Table of Contents

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Chapter 10. java.lang and Subpackages

This chapter covers the `java.lang` package which defines the core classes and interfaces that are indispensable to the Java platform and the Java programming language. It also covers more specialized subpackages:

`java.lang.annotation`

Defines the `Annotation` interface that all annotation types extend, and also defines meta-annotation types and related enumerated types. Added in Java 5.0.

`java.lang.instrument`

Provides support for Java-based "agents" that can instrument a Java program by transforming class files as they are loaded. Added in Java 5.0.

`java.lang.management`

Defines "management bean" interfaces for remote monitoring and management of a running Java interpreter.

`java.lang.ref`

Defines "reference" classes that are used to refer to objects without preventing the garbage collector from reclaiming those objects.

`java.lang.reflect`

Allows Java programs to examine the members of arbitrary classes, invoking methods, and querying and setting the value of fields.

Package java.lang

Java 1.0

The `java.lang` package contains the classes that are most central to the Java language. `Object` is the ultimate superclass of all Java classes and is therefore at the top of all class

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

hierarchies. `Class` is a class that describes a Java class. There is one `Class` object for each class that is loaded into Java.

`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` are immutable class wrappers around each of the primitive Java data types. These classes are useful when you need to manipulate primitive types as objects. They also contain useful conversion and utility methods. `Void` is a related class that defines a representation for the `void` method return type, but that defines no methods. `String` and `StringBuffer` are objects that represent strings. `String` is an immutable type, while `StringBuffer` can have its string changed in place. In Java 5.0, `StringBuilder` is like `StringBuffer` but without `synchronized` methods, which makes it the preferred choice in most applications. `String`, `StringBuffer` and `StringBuilder` implement the Java 1.4 interface `CharSequence` which allows instances of these classes to be manipulated through a simple shared API.

`String` and the various primitive type wrapper classes all implement the `Comparable` interface which defines an ordering for instances of those classes and enables sorting and searching algorithms (such as those of `java.util.Arrays` and `java.util.Collections`, for example). `Cloneable` is an important marker interface that specifies that the `Object.clone( )` method is allowed to make copies of an object.

The `Math` class (and, in Java 1.3, the `StrictMath` class) defines static methods for various floating-point mathematical functions.

The `Thread` class provides support for multiple threads of control running within the same Java interpreter. The `Runnable` interface is implemented by objects that have a `run( )` method that can serve as the body of a thread.

`System` provides low-level system methods. `Runtime` provides similar low-level methods, including an `exec( )` method that, along with the `Process` class, defines a platform-dependent API for running external processes. Java 5.0 allows `Process` objects to be created more easily with the `ProcessBuilder` class.

`Throwable` is the root class of the exception and error hierarchy. `Throwable` objects are used with the Java `throw` and `catch` statements. `java.lang` defines quite a few subclasses of `Throwable`. `Exception` and `Error` are the superclasses of all exceptions and errors. `RuntimeException` defines a special class or "unchecked exceptions" that do not need to be declared in a method's `throws` clause. The `Throwable` class was overhauled in Java 1.4, adding the ability to "chain" exceptions, and the ability to obtain the stack trace of an exception as an array of `StackTraceElement` objects.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 adds three important interfaces to this package. `Iterable` marks types that have an `iterator( )` method and enables iteration with the `for/in` looping statement introduced in Java 5.0. The `Appendable` interface is implemented by classes (such as `StringBuilder` and character output streams) that can have characters appended to them. Implementing this interface enables formatted text output with a `java.util.Formatter`. The `Readable` interface is implemented by classes (such as character input streams) that can sequentially copy characters into a buffer. It enables interaction with a `java.util.Scanner`.

Also new in Java 5.0 is `Enum`, which serves as the superclass of all enumerated types declared with the new `enum` keyword. `Deprecated`, `Override`, and `SuppressWarnings` are annotation types that provide metadata for the compiler.

Interfaces

```
public interface Appendable;
public interface CharSequence;
public interface Cloneable;
public interface Comparable<T>;
public interface Iterable<T>;
public interface Readable;
public interface Runnable;
public interface Thread.UncaughtExceptionHandler;
```

Enumerated Types

```
public enum Thread.State;
```

Annotation Types

```
public @interface Deprecated;
public @interface Override;
public @interface SuppressWarnings;
```

Classes

```
public class Object;
   abstract class AbstractStringBuilder implements Appendable, CharSequence;
      public final class StringBuffer extends AbstractStringBuilder implements
      CharSequence, Serializable;
      public final class StringBuilder extends AbstractStringBuilder implements
      CharSequence, Serializable;
   public final class Boolean implements Serializable, Comparable<Boolean>;
   public final class Character implements Serializable, Comparable<Character>;
   public static class Character.Subset;
      public static final class Character.UnicodeBlock extends Character.Subset;
   public final class Class<T> implements Serializable, java.lang.reflect.
      GenericDeclaration, java.lang.reflect.Type, java.lang.reflect.AnnotatedElement;
   public abstract class ClassLoader;
   public final class Compiler;
   public abstract class Enum<E extends Enum<E>> implements Comparable<E>,
   Serializable;
   public final class Math;
   public abstract class Number implements Serializable;
      public final class Byte extends Number implements Comparable<Byte>;
      public final class Double extends Number implements Comparable<Double>;
      public final class Float extends Number implements Comparable<Float>;
      public final class Integer extends Number implements Comparable<Integer>;
      public final class Long extends Number implements Comparable<Long>;
      public final class Short extends Number implements Comparable<Short>;
   public class Package implements java.lang.reflect.AnnotatedElement;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
   public abstract class Process;
   public final class ProcessBuilder;
   public class Runtime;
   public class SecurityManager;
   public final class StackTraceElement implements Serializable;
   public final class StrictMath;
   public final class String implements Serializable, Comparable<String>, CharSequence;
   public final class System;
   public class Thread implements Runnable;
   public class ThreadGroup implements Thread.UncaughtExceptionHandler;
   public class ThreadLocal<T>;
      public class InheritableThreadLocal<T> extends ThreadLocal<T>;
   public class Throwable implements Serializable;
   public final class Void;
public final class RuntimePermission extends java.security.BasicPermission;
```

Exceptions

```
public class Exception extends Throwable;
   public class ClassNotFoundException extends Exception;
   public class CloneNotSupportedException extends Exception;
   public class IllegalAccessException extends Exception;
   public class InstantiationException extends Exception;
   public class InterruptedException extends Exception;
   public class NoSuchFieldException extends Exception;
   public class NoSuchMethodException extends Exception;
   public class RuntimeException extends Exception;
      public class ArithmeticException extends RuntimeException;
      public class ArrayStoreException extends RuntimeException;
      public class ClassCastException extends RuntimeException;
      public class EnumConstantNotPresentException extends RuntimeException;
      public class IllegalArgumentException extends RuntimeException;
         public class IllegalThreadStateException extends IllegalArgumentException;
         public class NumberFormatException extends IllegalArgumentException;
      public class IllegalMonitorStateException extends RuntimeException;
      public class IllegalStateException extends RuntimeException;
      public class IndexOutOfBoundsException extends RuntimeException;
         public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException;
         public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException;
      public class NegativeArraySizeException extends RuntimeException;
      public class NullPointerException extends RuntimeException;
      public class SecurityException extends RuntimeException;
      public class TypeNotPresentException extends RuntimeException;
      public class UnsupportedOperationException extends RuntimeException;
```

Errors

```
public class Error extends Throwable;
   public class AssertionError extends Error;
   public class LinkageError extends Error;
      public class ClassCircularityError extends LinkageError;
      public class ClassFormatError extends LinkageError;
         public class UnsupportedClassVersionError extends ClassFormatError;
      public class ExceptionInInitializerError extends LinkageError;
      public class IncompatibleClassChangeError extends LinkageError;
         public class AbstractMethodError extends IncompatibleClassChangeError;
         public class IllegalAccessError extends IncompatibleClassChangeError;
         public class InstantiationError extends IncompatibleClassChangeError;
         public class NoSuchFieldError extends IncompatibleClassChangeError;
         public class NoSuchMethodError extends IncompatibleClassChangeError;
      public class NoClassDefFoundError extends LinkageError;
      public class UnsatisfiedLinkError extends LinkageError;
      public class VerifyError extends LinkageError;
   public class ThreadDeath extends Error;
   public abstract class VirtualMachineError extends Error;
      public class InternalError extends VirtualMachineError;
      public class OutOfMemoryError extends VirtualMachineError;
      public class StackOverflowError extends VirtualMachineError;
      public class UnknownError extends VirtualMachineError;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

AbstractMethodError java.lang

Java 1.0 ***serializable error***

Signals an attempt to invoke an abstract method.

Figure 10-1. java.lang.AbstractMethodError


```
public class AbstractMethodError extends IncompatibleClassChangeError {
// Public Constructors
     public AbstractMethodError( );
     public AbstractMethodError(String s);
}
```

AbstractStringBuilder java.lang

Java 5.0 ***appendable***

This package-private class is the abstract superclass of `StringBuffer` and `StringBuilder`. Because this class is not public, you may not use it directly. It is included in this quick-reference to fully document the shared API of its two subclasses.

Note that many of the methods of this class are declared to return an `AbstractStringBuilder` object. `StringBuilder` and `StringBuffer( )` override those methods and narrow the return type to `StringBuilder` or `StringBuffer`. (This is an example of "covariant returns," which are allowed in Java 5.0 and later.)

Figure 10-2. java.lang.AbstractStringBuilder


```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
// No Constructor
// Public Instance Methods
     public AbstractStringBuilder append(char[ ] str);
     public AbstractStringBuilder append(boolean b);
     public AbstractStringBuilder append(char c);
     public AbstractStringBuilder append(Object obj);
     public AbstractStringBuilder append(CharSequence s);
     public AbstractStringBuilder append(StringBuffer sb);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public AbstractStringBuilder append(String str);
    public AbstractStringBuilder append(int i);
    public AbstractStringBuilder append(double d);
    public AbstractStringBuilder append(float f);
    public AbstractStringBuilder append(long l);
    public AbstractStringBuilder append(char[ ] str, int offset, int len);
    public AbstractStringBuilder append(CharSequence s, int start, int end);
    public AbstractStringBuilder appendCodePoint(int codePoint);
    public int capacity( );
    public int codePointAt(int index);
    public int codePointBefore(int index);
    public int codePointCount(int beginIndex, int endIndex);
    public AbstractStringBuilder delete(int start, int end);
    public AbstractStringBuilder deleteCharAt(int index);
    public void ensureCapacity(int minimumCapacity);
    public void getChars(int srcBegin, int srcEnd, char[ ] dst, int dstBegin);
    public int indexOf(String str);
    public int indexOf(String str, int fromIndex);
    public AbstractStringBuilder insert(int offset, char c);
    public AbstractStringBuilder insert(int offset, boolean b);
    public AbstractStringBuilder insert(int dstOffset, CharSequence s);
    public AbstractStringBuilder insert(int offset, int i);
    public AbstractStringBuilder insert(int offset, double d);
    public AbstractStringBuilder insert(int offset, float f);
    public AbstractStringBuilder insert(int offset, long l);
    public AbstractStringBuilder insert(int offset, char[ ] str);
    public AbstractStringBuilder insert(int offset, Object obj);
    public AbstractStringBuilder insert(int offset, String str);
    public AbstractStringBuilder insert(int index, char[ ] str, int offset, int len);
    public AbstractStringBuilder insert(int dstOffset, CharSequence s, int start, int end);
    public int lastIndexOf(String str);
    public int lastIndexOf(String str, int fromIndex);
    public int offsetByCodePoints(int index, int codePointOffset);
    public AbstractStringBuilder replace(int start, int end, String str);
    public AbstractStringBuilder reverse( );
    public void setCharAt(int index, char ch);
    public void setLength(int newLength);
    public String substring(int start);
    public String substring(int start, int end);
    public void trimToSize( );
// Methods Implementing CharSequence
    public char charAt(int index);
    public int length( );
    public CharSequence subSequence(int start, int end);
    public abstract String toString( );
}
```

Subclasses

`StringBuffer`, `StringBuilder`

Returned By

Too many methods to list.

Appendable java.lang

Java 5.0 ***appendable***

Objects that implement this interface can have characters or character sequences appended to them. `Appendable` was added in Java 5.0 as a simple unifying API for `StringBuffer` and `StringBuilder`, `java.nio.CharBuffer`, and character output

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

stream subclasses of `java.io.Writer`. The `java.util.Formatter` class can send formatted output to any `Appendable` object. See also `Readable`.

```
public interface Appendable {
// Public Instance Methods
     Appendable append(char c) throws java.io.IOException;
     Appendable append(CharSequence csq) throws java.io.IOException;
     Appendable append(CharSequence csq, int start, int end) throws java.io.IOException;
}
```

Implementations

`java.io.PrintStream`, `java.io.Writer`, `java.nio.CharBuffer`

Passed To

`java.util.Formatter.Formatter( )`

Returned By

Too many methods to list.

ArithmeticException java.lang

Java 1.0 ***serializable unchecked***

A `RuntimeException` that signals an exceptional arithmetic condition, such as integer division by zero.

Figure 10-3. java.lang.ArithmeticException


```
public class ArithmeticException extends RuntimeException {
// Public Constructors
     public ArithmeticException( );
     public ArithmeticException(String s);
}
```

ArrayIndexOutOfBoundsException java.lang

Java 1.0 ***serializable unchecked***

Signals that an array index less than zero or greater than or equal to the array size has been used.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-4. java.lang.ArrayIndexOutOfBoundsException


```
public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException {
// Public Constructors
     public ArrayIndexOutOfBoundsException( );
     public ArrayIndexOutOfBoundsException(String s);
     public ArrayIndexOutOfBoundsException(int index);
}
```

Thrown By

Too many methods to list.

ArrayStoreException java.lang

Java 1.0 ***serializable unchecked***

Signals an attempt to store the wrong type of object into an array.

Figure 10-5. java.lang.ArrayStoreException


```
public class ArrayStoreException extends RuntimeException {
// Public Constructors
     public ArrayStoreException( );
     public ArrayStoreException(String s);
}
```

AssertionError java.lang

Java 1.4 ***serializable error***

An instance of this class is thrown if when an assertion fails. This happens when assertions are enabled, and the expression following an `assert` statement does not evaluate to `true`. If an assertion fails, and the `assert` statement has a second expression separated from the first by a colon, then the second expression is evaluated and the resulting value is passed to the `AssertionError( )` constructor, where it is converted to a string and used as the error message.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-6. java.lang.AssertionError


```
public class AssertionError extends Error {
// Public Constructors
     public AssertionError( );
     public AssertionError(long detailMessage);
     public AssertionError(float detailMessage);
     public AssertionError(double detailMessage);
     public AssertionError(int detailMessage);
     public AssertionError(Object detailMessage);
     public AssertionError(boolean detailMessage);
     public AssertionError(char detailMessage);
}
```

Boolean java.lang

Java 1.0 ***serializable comparable***

This class provides an immutable object wrapper around the `boolean` primitive type. Note that the `TRUE` and `FALSE` constants are `Boolean` objects; they are not the same as the `true` and `false` `boolean` values. As of Java 1.1, this class defines a `Class` constant that represents the `boolean` type. `booleanValue( )` returns the `boolean` value of a `Boolean` object. The class method `getBoolean( )` retrieves the `boolean` value of a named property from the system property list. The static method `valueOf( )` parses a string and returns the `Boolean` object it represents. Java 1.4 added two static methods that convert primitive `boolean` values to `Boolean` and `String` objects. In Java 5.0, the `parseBoolean( )` method behaves like `valueOf( )` but returns a primitive `boolean` value instead of a `Boolean` object.

Prior to Java 5.0, this class does not implement the `Comparable` interface.

Figure 10-7. java.lang.Boolean


```
public final class Boolean implements Serializable, Comparable<Boolean> {
// Public Constructors
     public Boolean(String s);
     public Boolean(boolean value);
// Public Constants
     public static final Boolean FALSE;
     public static final Boolean TRUE;
1.1  public static final Class<Boolean> TYPE;
// Public Class Methods
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public static boolean getBoolean(String name);
5.0 public static boolean parseBoolean(String s);
1.4 public static String toString(boolean b);
1.4 public static Boolean valueOf(boolean b);
    public static Boolean valueOf(String s);
// Public Instance Methods
    public boolean booleanValue( );
// Methods Implementing Comparable
5.0 public int compareTo(Boolean b);
// Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode( );
    public String toString( );
}
```

Byte java.lang

Java 1.1 ***serializable comparable***

This class provides an immutable object wrapper around the `byte` primitive type. It defines useful constants for the minimum and maximum values that can be stored by the `byte` type and a `Class` object constant that represents the `byte` type. It also provides various methods for converting `Byte` values to and from strings and other numeric types.

Most of the static methods of this class can convert a `String` to a `Byte` object or a `byte` value: the four `parseByte( )` and `valueOf( )` methods parse a number from the specified string using an optionally specified radix and return it in one of these two forms. The `decode( )` method parses a byte specified in base 10, base 8, or base 16 and returns it as a `Byte`. If the string begins with "0x" or "#", it is interpreted as a hexadecimal number. If it begins with "0", it is interpreted as an octal number. Otherwise, it is interpreted as a decimal number.

Note that this class has two `toString( )` methods. One is static and converts a `byte` primitive value to a string; the other is the usual `toString( )` method that converts a `Byte` object to a string. Most of the remaining methods convert a `Byte` to various primitive numeric types.

Figure 10-8. java.lang.Byte


```
public final class Byte extends Number implements Comparable<Byte> {
// Public Constructors
    public Byte(byte value);
    public Byte(String s) throws NumberFormatException;
// Public Constants
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public static final byte MAX_VALUE;   =127
    public static final byte MIN_VALUE;   =-128
5.0 public static final int SIZE;     =8
    public static final Class<Byte> TYPE;
// Public Class Methods
    public static Byte decode(String nm) throws NumberFormatException;
    public static byte parseByte(String s) throws NumberFormatException;
    public static byte parseByte(String s, int radix) throws NumberFormatException;
    public static String toString(byte b);
    public static Byte valueOf(String s) throws NumberFormatException;
5.0 public static Byte valueOf(byte b);
    public static Byte valueOf(String s, int radix) throws NumberFormatException;
// Methods Implementing Comparable
1.2 public int compareTo(Byte anotherByte);
// Public Methods Overriding Number
    public byte byteValue( );
    public double doubleValue( );
    public float floatValue( );  yu
    public int intValue( );
    public long longValue( );
    public short shortValue( );
// Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode( );
    public String toString( );
}
```

Character java.lang

Java 1.0 ***serializable comparable***

This class provides an immutable object wrapper around the primitive `char` data type. `charValue( )` returns the `char` value of a `Character` object. The `compareTo( )` method implements the `Comparable` interface so that `Character` objects can be ordered and sorted. The static methods are the most interesting thing about this class, however: they categorize `char` values based on the categories defined by the `Unicode` standard. (Some of the methods are only useful if you have a detailed understanding of that standard.) Static methods beginning with "is" test whether a character is in a given category. `isDigit( )` , `isLetter( )`, `isWhitespace( )`, `isUpperCase( )` and `isLowerCase( )` are some of the most useful. Note that these methods work for any Unicode character, not just with the familiar Latin letters and Arabic numbers of the ASCII character set. `getType( )` returns a constant that identifies the category of a character. `getDirectionality( )` returns a separate `DIRECTIONALITY_` constant that specifies the "directionality category" of a character.

In addition to testing the category of a character, this class also defines static methods for converting characters. `toUpperCase( )` returns the uppercase equivalent of the specified character (or returns the character itself if the character is uppercase or has no uppercase equivalent). `toLowerCase( )` converts instead to lowercase. `digit( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

returns the integer equivalent of a given character in a given radix (or base; for example, use 16 for hexadecimal). It works with any Unicode digit character, and also (for sufficiently large radix values) the ASCII letters a-z and A-Z. `forDigit( )` returns the ASCII character that corresponds to the specified value (0-35) for the specified radix. `getNumericValue( )` is similar, but also works with any Unicode character including those, such as Roman numerals, that represent numbers but are not decimal digits. Finally, the static `toString( )` method returns a `String` of length 1 that contains the specified `char` value.

Java 5.0 introduces many new methods to this class to accommodate Unicode supplementary characters that use 21 bits and do not fit in a single `char` value. The two representations for these supplementary characters are as an `int` codepoint in the range 0 through 0x10ffff, or as a sequence of two `char` values known as a "surrogate pair." The first `char` of such a pair should fall in the "high surrogate" range and the second `char` should fall in the "low surrogate" range. `toChars( )` converts an `int` codepoint into one or two `char` values. `toCodePoint( )` , `codePointAt( )`, and `codePointBefore( )` convert one or two `char` values into the corresponding `int` value. `codePointCount( )` returns the number of characters in a `char` array or `CharSequence`, counting surrogate pairs as a single supplementary character. `offsetByCodePoints( )` tells you how many `char` indexes to advance in a run of text if you want to skip over the specified number of code points. Finally, the various character type testing and case conversion methods such as `isWhitespace( )` and `toUpperCase( )` are available in new versions that take an `int` codepoint argument instead of a single `char` argument.

Figure 10-9. java.lang.Character


