized type representing each superinterface is created if it had not been created before. See the declaration of `ParameterizedType` for the semantics of the creation process for parameterized types.

If this `Class` object represents a class, the return value is an array containing objects representing all interfaces directly implemented by the class. The order of the interface objects in the array corresponds to the order of the interface names in the `implements` clause of the declaration of the class represented by this `Class` object.

If this `Class` object represents an interface, the array contains objects representing all interfaces directly extended by the interface. The order of the interface objects in the array corresponds to the order of the interface names in the `extends` clause of the declaration of the interface represented by this `Class` object.

If this `Class` object represents a class or interface that implements no interfaces, the method returns an array of length 0.

If this `Class` object represents a primitive type or void, the method returns an array of length 0.

If this `Class` object represents an array type, the interfaces `Cloneable` and `java.io.Serializable` are returned in that order.

Returns:
an array of interfaces directly implemented by this class

Throws:
`GenericSignatureFormatError` - if the generic class signature does not conform to the format specified in section 4.7.9 of *The Java Virtual Machine Specification*
`TypeNotPresentException` - if any of the generic superinterfaces refers to a non-existent type declaration
`MalformedParameterizedTypeException` - if any of the generic superinterfaces refer to a parameterized type that cannot be instantiated for any reason

Since:
1.5

getComponentType

```
public Class<?> getComponentType()
```

Returns the `Class` representing the component type of an array. If this class does not represent an array class this method returns null.

Returns:
the `Class` representing the component type of this class if this class is an array

Since:
1.1

See Also:
`Array`

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`, `static`, `abstract` and `interface`; they should be decoded using the methods of class `Modifier`.

If the underlying class is an array class:

- its `public`, `private` and `protected` modifiers are the same as those of its component type
- its `abstract` and `final` modifiers are always `true`
- its interface modifier is always `false`, even when the component type is an interface

If this `Class` object represents a primitive type or void, its `public`, `abstract`, and `final` modifiers are always `true`. For `Class` objects representing void, primitive types, and arrays, the values of other modifiers are `false` other than as specified above.

The modifier encodings are defined in section 4.1 of *The Java Virtual Machine Specification*.

Returns:
the `int` representing the modifiers for this class

See *Java Language Specification*:
8.1.1 Class Modifiers
9.1.1. Interface Modifiers

See *Java Virtual Machine Specification*:
4.1 The `ClassFile` Structure

Since:
1.1

See Also:
`Modifier`,
`accessFlags()`,
Java programming language and JVM modeling in core reflection

accessFlags

```
public Set<AccessFlag> accessFlags()
```

Returns an unmodifiable set of the access flags for this class, possibly empty.

If the underlying class is an array class:

- its `PUBLIC`, `PRIVATE` and `PROTECTED` access flags are the same as those of its component type
- its `ABSTRACT` and `FINAL` flags are present
- its `INTERFACE` flag is absent, even when the component type is an interface

If this `Class` object represents a primitive type or void, the flags are `PUBLIC`, `ABSTRACT`, and `FINAL`. For `Class` objects representing void, primitive types, and arrays, access flags are absent other than as specified above.

Returns:
an unmodifiable set of the access flags for this class, possibly empty

See *Java Virtual Machine Specification*:
4.1 The ClassFile Structure
4.7.6 The InnerClasses Attribute

Since:
20

See Also:
`getModifiers()`

getSigners

```
public Object[] getSigners()
```

Gets the signers of this class.

Returns:
the signers of this class, or null if there are no signers. In particular, this method returns null if this `Class` object represents a primitive type or void.

Since:
1.1

getEnclosingMethod

```
public Method getEnclosingMethod()
                          throws SecurityException
```

If this `Class` object represents a local or anonymous class within a method, returns a `Method` object representing the immediately enclosing method of the underlying class. Returns `null` otherwise. In particular, this method returns `null` if the underlying class is a local or anonymous class immediately enclosed by a class or interface declaration, instance initializer or static initializer.

Returns:
the immediately enclosing method of the underlying class, if that class is a local or anonymous class; otherwise `null`.

Throws:
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of the enclosing class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the methods within the enclosing class
- the caller's class loader is not the same as or an ancestor of the class loader for the enclosing class and invocation of `s.checkPackageAccess()` denies access to the package of the enclosing class

Since:
1.5

getEnclosingConstructor

```
public Constructor<?> getEnclosingConstructor()
                                       throws SecurityException
```

If this `Class` object represents a local or anonymous class within a constructor, returns a `Constructor` object representing the immediately enclosing constructor of the underlying class. Returns `null` otherwise. In particular, this method returns `null` if the underlying class is a local or anonymous class immediately enclosed by a class or interface declaration, instance initializer or static initializer.

Returns:
the immediately enclosing constructor of the underlying class, if that class is a local or anonymous class; otherwise `null`.

Throws:
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of the enclosing class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the constructors within the enclosing class
- the caller's class loader is not the same as or an ancestor of the class loader for the enclosing class and invocation of `s.checkPackageAccess()` denies access to the package of the enclosing class

Since:
1.5

getDeclaringClass

```
public Class<?> getDeclaringClass()
                           throws SecurityException
```

If the class or interface represented by this `Class` object is a member of another class, returns the `Class` object representing the class in which it was declared. This method returns null if this class or interface is not a member of any other class. If this `Class` object represents an array class, a primitive type, or void, then this method returns null.

Returns:
the declaring class for this class

Throws:
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the declaring class and invocation of `s.checkPackageAccess()` denies access to the package of the declaring class

Since:
1.1

getEnclosingClass

```
public Class<?> getEnclosingClass()
                           throws SecurityException
```

Returns the immediately enclosing class of the underlying class. If the underlying class is a top level class this method returns `null`.

Returns:
the immediately enclosing class of the underlying class

Throws:
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the enclosing class and invocation of `s.checkPackageAccess()` denies access to the package of the enclosing class

Since:
1.5

getSimpleName

```
public String getSimpleName()
```

Returns the simple name of the underlying class as given in the source code. An empty string is returned if the underlying class is anonymous or unnamed^PREVIEW. A synthetic class, one not present in source code, can have a non-empty name including special characters, such as "`$`".

The simple name of an array class is the simple name of the component type with "[]" appended. In particular the simple name of an array class whose component type is anonymous is "[]".

Returns:
the simple name of the underlying class

Since:
1.5

getTypeName

```
public String getTypeName()
```

Return an informative string for the name of this class or interface.

Specified by:
`getTypeName` in interface `Type`

Returns:
an informative string for the name of this class or interface

Since:
1.8

getCanonicalName

```
public String getCanonicalName()
```

Returns the canonical name of the underlying class as defined by *The Java Language Specification*. Returns `null` if the underlying class does not have a canonical name. Classes without canonical names include:

- a local class
- a anonymous class
- an unnamed class^PREVIEW
- a hidden class
- an array whose component type does not have a canonical name

The canonical name for a primitive class is the keyword for the corresponding primitive type (`byte`, `short`, `char`, `int`, and so on).

An array type has a canonical name if and only if its component type has a canonical name. When an array type has a canonical name, it is equal to the canonical name of the component type followed by "`[]`".

Returns:
the canonical name of the underlying class if it exists, and `null` otherwise.

See *Java Language Specification*:
6.7 Fully Qualified Names and Canonical Names

Since:
1.5

isUnnamedClass

```
public boolean isUnnamedClass()
```

***isUnnamedClass* is a reflective preview API of the Java platform.**
Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.

Returns `true` if and only if the underlying class is an unnamed class.

API Note:
An unnamed class is not an anonymous class.

Returns:
`true` if and only if the underlying class is an unnamed class

See *Java Language Specification*:
7.3 Compilation Units

Since:
21

isAnonymousClass

```
public boolean isAnonymousClass()
```

Returns `true` if and only if the underlying class is an anonymous class.

API Note:
An anonymous class is not a hidden class. An anonymous class is not an unnamed class^PREVIEW.

Returns:
`true` if and only if this class is an anonymous class.

See *Java Language Specification*:
15.9.5 Anonymous Class Declarations

Since:
1.5

isLocalClass

```
public boolean isLocalClass()
```

Returns `true` if and only if the underlying class is a local class.

Returns:
`true` if and only if this class is a local class.

See *Java Language Specification*:
14.3 Local Class Declarations

Since:
1.5

isMemberClass

```
public boolean isMemberClass()
```

Returns `true` if and only if the underlying class is a member class.

Returns:
`true` if and only if this class is a member class.

See *Java Language Specification*:
8.5 Member Type Declarations

Since:
1.5

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. This method returns an array of length 0 if this `Class` object has no public member classes or interfaces. This method also returns an array of length 0 if this `Class` object represents a primitive type, an array class, or void.

Returns:
the array of `Class` objects representing the public members of this class

Throws:
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class.

Since:
1.1

getFields

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

Returns an array containing `Field` objects reflecting all the accessible public fields of the class or interface represented by this `Class` object.

If this `Class` object represents a class or interface with no accessible public fields, then this method returns an array of length 0.

If this `Class` object represents a class, then this method returns the public fields of the class and of all its superclasses and superinterfaces.

If this `Class` object represents an interface, then this method returns the fields of the interface and of all its superinterfaces.

If this `Class` object represents an array type, a primitive type, or void, then this method returns an array of length 0.

The elements in the returned array are not sorted and are not in any particular order.

Returns:
the array of `Field` objects representing the public fields

Throws:
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class.

See *Java Language Specification*:
8.2 Class Members
8.3 Field Declarations

Since:
1.1

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 those inherited from superclasses and superinterfaces.

If this `Class` object represents an array type, then the returned array has a `Method` object for each of the public methods inherited by the array type from `Object`. It does not contain a `Method` object for `clone()`.

If this `Class` object represents an interface then the returned array does not contain any implicitly declared methods from `Object`. Therefore, if no methods are explicitly declared in this interface or any of its superinterfaces then the returned array has length 0. (Note that a `Class` object which represents a class always has public methods, inherited from `Object`.)

The returned array never contains methods with names "<init>" or "<clinit>".

The elements in the returned array are not sorted and are not in any particular order.

Generally, the result is computed as with the following 4 step algorithm. Let C be the class or interface represented by this `Class` object:

1. A union of methods is composed of:
   a. C's declared public instance and static methods as returned by `getDeclaredMethods()` and filtered to include only public methods.
   b. If C is a class other than `Object`, then include the result of invoking this algorithm recursively on the superclass of C.
   c. Include the results of invoking this algorithm recursively on all direct superinterfaces of C, but include only instance methods.
2. Union from step 1 is partitioned into subsets of methods with same signature (name, parameter types) and return type.
3. Within each such subset only the most specific methods are selected. Let method M be a method from a set of methods with same signature and return type. M is most specific if there is no such method N != M from the same set, such that N is more specific than M. N is more specific than M if:
   a. N is declared by a class and M is declared by an interface; or
   b. N and M are both declared by classes or both by interfaces and N's declaring type is the same as or a subtype of M's declaring type (clearly, if M's and N's declaring types are the same type, then M and N are the same method).
4. The result of this algorithm is the union of all selected methods from step 3.