```
public final class Character implements Serializable, Comparable<Character> {
// Public Constructors
     public Character(char value);
// Public Constants
1.1  public static final byte COMBINING_SPACING_MARK;                =8
1.1  public static final byte CONNECTOR_PUNCTUATION;                 =23
1.1  public static final byte CONTROL; =15
1.1  public static final byte CURRENCY_SYMBOL;                       =26
1.1  public static final byte DASH_PUNCTUATION;                      =20
1.1  public static final byte DECIMAL_DIGIT_NUMBER;                  =9
1.4  public static final byte DIRECTIONALITY_ARABIC_NUMBER;          =6
1.4  public static final byte DIRECTIONALITY_BOUNDARY_NEUTRAL;       =9
1.4  public static final byte DIRECTIONALITY_COMMON_NUMBER_SEPARATOR;     =7
1.4  public static final byte DIRECTIONALITY_EUROPEAN_NUMBER;        =3
1.4  public static final byte DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR;     =4
1.4  public static final byte DIRECTIONALITY_EUROPEAN_NUMBER_TERMINATOR;     =5
1.4  public static final byte DIRECTIONALITY_LEFT_TO_RIGHT;          =0
1.4  public static final byte DIRECTIONALITY_LEFT_TO_RIGHT_EMBEDDING;     =14
1.4  public static final byte DIRECTIONALITY_LEFT_TO_RIGHT_OVERRIDE;     =15
1.4  public static final byte DIRECTIONALITY_NONSPACING_MARK;        =8
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
1.4  public static final byte DIRECTIONALITY_OTHER_NEUTRALS;         =13
1.4  public static final byte DIRECTIONALITY_PARAGRAPH_SEPARATOR;     =10
1.4  public static final byte DIRECTIONALITY_POP_DIRECTIONAL_FORMAT;     =18
1.4  public static final byte DIRECTIONALITY_RIGHT_TO_LEFT;          =1
1.4  public static final byte DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC;     =2
1.4  public static final byte DIRECTIONALITY_RIGHT_TO_LEFT_EMBEDDING;     =16
1.4  public static final byte DIRECTIONALITY_RIGHT_TO_LEFT_OVERRIDE;     =17
1.4  public static final byte DIRECTIONALITY_SEGMENT_SEPARATOR;      =11
1.4  public static final byte DIRECTIONALITY_UNDEFINED;              =-1
1.4  public static final byte DIRECTIONALITY_WHITESPACE;             =12
1.1  public static final byte ENCLOSING_MARK;                       =7
1.1  public static final byte END_PUNCTUATION;                      =22
1.4  public static final byte FINAL_QUOTE_PUNCTUATION;              =30
1.1  public static final byte FORMAT;  =16
1.4  public static final byte INITIAL_QUOTE_PUNCTUATION;            =29
1.1  public static final byte LETTER_NUMBER;                        =10
1.1  public static final byte LINE_SEPARATOR;                       =13
1.1  public static final byte LOWERCASE_LETTER;                     =2
1.1  public static final byte MATH_SYMBOL;                          =25
5.0  public static final int MAX_CODE_POINT;                       =1114111
5.0  public static final char MAX_HIGH_SURROGATE;                  = \uDBFF
5.0  public static final char MAX_LOW_SURROGATE;                   = \uDFFF
     public static final int MAX_RADIX;    =36
5.0  public static final char MAX_SURROGATE;                       = \uDFFF
     public static final char MAX_VALUE;   = \uFFFF
5.0  public static final int MIN_CODE_POINT;                       =0
5.0  public static final char MIN_HIGH_SURROGATE;                  = \uD800
5.0  public static final char MIN_LOW_SURROGATE;                   = \uDC00
     public static final int MIN_RADIX;    =2
5.0  public static final int MIN_SUPPLEMENTARY_CODE_POINT;         =65536
5.0  public static final char MIN_SURROGATE;                       = \uD800
     public static final char MIN_VALUE;   = \0
1.1  public static final byte MODIFIER_LETTER;                     =4
1.1  public static final byte MODIFIER_SYMBOL;                     =27
1.1  public static final byte NON_SPACING_MARK;                    =6
1.1  public static final byte OTHER_LETTER;                        =5
1.1  public static final byte OTHER_NUMBER;                        =11
1.1  public static final byte OTHER_PUNCTUATION;                   =24
1.1  public static final byte OTHER_SYMBOL;                        =28
1.1  public static final byte PARAGRAPH_SEPARATOR;                 =14
1.1  public static final byte PRIVATE_USE;                         =18
5.0  public static final int SIZE;     =16
1.1  public static final byte SPACE_SEPARATOR;                     =12
1.1  public static final byte START_PUNCTUATION;                   =21
1.1  public static final byte SURROGATE;                           =19
1.1  public static final byte TITLECASE_LETTER;                    =3
1.1  public static final Class<Character> TYPE;
1.1  public static final byte UNASSIGNED;                          =0
1.1  public static final byte UPPERCASE_LETTER;                    =1
// Nested Types
1.2  public static class Subset;
1.2  public static final class UnicodeBlock extends Character.Subset;
// Public Class Methods
5.0  public static int charCount(int codePoint);
5.0  public static int codePointAt(char[ ] a, int index);
5.0  public static int codePointAt(CharSequence seq, int index);
5.0  public static int codePointAt(char[ ] a, int index, int limit);
5.0  public static int codePointBefore(CharSequence seq, int index);
5.0  public static int codePointBefore(char[ ] a, int index);
5.0  public static int codePointBefore(char[ ] a, int index, int start);
5.0  public static int codePointCount(char[ ] a, int offset, int count);
5.0  public static int codePointCount(CharSequence seq, int beginIndex, int endIndex);
5.0  public static int digit(int codePoint, int radix);
     public static int digit(char ch, int radix);
     public static char forDigit(int digit, int radix);
1.4  public static byte getDirectionality(char ch);
5.0  public static byte getDirectionality(int codePoint);
1.1  public static int getNumericValue(char ch);
5.0  public static int getNumericValue(int codePoint);
1.1  public static int getType(char ch);
5.0  public static int getType(int codePoint);
5.0  public static boolean isDefined(int codePoint);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
     public static boolean isDefined(char ch);
5.0  public static boolean isDigit(int codePoint);
     public static boolean isDigit(char ch);
5.0  public static boolean isHighSurrogate(char ch);
5.0  public static boolean isIdentifierIgnorable(int codePoint);
1.1  public static boolean isIdentifierIgnorable(char ch);
1.1  public static boolean isISOControl(char ch);
5.0  public static boolean isISOControl(int codePoint);
1.1  public static boolean isJavaIdentifierPart(char ch);
5.0  public static boolean isJavaIdentifierPart(int codePoint);
1.1  public static boolean isJavaIdentifierStart(char ch);
5.0  public static boolean isJavaIdentifierStart(int codePoint);
     public static boolean isLetter(char ch);
5.0  public static boolean isLetter(int codePoint);
     public static boolean isLetterOrDigit(char ch);
5.0  public static boolean isLetterOrDigit(int codePoint);
5.0  public static boolean isLowerCase(int codePoint);
     public static boolean isLowerCase(char ch);
5.0  public static boolean isLowSurrogate(char ch);
5.0  public static boolean isMirrored(int codePoint);
1.4  public static boolean isMirrored(char ch);
5.0  public static boolean isSpaceChar(int codePoint);
1.1  public static boolean isSpaceChar(char ch);
5.0  public static boolean isSupplementaryCodePoint(int codePoint);
5.0  public static boolean isSurrogatePair(char high, char low);
     public static boolean isTitleCase(char ch);
5.0  public static boolean isTitleCase(int codePoint);
1.1  public static boolean isUnicodeIdentifierPart(char ch);
5.0  public static boolean isUnicodeIdentifierPart(int codePoint);
5.0  public static boolean isUnicodeIdentifierStart(int codePoint);
1.1  public static boolean isUnicodeIdentifierStart(char ch);
     public static boolean isUpperCase(char ch);
5.0  public static boolean isUpperCase(int codePoint);
5.0  public static boolean isValidCodePoint(int codePoint);
5.0  public static boolean isWhitespace(int codePoint);
1.1  public static boolean isWhitespace(char ch);
5.0  public static int offsetByCodePoints(CharSequence seq, int index,
      int codePointOffset);
5.0  public static int offsetByCodePoints(char[ ] a, int start, int count,
      int index, int codePointOffset);
5.0  public static char reverseBytes(char ch);
5.0  public static char[ ] toChars(int codePoint);
5.0  public static int toChars(int codePoint, char[ ] dst, int dstIndex);
5.0  public static int toCodePoint(char high, char low);
     public static char toLowerCase(char ch);
5.0  public static int toLowerCase(int codePoint);
1.4  public static String toString(char c);
     public static char toTitleCase(char ch);
5.0  public static int toTitleCase(int codePoint);
     public static char toUpperCase(char ch);
5.0  public static int toUpperCase(int codePoint);
5.0  public static Character valueOf(char c);
// Public Instance Methods
     public char charValue( );
// Methods Implementing Comparable
1.2  public int compareTo(Character anotherCharacter);
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
// Deprecated Public Methods
#    public static boolean isJavaLetter(char ch);
#    public static boolean isJavaLetterOrDigit(char ch);
#    public static boolean isSpace(char ch);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Character.Subset java.lang

Java 1.2

This class represents a named subset of the Unicode character set. The `toString( )` method returns the name of the subset. This is a base class intended for further subclassing. Note, in particular, that it does not provide a way to list the members of the subset, nor a way to test for membership in the subset. See `Character.UnicodeBlock`.

```
public static class Character.Subset {
// Protected Constructors
     protected Subset(String name);
// Public Methods Overriding Object
     public final boolean equals(Object obj);
     public final int hashCode( );
     public final String toString( );
}
```

Subclasses

`Character.UnicodeBlock`

Character.UnicodeBlock java.lang

Java 1.2

This subclass of `Character.Subset` defines a number of constants that represent named subsets of the Unicode character set. The subsets and their names are the character blocks defined by the Unicode specification (see http://www.unicode.org/). Java 1.4 and 5.0 both update this class to a new version of the Unicode standard and define a number of new block constants. The static method `of( )` takes a character or `int` codepoint and returns the `Character.UnicodeBlock` to which it belongs, or `null` if it is not part of any defined block. When presented with an unknown Unicode character, this method provides a useful way to determine what alphabet it belongs to. In Java 5.0, the `forName( )` factory method allows lookup of a `UnicodeBlock` by name.

```
public static final class Character.UnicodeBlock extends Character.Subset {
// No Constructor
// Public Constants
5.0  public static final Character.UnicodeBlock AEGEAN_NUMBERS;
     public static final Character.UnicodeBlock ALPHABETIC_PRESENTATION_FORMS;
     public static final Character.UnicodeBlock ARABIC;
     public static final Character.UnicodeBlock ARABIC_PRESENTATION_FORMS_A;
     public static final Character.UnicodeBlock ARABIC_PRESENTATION_FORMS_B;
     public static final Character.UnicodeBlock ARMENIAN;
     public static final Character.UnicodeBlock ARROWS;
     public static final Character.UnicodeBlock BASIC_LATIN;
     public static final Character.UnicodeBlock BENGALI;
     public static final Character.UnicodeBlock BLOCK_ELEMENTS;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
     public static final Character.UnicodeBlock BOPOMOFO;
1.4  public static final Character.UnicodeBlock BOPOMOFO_EXTENDED;
     public static final Character.UnicodeBlock BOX_DRAWING;
1.4  public static final Character.UnicodeBlock BRAILLE_PATTERNS;
5.0  public static final Character.UnicodeBlock BUHID;
5.0  public static final Character.UnicodeBlock BYZANTINE_MUSICAL_SYMBOLS;
1.4  public static final Character.UnicodeBlock CHEROKEE;
     public static final Character.UnicodeBlock CJK_COMPATIBILITY;
     public static final Character.UnicodeBlock CJK_COMPATIBILITY_FORMS;
     public static final Character.UnicodeBlock CJK_COMPATIBILITY_IDEOGRAPHS;
5.0  public static final Character.UnicodeBlock CJK_COMPATIBILITY_IDEOGRAPHS_SUPPLEMENT;
1.4  public static final Character.UnicodeBlock CJK_RADICALS_SUPPLEMENT;
     public static final Character.UnicodeBlock CJK_SYMBOLS_AND_PUNCTUATION;
     public static final Character.UnicodeBlock CJK_UNIFIED_IDEOGRAPHS;
1.4  public static final Character.UnicodeBlock CJK_UNIFIED_IDEOGRAPHS_EXTENSION_A;
5.0  public static final Character.UnicodeBlock CJK_UNIFIED_IDEOGRAPHS_EXTENSION_B;
     public static final Character.UnicodeBlock COMBINING_DIACRITICAL_MARKS;
     public static final Character.UnicodeBlock COMBINING_HALF_MARKS;
     public static final Character.UnicodeBlock COMBINING_MARKS_FOR_SYMBOLS;
     public static final Character.UnicodeBlock CONTROL_PICTURES;
     public static final Character.UnicodeBlock CURRENCY_SYMBOLS;
5.0  public static final Character.UnicodeBlock CYPRIOT_SYLLABARY;
     public static final Character.UnicodeBlock CYRILLIC;
5.0  public static final Character.UnicodeBlock CYRILLIC_SUPPLEMENTARY;
5.0  public static final Character.UnicodeBlock DESERET;
     public static final Character.UnicodeBlock DEVANAGARI;
     public static final Character.UnicodeBlock DINGBATS;
     public static final Character.UnicodeBlock ENCLOSED_ALPHANUMERICS;
     public static final Character.UnicodeBlock ENCLOSED_CJK_LETTERS_AND_MONTHS;
1.4  public static final Character.UnicodeBlock ETHIOPIC;
     public static final Character.UnicodeBlock GENERAL_PUNCTUATION;
     public static final Character.UnicodeBlock GEOMETRIC_SHAPES;
     public static final Character.UnicodeBlock GEORGIAN;
5.0  public static final Character.UnicodeBlock GOTHIC;
     public static final Character.UnicodeBlock GREEK;
     public static final Character.UnicodeBlock GREEK_EXTENDED;
     public static final Character.UnicodeBlock GUJARATI;
     public static final Character.UnicodeBlock GURMUKHI;
     public static final Character.UnicodeBlock HALFWIDTH_AND_FULLWIDTH_FORMS;
     public static final Character.UnicodeBlock HANGUL_COMPATIBILITY_JAMO;
     public static final Character.UnicodeBlock HANGUL_JAMO;
     public static final Character.UnicodeBlock HANGUL_SYLLABLES;
5.0  public static final Character.UnicodeBlock HANUNOO;
     public static final Character.UnicodeBlock HEBREW;
5.0  public static final Character.UnicodeBlock HIGH_PRIVATE_USE_SURROGATES;
5.0  public static final Character.UnicodeBlock HIGH_SURROGATES;
     public static final Character.UnicodeBlock HIRAGANA;
1.4  public static final Character.UnicodeBlock IDEOGRAPHIC_DESCRIPTION_CHARACTERS;
     public static final Character.UnicodeBlock IPA_EXTENSIONS;
     public static final Character.UnicodeBlock KANBUN;
1.4  public static final Character.UnicodeBlock KANGXI_RADICALS;
     public static final Character.UnicodeBlock KANNADA;
     public static final Character.UnicodeBlock KATAKANA;
5.0  public static final Character.UnicodeBlock KATAKANA_PHONETIC_EXTENSIONS;
1.4  public static final Character.UnicodeBlock KHMER;
5.0  public static final Character.UnicodeBlock KHMER_SYMBOLS;
     public static final Character.UnicodeBlock LAO;
     public static final Character.UnicodeBlock LATIN_1_SUPPLEMENT;
     public static final Character.UnicodeBlock LATIN_EXTENDED_A;
     public static final Character.UnicodeBlock LATIN_EXTENDED_ADDITIONAL;
     public static final Character.UnicodeBlock LATIN_EXTENDED_B;
     public static final Character.UnicodeBlock LETTERLIKE_SYMBOLS;
5.0  public static final Character.UnicodeBlock LIMBU;
5.0  public static final Character.UnicodeBlock LINEAR_B_IDEOGRAMS;
5.0  public static final Character.UnicodeBlock LINEAR_B_SYLLABARY;
5.0  public static final Character.UnicodeBlock LOW_SURROGATES;
     public static final Character.UnicodeBlock MALAYALAM;
5.0  public static final Character.UnicodeBlock MATHEMATICAL_ALPHANUMERIC_SYMBOLS;
     public static final Character.UnicodeBlock MATHEMATICAL_OPERATORS;
5.0  public static final Character.UnicodeBlock MISCELLANEOUS_MATHEMATICAL_SYMBOLS_A;
5.0  public static final Character.UnicodeBlock MISCELLANEOUS_MATHEMATICAL_SYMBOLS_B;
     public static final Character.UnicodeBlock MISCELLANEOUS_SYMBOLS;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
5.0  public static final Character.UnicodeBlock MISCELLANEOUS_SYMBOLS_AND_ARROWS;
     public static final Character.UnicodeBlock MISCELLANEOUS_TECHNICAL;
1.4  public static final Character.UnicodeBlock MONGOLIAN;
5.0  public static final Character.UnicodeBlock MUSICAL_SYMBOLS;
1.4  public static final Character.UnicodeBlock MYANMAR;
     public static final Character.UnicodeBlock NUMBER_FORMS;
1.4  public static final Character.UnicodeBlock OGHAM;
5.0  public static final Character.UnicodeBlock OLD_ITALIC;
     public static final Character.UnicodeBlock OPTICAL_CHARACTER_RECOGNITION;
     public static final Character.UnicodeBlock ORIYA;
5.0  public static final Character.UnicodeBlock OSMANYA;
5.0  public static final Character.UnicodeBlock PHONETIC_EXTENSIONS;
     public static final Character.UnicodeBlock PRIVATE_USE_AREA;
1.4  public static final Character.UnicodeBlock RUNIC;
5.0  public static final Character.UnicodeBlock SHAVIAN;
1.4  public static final Character.UnicodeBlock SINHALA;
     public static final Character.UnicodeBlock SMALL_FORM_VARIANTS;
     public static final Character.UnicodeBlock SPACING_MODIFIER_LETTERS;
     public static final Character.UnicodeBlock SPECIALS;
     public static final Character.UnicodeBlock SUPERSCRIPTS_AND_SUBSCRIPTS;
5.0  public static final Character.UnicodeBlock SUPPLEMENTAL_ARROWS_A;
5.0  public static final Character.UnicodeBlock SUPPLEMENTAL_ARROWS_B;
5.0  public static final Character.UnicodeBlock SUPPLEMENTAL_MATHEMATICAL_OPERATORS;
5.0  public static final Character.UnicodeBlock SUPPLEMENTARY_PRIVATE_USE_AREA_A;
5.0  public static final Character.UnicodeBlock SUPPLEMENTARY_PRIVATE_USE_AREA_B;
1.4  public static final Character.UnicodeBlock SYRIAC;
5.0  public static final Character.UnicodeBlock TAGALOG;
5.0  public static final Character.UnicodeBlock TAGBANWA;
5.0  public static final Character.UnicodeBlock TAGS;
5.0  public static final Character.UnicodeBlock TAI_LE;
5.0  public static final Character.UnicodeBlock TAI_XUAN_JING_SYMBOLS;
     public static final Character.UnicodeBlock TAMIL;
     public static final Character.UnicodeBlock TELUGU;
1.4  public static final Character.UnicodeBlock THAANA;
     public static final Character.UnicodeBlock THAI;
     public static final Character.UnicodeBlock TIBETAN;
5.0  public static final Character.UnicodeBlock UGARITIC;
1.4  public static final Character.UnicodeBlock UNIFIED_CANADIAN_ABORIGINAL_SYLLABICS;
5.0  public static final Character.UnicodeBlock VARIATION_SELECTORS;
5.0  public static final Character.UnicodeBlock VARIATION_SELECTORS_SUPPLEMENT;
1.4  public static final Character.UnicodeBlock YI_RADICALS;
1.4  public static final Character.UnicodeBlock YI_SYLLABLES;
5.0  public static final Character.UnicodeBlock YIJING_HEXAGRAM_SYMBOLS;
// Public Class Methods
5.0  public static final Character.UnicodeBlock forName(String blockName);
5.0  public static Character.UnicodeBlock of(int codePoint);
     public static Character.UnicodeBlock of(char c);
// Deprecated Public Fields
#    public static final Character.UnicodeBlock SURROGATES_AREA;
}
```

CharSequence java.lang

Java 1.4

This interface defines a simple API for read-only access to sequences of characters. In the core platform it is implemented by the `String`, `StringBuffer` and `java.nio.CharBuffer` classes. `charAt( )` returns the character at a specified position in the sequence. `length( )` returns the number of characters in the sequence. `subSequence( )` returns a `CharSequence` that consists of the characters starting at,

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

and including, the specified *start* index, and continuing up to, but not including the specified *end* index. Finally, `toString( )` returns a `String` version of the sequence.