API Note:
There may be more than one method with a particular name and parameter types in a class because while the Java language forbids a class to declare multiple methods with the same signature but different return types, the Java virtual machine does not. This increased flexibility in the virtual machine can be used to implement various language features. For example, covariant returns can be implemented with bridge methods; the bridge method and the overriding method would have the same signature but different return types.

Returns:
the array of `Method` objects representing the public methods of this class

Throws:
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class.

See *Java Language Specification*:
8.2 Class Members
8.4 Method Declarations

Since:
1.1

getConstructors

```
public Constructor<?>[] getConstructors()
                                 throws SecurityException
```

Returns an array containing `Constructor` objects reflecting all the public constructors of the class represented by this `Class` object. An array of length 0 is returned if the class has no public constructors, or if the class is an array class, or if the class reflects a primitive type or void.

API Note:
While this method returns an array of `Constructor<T>` objects (that is an array of constructors from this class), the return type of this method is `Constructor<?>[]` and *not* `Constructor<T>[]` as might be expected. This less informative return type is necessary since after being returned from this method, the array could be modified to hold `Constructor` objects for different classes, which would violate the type guarantees of `Constructor<T>[]`.

Returns:
the array of `Constructor` objects representing the public constructors of this class

Throws:
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class.

Since:
1.1

See Also:
`getDeclaredConstructors()`

getField

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

Returns a `Field` object that reflects the specified public member 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 determined by the algorithm that follows. Let C be the class or interface represented by this `Class` object:

1. If C declares a public field with the name specified, that is the field to be reflected.
2. If no field was found in step 1 above, this algorithm is applied recursively to each direct superinterface of C. The direct superinterfaces are searched in the order they were declared.
3. If no field was found in steps 1 and 2 above, and C has a superclass S, then this algorithm is invoked recursively upon S. If C has no superclass, then a `NoSuchFieldException` is thrown.

If this `Class` object represents an array type, then this method does not find the `length` field of the array type.

Parameters:
`name` - the field name

Returns:
the `Field` object of this class specified by `name`

Throws:
`NoSuchFieldException` - if a field with the specified name is not found.
`NullPointerException` - if `name` is `null`
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class.

See *Java Language Specification*:
8.2 Class Members
8.3 Field Declarations

Since:
1.1

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 of the desired method. The `parameterTypes` parameter is an array of `Class` objects that identify the method's formal parameter types, in declared order. If `parameterTypes` is `null`, it is treated as if it were an empty array.

If this `Class` object represents an array type, then this method finds any public method inherited by the array type from `Object` except method `clone()`.

If this `Class` object represents an interface then this method does not find any implicitly declared method from `Object`. Therefore, if no methods are explicitly declared in this interface or any of its superinterfaces, then this method does not find any method.

This method does not find any method with name "<init>" or "<clinit>".

Generally, the method to be reflected is determined by the 4 step algorithm that follows. Let C be the class or interface represented by this `Class` object:

1. A union of methods is composed of:
   a. C's declared public instance and static methods as returned by `getDeclaredMethods()` and filtered to include only public methods that match given `name` and `parameterTypes`
   b. If C is a class other than `Object`, then include the result of invoking this algorithm recursively on the superclass of C.
   c. Include the results of invoking this algorithm recursively on all direct superinterfaces of C, but include only instance methods.
2. This union is partitioned into subsets of methods with same return type (the selection of methods from step 1 also guarantees that they have the same method name and parameter types).
3. Within each such subset only the most specific methods are selected. Let method M be a method from a set of methods with same VM signature (return type, name, parameter types). M is most specific if there is no such method N != M from the same set, such that N is more specific than M. N is more specific than M if:
   a. N is declared by a class and M is declared by an interface; or
   b. N and M are both declared by classes or both by interfaces and N's declaring type is the same as or a subtype of M's declaring type (clearly, if M's and N's declaring types are the same type, then M and N are the same method).
4. The result of this algorithm is chosen arbitrarily from the methods with most specific return type among all selected methods from step 3. Let R be a return type of a method M from the set of all selected methods from step 3. M is a method with most specific return type if there is no such method N != M from the same set, having return type S != R, such that S is a subtype of R as determined by R.class.`isAssignableFrom(java.lang.Class<?>)`(S.class).

API Note:
There may be more than one method with matching name and parameter types in a class because while the Java language forbids a class to declare multiple methods with the same signature but different return types, the Java virtual machine does not. This increased flexibility in the virtual machine can be used to implement various language features. For example, covariant returns can be implemented with bridge methods; the bridge method and the overriding method would have the same signature but different return types. This method would return the overriding method as it would have a more specific return type.

Parameters:
`name` - the name of the method
`parameterTypes` - the list of parameters

Returns:
the `Method` object that matches the specified `name` and `parameterTypes`

Throws:
`NoSuchMethodException` - if a matching method is not found or if the name is "<init>" or "<clinit>".
`NullPointerException` - if `name` is `null`
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class.

See *Java Language Specification*:
8.2 Class Members
8.4 Method Declarations

Since:
1.1

getConstructor

```
public Constructor<T> 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. If this `Class` object represents an inner class declared in a non-static context, the formal parameter types include the explicit enclosing instance as the first parameter.

The constructor to reflect is the public constructor of the class represented by this `Class` object whose formal parameter types match those specified by `parameterTypes`.

Parameters:
`parameterTypes` - the parameter array

Returns:
the `Constructor` object of the public constructor that matches the specified `parameterTypes`

Throws:
`NoSuchMethodException` - if a matching constructor is not found, including when this `Class` object represents an interface, a primitive type, an array class, or void.
`SecurityException` - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class.

Since:
1.1

See Also:
`getDeclaredConstructor(Class<?>[])`

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. This method 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, an array class, or void.

Returns:
the array of `Class` objects representing all the declared members of this class

Throws:
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the declared classes within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class

See *Java Language Specification*:
8.5 Member Type Declarations

Since:
1.1

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.

If this `Class` object represents a class or interface with no declared fields, then this method returns an array of length 0.

If this `Class` object represents an array type, a primitive type, or void, then this method returns an array of length 0.

The elements in the returned array are not sorted and are not in any particular order.

Returns:
the array of `Field` objects representing all the declared fields of this class

Throws:
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the declared fields within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class

See *Java Language Specification*:
8.2 Class Members
8.3 Field Declarations

Since:
1.1

getRecordComponents

```
public RecordComponent[] getRecordComponents()
```

Returns an array of `RecordComponent` objects representing all the record components of this record class, or `null` if this class is not a record class.

The components are returned in the same order that they are declared in the record header. The array is empty if this record class has no components. If the class is not a record class, that is `isRecord()` returns `false`, then this method returns `null`. Conversely, if `isRecord()` returns `true`, then this method returns a non-null value.

API Note:
The following method can be used to find the record canonical constructor:

```
static <T extends Record> Constructor<T> getCanonicalConstructor(Class<T> cls)
    throws NoSuchMethodException {
  Class<?>[] paramTypes =
    Arrays.stream(cls.getRecordComponents())
          .map(RecordComponent::getType)
          .toArray(Class<?>[]::new);
  return cls.getDeclaredConstructor(paramTypes);
}
```

Returns:
An array of `RecordComponent` objects representing all the record components of this record class, or `null` if this class is not a record class

Throws:
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the declared methods within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class

See *Java Language Specification*:
8.10 Record Classes

Since:
16

getDeclaredMethods

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

Returns an array containing `Method` objects reflecting all the declared methods of the class or interface represented by this `Class` object, including public, protected, default (package) access, and private methods, but excluding inherited methods. The declared methods may include methods *not* in the source of the class or interface, including bridge methods and other synthetic methods added by compilers.

If this `Class` object represents a class or interface that has multiple declared methods with the same name and parameter types, but different return types, then the returned array has a `Method` object for each such method.

If this `Class` object represents a class or interface that has a class initialization method "<clinit>", then the returned array does *not* have a corresponding `Method` object.

If this `Class` object represents a class or interface with no declared methods, then the returned array has length 0.

If this `Class` object represents an array type, a primitive type, or void, then the returned array has length 0.

The elements in the returned array are not sorted and are not in any particular order.

Returns:
the array of `Method` objects representing all the declared methods of this class

Throws:
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the declared methods within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class

See *Java Language Specification*:
8.2 Class Members
8.4 Method Declarations

Since:
1.1

See Also:
Java programming language and JVM modeling in core reflection

getDeclaredConstructors

```
public Constructor<?>[] getDeclaredConstructors()
                                         throws SecurityException
```

Returns an array of `Constructor` objects reflecting all the constructors implicitly or explicitly declared by the class represented by this `Class` object. These are public, protected, default (package) access, and private constructors. The elements in the array returned are not sorted and are not in any particular order. If the class has a default constructor (JLS 8.8.9), it is included in the returned array. If a record class has a canonical constructor (JLS 8.10.4.1, 8.10.4.2), it is included in the returned array. This method returns an array of length 0 if this `Class` object represents an interface, a primitive type, an array class, or void.

Returns:
the array of `Constructor` objects representing all the declared constructors of this class

Throws:
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the declared constructors within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class

See *Java Language Specification*:
8.8 Constructor Declarations

Since:
1.1

See Also:
`getConstructors()`

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.

If this `Class` object represents an array type, then this method does not find the `length` field of the array type.

Parameters:
`name` - the name of the field

Returns:
the `Field` object for the specified field in this class

Throws:
`NoSuchFieldException` - if a field with the specified name is not found.
`NullPointerException` - if `name` is `null`
`SecurityException` - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMembers")` denies access to the declared field
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to the package of this class

See *Java Language Specification*:
8.2 Class Members
8.3 Field Declarations

Since:
1.1

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. If more than one method with the same parameter types is declared in a class, and one of these methods has a return type that is more specific than any of the others, that method is returned; otherwise one of the methods is chosen arbitrarily. If the name is "<init>" or "<clinit>" a NoSuchMethodException is raised.

If this Class object represents an array type, then this method does not find the clone() method.

Parameters:
name - the name of the method
parameterTypes - the parameter array

Returns:
the Method object for the method of this class matching the specified name and parameters

Throws:
NoSuchMethodException - if a matching method is not found.
NullPointerException - if name is null
SecurityException - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of s.checkPermission method with RuntimePermission("accessDeclaredMembers") denies access to the declared method
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of s.checkPackageAccess() denies access to the package of this class

See *Java Language Specification*:
8.2 Class Members
8.4 Method Declarations

Since:
1.1

getDeclaredConstructor