Note that `CharSequence` implementations do not typically have interoperable `equals( )` or `hashCode( )` methods, and it is not usually possible to compare two `CharSequence` objects or use multiple sequences in a set or hashtable unless they are instances of the same implementing class.

```
public interface CharSequence {
// Public Instance Methods
     char charAt(int index);
     int length( );
     CharSequence subSequence(int start, int end);
     String toString( );
}
```

Implementations

`String`, `StringBuffer`, `StringBuilder`, `java.nio.CharBuffer`

Passed To

Too many methods to list.

Returned By

`String.subSequence( )`, `StringBuffer.subSequence( )`, `java.nio.CharBuffer.subSequence( )`

Class<T> **java.lang**

Java 1.0 ***serializable***

This class represents a Java type. There is one `Class` object for each class that is loaded into the Java Virtual Machine, and, as of Java 1.1, there are special `Class` objects that represent the Java primitive types. The `TYPE` constants defined by `Boolean`, `Integer`, and the other primitive wrapper classes hold these special `Class` objects. Array types are also represented by `Class` objects in Java 1.1.

There is no constructor for this class. You can obtain a `Class` object by calling the `getClass( )` method of any instance of the desired class. In Java 1.1 and later, you can also refer to a `Class` object by appending `.class` to the name of a class. Finally, and most interestingly, a class can be dynamically loaded by passing its fully qualified name (i.e., package name plus class name) to the static `Class.forName( )` method. This method loads the named class (if it is not already loaded) into the Java interpreter and returns a `Class` object for it. Classes can also be loaded with a `ClassLoader` object.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The `newInstance( )` method creates an instance of a given class; this allows you to create instances of dynamically loaded classes for which you cannot use the `new` keyword. Note that this method works only when the target class has a no-argument constructor. See `newInstance( )` in `java.lang.reflect.Constructor` for a more powerful way to instantiate dynamically loaded classes. In Java 5.0, `Class` is a generic type and the type variable *`T`* specifies the type that is returned by the `newInstance( )` method.

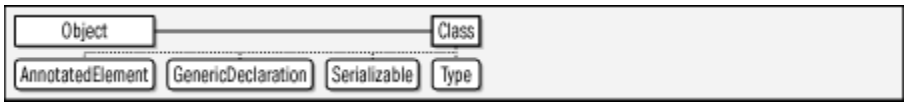
`getName( )` returns the name of the class. `getSuperclass( )` returns its superclass. `isInterface( )` tests whether the `Class` object represents an interface, and `getInterfaces( )` returns an array of the interfaces that this class implements. In Java 1.2 and later, `getPackage( )` returns a `Package` object that represents the package containing the class. `getProtectionDomain( )` returns the `java.security.ProtectionDomain` to which this class belongs. The various other `get( )` and `is( )` methods return other information about the represented class; they form part of the Java Reflection API, along with the classes in `java.lang.reflect`.

Java 5.0 adds a number of methods to support the new language features it defines. `isAnnotation( )` tests whether a type is an annotation type. `Class` implements `java.lang.reflect.AnnotatedElement` in Java 5.0 and the `getAnnotation( )` and related methods allow the retrieval of annotations (with runtime retention) on the class. `isEnum( )` tests whether a `Class` object represents an enumerated type and `getEnumConstants( )` returns an array of the constants defined by an enumerated type. `getTypeParameters( )` returns the type variables declared by a generic type. `getGenericSuperclass( )` and `getGenericInterfaces( )` are the generic variants of the `getSuperclass( )` and `getInterfaces( )` methods, returning the generic type information that appears in the `extends` and `implements` clause of the class declaration. See `java.lang.reflect.Type` for more information.

Java 5.0 also adds methods that are useful for reflection on inner classes. `isMemberClass( )` , `isLocalClass( )`, and `isAnonymousClass( )` determine whether a `Class` represents one of these kinds of nested types. `getEnclosingClass( )` , `getEnclosingMethod( )`, and `getEnclosingConstructor( )` return the type, method, or constructor that an inner class is nested within. Finally, `getSimpleName( )` returns the name of a type as it would appear in Java source code. This is typically more useful than the Java VM formatted names returned by `getName( )`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-10. java.lang.Class<T>




```
public final class Class<T>
        implements Serializable, java.lang.reflect.GenericDeclaration,
        java.lang.reflect.Type, java.lang.reflect.AnnotatedElement {
// No Constructor
// Public Class Methods
     public static Class<?> forName(String className)
     throws ClassNotFoundException;
1.2  public static Class<?> forName(String name, boolean initialize,
        ClassLoader loader) throws ClassNotFoundException;
// Public Instance Methods
5.0  public <U> Class<? extends U> asSubclass(Class<U> clazz);
5.0  public T cast(Object obj);
1.4  public boolean desiredAssertionStatus( );
5.0  public String getCanonicalName( );
1.1  public Class[ ] getClasses( );
     public ClassLoader getClassLoader( );
1.1  public Class<?> getComponentType( );                    native
1.1  public java.lang.reflect.Constructor<T> getConstructor(Class ...
        parameterTypes) throws NoSuchMethodException, SecurityException
1.1  public java.lang.reflect.Constructor[ ] getConstructors( )
        throws SecurityException;
1.1  public Class[ ] getDeclaredClasses( )
        throws SecurityException;
1.1  public java.lang.reflect.Constructor<T> getDeclaredConstructor(Class ...
parameterTypes) throws NoSuchMethodException, SecurityException;
1.1  public java.lang.reflect.Constructor[ ] getDeclaredConstructors( )
        throws SecurityException;
1.1  public java.lang.reflect.Field getDeclaredField(String name)
        throws NoSuchFieldException, SecurityException;
1.1  public java.lang.reflect.Field[ ] getDeclaredFields( )
        throws SecurityException;
1.1  public java.lang.reflect.Method getDeclaredMethod(String name, Class...
parameterTypes) throws NoSuchMethodException, SecurityException;
1.1  public java.lang.reflect.Method[ ] getDeclaredMethods( )
        throws SecurityException;
1.1  public Class<?> getDeclaringClass( );                   native
5.0  public Class<?> getEnclosingClass( );
5.0  public java.lang.reflect.Constructor<?> getEnclosingConstructor( );
5.0  public java.lang.reflect.Method getEnclosingMethod( );
5.0  public T[ ] getEnumConstants( );
1.1  public java.lang.reflect.Field getField(String name)
        throws NoSuchFieldException, SecurityException;
1.1  public java.lang.reflect.Field[ ] getFields( ) throws SecurityException;
5.0  public java.lang.reflect.Type[ ] getGenericInterfaces( );
5.0  public java.lang.reflect.Type getGenericSuperclass( );
     public Class[ ] getInterfaces( );                          native
1.1  public java.lang.reflect.Method getMethod(String name, Class...
parameterTypes) throws NoSuchMethodException, SecurityException;
1.1  public java.lang.reflect.Method[ ] getMethods( ) throws SecurityException;
1.1  public int getModifiers( );                                native
     public String getName( );
1.2  public Package getPackage( );
1.2  public java.security.ProtectionDomain getProtectionDomain( );
1.1  public java.net.URL getResource(String name);
1.1  public java.io.InputStream getResourceAsStream(String name);
1.1  public Object[ ] getSigners( );                          native
5.0  public String getSimpleName( );
     public Class<? super T> getSuperclass( );                  native
5.0  public boolean isAnnotation( );
5.0  public boolean isAnonymousClass( );
1.1  public boolean isArray( );                               native
1.1  public boolean isAssignableFrom(Class<?> cls);         native
5.0  public boolean isEnum( );
1.1  public boolean isInstance(Object obj);                    native
     public boolean isInterface( );                            native
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
5.0  public boolean isLocalClass( );
5.0  public boolean isMemberClass( );
1.1  public boolean isPrimitive( );                                    native
5.0  public boolean isSynthetic( );
     public T newInstance( )
        throws InstantiationException, IllegalAccessException;
// Methods Implementing AnnotatedElement
5.0  public <A extends java.lang.annotation.Annotation> A getAnnotation
(Class<A> annotationClass);
5.0  public java.lang.annotation.Annotation[ ] getAnnotations( );
5.0  public java.lang.annotation.Annotation[ ] getDeclaredAnnotations( );
5.0  public boolean isAnnotationPresent(Class<? extends java.lang.annotation.
Annotation> annotationClass);
// Methods Implementing GenericDeclaration
5.0  public java.lang.reflect.TypeVariable<Class<T>>[ ] getTypeParameters( );
// Public Methods Overriding Object
     public String toString( );
}
```

Passed To

Too many methods to list.

Returned By

Too many methods to list.

Type Of

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

ClassCastException java.lang

Java 1.0 ***serializable unchecked***

Signals an invalid cast of an object to a type of which it is not an instance.

Figure 10-11. java.lang.ClassCastException


```
public class ClassCastException extends RuntimeException {
// Public Constructors
     public ClassCastException( );
     public ClassCastException(String s);
}
```

Thrown By

`org.xml.sax.helpers.ParserFactory.makeParser( )`

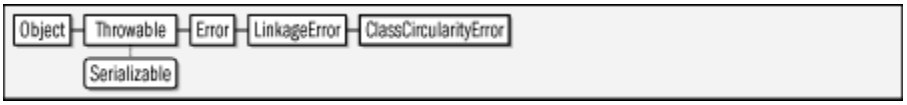
ClassCircularityError java.lang

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.0 ***serializable error***

Signals that a circular dependency has been detected while performing initialization for a class.

Figure 10-12. java.lang.ClassCircularityError




```
public class ClassCircularityError extends LinkageError {
// Public Constructors
     public ClassCircularityError( );
     public ClassCircularityError(String s);
}
```

ClassFormatError **java.lang**

Java 1.0 ***serializable error***

Signals an error in the binary format of a class file.

Figure 10-13. java.lang.ClassFormatError


```
public class ClassFormatError extends LinkageError {
// Public Constructors
     public ClassFormatError( );
     public ClassFormatError(String s);
}
```

Subclasses

`UnsupportedClassVersionError`, `java.lang.reflect.GenericSignatureFormatError`

Thrown By

`ClassLoader.defineClass( )`

ClassLoader **java.lang**

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.0

This class is the abstract superclass of objects that know how to load Java classes into a Java VM. Given a `ClassLoader` object, you can dynamically load a class by calling the public `loadClass( )` method, specifying the full name of the desired class. You can obtain a resource associated with a class by calling `getResource( )`, `getResources( )`, and `getResourceAsStream( )`. Many applications do not need to use `ClassLoader` directly; these applications use the `Class.forName( )` and `Class.getResource( )` methods to dynamically load classes and resources using the `ClassLoader` object that loaded the application itself.

In order to load classes over the network or from any source other than the class path, you must use a custom `ClassLoader` object that knows how to obtain data from that source. A `java.net.URLClassLoader` is suitable for this purpose for almost all applications. Only rarely should an application need to define a `ClassLoader` subclass of its own. When this is necessary, the subclass should typically extend `java.security.SecureClassLoader` and override the `findClass( )` method. This method must find the bytes that comprise the named class, then pass them to the `defineClass( )` method and return the resulting `Class` object. In Java 1.2 and later, the `findClass( )` method must also define the `Package` object associated with the class, if it has not already been defined. It can use `getPackage( )` and `definePackage( )` for this purpose. Custom subclasses of `ClassLoader` should also override `findResource( )` and `findResources( )` to enable the public `getResource( )` and `getResources( )` methods.

In Java 1.4 and later you can specify whether the classes loaded through a `ClassLoader` should have assertions (`assert` statements) enabled. `setDefaultAssertionStatus( )` enables or disables assertions for all loaded classes. `setPackageAssertionStatus( )` and `setClassAssertionStatus( )` allow you to override the default assertion status for a named package or a named class. Finally, `clearAssertionStatus( )` sets the default status to `false` and discards the assertions status for any named packages and classes.

```
public abstract class ClassLoader {
// Protected Constructors
     protected ClassLoader( );
1.2  protected ClassLoader(ClassLoader parent);
// Public Class Methods
1.2  public static ClassLoader getSystemClassLoader( );
1.1  public static java.net.URL getSystemResource(String name);
1.1  public static java.io.InputStream getSystemResourceAsStream(String name);
1.2  public static java.util.Enumeration<java.net.URL> getSystemResources(String name)
throws java.io.IOException;
// Public Instance Methods
1.4  public void clearAssertionStatus( );                        synchronized
1.2  public final ClassLoader getParent( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
1.1  public java.net.URL getResource(String name);
1.1  public java.io.InputStream getResourceAsStream(String name);
1.2  public java.util.Enumeration<java.net.URL> getResources(String name) throws
java.io.IOException;
1.1  public Class<?> loadClass(String name) throws ClassNotFoundException;
1.4  public void setClassAssertionStatus(String className, boolean enabled);     synchronized
1.4  public void setDefaultAssertionStatus(boolean enabled);        synchronized
1.4  public void setPackageAssertionStatus(String packageName, boolean enabled);     synchronized
// Protected Instance Methods
5.0  protected final Class<?> defineClass(String name, java.nio.ByteBuffer b,
java.security.ProtectionDomain protectionDomain)
throws ClassFormatError;
1.1  protected final Class<?> defineClass(String name, byte[ ] b, int off, int len)
throws ClassFormatError;
1.2  protected final Class<?> defineClass(String name, byte[ ] b, int off, int len,
java.security.ProtectionDomain protectionDomain)
throws ClassFormatError;
1.2  protected Package definePackage(String name, String specTitle, String specVersion,
String specVendor, String implTitle, String implVersion, String implVendor, java.net.URL sealBase)
throws IllegalArgumentException;
1.2  protected Class<?> findClass(String name) throws ClassNotFoundException;
1.2  protected String findLibrary(String libname);                  constant
1.1  protected final Class<?> findLoadedClass(String name);
1.2  protected java.net.URL findResource(String name);              constant
1.2  protected java.util.Enumeration<java.net.URL> findResources(String name) throws
java.io.IOException;
     protected final Class<?> findSystemClass(String name) throws ClassNotFoundException;
1.2  protected Package getPackage(String name);
1.2  protected Package[ ] getPackages( );
     protected Class<?> loadClass(String name, boolean resolve)
throws ClassNotFoundException;     synchronized
     protected final void resolveClass(Class<?> c);
1.1  protected final void setSigners(Class<?> c, Object[ ] signers);
// Deprecated Protected Methods
#    protected final Class<?> defineClass(byte[ ] b, int off, int len) throws ClassFormatError;
}
```

Subclasses

java.security.SecureClassLoader

Passed To

Class.forName( ),Thread.setContextClassLoader( ),
java.lang.instrument.ClassFileTransformer.transform( ),
java.lang.instrument.Instrumentation.getInitiatedClasses( ),
java.lang.reflect.Proxy.{getProxyClass( ),newProxyInstance( )},
java.net.URLClassLoader.{newInstance( ),URLClassLoader( )},
java.security.ProtectionDomain.ProtectionDomain( ),
java.security.SecureClassLoader.SecureClassLoader( ),
java.util.ResourceBundle.getBundle( )

Returned By

Class.getClassLoader( ),SecurityManager.currentClassLoader( ),
Thread.getContextClassLoader( ),
java.security.ProtectionDomain.getClassLoader( )

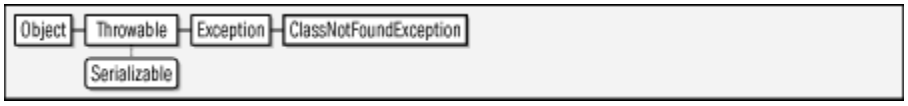
ClassNotFoundException java.lang

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.0 ***serializable checked***

Signals that a class to be loaded cannot be found. If an exception of this type was caused by some underlying exception, you can query that lower-level exeption with `getException( )` or with the newer, more general `getCause( )`.

Figure 10-14. java.lang.ClassNotFoundException




```
public class ClassNotFoundException extends Exception {
// Public Constructors
     public ClassNotFoundException( );
     public ClassNotFoundException(String s);
1.2  public ClassNotFoundException(String s, Throwable ex);
// Public Instance Methods
1.2  public Throwable getException( );                              default:null
// Public Methods Overriding Throwable
1.4  public Throwable getCause( );                                  default:null
}
```

Thrown By

Too many methods to list.

Cloneable **java.lang**

Java 1.0 ***cloneable***

This interface defines no methods or variables, but indicates that the class that implements it may be cloned (i.e., copied) by calling the `Object` method `clone( )`. Calling `clone( )` for an object that does not implement this interface (and does not override `clone( )` with its own implementation) causes a `CloneNotSupportedException` to be thrown.

```
public interface Cloneable {
}
```

Implementations

Too many classes to list.

CloneNotSupportedException **java.lang**

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.0 ***serializable checked***

Signals that the `clone( )` method has been called for an object of a class that does not implement the `Cloneable` interface.

Figure 10-15. java.lang.CloneNotSupportedException


```
public class CloneNotSupportedException extends Exception {
// Public Constructors
     public CloneNotSupportedException( );
     public CloneNotSupportedException(String s);
}
```

Thrown By

`Enum.clone( ),Object.clone( ), java.security.MessageDigest.clone( ), java.security.MessageDigestSpi.clone( ), java.security.Signature.clone( ), java.security.SignatureSpi.clone( ), java.util.AbstractMap.clone( ),java.util.EnumMap.clone( ), java.util.EnumSet.clone( ),javax.crypto.Mac.clone( ), javax.crypto.MacSpi.clone( )`

Comparable<T> java.lang

Java 1.2 ***comparable***

This interface defines a single method, `compareTo( )`, that is responsible for comparing one object to another and determining their relative order, according to some natural ordering for that class of objects. Any general-purpose class that represents a value that can be sorted or ordered should implement this interface. Any class that does implement this interface can make use of various powerful methods such as `java.util.Collections.sort( )` and `java.util.Arrays.binarySearch( )`. Many of the key classes in the Java API implement this interface. In Java 5.0, this interface has been made generic. The type variable *`T`* represents the type of the object that is passed to the `compareTo( )` method.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The `compareTo( )` method compares this object to the object passed as an argument. It should assume that the supplied object is of the appropriate type; if it is not, it should throw a `ClassCastException`. If this object is less than the supplied object or should appear before the supplied object in a sorted list, `compareTo( )` should return a negative number. If this object is greater than the supplied object or should come after the supplied object in a sorted list, `compareTo( )` should return a positive integer. If the two objects are equivalent, and their relative order in a sorted list does not matter, `compareTo( )` should return 0. If `compareTo( )` returns 0 for two objects, the `equals( )` method should typically return `true`. If this is not the case, the `Comparable` objects are not suitable for use in `java.util.TreeSet` and `java.util.TreeMap` classes.

See `java.util.Comparator` for a way to define an ordering for objects that do not implement `Comparable` or to define an ordering other than the natural ordering defined by a `Comparable` class.

```
public interface Comparable<T> {
// Public Instance Methods
     int compareTo(T o);
}
```

Implementations

Too many classes to list.

Compiler — java.lang

Java 1.0

The static methods of this class provide an interface to the just-in-time (JIT) byte-code-to-native code compiler in use by the Java interpreter. If no JIT compiler is in use by the VM, these methods do nothing. `compileClass( )` asks the JIT compiler to compile the specified class. `compileClasses( )` asks the JIT compiler to compile all classes that match the specified name. These methods return `true` if the compilation was successful, or `false` if it failed or if there is no JIT compiler on the system. `enable( )` and `disable( )` turn just-in-time compilation on and off. `command( )` asks the JIT compiler to perform some compiler-specific operation; this is a hook for vendor extensions. No standard operations have been defined.

```
public final class Compiler {
// No Constructor
// Public Class Methods
     public static Object command(Object any);                          native
     public static boolean compileClass(Class<?> clazz);        native
     public static boolean compileClasses(String string);               native
     public static void disable( );                                       native
     public static void enable( );                                        native
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Deprecated java.lang

Java 5.0 ***@Documented @Retention(RUNTIME) annotation***

This annotation type marks the annotated program element as deprecated. The Java compiler issues a warning if the annotated element is used or overrided in code that is not itself `@Deprecated`.

In Java 5.0, the `@Deprecated` annotation works in the same way as the `@deprecated` javadoc tag. In future releases of Java, the compiler may ignore `@deprecated` javadoc tag and rely only on the `@Deprecated` annotation.

This annotation type has source retention and does not have an `@Target` meta-annotation, which means it may be applied to any program element. `Deprecated` has an `@Documented` meta-annotation, meaning that the presence of an `@Deprecated` annotation should be a documented part of the annotated element's API.

Figure 10-16. java.lang.Deprecated


```
public @interface Deprecated {
}
```

Double java.lang

Java 1.0 ***serializable comparable***

This class provides an immutable object wrapper around the `double` primitive data type. `doubleValue( )` returns the primitive `double` value of a `Double` object, and there are other methods (which override `Number` methods and whose names all end in "Value") for returning a the wrapped `double` value as a variety of other primitive types.

This class also provides some useful constants and static methods for testing `double` values. `MIN_VALUE` and `MAX_VALUE` are the smallest (closest to zero) and largest representable `double` values. `POSITIVE_INFINITY` and `NEGATIVE_INFINITY` are the

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

double representations of infinity and negative infinity, and NaN is special double "not a number" value. isInfinite( ) in class and instance method forms tests whether a double or a Double has an infinite value. Similarly, isNaN( ) tests whether a double or Double is not-a-number; this is a comparison that cannot be done directly because the NaN constant never tests equal to any other value, including itself.

The static parseDouble( ) method converts a String to a double. The static valueOf( ) converts a String to a Double, and is basically equivalent to the Double( ) constructor that takes a String argument. The static and instance toString( ) methods perform the opposite conversion: they convert a double or a Double to a String. See also java.text.NumberFormat for more flexible number parsing and formatting.

The compareTo( ) method makes Double object Comparable which is useful for ordering and sorting. The static compare( ) method is similar (its return values have the same meaning as those of Comparable.compareTo( )) but works on primitive values rather than objects and is useful when ordering and sorting arrays of double values.

doubleToLongBits( ), doubleToRawBits( ) and longBitsToDouble( ) allow you to manipulate the bit representation (defined by IEEE 754) of a double directly (which is not something that most applications ever need to do).

Figure 10-17. java.lang.Double


```
public final class Double extends Number implements Comparable<Double> {
// Public Constructors
     public Double(String s) throws NumberFormatException;
     public Double(double value);
// Public Constants
     public static final double MAX_VALUE; =1.7976931348623157E308
     public static final double MIN_VALUE; =4.9E-324
     public static final double NaN;       =NaN
     public static final double NEGATIVE_INFINITY;                    =-Infinity
     public static final double POSITIVE_INFINITY;                    =Infinity
5.0  public static final int SIZE;     =64
1.1  public static final Class<Double> TYPE;
// Public Class Methods
1.4  public static int compare(double d1, double d2);
     public static long doubleToLongBits(double value);               native
1.3  public static long doubleToRawLongBits(double value);         native
     public static boolean isInfinite(double v);
     public static boolean isNaN(double v);
     public static double longBitsToDouble(long bits);                native
1.2  public static double parseDouble(String s) throws NumberFormatException;
5.0  public static String toHexString(double d);
     public static String toString(double d);
     public static Double valueOf(String s) throws NumberFormatException;
5.0  public static Double valueOf(double d);
// Public Instance Methods
     public boolean isInfinite( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public boolean isNaN( );
// Methods Implementing Comparable
1.2 public int compareTo(Double anotherDouble);
// Public Methods Overriding Number
1.1 public byte byteValue( );
    public double doubleValue( );
    public float floatValue( );
    public int intValue( );
    public long longValue( );
1.1 public short shortValue( );
// Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode( );
    public String toString( );
}
```

Enum<E extends Enum<E>> **java.lang**

Java 5.0 ***serializable comparable***

This class is the common superclass of all enumerated types. It is not itself an `enum` type, however, and a Java compiler does not allow other classes to extend it. Subclasses of `Enum` may be only created with `enum` declarations. `Enum` is a generic type, and the type variable *`E`* represents the concrete enumerated type that actually extends `Enum`. This type variable exists so that `Enum` can implement `Comparable<E>`.

Every enumerated constant has a name (the name it was declared with) and an ordinal value—the first constant in an `enum` declaration has an ordinal of 0, the second has an ordinal of 1, and so on. The `final` methods `name( )` and `ordinal( )` return these values. Most users of enumerated constants will use `toString( )` instead of `name( )`. The implementation of `toString( )` defined by `Enum` returns the same value as `name( )`. The `toString( )` method is not `final`, however, and it can be overridden in `enum` declarations.

`Enum` implements a number of `Object` and `Comparable` methods and makes its implementations `final` so that they are inherited by all `enum` types and may not be overridden. `equals( )` compares enumerated constants with the = = operator, and `hashCode( )` returns the `System.identityHashCode( )` value. In order to make this identity-based `equals( )` implementation work, `Enum` overrides the protected `clone( )` method to throw `CloneNotSupportedException`, preventing additional copies of enumerated values from being created. Finally, the `compareTo( )` method of the `Comparable` interface is defined to compare enumerated values based on their `ordinal( )` value.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`getDeclaringClass( )` returns the `Class` object that represents the `enum` type of which the constant is a part. It is like the `getClass( )` method inherited from `Object`, but the return values of these two methods will be different for enumerated constants that have value-specific class bodies, since those constants are instances of an anonymous subclass of the `enum` type.

The static `valueOf( )` method is passed the type and name of an enumerated constant and returns the object that represents that constant (or throws an `IllegalArgumentException`).

Figure 10-18. java.lang.Enum<E extends Enum<E>>


```
public abstract class Enum<E extends Enum<E>> implements Comparable<E>, Serializable {
// Protected Constructors
     protected Enum(String name, int ordinal);
// Public Class Methods
     public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name);
// Public Instance Methods
     public final Class<E> getDeclaringClass( );
     public final String name( );
     public final int ordinal( );
// Methods Implementing Comparable
     public final int compareTo(E o);
// Public Methods Overriding Object
     public final boolean equals(Object other);
     public final int hashCode( );
     public String toString( );
// Protected Methods Overriding Object
     protected final Object clone( ) throws CloneNotSupportedException;
}
```

Subclasses

`Thread.State`, `java.lang.annotation.ElementType`, `java.lang.annotation.RetentionPolicy`, `java.lang.management.MemoryType`, `java.math.RoundingMode`, `java.net.Authenticator.RequestorType`, `java.net.Proxy.Type`, `java.security.KeyRep.Type`, `java.util.Formatter.BigDecimalLayoutForm`, `java.util.concurrent.TimeUnit`, `javax.net.ssl.SSLEngineResult.HandshakeStatus`, `javax.net.ssl.SSLEngineResult.Status`

Passed To

Too many methods to list.

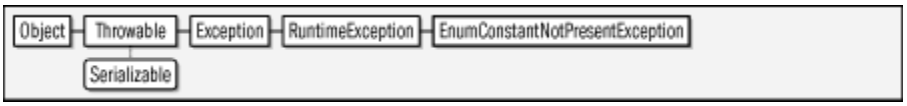
Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

EnumConstantNotPresentException java.lang

Java 5.0 ***serializable unchecked***

This unchecked exception is thrown when Java code attempts to use an enum constant that no longer exists. This can happen only if the enumerated constant was removed from its enumerated type after the referencing code was compiled. The methods of the exception provide the `Class` of the enumerated type and the name of the nonexistent constant.

Figure 10-19. java.lang.EnumConstantNotPresentException




```
public class EnumConstantNotPresentException extends RuntimeException {
// Public Constructors
     public EnumConstantNotPresentException(Class<? extends Enum> enumType,
     String constantName);
// Public Instance Methods
     public String constantName( );
     public Class<? extends Enum> enumType( );
}
```

Error java.lang

Java 1.0 ***serializable error***

This class forms the root of the error hierarchy in Java. Subclasses of `Error`, unlike subclasses of `Exception`, should not be caught and generally cause termination of the program. Subclasses of `Error` need not be declared in the `throws` clause of a method definition. This class inherits methods from `Throwable` but declares none of its own. Each of its constructors simply invokes the corresponding `Throwable( )` constructor. See `Throwable` for details.

Figure 10-20. java.lang.Error


```
public class Error extends Throwable {
// Public Constructors
     public Error( );
1.4  public Error(Throwable cause);
```

```
     public Error(String message);
1.4  public Error(String message, Throwable cause);
}
```

Subclasses

`AssertionError`, `LinkageError`, `ThreadDeath`, `VirtualMachineError`, `java.lang.annotation.AnnotationFormatError`, `java.nio.charset.CoderMalfunctionError`, `javax.xml.parsers.FactoryConfigurationError`, `javax.xml.transform.TransformerFactoryConfigurationError`

Exception — java.lang

Java 1.0 — ***serializable checked***

This class forms the root of the exception hierarchy in Java. An `Exception` signals an abnormal condition that must be specially handled to prevent program termination. Exceptions may be caught and handled. An exception that is not a subclass of `RuntimeException` must be declared in the `throws` clause of any method that can throw it. This class inherits methods from `Throwable` but declares none of its own. Each of its constructors simply invokes the corresponding `Throwable( )` constructor. See `Throwable` for details.

Figure 10-21. java.lang.Exception


```
public class Exception extends Throwable {
// Public Constructors
     public Exception( );
1.4  public Exception(Throwable cause);
     public Exception(String message);
1.4  public Exception(String message, Throwable cause);
}
```

Subclasses

Too many classes to list.

Passed To

`java.io.WriteAbortedException.WriteAbortedException( )`, `java.nio.charset.CoderMalfunctionError.CoderMalfunctionError( )`, `java.security.PrivilegedActionException.PrivilegedActionException( )`, `java.util.logging.ErrorManager.error( )`,

`java.util.logging.Handler.reportError( ),`
`javax.xml.parsers.FactoryConfigurationError.FactoryConfigurationError( ),`
`javax.xml.transform.TransformerFactoryConfigurationError.TransformerFactoryConfigurationError( ),`
`org.xml.sax.SAXException.SAXException( ),`
`org.xml.sax.SAXParseException.SAXParseException( )`

Returned By

`java.security.PrivilegedActionException.getException( ),`
`javax.xml.parsers.FactoryConfigurationError.getException( ),`
`javax.xml.transform.TransformerFactoryConfigurationError.getException( ),org.xml.sax.SAXException.getException( )`

Thrown By

`java.security.PrivilegedExceptionAction.run( ),`
`java.util.concurrent.Callable.call( )`

Type Of

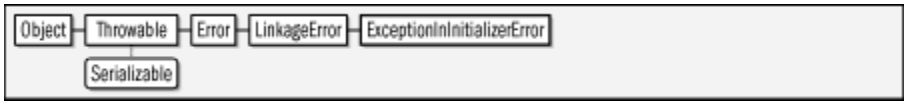
`java.io.WriteAbortedException.detail`

ExceptionInInitializerError java.lang

Java 1.1 ***serializable error***

This error is thrown by the Java Virtual Machine when an exception occurs in the static initializer of a class. You can use the `getException( )` method to obtain the `Throwable` object that was thrown from the initializer. In Java 1.4 and later, `getException( )` has been superseded by the more general `getCause( )` method of the `Throwable` class.

Figure 10-22. java.lang.ExceptionInInitializerError




```
public class ExceptionInInitializerError extends LinkageError {
// Public Constructors
     public ExceptionInInitializerError( );
     public ExceptionInInitializerError(String s);
     public ExceptionInInitializerError(Throwable thrown);
// Public Instance Methods
     public Throwable getException( );                                    default:null
// Public Methods Overriding Throwable
1.4  public Throwable getCause( );                                      default:null
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Float **java.lang**

Java 1.0 ***serializable comparable***

This class provides an immutable object wrapper around a primitive `float` value. `floatValue( )` returns the primitive `float` value of a `Float` object, and there are methods for returning the value of a `Float` as a variety of other primitive types. This class is very similar to `Double`, and defines the same set of useful methods and constants as that class does. See `Double` for details.

Figure 10-23. java.lang.Float


```
public final class Float extends Number implements Comparable<Float> {
// Public Constructors
     public Float(double value);
     public Float(String s) throws NumberFormatException;
     public Float(float value);
// Public Constants
     public static final float MAX_VALUE;  =3.4028235E38
     public static final float MIN_VALUE;  =1.4E-45
     public static final float NaN;        =NaN
     public static final float NEGATIVE_INFINITY;                        =-Infinity
     public static final float POSITIVE_INFINITY;                        =Infinity
5.0  public static final int SIZE;     =32
1.1  public static final Class<Float> TYPE;
// Public Class Methods
1.4  public static int compare(float f1, float f2);
     public static int floatToIntBits(float value);                    native
1.3  public static int floatToRawIntBits(float value);              native
     public static float intBitsToFloat(int bits);                     native
     public static boolean isInfinite(float v);
     public static boolean isNaN(float v);
1.2  public static float parseFloat(String s) throws NumberFormatException;
5.0  public static String toHexString(float f);
     public static String toString(float f);
     public static Float valueOf(String s) throws NumberFormatException;
5.0  public static Float valueOf(float f);
// Public Instance Methods
     public boolean isInfinite( );
     public boolean isNaN( );
// Methods Implementing Comparable
1.2  public int compareTo(Float anotherFloat);
// Public Methods Overriding Number
1.1  public byte byteValue( );
     public double doubleValue( );
     public float floatValue( );
     public int intValue( );
     public long longValue( );
1.1  public short shortValue( );
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

IllegalAccessError java.lang

Java 1.0 ***serializable error***

Signals an attempted use of a class, method, or field that is not accessible.

Figure 10-24. java.lang.IllegalAccessError

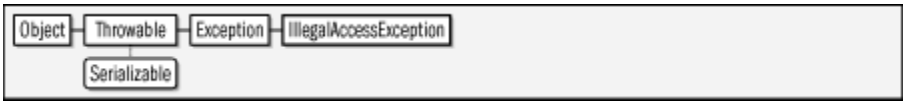

```
public class IllegalAccessError extends IncompatibleClassChangeError {
// Public Constructors
     public IllegalAccessError( );
     public IllegalAccessError(String s);
}
```

IllegalAccessException java.lang

Java 1.0 ***serializable checked***

Signals that a class or initializer is not accessible. Thrown by `Class.newInstance( )`.

Figure 10-25. java.lang.IllegalAccessException




```
public class IllegalAccessException extends Exception {
// Public Constructors
     public IllegalAccessException( );
     public IllegalAccessException(String s);
}
```

Thrown By

Too many methods to list.

IllegalArgumentException java.lang

Java 1.0 ***serializable unchecked***

Signals an illegal argument to a method. See subclasses `IllegalThreadStateException` and `NumberFormatException`.

Figure 10-26. java.lang.IllegalArgumentException


```
public class IllegalArgumentException extends RuntimeException {
// Public Constructors
     public IllegalArgumentException( );
5.0  public IllegalArgumentException(Throwable cause);
     public IllegalArgumentException(String s);
5.0  public IllegalArgumentException(String message, Throwable cause);
}
```

Subclasses

`IllegalThreadStateException`, `NumberFormatException`, `java.nio.channels.IllegalSelectorException`, `java.nio.channels.UnresolvedAddressException`, `java.nio.channels.UnsupportedAddressTypeException`, `java.nio.charset.IllegalCharsetNameException`, `java.nio.charset.UnsupportedCharsetException`, `java.security.InvalidParameterException`, `java.util.IllegalFormatException`, `java.util.regex.PatternSyntaxException`

Thrown By

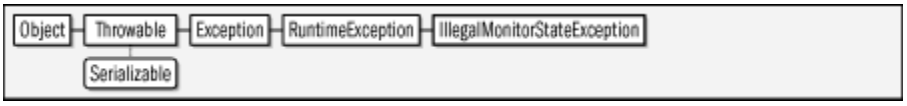
Too many methods to list.

IllegalMonitorStateException **java.lang**

Java 1.0 ***serializable unchecked***

Signals an illegal monitor state. It is thrown by the `Object` `notify( )` and `wait( )` methods used for thread synchronization.

Figure 10-27. java.lang.IllegalMonitorStateException



Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class IllegalMonitorStateException extends RuntimeException {
// Public Constructors
     public IllegalMonitorStateException( );
     public IllegalMonitorStateException(String s);
}
```

IllegalStateException java.lang

Java 1.1 ***serializable unchecked***

Signals that a method has been invoked on an object that is not in an appropriate state to perform the requested operation.

Figure 10-28. java.lang.IllegalStateException


```
public class IllegalStateException extends RuntimeException {
// Public Constructors
     public IllegalStateException( );
5.0  public IllegalStateException(Throwable cause);
     public IllegalStateException(String s);
5.0  public IllegalStateException(String message, Throwable cause);
}
```

Subclasses

java.nio.InvalidMarkException, java.nio.channels.AlreadyConnectedException,java.nio.channels.CancelledKeyException, java.nio.channels.ClosedSelectorException,java.nio.channels.ConnectionPendingException, java.nio.channels.IllegalBlockingModeException,java.nio.channels.NoConnectionPendingException, java.nio.channels.NonReadableChannelException, java.nio.channels.NonWritableChannelException, java.nio.channels.NotYetBoundException,java.nio.channels.NotYetConnectedException, java.nio.channels.OverlappingFileLockException,java.util.FormatterClosedException, java.util.concurrent.CancellationException

Thrown By

Too many methods to list.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

IllegalThreadStateException java.lang

Java 1.0 ***serializable unchecked***

Signals that a thread is not in the appropriate state for an attempted operation to succeed.

Figure 10-29. java.lang.IllegalThreadStateException

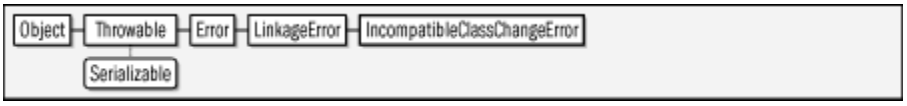

```
public class IllegalThreadStateException extends IllegalArgumentException {
// Public Constructors
     public IllegalThreadStateException( );
     public IllegalThreadStateException(String s);
}
```

IncompatibleClassChangeError java.lang

Java 1.0 ***serializable error***

This is the superclass of a group of related error types. It signals an illegal use of a legal class.

Figure 10-30. java.lang.IncompatibleClassChangeError




```
public class IncompatibleClassChangeError extends LinkageError {
// Public Constructors
     public IncompatibleClassChangeError( );
     public IncompatibleClassChangeError(String s);
}
```

Subclasses

`AbstractMethodError`, `IllegalAccessError`, `InstantiationError`, `NoSuchFieldError`, `NoSuchMethodError`

IndexOutOfBoundsException java.lang

Java 1.0 ***serializable unchecked***

Signals that an index is out of bounds. See the subclasses `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException`.

Figure 10-31. java.lang.IndexOutOfBoundsException


```
public class IndexOutOfBoundsException extends RuntimeException {
// Public Constructors
     public IndexOutOfBoundsException( );
     public IndexOutOfBoundsException(String s);
}
```

Subclasses

`ArrayIndexOutOfBoundsException`, `StringIndexOutOfBoundsException`

InheritableThreadLocal<T> **java.lang**

Java 1.2

This class holds a thread-local value that is inherited by child threads. See `ThreadLocal` for a discussion of thread-local values. Note that the inheritance referred to in the name of this class is not from superclass to subclass; it is inheritance from parent thread to child thread. Like its superclass, this class has been made generic in Java 5.0. The type variable *`T`* represents the type of the referenced object.

This class is best understood by example. Suppose that an application has defined an `InheritableThreadLocal` object and that a certain thread (the parent thread) has a thread-local value stored in that object. Whenever that thread creates a new thread (a child thread), the `InheritableThreadLocal` object is automatically updated so that the new child thread has the same value associated with it as the parent thread. Note that the value associated with the child thread is independent from the value associated with the parent thread. If the child thread subsequently alters its value by calling the `set( )` method of the `InheritableThreadLocal`, the value associated with the parent thread does not change.

By default, a child thread inherits a parent's values unmodified. By overriding the `childValue( )` method, however, you can create a subclass of

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`InheritableThreadLocal` in which the child thread inherits some arbitrary function of the parent thread's value.