```
public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
                                      throws NoSuchMethodException,
                                             SecurityException
```

Returns a Constructor object that reflects the specified 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. If this Class object represents an inner class declared in a non-static context, the formal parameter types include the explicit enclosing instance as the first parameter.

Parameters:
parameterTypes - the parameter array

Returns:
The Constructor object for the constructor with the specified parameter list

Throws:
NoSuchMethodException - if a matching constructor is not found, including when this Class object represents an interface, a primitive type, an array class, or void.
SecurityException - If a security manager, *s*, is present and any of the following conditions is met:
- the caller's class loader is not the same as the class loader of this class and invocation of s.checkPermission method with RuntimePermission("accessDeclaredMembers") denies access to the declared constructor
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of s.checkPackageAccess() denies access to the package of this class

Since:
1.1

See Also:
getConstructor(Class<?>[])

getResourceAsStream

```
public InputStream getResourceAsStream(String name)
```

Finds a resource with a given name.

If this class is in a named Module then this method will attempt to find the resource in the module. This is done by delegating to the module's class loader findResource(String,String) method, invoking it with the module name and the absolute name of the resource. Resources in named modules are subject to the rules for encapsulation specified in the Module getResourceAsStream method and so this method returns null when the resource is a non-".class" resource in a package that is not open to the caller's module.

Otherwise, if this class is not in a named module then the rules for searching resources associated with a given class are implemented by the defining class loader of the class. This method delegates to this Class object's class loader. If this Class object was loaded by the bootstrap class loader, the method delegates to ClassLoader.getSystemResourceAsStream(java.lang.String).

Before delegation, an absolute resource name is constructed from the given resource name using this algorithm:

- If the name begins with a '/' ('\u002f'), then the absolute name of the resource is the portion of the name following the '/'.
- Otherwise, the absolute name is of the following form:

  ```
  modified_package_name/name
  ```

  Where the modified_package_name is the package name of this object with '/' substituted for '.' ('\u002e').

Parameters:
name - name of the desired resource

Returns:
A InputStream object; null if no resource with this name is found, the resource is in a package that is not open to at least the caller module, or access to the resource is denied by the security manager.

Throws:
NullPointerException - If name is null

Since:
1.1

See Also:
Module.getResourceAsStream(String)

getResource

```
public URL getResource(String name)
```

Finds a resource with a given name.

If this class is in a named Module then this method will attempt to find the resource in the module. This is done by delegating to the module's class loader findResource(String,String) method, invoking it with the module name and the absolute name of the resource. Resources in named modules are subject to the rules for encapsulation specified in the Module getResourceAsStream method and so this method returns null when the resource is a non-".class" resource in a package that is not open to the caller's module.

Otherwise, if this class is not in a named module then the rules for searching resources associated with a given class are implemented by the defining class loader of the class. This method delegates to this Class object's class loader. If this Class object was loaded by the bootstrap class loader, the method delegates to ClassLoader.getSystemResource(java.lang.String).

Before delegation, an absolute resource name is constructed from the given resource name using this algorithm:

- If the name begins with a '/' ('\u002f'), then the absolute name of the resource is the portion of the name following the '/'.
- Otherwise, the absolute name is of the following form:

  ```
  modified_package_name/name
  ```

  Where the modified_package_name is the package name of this object with '/' substituted for '.' ('\u002e').

Parameters:
name - name of the desired resource

Returns:
A URL object; null if no resource with this name is found, the resource cannot be located by a URL, the resource is in a package that is not open to at least the caller module, or access to the resource is denied by the security manager.

Throws:
NullPointerException - If name is null

Since:
1.1

getProtectionDomain

```
public ProtectionDomain getProtectionDomain()
```

Returns the ProtectionDomain of this class. If there is a security manager installed, this method first calls the security manager's checkPermission method with a RuntimePermission("getProtectionDomain") permission to ensure it's ok to get the ProtectionDomain.

Returns:
the ProtectionDomain of this class

Throws:
SecurityException - if a security manager exists and its checkPermission method doesn't allow getting the ProtectionDomain.

Since:
1.2

See Also:
ProtectionDomain,
SecurityManager.checkPermission(java.security.Permission),
RuntimePermission

desiredAssertionStatus

```
public boolean desiredAssertionStatus()
```

Returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked. If this class has had its assertion status set, the most recent setting will be returned; otherwise, if any package default assertion status pertains to this class, the most recent setting for the most specific pertinent package default assertion status is returned; otherwise, if this class is not a system class (i.e., it has a class loader) its class loader's default assertion status is returned; otherwise, the system class default assertion status is returned.

API Note:
Few programmers will have any need for this method; it is provided for the benefit of the JDK itself. (It allows a class to determine at the time that it is initialized whether assertions should be enabled.) Note that this method is not guaranteed to return the actual assertion status that was (or will be) associated with the specified class when it was (or will be) initialized.

Returns:
the desired assertion status of the specified class.

Since:
1.4

See Also:
ClassLoader.setClassAssertionStatus(java.lang.String, boolean),
ClassLoader.setPackageAssertionStatus(java.lang.String, boolean),
ClassLoader.setDefaultAssertionStatus(boolean)

isEnum