Figure 10-32. java.lang.InheritableThreadLocal<T>

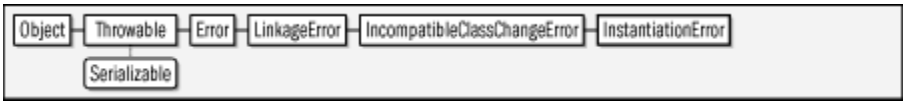

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {
// Public Constructors
     public InheritableThreadLocal( );
// Protected Instance Methods
     protected T childValue(T parentValue);
}
```

InstantiationError java.lang

Java 1.0 ***serializable error***

Signals an attempt to instantiate an interface or abstract class.

Figure 10-33. java.lang.InstantiationError



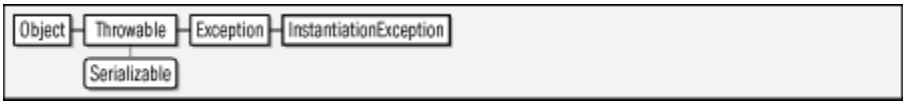

```
public class InstantiationError extends IncompatibleClassChangeError {
// Public Constructors
     public InstantiationError( );
     public InstantiationError(String s);
}
```

InstantiationException java.lang

Java 1.0 ***serializable checked***

Signals an attempt to instantiate an interface or an abstract class.

Figure 10-34. java.lang.InstantiationException



Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class InstantiationException extends Exception {
// Public Constructors
     public InstantiationException( );
     public InstantiationException(String s);
}
```

Thrown By

`Class.newInstance( )`, `java.lang.reflect.Constructor.newInstance( )`, `org.xml.sax.helpers.ParserFactory.makeParser( )`

Integer java.lang

Java 1.0 ***serializable comparable***

This class provides an immutable object wrapper around the `int` primitive data type. This class also contains useful minimum and maximum constants and useful conversion methods. `parseInt( )` and `valueOf( )` convert a string to an `int` or to an `Integer`, respectively. Each can take a radix argument to specify the base the value is represented in. `decode( )` also converts a `String` to an `Integer`. It assumes a hexadecimal number if the string begins with "0X" or "0x", or an octal number if the string begins with "0". Otherwise, a decimal number is assumed. `toString( )` converts in the other direction, and the `static` version takes a radix argument. `toBinaryString( )`, `toOctalString( )`, and `toHexString( )` convert an `int` to a string using base 2, base 8, and base 16. These methods treat the integer as an unsigned value. Other routines return the value of an `Integer` as various primitive types, and, finally, the `getInteger( )` methods return the integer value of a named property from the system property list, or the specified default value.

Java 5.0 adds a number of static methods that operate on the bits of an `int` value. `rotateLeft( )` and `rotateRight( )` shift the bits the specified distance in the specified direction, with bits shifted off one end being shifted in on the other end. `signum( )` returns the sign of the integer as -1, 0, or 1. `highestOneBit( )`, `numberOfTrailingZeros( )`, `bitCount( )` and related methods can be useful if you use an `int` value as a set of bits and want to iterate through the ones bits in the set.

Figure 10-35. java.lang.Integer


```
public final class Integer extends Number implements Comparable<Integer> {
// Public Constructors
     public Integer(int value);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

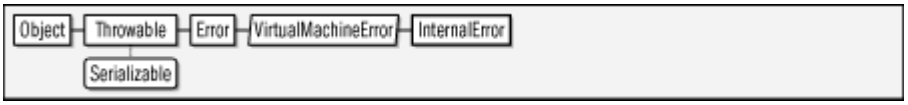
```
     public Integer(String s) throws NumberFormatException;
// Public Constants
     public static final int MAX_VALUE;    =2147483647
     public static final int MIN_VALUE;    =-2147483648
5.0  public static final int SIZE;     =32
1.1  public static final Class<Integer> TYPE;
// Public Class Methods
5.0  public static int bitCount(int i);
1.1  public static Integer decode(String nm) throws NumberFormatException;
     public static Integer getInteger(String nm);
     public static Integer getInteger(String nm, int val);
     public static Integer getInteger(String nm, Integer val);
5.0  public static int highestOneBit(int i);
5.0  public static int lowestOneBit(int i);
5.0  public static int numberOfLeadingZeros(int i);
5.0  public static int numberOfTrailingZeros(int i);
     public static int parseInt(String s) throws NumberFormatException;
     public static int parseInt(String s, int radix) throws NumberFormatException;
5.0  public static int reverse(int i);
5.0  public static int reverseBytes(int i);
5.0  public static int rotateLeft(int i, int distance);
5.0  public static int rotateRight(int i, int distance);
5.0  public static int signum(int i);
     public static String toBinaryString(int i);
     public static String toHexString(int i);
     public static String toOctalString(int i);
     public static String toString(int i);
     public static String toString(int i, int radix);
5.0  public static Integer valueOf(int i);
     public static Integer valueOf(String s) throws NumberFormatException;
     public static Integer valueOf(String s, int radix) throws NumberFormatException;
// Methods Implementing Comparable
1.2  public int compareTo(Integer anotherInteger);
// Public Methods Overriding Number
1.1  public byte byteValue( );
     public double doubleValue( );
     public float floatValue( );
     public int intValue( );
     public long longValue( );
1.1  public short shortValue( );
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```

InternalError java.lang

Java 1.0 ***serializable error***

Signals an internal error in the Java interpreter.

Figure 10-36. java.lang.InternalError




```
public class InternalError extends VirtualMachineError {
// Public Constructors
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
     public InternalError( );
     public InternalError(String s);
}
```

InterruptedException java.lang

Java 1.0 ***serializable checked***

Signals that the thread has been interrupted.

Figure 10-37. java.lang.InterruptedException


```
public class InterruptedException extends Exception {
// Public Constructors
     public InterruptedException( );
     public InterruptedException(String s);
}
```

Thrown By

Too many methods to list.

Iterable<T> java.lang

Java 5.0

This interface defines a single method for returning a `java.util.Iterator` object. `Iterable` was added in Java 5.0 to support the `for/in` loop, which is also new in Java 5.0. The `Collection`, `List`, `Set`, and `Queue` collection interfaces of `java.util` extend this interface, making all collections other than maps `Iterable`. You can implement this interface in your own classes if you want to allow them to be iterated with the `for/in` loop.

The type variable `T` specifies the type parameter of the returned `Iterator` object, which, in turn, specifies the element type of the collection being iterated over.

```
public interface Iterable<T> {
// Public Instance Methods
     java.util.Iterator<T> iterator( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Implementations

`java.util.Collection`

LinkageError java.lang

Java 1.0 ***serializable error***

The superclass of a group of errors that signal problems linking a class or resolving dependencies between classes.

Figure 10-38. java.lang.LinkageError


```
public class LinkageError extends Error {
// Public Constructors
     public LinkageError( );
     public LinkageError(String s);
}
```

Subclasses

`ClassCircularityError`, `ClassFormatError`, `ExceptionInInitializerError`, `IncompatibleClassChangeError`, `NoClassDefFoundError`, `UnsatisfiedLinkError`, `VerifyError`

Long java.lang

Java 1.0 ***serializable comparable***

This class provides an immutable object wrapper around the `long` primitive data type. This class also contains useful minimum and maximum constants and useful conversion methods. `parseLong( )` and `valueOf( )` convert a string to a `long` or to a `Long`, respectively. Each can take a radix argument to specify the base the value is represented in. `toString( )` converts in the other direction and may also take a radix argument. `toBinaryString( )`, `toOctalString( )`, and `toHexString( )` convert a `long` to a string using base 2, base 8, and base 16. These methods treat the `long` as an unsigned value. Other routines return the value of a `Long` as various primitive types, and, finally, the `getLong( )` methods return the `long` value of a named property or the value of the specified default.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 adds a number of static methods that operate on the bits of a `long` value. Except for their argument type and return type, they are the same as the `Integer` methods of the same name.

Figure 10-39. java.lang.Long


```
public final class Long extends Number implements Comparable<Long> {
// Public Constructors
     public Long(long value);
     public Long(String s) throws NumberFormatException;
// Public Constants
     public static final long MAX_VALUE;   =9223372036854775807
     public static final long MIN_VALUE;   =-9223372036854775808
5.0  public static final int SIZE;     =64
1.1  public static final Class<Long> TYPE;
// Public Class Methods
5.0  public static int bitCount(long i);
1.2  public static Long decode(String nm) throws NumberFormatException;
     public static Long getLong(String nm);
     public static Long getLong(String nm, Long val);
     public static Long getLong(String nm, long val);
5.0  public static long highestOneBit(long i);
5.0  public static long lowestOneBit(long i);
5.0  public static int numberOfLeadingZeros(long i);
5.0  public static int numberOfTrailingZeros(long i);
     public static long parseLong(String s) throws NumberFormatException;
     public static long parseLong(String s, int radix) throws NumberFormatException;
5.0  public static long reverse(long i);
5.0  public static long reverseBytes(long i);
5.0  public static long rotateLeft(long i, int distance);
5.0  public static long rotateRight(long i, int distance);
5.0  public static int signum(long i);
     public static String toBinaryString(long i);
     public static String toHexString(long i);
     public static String toOctalString(long i);
     public static String toString(long i);
     public static String toString(long i, int radix);
5.0  public static Long valueOf(long l);
     public static Long valueOf(String s) throws NumberFormatException;
     public static Long valueOf(String s, int radix) throws NumberFormatException;
// Methods Implementing Comparable
1.2  public int compareTo(Long anotherLong);
// Public Methods Overriding Number
1.1  public byte byteValue( );
     public double doubleValue( );
     public float floatValue( );
     public int intValue( );
     public long longValue( );
1.1  public short shortValue( );
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.0

This class defines constants for the mathematical values *e* and π and defines static methods for floating-point trigonometry, exponentiation, and other operations. It is the equivalent of the C <*math.h*> functions. It also contains methods for computing minimum and maximum values and for generating pseudorandom numbers.

Most methods of `Math` operate on `float` and `double` floating-point values. Remember that these values are only approximations of actual real numbers. To allow implementations to take full advantage of the floating-point capabilities of a native platform, the methods of `Math` are not required to return exactly the same values on all platforms. In other words, the results returned by different implementations may differ slightly in the least-significant bits. As of Java 1.3, applications that require strict platform-independence of results should use `StrictMath` instead.

Java 5.0 adds several methods including `log10( )` to compute the base-ten logarithm, `cbrt( )` to compute the cube root of a number, and `signum( )` to compute the sign of a number as well as `sinh( )` , `cosh( )`, and `tanh( )` hyperbolic trigonometric functions.

```
public final class Math {
// No Constructor
// Public Constants
     public static final double E;         =2.718281828459045
     public static final double PI;        =3.141592653589793
// Public Class Methods
     public static int abs(int a);
     public static long abs(long a);
     public static float abs(float a);
     public static double abs(double a);
     public static double acos(double a);
     public static double asin(double a);
     public static double atan(double a);
     public static double atan2(double y, double x);
5.0  public static double cbrt(double a);
     public static double ceil(double a);
     public static double cos(double a);
5.0  public static double cosh(double x);
     public static double exp(double a);
5.0  public static double expm1(double x);
     public static double floor(double a);
5.0  public static double hypot(double x, double y);
     public static double IEEEremainder(double f1, double f2);
     public static double log(double a);
5.0  public static double log10(double a);
5.0  public static double log1p(double x);
     public static int max(int a, int b);
     public static long max(long a, long b);
     public static float max(float a, float b);
     public static double max(double a, double b);
     public static int min(int a, int b);
     public static long min(long a, long b);
     public static float min(float a, float b);
     public static double min(double a, double b);
     public static double pow(double a, double b);
     public static double random( );
     public static double rint(double a);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

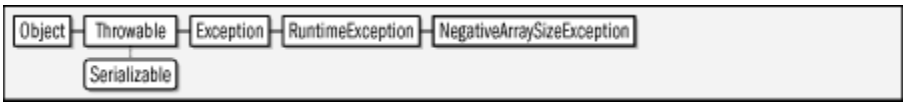
```
     public static int round(float a);
     public static long round(double a);
5.0  public static float signum(float f);
5.0  public static double signum(double d);
     public static double sin(double a);
5.0  public static double sinh(double x);
     public static double sqrt(double a);
     public static double tan(double a);
5.0  public static double tanh(double x);
1.2  public static double toDegrees(double angrad);
1.2  public static double toRadians(double angdeg);
5.0  public static float ulp(float f);
5.0  public static double ulp(double d);
}
```

NegativeArraySizeException java.lang

Java 1.0 ***serializable unchecked***

Signals an attempt to allocate an array with fewer than zero elements.

Figure 10-40. java.lang.NegativeArraySizeException




```
public class NegativeArraySizeException extends RuntimeException {
// Public Constructors
     public NegativeArraySizeException( );
     public NegativeArraySizeException(String s);
}
```

Thrown By

`java.lang.reflect.Array.newInstance( )`

NoClassDefFoundError java.lang

Java 1.0 ***serializable error***

Signals that the definition of a specified class cannot be found.

Figure 10-41. java.lang.NoClassDefFoundError

```
public class NoClassDefFoundError extends LinkageError {
// Public Constructors
     public NoClassDefFoundError( );
     public NoClassDefFoundError(String s);
}
```

NoSuchFieldError java.lang

Java 1.0 ***serializable error***

Signals that a specified field cannot be found.

Figure 10-42. java.lang.NoSuchFieldError


```
public class NoSuchFieldError extends IncompatibleClassChangeError {
// Public Constructors
     public NoSuchFieldError( );
     public NoSuchFieldError(String s);
}
```

NoSuchFieldException java.lang

Java 1.1 ***serializable checked***

This exception signals that the specified field does not exist in the specified class.

Figure 10-43. java.lang.NoSuchFieldException


```
public class NoSuchFieldException extends Exception {
// Public Constructors
     public NoSuchFieldException( );
     public NoSuchFieldException(String s);
}
```

Thrown By

`Class.{getDeclaredField( ),getField( )}`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

NoSuchMethodError java.lang

Java 1.0 ***serializable error***

Signals that a specified method cannot be found.

Figure 10-44. java.lang.NoSuchMethodError

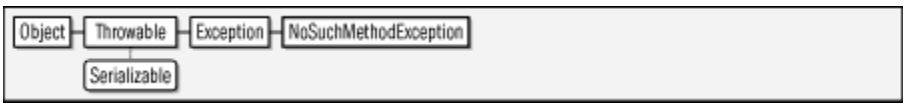

```
public class NoSuchMethodError extends IncompatibleClassChangeError {
// Public Constructors
     public NoSuchMethodError( );
     public NoSuchMethodError(String s);
}
```

NoSuchMethodException java.lang

Java 1.0 ***serializable checked***

Signals that the specified method does not exist in the specified class.

Figure 10-45. java.lang.NoSuchMethodException




```
public class NoSuchMethodException extends Exception {
// Public Constructors
     public NoSuchMethodException( );
     public NoSuchMethodException(String s);
}
```

Thrown By

`Class.{getConstructor( ),getDeclaredConstructor( ), getDeclaredMethod( ),getMethod( )}`

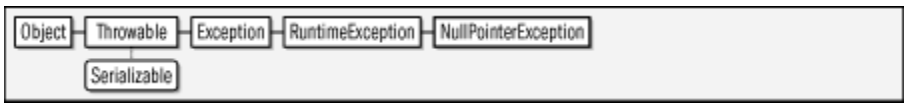
NullPointerException java.lang

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.0 ***serializable unchecked***

Signals an attempt to access a field or invoke a method of a `null` object.

Figure 10-46. java.lang.NullPointerException




```
public class NullPointerException extends RuntimeException {
// Public Constructors
     public NullPointerException( );
     public NullPointerException(String s);
}
```

Thrown By

`org.xml.sax.helpers.ParserFactory.makeParser( )`

Number **java.lang**

Java 1.0 ***serializable***

This is an abstract class that is the superclass of `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`. It defines the conversion functions those types implement.

Figure 10-47. java.lang.Number


```
public abstract class Number implements Serializable {
// Public Constructors
     public Number( );
// Public Instance Methods
1.1  public byte byteValue( );
     public abstract double doubleValue( );
     public abstract float floatValue( );
     public abstract int intValue( );
     public abstract long longValue( );
1.1  public short shortValue( );
}
```

Subclasses

`Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.concurrent.atomic.AtomicInteger`, `java.util.concurrent.atomic.AtomicLong`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Returned By

`java.text.ChoiceFormat.parse( )`, `java.text.DecimalFormat.parse( )`, `java.text.NumberFormat.parse( )`, `javax.xml.datatype.Duration.getField( )`

NumberFormatException java.lang

Java 1.0 ***serializable unchecked***

Signals an illegal number format.

Figure 10-48. java.lang.NumberFormatException


```
public class NumberFormatException extends IllegalArgumentException {
// Public Constructors
     public NumberFormatException( );
     public NumberFormatException(String s);
}
```

Thrown By

Too many methods to list.

Object java.lang

Java 1.0

This is the root class in Java. All classes are subclasses of `Object`, and thus all objects can invoke the `public` and `protected` methods of this class. For classes that implement the `Cloneable` interface, `clone( )` makes a byte-for-byte copy of an `Object`. `getClass( )` returns the `Class` object associated with any `Object`, and the `notify( )`, `notifyAll( )`, and `wait( )` methods are used for thread synchronization on a given `Object`.

A number of these `Object` methods should be overridden by subclasses of `Object`. For example, a subclass should provide its own definition of the `toString( )` method so that it can be used with the string concatenation operator and with the `PrintWriter.println( )` methods. Defining the `toString( )` method for all objects also helps with debugging.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The default implementation of the `equals( )` method simply uses the `= =` operator to test whether this object reference and the specified object reference refer to the same object. Many subclasses override this method to compare the individual fields of two distinct objects (i.e., they override the method to test for the equivalence of distinct objects rather than the equality of object references). Some classes, particularly those that override `equals( )`, may also want to override the `hashCode( )` method to provide an appropriate hashcode to be used when storing instances in a `Hashtable` data structure.

A class that allocates system resources other than memory (such as file descriptors or windowing system graphic contexts) should override the `finalize( )` method to release these resources when the object is no longer referred to and is about to be garbage-collected.

```
public class Object {
// Public Constructors
     public Object( );                                                    empty
// Public Instance Methods
     public boolean equals(Object obj);
     public final Class<? extends Object> getClass( );           native
     public int hashCode( );                                              native
     public final void notify( );                                         native
     public final void notifyAll( );                                      native
     public String toString( );
     public final void wait( ) throws InterruptedException;
     public final void wait(long timeout) throws InterruptedException;     native
     public final void wait(long timeout, int nanos) throws InterruptedException;
// Protected Instance Methods
     protected Object clone( ) throws CloneNotSupportedException;         native
     protected void finalize( ) throws Throwable;                         empty
}
```

Subclasses

Too many classes to list.

Passed To

Too many methods to list.

Returned By

Too many methods to list.

Type Of

`java.io.Reader.lock`, `java.io.Writer.lock`, `java.util.EventObject.source`, `java.util.Vector.elementData`, `java.util.prefs.AbstractPreferences.lock`

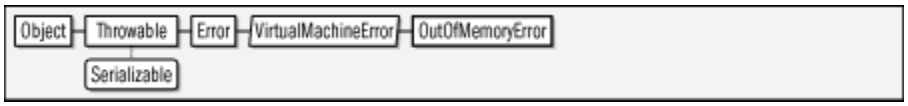
OutOfMemoryError — java.lang

Java 1.0 — ***serializable error***

Signals that the interpreter has run out of memory (and that garbage collection is unable to free any memory).

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-49. java.lang.OutOfMemoryError




```
public class OutOfMemoryError extends VirtualMachineError {
// Public Constructors
     public OutOfMemoryError( );
     public OutOfMemoryError(String s);
}
```

Override java.lang

Java 5.0 ***@Target(METHOD) @Retention(SOURCE) annotation***

An annotation of this type may be applied to methods and indicates that the programmer intends for the method to override a method from a superclass. In effect, it is an assertion for the compiler to verify. If a method annotated `@Override` does not, in fact, override another method (perhaps because the method name was misspelled or an argument was incorrectly typed), the compiler issues an error. This annotation type has source retention.

Figure 10-50. java.lang.Override


```
public @interface Override {
}
```

Package java.lang

Java 1.2

This class represents a Java package. You can obtain the `Package` object for a given `Class` by calling the `getPackage( )` method of the `Class` object. The static `Package.getPackage( )` method returns a `Package` object for the named package, if any such package has been loaded by the current class loader. Similarly, the static `Package.getPackages( )` returns all `Package` objects that have been loaded by the current class loader. Note that a `Package` object is not defined unless at least one class

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

has been loaded from that package. Although you can obtain the `Package` of a given `Class`, you cannot obtain an array of `Class` objects contained in a specified `Package`.

If the classes that comprise a package are contained in a JAR file that has the appropriate attributes set in its manifest file, the `Package` object allows you to query the title, vendor, and version of both the package specification and the package implementation; all six values are strings. The specification version string has a special format. It consists of one or more integers, separated from each other by periods. Each integer can have leading zeros, but is not considered an octal digit. Increasing numbers indicate later versions. The `isCompatibleWith( )` method calls `getSpecificationVersion( )` to obtain the specification version and compares it with the version string supplied as an argument. If the package-specification version is the same as or greater than the specified string, `isCompatibleWith( )` returns `true`. This allows you to test whether the version of a package (typically a standard extension) is new enough for the purposes of your application.

Packages may be sealed, which means that all classes in the package must come from the same JAR file. If a package is sealed, the no-argument version of `isSealed( )` returns `true`. The one-argument version of `isSealed( )` returns `true` if the specified URL represents the JAR file from which the package is loaded.

Figure 10-51. java.lang.Package


```
public class Package implements java.lang.reflect.AnnotatedElement {
// No Constructor
// Public Class Methods
     public static Package getPackage(String name);
     public static Package[ ] getPackages( );
// Public Instance Methods
     public String getImplementationTitle( );
     public String getImplementationVendor( );
     public String getImplementationVersion( );
     public String getName( );
     public String getSpecificationTitle( );
     public String getSpecificationVendor( );
     public String getSpecificationVersion( );
     public boolean isCompatibleWith(String desired) throws NumberFormatException;
     public boolean isSealed( );
     public boolean isSealed(java.net.URL url);
// Methods Implementing AnnotatedElement
5.0  public <A extends java.lang.annotation.Annotation> A getAnnotation(Class<A>
annotationClass);
5.0  public java.lang.annotation.Annotation[ ] getAnnotations( );
5.0  public java.lang.annotation.Annotation[ ] getDeclaredAnnotations( );
5.0  public boolean isAnnotationPresent(Class<? extends java.lang.annotation.
Annotation> annotationClass);
// Public Methods Overriding Object
     public int hashCode( );
     public String toString( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Returned By

`Class.getPackage( )`, `ClassLoader.{definePackage( )`, `getPackage( )`, `getPackages( )}`, `java.net.URLClassLoader.definePackage( )`

Process java.lang

Java 1.0

This class describes a process that is running externally to the Java interpreter. Note that a `Process` is very different from a `Thread`; the `Process` class is abstract and cannot be instantiated. Call one of the `Runtime.exec( )` methods to start a process and return a corresponding `Process` object.