```
public boolean isEnum()
```

Returns true if and only if this class was declared as an enum in the source code. Note that Enum is not itself an enum class. Also note that if an enum constant is declared with a class body, the class of that enum constant object is an anonymous class and *not* the class of the declaring enum class. The Enum.getDeclaringClass() method of an enum constant can be used to get the class of the enum class declaring the constant.

Returns:
true if and only if this class was declared as an enum in the source code

See *Java Language Specification*:
8.9.1 Enum Constants

Since:
1.5

isRecord

```
public boolean isRecord()
```

Returns true if and only if this class is a record class.

The direct superclass of a record class is java.lang.Record. A record class is final. A record class has (possibly zero) record components; getRecordComponents() returns a non-null but possibly empty value for a record.

Note that class Record is not a record class and thus invoking this method on class Record returns false.

Returns:
true if and only if this class is a record class, otherwise false

See *Java Language Specification*:
8.10 Record Classes

Since:
16

getEnumConstants

```
public T[] getEnumConstants()
```

Returns the elements of this enum class or null if this Class object does not represent an enum class.

Returns:
an array containing the values comprising the enum class represented by this Class object in the order they're declared, or null if this Class object does not represent an enum class

See *Java Language Specification*:
8.9.1 Enum Constants

Since:
1.5

cast

```
public T cast(Object obj)
```

Casts an object to the class or interface represented by this Class object.

Parameters:
obj - the object to be cast

Returns:
the object after casting, or null if obj is null

Throws:
ClassCastException - if the object is not null and is not assignable to the type T.

Since:
1.5

asSubclass

```
public <U> Class<? extends U> asSubclass(Class<U> clazz)
```

Casts this Class object to represent a subclass of the class represented by the specified class object. Checks that the cast is valid, and throws a ClassCastException if it is not. If this method succeeds, it always returns a reference to this Class object.

This method is useful when a client needs to "narrow" the type of a Class object to pass it to an API that restricts the Class objects that it is willing to accept. A cast would generate a compile-time warning, as the correctness of the cast could not be checked at runtime (because generic types are implemented by erasure).

Type Parameters:
U - the type to cast this Class object to

Parameters:
clazz - the class of the type to cast this Class object to

Returns:
this Class object, cast to represent a subclass of the specified class object.

Throws:
ClassCastException - if this Class object does not represent a subclass of the specified class (here "subclass" includes the class itself).

Since:
1.5

getAnnotation

```
public <A extends Annotation> A getAnnotation(Class<A> annotationClass)
```

Returns this element's annotation for the specified type if such an annotation is *present*, else null.

Note that any annotation returned by this method is a declaration annotation.

Specified by:
getAnnotation in interface AnnotatedElement

Type Parameters:
A - the type of the annotation to query for and return if present

Parameters:
annotationClass - the Class object corresponding to the annotation type

Returns:
this element's annotation for the specified annotation type if present on this element, else null

Throws:
NullPointerException - if the given annotation class is null

Since:
1.5

isAnnotationPresent

```
public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)
```

Returns true if an annotation for the specified type is *present* on this element, else false. This method is designed primarily for convenient access to marker annotations.

The truth value returned by this method is equivalent to: getAnnotation(annotationClass) != null

Specified by:
isAnnotationPresent in interface AnnotatedElement

Parameters:
annotationClass - the Class object corresponding to the annotation type

Returns:
true if an annotation for the specified annotation type is present on this element, else false

Throws:
NullPointerException - if the given annotation class is null

Since:
1.5

getAnnotationsByType

```
public <A extends Annotation> A[] getAnnotationsByType(Class<A> annotationClass)
```

Returns annotations that are *associated* with this element. If there are no annotations *associated* with this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getAnnotation(Class) is that this method detects if its argument is a *repeatable annotation type* (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation. The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers.

Note that any annotations returned by this method are declaration annotations.

Specified by:
getAnnotationsByType in interface AnnotatedElement

Type Parameters:
A - the type of the annotation to query for and return if present

Parameters:
annotationClass - the Class object corresponding to the annotation type

Returns:
all this element's annotations for the specified annotation type if associated with this element, else an array of length zero

Throws:
NullPointerException - if the given annotation class is null

Since:
1.8

getAnnotations

```
public Annotation[] getAnnotations()
```

Returns annotations that are *present* on this element. If there are no annotations *present* on this element, the return value is an array of length 0. The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers.

Note that any annotations returned by this method are declaration annotations.

Specified by:
getAnnotations in interface AnnotatedElement

Returns:
annotations present on this element

Since:
1.5

getDeclaredAnnotation

```
public <A extends Annotation> A getDeclaredAnnotation(Class<A> annotationClass)
```

Returns this element's annotation for the specified type if such an annotation is *directly present*, else null. This method ignores inherited annotations. (Returns null if no annotations are directly present on this element.)

Note that any annotation returned by this method is a declaration annotation.

Specified by:
getDeclaredAnnotation in interface AnnotatedElement

Type Parameters:
A - the type of the annotation to query for and return if directly present

Parameters:
annotationClass - the Class object corresponding to the annotation type

Returns:
this element's annotation for the specified annotation type if directly present on this element, else null

Throws:
NullPointerException - if the given annotation class is null

Since:
1.8

getDeclaredAnnotationsByType