`waitFor( )` blocks until the process exits. `exitValue( )` returns the exit code of the process. `destroy( )` kills the process. `getErrorStream( )` returns an `InputStream` from which you can read any bytes the process sends to its standard error stream. `getInputStream( )` returns an `InputStream` from which you can read any bytes the process sends to its standard output stream. `getOutputStream( )` returns an `OutputStream` you can use to send bytes to the standard input stream of the process.

```
public abstract class Process {
// Public Constructors
     public Process( );
// Public Instance Methods
     public abstract void destroy( );
     public abstract int exitValue( );
     public abstract java.io.InputStream getErrorStream( );
     public abstract java.io.InputStream getInputStream( );
     public abstract java.io.OutputStream getOutputStream( );
     public abstract int waitFor( ) throws InterruptedException;
}
```

Returned By

`ProcessBuilder.start( )`, `Runtime.exec( )`

ProcessBuilder java.lang

Java 5.0

This class launches operating system processes, producing `Process` objects. Specify the operating system command when you invoke the `ProcessBuilder( )` constructor or with the `command( )` method. Commands are specified with one or more strings, typically the filename of the executable to run followed by the command-line arguments for the

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

executable. Specify these strings in a `List`, a `String[ ]`, or, most conveniently, using a variable-length argument list of strings.

Before launching the command you have specified, you can configure the `ProcessBuilder`. Query the current working directory with the no-argument version of `directory( )` and set it with the one-argument version of the method. Query the mapping of environment variables to values with the `environment( )` method. You can alter the mappings in the returned `Map` to specify the environment you want the child process to run in. Pass `true` to `redirectErrorStream( )` if you would like both the standard output and the standard error stream of the child process to be merged into a single stream that you can obtain with `Process.getInputStream( )`. If you do so, you do not have to arrange to read two separate input streams to get the output of the process.

Once you have specified a command and configured your `ProcessBuilder` as desired, call the `start( )` method to launch the process. You then use methods of the returned `Process` to provide input to the process, read output from the process, or wait for the process to exit. `start( )` may throw an `IOException`. This may occur, for example, if the executable filename you have specified does not exist. The `command( )` and `directory( )` methods do not perform error checking on the values you provide them; these checks are performed by the `start( )` method, so it is also possible for `start( )` to throw exceptions based on bad input to the configuration methods.

Note that a `ProcessBuilder` can be reused: once you have established a working directory and environment variables, you can change the `command( )` and launch multiple processes with repeated calls to `start( )`.

```
public final class ProcessBuilder {
// Public Constructors
     public ProcessBuilder(java.util.List<String> command);
     public ProcessBuilder(String... command);
// Public Instance Methods
     public java.util.List<String> command( );
     public ProcessBuilder command(String... command);
     public ProcessBuilder command(java.util.List<String> command);
     public java.io.File directory( );
     public ProcessBuilder directory(java.io.File directory);
     public java.util.Map<String,String> environment( );
     public boolean redirectErrorStream( );
     public ProcessBuilder redirectErrorStream(boolean redirectErrorStream);
     public Process start( ) throws java.io.IOException;
}
```

Readable **java.lang**

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***readable***

Objects that implement this interface can serve as a source of characters and can transfer one or more at a time to a `java.nio.CharBuffer`. `Readable` was added in Java 5.0 as a simple unifying API for `java.nio.CharBuffer` and character input stream subclasses of `java.io.Reader`. The `java.util.Scanner` class can parse input from any `Readable` object. See also `Appendable`.

```
public interface Readable {
// Public Instance Methods
     int read(java.nio.CharBuffer cb) throws java.io.IOException;
}
```

Implementations

`java.io.Reader`, `java.nio.CharBuffer`

Passed To

`java.util.Scanner.Scanner( )`

Runnable **java.lang**

Java 1.0 ***runnable***

This interface specifies the `run( )` method that is required to use with the `Thread` class. Any class that implements this interface can provide the body of a thread. See `Thread` for more information.

```
public interface Runnable {
// Public Instance Methods
     void run( );
}
```

Implementations

`Thread`, `java.util.TimerTask`, `java.util.concurrent.FutureTask`

Passed To

Too many methods to list.

Returned By

`javax.net.ssl.SSLEngine.getDelegatedTask( )`

Runtime **java.lang**

Java 1.0

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This class encapsulates a number of platform-dependent system functions. The static method `getRuntime( )` returns the `Runtime` object for the current platform; this object can perform system functions in a platform-independent way.

`exit( )` causes the Java interpreter to exit and return a specified return code. This method is usually invoked through `System.exit( )`. In Java 1.3, `addShutdownHook( )` registers an unstarted `Thread` object that is run when the virtual machine shuts down, either through a call to `exit( )` or through a user interrupt (a CTRL-C, for example). The purpose of a shutdown hook is to perform necessary cleanup, such as shutting down network connections, deleting temporary files, and so on. Any number of hooks can be registered with `addShutdownHook( )`. Before the interpreter exits, it starts all registered shutdown-hook threads and lets them run concurrently. Any hooks you write should perform their cleanup operation and exit promptly so they do not delay the shutdown process. To remove a shutdown hook before it is run, call `removeShutdownHook( )`. To force an immediate exit that does not invoke the shutdown hooks, call `halt( )`.

`exec( )` starts a new process running externally to the interpreter. Note that any processes run outside of Java may be system-dependent.

`freeMemory( )` returns the approximate amount of free memory. `totalMemory( )` returns the total amount of memory available to the Java interpreter. `gc( )` forces the garbage collector to run synchronously, which may free up more memory. Similarly, `runFinalization( )` forces the `finalize( )` methods of unreferenced objects to be run immediately. This may free up system resources those objects were holding.

`load( )` loads a dynamic library with a fully specified pathname. `loadLibrary( )` loads a dynamic library with only the library name specified; it looks in platform-dependent locations for the specified library. These libraries generally contain native code definitions for native methods.

`traceInstructions( )` and `traceMethodCalls( )` enable and disable tracing by the interpreter. These methods are used for debugging or profiling an application. It is not specified how the VM emits the trace information, and VMs are not even required to support this feature.

Note that some of the `Runtime` methods are more commonly called via the static methods of the `System` class.

```
public class Runtime {
// No Constructor
// Public Class Methods
     public static Runtime getRuntime( );
// Public Instance Methods
1.3  public void addShutdownHook(Thread hook);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
1.4  public int availableProcessors( );                              native
     public Process exec(String[ ] cmdarray) throws java.io.IOException;
     public Process exec(String command) throws java.io.IOException;
     public Process exec(String command, String[ ] envp) throws java.io.IOException;
     public Process exec(String[ ] cmdarray, String[ ] envp) throws java.io.IOException;
1.3  public Process exec(String[ ] cmdarray, String[ ] envp, java.io.File dir)
     throws java.io.IOException;
1.3  public Process exec(String command, String[ ] envp, java.io.File dir) throws
     java.io.IOException;
     public void exit(int status);
     public long freeMemory( );                                        native
     public void gc( );                                                native
1.3  public void halt(int status);
     public void load(String filename);
     public void loadLibrary(String libname);
1.4  public long maxMemory( );                                       native
1.3  public boolean removeShutdownHook(Thread hook);
     public void runFinalization( );
     public long totalMemory( );                                       native
     public void traceInstructions(boolean on);                      native
     public void traceMethodCalls(boolean on);                       native
// Deprecated Public Methods
#    public java.io.InputStream getLocalizedInputStream(java.io.InputStream in);
#    public java.io.OutputStream getLocalizedOutputStream(java.io.OutputStream out);
1.1#  public static void runFinalizersOnExit(boolean value);
}
```

RuntimeException java.lang

Java 1.0 ***serializable unchecked***

This exception type is not used directly, but serves as a superclass of a group of run-time exceptions that need not be declared in the `throws` clause of a method definition. These exceptions need not be declared because they are runtime conditions that can generally occur in any Java method. Thus, declaring them would be unduly burdensome, and Java does not require it.

This class inherits methods from `Throwable` but declares none of its own. Each of the `RuntimeException` constructors simply invokes the corresponding `Exception( )` and `Throwable( )` constructor. See `Throwable` for details.

Figure 10-52. java.lang.RuntimeException


```
public class RuntimeException extends Exception {
// Public Constructors
     public RuntimeException( );
1.4  public RuntimeException(Throwable cause);
     public RuntimeException(String message);
1.4  public RuntimeException(String message, Throwable cause);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Subclasses

Too many classes to list.

RuntimePermission — java.lang

Java 1.2 — ***serializable permission***

This class is a `java.security.Permission` that represents access to various important system facilities. A `RuntimePermission` has a name, or target, that represents the facility for which permission is being sought or granted. The name "exitVM" represents permission to call `System.exit( )`, and the name "accessClassInPackage.java.lang" represents permission to read classes from the `java.lang` package. The name of a `RuntimePermission` may use a ".*" suffix as a wildcard. For example, the name "accessClassInPackage.java.*" represents permission to read classes from any package whose name begins with "java.". `RuntimePermission` does not use action list strings as some `Permission` classes do; the name of the permission alone is enough.

The following are supported `RuntimePermssion` names:

accessClassInPackage.*package*	getProtectionDomain	setFactory
accessDeclaredMembers	loadLibrary.*library_name*	setIO
createClassLoader	modifyThread	setSecurityManager
createSecurityManager	modifyThreadGroup	stopThread
defineClassInPackage.*package*	queuePrintJob	writeFileDescriptor
exitVM	readFileDescriptor	
getClassLoader	set-ContextClassLoader	

System administrators configuring security policies should be familiar with these permission names, the operations they govern access to, and with the risks inherent in granting any of them. Although system programmers may need to work with this class, application programmers should never need to use `RuntimePermssion` directly.

Figure 10-53. java.lang.RuntimePermission


```
public final class RuntimePermission extends java.security.BasicPermission {
// Public Constructors
     public RuntimePermission(String name);
     public RuntimePermission(String name, String actions);
}
```

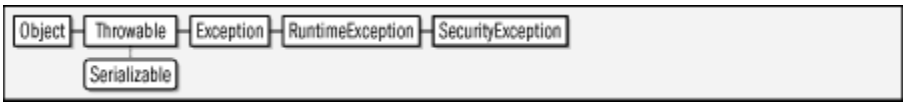

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

SecurityException **java.lang**

Java 1.0 ***serializable unchecked***

Signals that an operation is not permitted for security reasons.

Figure 10-54. java.lang.SecurityException




```
public class SecurityException extends RuntimeException {
// Public Constructors
     public SecurityException( );
5.0  public SecurityException(Throwable cause);
     public SecurityException(String s);
5.0  public SecurityException(String message, Throwable cause);
}
```

Subclasses

`java.security.AccessControlException`

Thrown By

Too many methods to list.

SecurityManager **java.lang**

Java 1.0

This class defines the methods necessary to implement a security policy for the safe execution of untrusted code. Before performing potentially sensitive operations, Java calls methods of the `SecurityManager` object currently in effect to determine whether the operations are permitted. These methods throw a `SecurityException` if the operation is not permitted. Typical applications do not need to use or subclass `SecurityManager`. It is typically used only by web browsers, applet viewers, and other programs that need to run untrusted code in a controlled environment.

Prior to Java 1.2, this class is `abstract`, and the default implementation of each `check( )` method throws a `SecurityException` unconditionally. The Java security mechanism has been overhauled as of Java 1.2. As part of the overhaul, this class is no longer `abstract` and its methods have useful default implementations, so there is rarely

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

a need to subclass it. `checkPermission( )` operates by invoking the `checkPermission( )` method of the system `java.security.AccessController` object. In Java 1.2 and later, all other `check( )` methods of `SecurityManager` are now implemented on top of `checkPermission( )`.

```
public class SecurityManager {
// Public Constructors
     public SecurityManager( );
// Public Instance Methods
     public void checkAccept(String host, int port);
     public void checkAccess(ThreadGroup g);
     public void checkAccess(Thread t);
1.1  public void checkAwtEventQueueAccess( );
     public void checkConnect(String host, int port);
     public void checkConnect(String host, int port, Object context);
     public void checkCreateClassLoader( );
     public void checkDelete(String file);
     public void checkExec(String cmd);
     public void checkExit(int status);
     public void checkLink(String lib);
     public void checkListen(int port);
1.1  public void checkMemberAccess(Class<?> clazz, int which);
1.1  public void checkMulticast(java.net.InetAddress maddr);
     public void checkPackageAccess(String pkg);
     public void checkPackageDefinition(String pkg);
1.2  public void checkPermission(java.security.Permission perm);
1.2  public void checkPermission(java.security.Permission perm, Object context);
1.1  public void checkPrintJobAccess( );
     public void checkPropertiesAccess( );
     public void checkPropertyAccess(String key);
     public void checkRead(String file);
     public void checkRead(java.io.FileDescriptor fd);
     public void checkRead(String file, Object context);
1.1  public void checkSecurityAccess(String target);
     public void checkSetFactory( );
1.1  public void checkSystemClipboardAccess( );
     public boolean checkTopLevelWindow(Object window);
     public void checkWrite(java.io.FileDescriptor fd);
     public void checkWrite(String file);
     public Object getSecurityContext( );                    default:AccessControlContext
1.1  public ThreadGroup getThreadGroup( );
// Protected Instance Methods
     protected Class[ ] getClassContext( );                       native
// Deprecated Public Methods
1.1#  public void checkMulticast(java.net.InetAddress maddr, byte ttl);
#    public boolean getInCheck( );                                 default:false
// Deprecated Protected Methods
#    protected int classDepth(String name);                       native
#    protected int classLoaderDepth( );
#    protected ClassLoader currentClassLoader( );
1.1#  protected Class<?> currentLoadedClass( );
#    protected boolean inClass(String name);
#    protected boolean inClassLoader( );
// Deprecated Protected Fields
#    protected boolean inCheck;
}
```

Passed To

`System.setSecurityManager( )`

Returned By

`System.getSecurityManager( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Short **java.lang**

Java 1.1 ***serializable comparable***

This class provides an object wrapper around the `short` primitive type. It defines useful constants for the minimum and maximum values that can be stored by the `short` type, and also a `Class` object constant that represents the `short` type. It also provides various methods for converting `Short` values to and from strings and other numeric types.

Most of the static methods of this class can convert a `String` to a `Short` object or a `short` value; the four `parseShort( )` and `valueOf( )` methods parse a number from the specified string using an optionally specified radix and return it in one of these two forms. The `decode( )` method parses a number specified in base 10, base 8, or base 16 and returns it as a `Short`. If the string begins with "0x" or "#", it is interpreted as a hexadecimal number; if it begins with "0", it is interpreted as an octal number. Otherwise, it is interpreted as a decimal number.

Note that this class has two different `toString( )` methods. One is static and converts a `short` primitive value to a string. The other is the usual `toString( )` method that converts a `Short` object to a string. Most of the remaining methods convert a `Short` to various primitive numeric types.

Figure 10-55. java.lang.Short


```
public final class Short extends Number implements Comparable<Short> {
// Public Constructors
     public Short(short value);
     public Short(String s) throws NumberFormatException;
// Public Constants
     public static final short MAX_VALUE;  =32767
     public static final short MIN_VALUE;  =-32768
5.0  public static final int SIZE;     =16
     public static final Class<Short> TYPE;
// Public Class Methods
     public static Short decode(String nm) throws NumberFormatException;
     public static short parseShort(String s) throws NumberFormatException;
     public static short parseShort(String s, int radix) throws NumberFormatException;
5.0  public static short reverseBytes(short i);
     public static String toString(short s);
     public static Short valueOf(String s) throws NumberFormatException;
5.0  public static Short valueOf(short s);
     public static Short valueOf(String s, int radix) throws NumberFormatException;
// Methods Implementing Comparable
1.2  public int compareTo(Short anotherShort);
// Public Methods Overriding Number
     public byte byteValue( );
     public double doubleValue( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

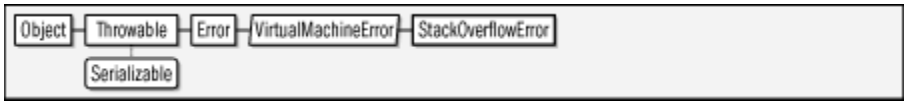
```
    public float floatValue( );
    public int intValue( );
    public long longValue( );
    public short shortValue( );
// Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode( );
    public String toString( );
}
```

StackOverflowError java.lang

Java 1.0 ***serializable error***

Signals that a stack overflow has occurred within the Java interpreter.

Figure 10-56. java.lang.StackOverflowError




```
public class StackOverflowError extends VirtualMachineError {
// Public Constructors
    public StackOverflowError( );
    public StackOverflowError(String s);
}
```

StackTraceElement java.lang

Java 1.4 ***serializable***

Instances of this class are returned in an array by `Throwable.getStackTrace( )`. Each instance represents one frame in the stack trace associated with an exception or error. `getClassName( )` and `getMethodName( )` return the name of the class (including package name) and method that contain the point of execution that the stack frame represents. If the class file contains sufficient information, `getFileName( )` and `getLineNumber( )` return the source file and line number associated with the frame. `getFileName( )` returns `null` and `getLineNumber( )` returns a negative value if source or line number information is not available. `isNativeMethod( )` returns `true` if the named method is a native method (and therefore does not have a meaningful source file or line number).

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-57. java.lang.StackTraceElement


```
public final class StackTraceElement implements Serializable {
// Public Constructors
5.0  public StackTraceElement(String declaringClass, String methodName,
     String fileName, int lineNumber);
// Public Instance Methods
     public String getClassName( );
     public String getFileName( );
     public int getLineNumber( );
     public String getMethodName( );
     public boolean isNativeMethod( );
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```

Passed To

`Throwable.setStackTrace( )`

Returned By

`Thread.getStackTrace( ),Throwable.getStackTrace( ), java.lang.management.ThreadInfo.getStackTrace( )`

StrictMath java.lang

Java 1.3

This class is identical to the `Math` class, but additionally requires that its methods strictly adhere to the behavior of certain published algorithms. The methods of `StrictMath` are intended to operate identically on all platforms, and must produce exactly the same result (down to the very least significant bit) as certain well-known standard algorithms. When strict platform-independence of floating-point results is not required, use the `Math` class for better performance.