```
public <A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> annotationClass)
```

Returns this element's annotation(s) for the specified type if such annotations are either *directly present* or *indirectly present*. This method ignores inherited annotations. If there are no specified annotations directly or indirectly present on this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getDeclaredAnnotation(Class) is that this method detects if its argument is a *repeatable annotation type* (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation if one is present. The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers.

Note that any annotations returned by this method are declaration annotations.

Specified by:
getDeclaredAnnotationsByType in interface AnnotatedElement

Type Parameters:
A - the type of the annotation to query for and return if directly or indirectly present

Parameters:
annotationClass - the Class object corresponding to the annotation type

Returns:
all this element's annotations for the specified annotation type if directly or indirectly present on this element, else an array of length zero

Throws:
NullPointerException - if the given annotation class is null

Since:
1.8

getDeclaredAnnotations

```
public Annotation[] getDeclaredAnnotations()
```

Returns annotations that are *directly present* on this element. This method ignores inherited annotations. If there are no annotations *directly present* on this element, the return value is an array of length 0. The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers.

Note that any annotations returned by this method are declaration annotations.

Specified by:
getDeclaredAnnotations in interface AnnotatedElement

Returns:
annotations directly present on this element

Since:
1.5

getAnnotatedSuperclass

```
public AnnotatedType getAnnotatedSuperclass()
```

Returns an AnnotatedType object that represents the use of a type to specify the superclass of the entity represented by this Class object. (The *use* of type Foo to specify the superclass in '... extends Foo' is distinct from the *declaration* of class Foo.)

If this Class object represents a class whose declaration does not explicitly indicate an annotated superclass, then the return value is an AnnotatedType object representing an element with no annotations.

If this Class represents either the Object class, an interface type, an array type, a primitive type, or void, the return value is null.

Returns:
an object representing the superclass

Since:
1.8

getAnnotatedInterfaces

```
public AnnotatedType[] getAnnotatedInterfaces()
```

Returns an array of AnnotatedType objects that represent the use of types to specify superinterfaces of the entity represented by this Class object. (The *use* of type Foo to specify a superinterface in '... implements Foo' is distinct from the *declaration* of interface Foo.)

If this Class object represents a class, the return value is an array containing objects representing the uses of interface types to specify interfaces implemented by the class. The order of the objects in the array corresponds to the order of the interface types used in the 'implements' clause of the declaration of this Class object.

If this Class object represents an interface, the return value is an array containing objects representing the uses of interface types to specify interfaces directly extended by the interface. The order of the objects in the array corresponds to the order of the interface types used in the 'extends' clause of the declaration of this Class object.

If this Class object represents a class or interface whose declaration does not explicitly indicate any annotated superinterfaces, the return value is an array of length 0.

If this Class object represents either the Object class, an array type, a primitive type, or void, the return value is an array of length 0.

Returns:
an array representing the superinterfaces

Since:
1.8

getNestHost

```
public Class<?> getNestHost()
```

Returns the nest host of the nest to which the class or interface represented by this Class object belongs. Every class and interface belongs to exactly one nest. If the nest host of this class or interface has previously been determined, then this method returns the nest host. If the nest host of this class or interface has not previously been determined, then this method determines the nest host using the algorithm of JVMS 5.4.4, and returns it. Often, a class or interface belongs to a nest consisting only of itself, in which case this method returns this to indicate that the class or interface is the nest host.

If this Class object represents a primitive type, an array type, or void, then this method returns this, indicating that the represented entity belongs to the nest consisting only of itself, and is the nest host.

Returns:
the nest host of this class or interface

Throws:
SecurityException - If the returned class is not the current class, and if a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for the returned class and invocation of s.checkPackageAccess() denies access to the package of the returned class

See *Java Virtual Machine Specification*:
4.7.28 The NestHost Attribute
4.7.29 The NestMembers Attribute
5.4.4 Access Control

Since:
11

isNestmateOf

```
public boolean isNestmateOf(Class<?> c)
```

Determines if the given Class is a nestmate of the class or interface represented by this Class object. Two classes or interfaces are nestmates if they have the same nest host.

Parameters:
c - the class to check

Returns:
true if this class and c are members of the same nest; and false otherwise.

Since:
11

getNestMembers

```
public Class<?>[] getNestMembers()
```

Returns an array containing Class objects representing all the classes and interfaces that are members of the nest to which the class or interface represented by this Class object belongs. First, this method obtains the nest host, H, of the nest to which the class or interface represented by this Class object belongs. The zeroth element of the returned array is H. Then, for each class or interface C which is recorded by H as being a member of its nest, this method attempts to obtain the Class object for C (using the defining class loader of the current Class object), and then obtains the nest host of the nest to which C belongs. The classes and interfaces which are recorded by H as being members of its nest, and for which H can be determined as their nest host, are indicated by subsequent elements of the returned array. The order of such elements is unspecified. Duplicates are permitted.

If this Class object represents a primitive type, an array type, or void, then this method returns a single-element array containing this.

API Note:
The returned array includes only the nest members recorded in the NestMembers attribute, and not any hidden classes that were added to the nest via Lookup::defineHiddenClass.

Returns:
an array of all classes and interfaces in the same nest as this class or interface

Throws:
SecurityException - If any returned class is not the current class, and if a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for that returned class and invocation of s.checkPackageAccess() denies access to the package of that returned class

See *Java Virtual Machine Specification*:
4.7.28 The NestHost Attribute
4.7.29 The NestMembers Attribute

Since:
11

See Also:
getNestHost()

descriptorString

```
public String descriptorString()
```

Returns the descriptor string of the entity (class, interface, array class, primitive type, or void) represented by this Class object.

If this Class object represents a class or interface, not an array class, then:

- If the class or interface is not hidden, then the result is a field descriptor (JVMS 4.3.2) for the class or interface. Calling ClassDesc::ofDescriptor with the result descriptor string produces a ClassDesc describing this class or interface.
- If the class or interface is hidden, then the result is a string of the form:

  ```
  "L" +N+ "." + <suffix> + ";"
  ```

  where *N* is the binary name encoded in internal form indicated by the class file passed to Lookup::defineHiddenClass, and <suffix> is an unqualified name. A hidden class or interface has no nominal descriptor. The result string is not a type descriptor.

If this Class object represents an array class, then the result is a string consisting of one or more '[' characters representing the depth of the array nesting, followed by the descriptor string of the element type.

- If the element type is not a hidden class or interface, then this array class can be described nominally. Calling ClassDesc::ofDescriptor with the result descriptor string produces a ClassDesc describing this array class.
- If the element type is a hidden class or interface, then this array class cannot be described nominally. The result string is not a type descriptor.

If this Class object represents a primitive type or void, then the result is a field descriptor string which is a one-letter code corresponding to a primitive type or void ("B", "C", "D", "F", "I", "J", "S", "Z", "V") (JVMS 4.3.2).

Specified by:
descriptorString in interface TypeDescriptor

Returns:
the descriptor string for this Class object

See *Java Virtual Machine Specification*:
4.3.2 Field Descriptors

Since:
12

componentType

```
public Class<?> componentType()
```

Returns the component type of this Class, if it describes an array type, or null otherwise.

Specified by:
componentType in interface TypeDescriptor.OfField<T>

Implementation Requirements:
Equivalent to getComponentType().

Returns:
a Class describing the component type, or null if this Class does not describe an array type

Since:
12

arrayType

```
public Class<?> arrayType()
```

Returns a Class for an array type whose component type is described by this Class.

Specified by:
arrayType in interface TypeDescriptor.OfField<T>

Returns:
a Class describing the array type

Throws:
UnsupportedOperationException - if this component type is void or if the number of dimensions of the resulting array type would exceed 255.

See *Java Virtual Machine Specification*:
4.3.2 Field Descriptors
4.4.1 The CONSTANT_Class_info Structure

Since:
12

describeConstable

```
public Optional<ClassDesc> describeConstable()
```

Returns a nominal descriptor for this instance, if one can be constructed, or an empty Optional if one cannot be.

Specified by:
describeConstable in interface Constable

Returns:
An Optional containing the resulting nominal descriptor, or an empty Optional if one cannot be constructed.

Since:
12

isHidden

```
public boolean isHidden()
```

Returns true if and only if the underlying class is a hidden class.

Returns:
true if and only if this class is a hidden class.

Since:
15

See Also:
MethodHandles.Lookup.defineHiddenClass(byte[], boolean, java.lang.invoke.MethodHandles.Lookup.ClassOption...)

getPermittedSubclasses

```
public Class<?>[] getPermittedSubclasses()
```

Returns an array containing Class objects representing the direct subinterfaces or subclasses permitted to extend or implement this class or interface if it is sealed. The order of such elements is unspecified. The array is empty if this sealed class or interface has no permitted subclass. If this Class object represents a primitive type, void, an array type, or a class or interface that is not sealed, that is isSealed() returns false, then this method returns null. Conversely, if isSealed() returns true, then this method returns a non-null value. For each class or interface C which is recorded as a permitted direct subinterface or subclass of this class or interface, this method attempts to obtain the Class object for C (using the defining class loader of the current Class object). The Class objects which can be obtained and which are direct subinterfaces or subclasses of this class or interface, are indicated by elements of the returned array. If a Class object cannot be obtained, it is silently ignored, and not included in the result array.

Returns:
an array of Class objects of the permitted subclasses of this class or interface, or null if this class or interface is not sealed.

Throws:
SecurityException - If a security manager, *s*, is present and the caller's class loader is not the same as or an ancestor of the class loader for that returned class and invocation of s.checkPackageAccess() denies access to the package of any class in the returned array.

See *Java Language Specification*:
8.1 Class Declarations
9.1 Interface Declarations

Since:
17

isSealed

```
public boolean isSealed()
```

Returns true if and only if this Class object represents a sealed class or interface. If this Class object represents a primitive type, void, or an array type, this method returns false. A sealed class or interface has (possibly zero) permitted subclasses; getPermittedSubclasses() returns a non-null but possibly empty value for a sealed class or interface.

Returns:
true if and only if this Class object represents a sealed class or interface.

See *Java Language Specification*:
8.1 Class Declarations
9.1 Interface Declarations

Since:
17

Report a bug or suggest an enhancement
For further API reference and developer documentation see the Java SE Documentation, which contains more detailed, developer-targeted descriptions with conceptual overviews, definitions of terms, workarounds, and working code examples. Other versions.
Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.
Copyright © 1993, 2023, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.
All rights reserved. Use is subject to license terms and the documentation redistribution policy.
DRAFT 22-internal-adhoc.jlaskey.open