```
public final class StrictMath {
// No Constructor
// Public Constants
     public static final double E;         =2.718281828459045
     public static final double PI;        =3.141592653589793
// Public Class Methods
     public static int abs(int a);
     public static long abs(long a);
     public static float abs(float a);
     public static double abs(double a);
     public static double acos(double a);                              native
     public static double asin(double a);                              native
     public static double atan(double a);                              native
     public static double atan2(double y, double x);                   native
5.0  public static double cbrt(double a);                            native
     public static double ceil(double a);                              native
     public static double cos(double a);                               native
5.0  public static double cosh(double x);                            native
     public static double exp(double a);                               native
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
5.0  public static double expm1(double x);                          native
     public static double floor(double a);                             native
5.0  public static double hypot(double x, double y);               native
     public static double IEEEremainder(double f1, double f2);         native
     public static double log(double a);                                native
5.0  public static double log10(double a);                         native
5.0  public static double log1p(double x);                         native
     public static int max(int a, int b);
     public static long max(long a, long b);
     public static float max(float a, float b);
     public static double max(double a, double b);
     public static int min(int a, int b);
     public static long min(long a, long b);
     public static float min(float a, float b);
     public static double min(double a, double b);
     public static double pow(double a, double b);                      native
     public static double random( );
     public static double rint(double a);
     public static int round(float a);
     public static long round(double a);
5.0  public static float signum(float f);
5.0  public static double signum(double d);
     public static double sin(double a);                                native
5.0  public static double sinh(double x);                           native
     public static double sqrt(double a);                               native
     public static double tan(double a);                                native
5.0  public static double tanh(double x);                           native
     public static double toDegrees(double angrad);
 strictfp
     public static double toRadians(double angdeg);
 strictfp
5.0  public static float ulp(float f);
5.0  public static double ulp(double d);
}
```

String java.lang

Java 1.0 ***serializable comparable***

The `String` class represents a read-only string of characters. A `String` object is created by the Java compiler whenever it encounters a string in double quotes; this method of creation is typically simpler than using a constructor. The static `valueOf( )` factory methods create new `String` objects that hold the textual representation of various Java primitive types. There are also `valueOf( )` methods, `copyValueOf( )` methods and `String( )` constructors for creating a `String` object that holds a copy of the text contained in another `String`, `StringBuffer`, `StringBuilder`, or a `char` or `int` array. You can also use the `String( )` constructor to create a `String` object from an array of bytes. If you do this, you may explicitly specify the name of the charset (or character encoding) to be used to decode the bytes into characters, or you can rely on the default charset for your platform. (See `java.nio.charset.Charset` for more on charset names.)

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

In Java 5.0, the static `format( )` methods provide another useful way to create `String` objects that hold formatted text. These utility methods create and use a new `java.util.Formatter` object and behave like the `sprintf( )` function in the C programming language.

`length( )` returns the number of characters in a string. `charAt( )` extracts a character from a string. You can use these two methods to iterate through the characters of a string. You can obtain a `char` array that holds the characters of a string with `toCharArray( )`, or use `getChars( )` to copy just a selected region of the string into an existing array. Use `getBytes( )` if you want to obtain an array of bytes that contains the encoded form of the characters in a string, using either the platform's default encoding or a named encoding.

This class defines many methods for comparing strings and substrings. `equals( )` returns `true` if two `String` objects contain the same text, and `equalsIgnoreCase( )` returns `true` if two strings are equal when uppercase and lowercase differences are ignored. As of Java 1.4, the `contentEquals( )` method compares a string to a specified `StringBuffer` object, returning `true` if they contain the same text. `startsWith( )` and `endsWith( )` return `true` if a string starts with the specified prefix string or ends with the specified suffix string. A two-argument version of `startsWith( )` allows you to specify a position within this string at which the prefix comparison is to be done. The `regionMatches( )` method is a generalized version of this `startsWith( )` method. It returns `true` if the specified region of the specified string matches the characters that begin at a specified position within this string. The five-argument version of this method allows you to perform this comparison ignoring the case of the characters being compared. The final string comparison method is `matches( )`, which, as described below, compares a string to a regular expression pattern.

`compareTo( )` is another string comparison method, but it is used for comparing the order of two strings, rather than simply comparing them for equality. `compareTo( )` implements the `Comparable` interface and enables sorting of lists and arrays of `String` objects. See `Comparable` for more information. `compareToIgnoreCase( )` is like `compareTo( )` but ignores the case of the two strings when doing the comparison. The `CASE_INSENSITIVE_ORDER` constant is a `Comparator` for sorting strings in a way that ignores the case of their characters. (The `java.util.Comparator` interface is similar to the `Comparable` interface but allows the definition of object orderings that are different from the default ordering defined by `Comparable`.) The `compareTo( )` and `compareToIgnoreCase( )` methods and the `CASE_INSENSITIVE_ORDER` `Comparator` object order strings based only on the numeric ordering of the Unicode encoding of their characters. This is not always the preferred "alphabetical ordering" in

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

some languages. See `java.text.Collator` for a more general technique for collating strings.

`indexOf( )` and `lastIndexOf( )` search forward and backward in a string for a specified character or substring. They return the position of the match, or -1 if there is no match. The one argument versions of these methods start at the beginning or end of the string, and the two-argument versions start searching from a specified character position.

Java 5.0 adds new comparison methods that work with any `CharSequence`. A new version of `contentEquals( )` enables the comparison of a string with any `CharSequence`, including `StringBuilder` objects. The `contains( )` method returns `true` if the string contains any sequence of characters equal to the specified `CharSequence`.

`substring( )` returns a string that consists of the characters from (and including) the specified start position to (but not including) the specified end position. A one-argument version returns all characters from (and including) the specified start position to the end of the string. As of Java 1.4, the `String` class implements the `CharSequence` interface and defines the `subSequence( )` method, which works just like the two-argument version of `substring( )` but returns the specified characters as a `CharSequence` rather than as a `String`.

Several methods return new strings that contain modified versions of the text held by the original string (the original string remains unchanged). `replace( )` creates a new string with all occurrences of one character replaced by another. Java 5.0 adds a generalized version of `replace( )` that replaces all occurrences of one `CharSequence` with another. More general methods, `replaceAll( )` and `replaceFirst( )`, use regular expression pattern matching; they are described later in this section. `toUpperCase( )` and `toLowerCase( )` return a new string in which all characters are converted to upper- or lowercase respectively. These case-conversion methods take an optional `Locale` argument to perform locale-specific case conversion. `trim( )` is a utility method that returns a new string in which all leading and trailing whitespace has been removed. `concat( )` returns the new string formed by concatenating or appending the specified string to this string. String concatenation is more commonly done, however, with the `+` operator.

Note that `String` objects are immutable; there is no `setCharAt( )` method to change the contents. The methods that return a `String` do not modify the string they are invoked on but instead return a new `String` object that holds a modified copy of the text of the original. Use a `StringBuffer` if you want to manipulate the contents of a string or call `toCharArray( )` or `getChars( )` to convert a string to an array of `char` values.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.4 introduced support for pattern matching with regular expressions. `matches( )` returns true if this string exactly matches the pattern specified by the regular expression argument. `replaceAll( )` and `replaceFirst( )` create a new string in which all occurrences or the first occurrence of a substring that matches the specified regular expression is replaced with the specified replacement string. The `split( )` methods return an array of substrings of this string, formed by splitting this string at positions that match the specified regular expression. These regular expression methods are all convenience methods that simply call methods of the same name in the `java.util.regex` package. See the `Pattern` and `Matcher` classes in that package for further details.

Many programs use strings as commonly as they use Java primitive values. Because the `String` type is an object rather than a primitive value, however, you cannot in general use the `==` operator to compare two strings for equality. Instead, even though strings are immutable, you must use the more expensive `equals( )` method. For programs that perform a lot of string comparison, the `intern( )` provides a way to speed up those comparisons. The `String` class maintains a set of `String` objects that includes all double-quoted string literals and all compile-time constant strings defined in a Java program. The set is guaranteed not to contain duplicates, and the set is used to ensure that duplicate `String` objects are not created unnecessarily. The `intern( )` method looks up a string in or adds a new string to this set of unique strings. It searches the set for a string that contains exactly the same characters as the string you invoked the method on. If such a string is found, `intern( )` returns it. If no matching string is found, the string you invoked `intern( )` on is itself stored in the set ("interned") and becomes the return value of the method. What this means is that you can safely compare any strings returned by the `intern( )` method using the `==` and `!=` operators instead of `equals( )`. You can also successfully compare any string returned by `intern( )` to any string constant with `==` and `!=`.

In Java 5.0, Unicode supplementary characters may be represented as a single `int` codepoint value or as a sequence of two `char` values known as a "surrogate pair." See `Character` for more on supplementary characters and methods for working with them. `String` methods for working with supplementary characters, such as `codePointAt( )`, `codePointCount( )`, and `offsetByCodePoints( )`, are similar to those defined by `Character`.

Figure 10-58. java.lang.String

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public final class String implements Serializable, Comparable<String>, CharSequence {
// Public Constructors
     public String( );
5.0  public String(StringBuilder builder);
     public String(StringBuffer buffer);
     public String(char[ ] value);
     public String(String original);
1.1  public String(byte[ ] bytes);
1.1  public String(byte[ ] bytes, String charsetName)
     throws java.io.UnsupportedEncodingException;
#    public String(byte[ ] ascii, int hibyte);
     public String(char[ ] value, int offset, int count);
1.1  public String(byte[ ] bytes, int offset, int length);
5.0  public String(int[ ] codePoints, int offset, int count);
#    public String(byte[ ] ascii, int hibyte, int offset, int count);
1.1  public String(byte[ ] bytes, int offset, int length, String charsetName)
     throws java.io.UnsupportedEncodingException;
// Public Constants
1.2  public static final java.util.Comparator<String> CASE_INSENSITIVE_ORDER;
// Public Class Methods
     public static String copyValueOf(char[ ] data);
     public static String copyValueOf(char[ ] data, int offset, int count);
5.0  public static String format(String format, Object... args);
5.0  public static String format(java.util.Locale l, String format, Object... args);
     public static String valueOf(float f);
     public static String valueOf(long l);
     public static String valueOf(Object obj);
     public static String valueOf(double d);
     public static String valueOf(boolean b);
     public static String valueOf(char[ ] data);
     public static String valueOf(int i);
     public static String valueOf(char c);
     public static String valueOf(char[ ] data, int offset, int count);
// Public Instance Methods
     public char charAt(int index);          Implements:CharSequence
5.0  public int codePointAt(int index);
5.0  public int codePointBefore(int index);
5.0  public int codePointCount(int beginIndex, int endIndex);
     public int compareTo(String anotherString);               Implements:Comparable
1.2  public int compareToIgnoreCase(String str);
     public String concat(String str);
5.0  public boolean contains(CharSequence s);
1.4  public boolean contentEquals(StringBuffer sb);
5.0  public boolean contentEquals(CharSequence cs);
     public boolean endsWith(String suffix);
     public boolean equalsIgnoreCase(String anotherString);
1.1  public byte[ ] getBytes( );
1.1  public byte[ ] getBytes(String charsetName) throws java.io.
     UnsupportedEncodingException;
     public void getChars(int srcBegin, int srcEnd, char[ ] dst, int dstBegin);
     public int indexOf(int ch);
     public int indexOf(String str);
     public int indexOf(int ch, int fromIndex);
     public int indexOf(String str, int fromIndex);
     public String intern( );                                          native
     public int lastIndexOf(String str);
     public int lastIndexOf(int ch);
     public int lastIndexOf(String str, int fromIndex);
     public int lastIndexOf(int ch, int fromIndex);
     public int length( );                    Implements:CharSequence
1.4  public boolean matches(String regex);
5.0  public int offsetByCodePoints(int index, int codePointOffset);
     public boolean regionMatches(int toffset, String other, int ooffset, int len);
     public boolean regionMatches(boolean ignoreCase, int toffset, String other,
                    int ooffset, int len);
     public String replace(char oldChar, char newChar);
5.0  public String replace(CharSequence target, CharSequence replacement);
1.4  public String replaceAll(String regex, String replacement);
1.4  public String replaceFirst(String regex, String replacement);
1.4  public String[ ] split(String regex);
1.4  public String[ ] split(String regex, int limit);
     public boolean startsWith(String prefix);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public boolean startsWith(String prefix, int toffset);
    public String substring(int beginIndex);
    public String substring(int beginIndex, int endIndex);
    public char[ ] toCharArray( );
    public String toLowerCase( );
1.1 public String toLowerCase(java.util.Locale locale);
    public String toString( );              Implements:CharSequence
    public String toUpperCase( );
1.1 public String toUpperCase(java.util.Locale locale);
    public String trim( );
// Methods Implementing CharSequence
    public char charAt(int index);
    public int length( );
1.4 public CharSequence subSequence(int beginIndex, int endIndex);
    public String toString( );
// Methods Implementing Comparable
    public int compareTo(String anotherString);
// Public Methods Overriding Object
    public boolean equals(Object anObject);
    public int hashCode( );
// Deprecated Public Methods
#   public void getBytes(int srcBegin, int srcEnd, byte[ ] dst, int dstBegin);
}
```

Passed To

Too many methods to list.

Returned By

Too many methods to list.

Type Of

Too many fields to list.

StringBuffer java.lang

Java 1.0 ***serializable appendable***

This class represents a mutable string of characters that can grow or shrink as necessary. Its mutability makes it suitable for processing text in place, which is not possible with the immutable `String` class. Its resizability and the various methods it implements make it easier to use than a `char[ ]`. Create a `StringBuffer` with the `StringBuffer( )` constructor. You may pass a `String` that contains the initial text for the buffer to this constructor, but if you do not, the buffer will start out empty. You may also specify the initial capacity for the buffer if you can estimate the number of characters the buffer will eventually hold.

The methods of this class are `synchronized`, which makes `StringBuffer` objects suitable for use by multiple threads. In Java 5.0 and later, when working with a single thread, `StringBuilder` is preferred over this class because it does not have the overhead of synchronized methods. `StringBuilder` implements the same methods as `StringBuffer` and can be used in the same way.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Query the character stored at a given index with `charAt( )` and set or delete that character with `setCharAt( )` or `deleteCharAt( )`. Use `length( )` to return the length of the buffer, and use `setLength( )` to set the length of the buffer, truncating it or filling it with null characters ('\u0000') as necessary. `capacity( )` returns the number of characters a `StringBuffer` can hold before its internal buffer needs to be reallocated. If you expect a `StringBuffer` to grow substantially and can approximate its eventual size, you can use `ensureCapacity( )` to preallocate sufficient internal storage.

Use the various `append( )` methods to append text to the end of the buffer. Use `insert( )` to insert text at a specified position within the buffer. Note that in addition to strings, primitive values, character arrays, and arbitrary objects may be passed to `append( )` and `insert( )`. These values are converted to strings before they are appended or inserted. Use `delete( )` to delete a range of characters from the buffer and use `replace( )` to replace a range of characters with a specified `String`.

Use `substring( )` to convert a portion of a `StringBuffer` to a `String`. The two versions of this method work just like the same-named methods of `String`. Call `toString( )` to obtain the contents of a `StringBuffer` as a `String` object. Or use `getChars( )` to extract the specified range of characters from the `StringBuffer` and store them into the specified character array starting at the specified index of that array.

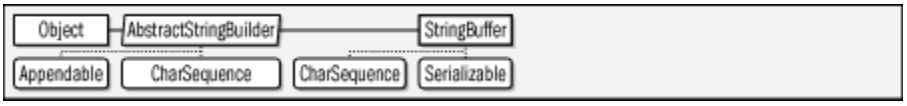
As of Java 1.4, `StringBuffer` implements `CharSequence` and so also defines a `subSequence( )` method that is like `substring( )` but returns its value as a `CharSequence`. Java 1.4 also added `indexOf( )` and `lastIndexOf( )` methods that search forward or backward (from the optionally specified index) in a `StringBuffer` for a sequence of characters that matches the specified `String`. These methods return the index of the matching string or `-1` if no match was found. See also the similarly named methods of `String` after which these methods are modeled.

In Java 5.0, this class has a new constructor and new methods for working with `CharSequence` objects. It implements the `Appendable` interface for use with `java.util.Formatter` and includes new methods for working with 21-bit Unicode characters as `int` codepoints.

String concatenation in Java is performed with the `+` operator and is implemented, prior to Java 5.0, using the `append( )` method of a `StringBuffer`. In Java 5.0 and later, `StringBuilder` is used instead. After a string is processed in a `StringBuffer` object, it can be efficiently converted to a `String` object for subsequent use. The `StringBuffer.toString( )` method is typically implemented so that it does not copy the internal array of characters. Instead, it shares that array with the new `String` object,

making a new copy for itself only if and when further modifications are made to the `StringBuffer` object.

Figure 10-59. java.lang.StringBuffer




```
public final class StringBuffer extends AbstractStringBuilder implements CharSequence,
Serializable {
// Public Constructors
     public StringBuffer( );
     public StringBuffer(String str);
     public StringBuffer(int capacity);
5.0  public StringBuffer(CharSequence seq);
// Public Instance Methods
     public StringBuffer append(String str);                      synchronized
1.4  public StringBuffer append(StringBuffer sb);                   synchronized
5.0  public StringBuffer append(CharSequence s);
     public StringBuffer append(Object obj);                      synchronized
     public StringBuffer append(char[ ] str);                    synchronized
     public StringBuffer append(long lng);                        synchronized
     public StringBuffer append(float f);                         synchronized
     public StringBuffer append(double d);                        synchronized
     public StringBuffer append(boolean b);                       synchronized
     public StringBuffer append(char c);                 synchronized
     public StringBuffer append(int i);                  synchronized
     public StringBuffer append(char[ ] str, int offset, int len);     synchronized
5.0  public StringBuffer append(CharSequence s, int start, int end);     synchronized
5.0  public StringBuffer appendCodePoint(int codePoint);            synchronized
     public char charAt(int index);      Implements:CharSequence synchronized
1.2  public StringBuffer delete(int start, int end);                synchronized
1.2  public StringBuffer deleteCharAt(int index);                   synchronized
     public StringBuffer insert(int offset, char c);                    synchronized
     public StringBuffer insert(int offset, boolean b);
     public StringBuffer insert(int offset, long l);
     public StringBuffer insert(int offset, int i);
     public StringBuffer insert(int offset, String str);                synchronized
     public StringBuffer insert(int offset, Object obj);                synchronized
5.0  public StringBuffer insert(int dstOffset, CharSequence s);
     public StringBuffer insert(int offset, char[ ] str);        synchronized
     public StringBuffer insert(int offset, double d);
     public StringBuffer insert(int offset, float f);
1.2  public StringBuffer insert(int index, char[ ] str, int offset, int len);     synchronized
5.0  public StringBuffer insert(int dstOffset, CharSequence s, int start,
     int end);     synchronized
     public int length( );                   Implements:CharSequence synchronized
1.2  public StringBuffer replace(int start, int end, String str);     synchronized
     public StringBuffer reverse( );                       synchronized
     public String toString( );            Implements:CharSequence synchronized
// Methods Implementing CharSequence
     public char charAt(int index);                      synchronized
     public int length( );                                 synchronized
1.4  public CharSequence subSequence(int start, int end);           synchronized
     public String toString( );                             synchronized
// Public Methods Overriding AbstractStringBuilder
     public int capacity( );                                synchronized
5.0  public int codePointAt(int index);                        synchronized
5.0  public int codePointBefore(int index);                    synchronized
5.0  public int codePointCount(int beginIndex, int endIndex);       synchronized
     public void ensureCapacity(int minimumCapacity);                   synchronized
     public void getChars(int srcBegin,
     int srcEnd, char[ ] dst, int dstBegin);     synchronized
1.4  public int indexOf(String str);
1.4  public int indexOf(String str, int fromIndex);                 synchronized
1.4  public int lastIndexOf(String str);
1.4  public int lastIndexOf(String str, int fromIndex);             synchronized
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
5.0  public int offsetByCodePoints(int index, int codePointOffset);     synchronized
     public void setCharAt(int index, char ch);                    synchronized
     public void setLength(int newLength);                         synchronized
1.2  public String substring(int start);                         synchronized
1.2  public String substring(int start, int end);                    synchronized
5.0  public void trimToSize( );                           synchronized
}
```

Passed To

Too many methods to list.

Returned By

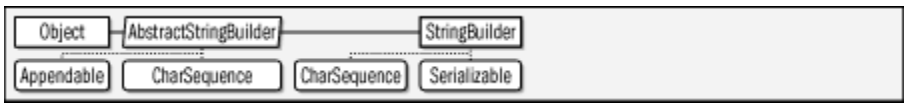
Too many methods to list.

StringBuilder java.lang

Java 5.0 ***serializable appendable***

This class defines the same methods as `StringBuffer` but does not declare those methods `synchronized`, which can result in better performance in the common case in which only a single thread is using the object. `StringBuilder` is a drop-in replacement for `StringBuffer` and should be used in preference to `StringBuffer` except where thread safety is required. See `StringBuffer` for an overview of the methods shared by these two classes.

Figure 10-60. java.lang.StringBuilder




```
public final class StringBuilder extends AbstractStringBuilder implements CharSequence,
Serializable {
// Public Constructors
     public StringBuilder( );
     public StringBuilder(int capacity);
     public StringBuilder(String str);
     public StringBuilder(CharSequence seq);
// Public Instance Methods
     public StringBuilder append(long lng);
     public StringBuilder append(float f);
     public StringBuilder append(double d);
     public StringBuilder append(int i);
     public StringBuilder append(String str);
     public StringBuilder append(StringBuffer sb);
     public StringBuilder append(CharSequence s);
     public StringBuilder append(Object obj);
     public StringBuilder append(char c);
     public StringBuilder append(boolean b);
     public StringBuilder append(char[ ] str);
     public StringBuilder append(CharSequence s, int start, int end);
     public StringBuilder append(char[ ] str, int offset, int len);
     public StringBuilder appendCodePoint(int codePoint);
     public StringBuilder delete(int start, int end);
     public StringBuilder deleteCharAt(int index);
     public StringBuilder insert(int offset, boolean b);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        public StringBuilder insert(int offset, char c);
        public StringBuilder insert(int offset, int i);
        public StringBuilder insert(int dstOffset, CharSequence s);
        public StringBuilder insert(int offset, Object obj);
        public StringBuilder insert(int offset, String str);
        public StringBuilder insert(int offset, char[ ] str);
        public StringBuilder insert(int offset, double d);
        public StringBuilder insert(int offset, long l);
        public StringBuilder insert(int offset, float f);
        public StringBuilder insert(int index, char[ ] str, int offset, int len);
        public StringBuilder insert(int dstOffset, CharSequence s, int start, int end);
        public StringBuilder replace(int start, int end, String str);
        public StringBuilder reverse( );
    // Methods Implementing CharSequence
        public String toString( );
    // Public Methods Overriding AbstractStringBuilder
        public int indexOf(String str);
        public int indexOf(String str, int fromIndex);
        public int lastIndexOf(String str);
        public int lastIndexOf(String str, int fromIndex);
    }
```

Passed To

`String.String( )`

StringIndexOutOfBoundsException java.lang

Java 1.0 ***serializable unchecked***

Signals that the index used to access a character of a `String` or `StringBuffer` is less than zero or is too large.

Figure 10-61. java.lang.StringIndexOutOfBoundsException


```
public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException {
// Public Constructors
    public StringIndexOutOfBoundsException( );
    public StringIndexOutOfBoundsException(int index);
    public StringIndexOutOfBoundsException(String s);
}
```

SuppressWarnings java.lang

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE}) @Retention(SOURCE) annotation***

An annotation of this type tells the Java compiler not to generate specified kinds of warning messages for code within the annotated program element. Annotations of this type have source retention and may be applied to any program element except packages and other annotation types. An `@SuppressWarnings` annotation has an array of `String` objects as its `value`. These strings specify the names of the warnings to be suppressed. The available warnings (and their names) depend on the compiler implementation, and compilers will ignore warning names they do not support. Compiler vendors are expected to cooperate in defining at least a core set of common warning names. In Java 5.0, the `@SuppressWarnings` warning names supported by the *javac* compiler are the same as the warning flags that can be specfied with the `-Xlint` compiler flag.

Figure 10-62. java.lang.SuppressWarnings


```
public @interface SuppressWarnings {
// Public Instance Methods
     String[ ] value( );
}
```

System **java.lang**

Java 1.0

This class defines a platform-independent interface to system facilities, including system properties and system input and output streams. All methods and variables of this class are static, and the class cannot be instantiated. Because the methods defined by this class are low-level system methods, most require special permissions and cannot be executed by untrusted code.

`getProperty( )` looks up a named property on the system properties list, returning the optionally specified default value if no property definition is found. `getProperties( )` returns the entire properties list. `setProperties( )` sets a `Properties` object on the properties list. In Java 1.2 and later, `setProperty( )` sets the value of a system property. In Java 5.0, you can clear a property setting with `clearProperty( )`. The following table lists system properties that are always defined.

Untrusted code may be unable to read some or all of these properties. Additional properties can be defined using the `-D` option when invoking the Java interpreter.

Property name	Description
`file.separator`	Platform directory separator character
`path.separator`	Platform path separator character
`line.separator`	Platform line separator character(s)
`user.name`	Current user s account name
`user.home`	Home directory of current user
`user.dir`	The current working directory
`java.class.path`	Where classes are loaded from
`java.class.version`	Version of the Java class file format
`java.compiler`	The name of the just-in-time compiler
`java.ext.dirs`	Path to directories that hold extensions
`java.home`	The directory Java is installed in
`java.io.tmpdir`	The directory that temporary files are written to
`java.library.path`	Directories to search for native libraries
`java.specification.version`	Version of the Java API specification
`java.specification.vendor`	Vendor of the Java API specification
`java.specification.name`	Name of the Java API specification
`java.version`	Version of the Java API implementation
`java.vendor`	Vendor of this Java API implementation
`java.vendor.url`	URL of the vendor of this Java API implementation
`java.vm.specification.version`	Version of the Java VM specification
`java.vm.specification.vendor`	Vendor of the Java VM specification
`java.vm.specification.name`	Name of the Java VM specification
`java.vm.version`	Version of the Java VM implementation
`java.vm.vendor`	Vendor of the Java VM implementation
`java.vm.name`	Name of the Java VM implementation
`os.name`	Name of the host operating system
`os.arch`	Host operating system architecture
`os.version`	Version of the host operating system

The `in` , `out`, and `err` fields hold the standard input, output, and error streams for the system. These fields are frequently used in calls such as `System.out.println( )`. In Java 1.1, `setIn( )`, `setOut( )`, and `setErr( )` allow these streams to be redirected.

`System` also defines various other useful static methods. `exit( )` causes the Java VM to exit. `arraycopy( )` efficiently copies an array or a portion of an array into a destination array. `currentTimeMillis( )` returns the current time in milliseconds since midnight GMT, January 1, 1970 GMT. In Java 5.0, `nanoTime( )` returns a time in nanoseconds. Unlike `currentTimeMillis( )` this time is not relative to any fixed point and so is useful only for elapsed time computations.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`getenv( )` returns the value of a platform-dependent environment variable, or (in Java 5.0) returns a `Map` of all environment variables. The one-argument version of `getenv( )` was previously deprecated but has been restored in Java 5.0.

`identityHashCode( )` computes the hashcode for an object in the same way that the default `Object.hashCode( )` method does. It does this regardless of whether or how the `hashCode( )` method has been overridden.

In Java 5.0, `inheritedChannel( )` returns a `java.nio.channels.Channel` object that represents a network connection passed to the Java process by the invoking process. This allows Java programs to be used with the Unix *inetd* daemon, for example.

`load( )` and `loadLibrary( )` can read libraries of native code into the system. `mapLibraryName( )` converts a system-independent library name into a system-dependent library filename. Finally, `getSecurityManager( )` and `setSecurityManager( )` get and set the system `SecurityManager` object responsible for the system security policy.

See also `Runtime`, which defines several other methods that provide low-level access to system facilities.

```
public final class System {
// No Constructor
// Public Constants
     public static final java.io.PrintStream err;
     public static final java.io.InputStream in;
     public static final java.io.PrintStream out;
// Public Class Methods
     public static void arraycopy(Object src, int srcPos, Object dest, int destPos,
     int length);     native
5.0  public static String clearProperty(String key);
     public static long currentTimeMillis( );                              native
     public static void exit(int status);
     public static void gc( );
5.0  public static java.util.Map<String,String> getenv( );
     public static String getenv(String name);
     public static java.util.Properties getProperties( );
     public static String getProperty(String key);
     public static String getProperty(String key, String def);
     public static SecurityManager getSecurityManager( );
1.1  public static int identityHashCode(Object x);                  native
5.0  public static java.nio.channels.Channel inheritedChannel( ) throws java.io.IOException;
     public static void load(String filename);
     public static void loadLibrary(String libname);
1.2  public static String mapLibraryName(String libname);           native
5.0  public static long nanoTime( );                                  native
     public static void runFinalization( );
1.1  public static void setErr(java.io.PrintStream err);
1.1  public static void setIn(java.io.InputStream in);
1.1  public static void setOut(java.io.PrintStream out);
     public static void setProperties(java.util.Properties props);
1.2  public static String setProperty(String key, String value);
     public static void setSecurityManager(SecurityManager s);
// Deprecated Public Methods
1.1#  public static void runFinalizersOnExit(boolean value);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Thread **java.lang**

Java 1.0 ***runnable***

This class encapsulates all information about a single thread of control running on the Java interpreter. To create a thread, you must either pass a `Runnable` object (i.e., an object that implements the `Runnable` interface by defining a `run( )` method) to the `Thread` constructor or subclass `Thread` so that it defines its own `run( )` method. The `run( )` method of the `Thread` or of the specified `Runnable` object is the body of the thread. It begins executing when the `start( )` method of the `Thread` object is called. The thread runs until the `run( )` method returns. `isAlive( )` returns `true` if a thread has been started, and the `run( )` method has not yet exited.

The static methods of this class operate on the currently running thread. `currentThread( )` returns the `Thread` object of the currently running code. `sleep( )` makes the current thread stop for a specified amount of time. `yield( )` makes the current thread give up control to any other threads of equal priority that are waiting to run. `holdsLock( )` tests whether the current thread holds a lock (through a `synchronized` method or statement) on the specified object; this Java 1.4 method is often useful with an `assert` statement.

The instance methods may be called by one thread to operate on a different thread. `checkAccess( )` checks whether the running thread has permission to modify a `Thread` object and throws a `SecurityException` if it does not. `join( )` waits for a thread to die. `interrupt( )` wakes up a waiting or sleeping thread (with an `InterruptedException`) or sets an interrupted flag on a nonsleeping thread. A thread can test its own interrupted flag with the static `interrupted( )` method or can test the flag of another thread with `isInterrupted( )`. Calling `interrupted( )` implicitly clears the interrupted flag, but calling `isInterrupted( )` does not. Methods related to `sleep( )` and `interrupt( )` are the `wait( )` and `notify( )` methods defined by the `Object` class. Calling `wait( )` causes the current thread to block until the object's `notify( )` method is called by another thread.

`setName( )` sets the name of a thread, which is purely optional. `setPriority( )` sets the priority of the thread. Higher priority threads run before lower priority threads. Java does not specify what happens to multiple threads of equal priority; some systems perform time-slicing and share the CPU between such threads. On other systems, one compute-

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

bound thread that does not call `yield( )` may starve another thread of the same priority. `setDaemon( )` sets a boolean flag that specifies whether this thread is a daemon or not. The Java VM keeps running as long as at least one nondaemon thread is running. Call `getThreadGroup( )` to obtain the `ThreadGroup` of which a thread is part. In Java 1.2 and later, use `setContextClassLoader( )` to specify the `ClassLoader` to be used to load any classes required by the thread.

`suspend( )`, `resume( )`, and `stop( )` suspend, resume, and stop a given thread, respectively, but all three methods are deprecated because they are inherently unsafe and can cause deadlock. If a thread must be stoppable, have it periodically check a flag and exit if the flag is set.

In Java 1.4 and later, the four-argument `Thread( )` constructor allows you to specify the "stack size" parameter for the thread. Typically, larger stack sizes allow threads to recurse more deeply before running out of stack space. Smaller stack sizes reduce the fixed per-thread memory requirements and may allow more threads to exist concurrently. The meaning of this argument is implementation dependent, and implementations may even ignore it.

Java 5.0 adds important new features to this class. `getId( )` returns a unique `long` identifier for the thread. `getState( )` returns the state of the thread as an enumerated constant of type `Thread.State`. `Thread.UncaughtExceptionHandler` defines an API for handling exceptions that cause the `run( )` method of the thread to exit. Register a handler of this type with `setUncaughtExceptionHandler( )` or register a default handler with the static methods `setDefaultUncaughtExceptionHandler( )`. Obtain a snapshot of a thread's current stack trace with `getStackTrace( )`. This returns an array of `StackTraceElement` objects: the first element of the array is the most recent method invocation and the last element is the least recent. The static `getAllStackTraces( )` returns stack traces for all running threads (the traces may be obtained at different times for different threads).

Figure 10-63. java.lang.Thread


```
public class Thread implements Runnable {
// Public Constructors
     public Thread( );
     public Thread(String name);
     public Thread(Runnable target);
     public Thread(Runnable target, String name);
     public Thread(ThreadGroup group, String name);
     public Thread(ThreadGroup group, Runnable target);
     public Thread(ThreadGroup group, Runnable target, String name);
1.4  public Thread(ThreadGroup group, Runnable target, String name, long stackSize);
// Public Constants
     public static final int MAX_PRIORITY; =10
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public static final int MIN_PRIORITY; =1
    public static final int NORM_PRIORITY;                              =5
// Nested Types
5.0  public enum State;
5.0  public interface UncaughtExceptionHandler;
// Public Class Methods
    public static int activeCount( );
    public static Thread currentThread( );                               native
    public static void dumpStack( );
    public static int enumerate(Thread[ ] tarray);
5.0  public static java.util.Map<Thread,StackTraceElement[ ]> getAllStackTraces( );
5.0  public static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler( );
1.4  public static boolean holdsLock(Object obj);                   native
    public static boolean interrupted( );
5.0  public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh);
    public static void sleep(long millis) throws InterruptedException;     native
    public static void sleep(long millis, int nanos) throws InterruptedException;
    public static void yield( );                                         native
// Public Instance Methods
    public final void checkAccess( );
1.2  public ClassLoader getContextClassLoader( );
5.0  public long getId( );                          default:7
    public final String getName( );                                      default:"Thread-0"
    public final int getPriority( );                                     default:5
5.0  public StackTraceElement[ ] getStackTrace( );
5.0  public Thread.State getState( );
    public final ThreadGroup getThreadGroup( );
5.0  public Thread.UncaughtExceptionHandler getUncaughtExceptionHandler( ); default:ThreadGroup
    public void interrupt( );
    public final boolean isAlive( );                                     native default:false
    public final boolean isDaemon( );                                    default:false
    public boolean isInterrupted( );                                     default:false
    public final void join( ) throws InterruptedException;
    public final void join(long millis) throws InterruptedException;     synchronized
    public final void join(long millis, int nanos) throws InterruptedException;     synchronized
1.2  public void setContextClassLoader(ClassLoader cl);
    public final void setDaemon(boolean on);
    public final void setName(String name);
    public final void setPriority(int newPriority);
5.0  public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh);
    public void start( );                               synchronized
// Methods Implementing Runnable
    public void run( );
// Public Methods Overriding Object
    public String toString( );
// Deprecated Public Methods
#    public int countStackFrames( );                                    native
#    public void destroy( );
#    public final void resume( );
#    public final void stop( );
#    public final void stop(Throwable obj);                    synchronized
#    public final void suspend( );
}
```

Passed To

Runtime.{addShutdownHook( ),removeShutdownHook( )}, SecurityManager.checkAccess( ), Thread.UncaughtExceptionHandler.uncaughtException( ),ThreadGroup.{enumerate( ),uncaughtException( )}, java.util.concurrent.ThreadPoolExecutor.beforeExecute( ), java.util.concurrent.TimeUnit.timedJoin( ), java.util.concurrent.locks.AbstractQueuedSynchronizer.isQueued( ),java.util.concurrent.locks.LockSupport.unpark( ),

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`java.util.concurrent.locks.ReentrantLock.hasQueuedThread( ),`
`java.util.concurrent.locks.ReentrantReadWriteLock.hasQueuedThread( )`

Returned By

`java.util.concurrent.ThreadFactory.newThread( ),`
`java.util.concurrent.locks.AbstractQueuedSynchronizer.getFirstQueuedThread( ),`
`java.util.concurrent.locks.ReentrantLock.getOwner( ),`
`java.util.concurrent.locks.ReentrantReadWriteLock.getOwner( )`

Thread.State java.lang

Java 5.0 ***serializable comparable enum***

This enumerated type defines the possible states of a thread. Call the `getState( )` method of a `Thread` object to obtain one of the enumerated constants defined here. A `NEW` thread has not been started yet, and a `TERMINATED` thread has exited. A `BLOCKED` thread is waiting to enter a `synchronized` method or block. A `WAITING` thread is waiting in `Object.wait( )`, `Thread.join( )`, or a similar method. A `TIMED_WAITING` thread is waiting but is subject to a timeout, such as in `Thread.sleep( )` or the timed versions of `Object.wait( )` and `Thread.join( )`. Finally, a thread that has been started and has not yet exited and is not blocked or waiting is `RUNNABLE`. This does not mean that the operating system is currently running it or that it is even making any forward progress, but that it is at least available to run when the operating system gives it the CPU.

```
public enum Thread.State {
// Enumerated Constants
     NEW,
     RUNNABLE,
     BLOCKED,
     WAITING,
     TIMED_WAITING,
     TERMINATED;
// Public Class Methods
     public static Thread.State valueOf(String name);
     public static final Thread.State[ ] values( );
}
```

Returned By

`Thread.getState( ),`
`java.lang.management.ThreadInfo.getThreadState( )`

Thread.UncaughtExceptionHandler java.lang

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0

This interface defines a handler to be invoked when a thread throws an exception that remains uncaught. When this happens, the `uncaughtException( )` method of the registered handler is invoked with the `Thread` object that threw the exception and the `Throwable` exception object as arguments. The handler is run by the thread that received the exception, and that thread will exit as soon as the handler exits. If `uncaughtException( )` itself throws an exception, that exception will be ignored.

An object that implements this interface may be registered for a `Thread` with the `setUncaughtExceptionHandler( )` method. A default `UncaughtExceptionHandler` may be registered with the static method `Thread.setDefaultUncaughtExceptionHandler( )`. If no handler or default handler is registered, the `uncaughtException( )` method of the containing `ThreadGroup` is used instead.

```
public interface Thread.UncaughtExceptionHandler {
// Public Instance Methods
     void uncaughtException(Thread t, Throwable e);
}
```

Implementations

`ThreadGroup`

Passed To

`Thread.{setDefaultUncaughtExceptionHandler( ), setUncaughtExceptionHandler( )}`

Returned By

`Thread.{getDefaultUncaughtExceptionHandler( ), getUncaughtExceptionHandler( )}`

ThreadDeath java.lang

Java 1.0 ***serializable error***

Signals that a thread should terminate. This error is thrown in a thread when the `Thread.stop( )` method is called for that thread. This is an unusual `Error` type that simply causes a thread to be terminated, but does not print an error message or cause the interpreter to exit. You can catch `ThreadDeath` errors to do any necessary cleanup for a thread, but if you do, you must rethrow the error so that the thread actually terminates.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-64. java.lang.ThreadDeath


```
public class ThreadDeath extends Error {
// Public Constructors
     public ThreadDeath( );
}
```

ThreadGroup java.lang

Java 1.0

This class represents a group of threads and allows that group to be manipulated as a whole. A `ThreadGroup` can contain `Thread` objects, as well as other child `ThreadGroup` objects. All `ThreadGroup` objects are created as children of some other `ThreadGroup`, and thus there is a parent/child hierarchy of `ThreadGroup` objects. Use `getParent( )` to obtain the parent `ThreadGroup`, and use `activeCount( )`, `activeGroupCount( )`, and the various `enumerate( )` methods to list the child `Thread` and `ThreadGroup` objects. Most applications can simply rely on the default system thread group. System-level code and applications such as servers that need to create a large number of threads may find it convenient to create their own `ThreadGroup` objects, however.

`interrupt( )` interrupts all threads in the group at once. `setMaxPriority( )` specifies the maximum priority any thread in the group can have. `checkAccess( )` checks whether the calling thread has permission to modify the given thread group. The method throws a `SecurityException` if the current thread does not have access. `uncaughtException( )` contains the code that is run when a thread terminates because of an uncaught exception or error. You can customize this method by subclassing `ThreadGroup`.

Figure 10-65. java.lang.ThreadGroup


```
public class ThreadGroup implements Thread.UncaughtExceptionHandler {
// Public Constructors
     public ThreadGroup(String name);
     public ThreadGroup(ThreadGroup parent, String name);
// Public Instance Methods
     public int activeCount( );
     public int activeGroupCount( );
     public final void checkAccess( );
     public final void destroy( );
     public int enumerate(ThreadGroup[ ] list);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        public int enumerate(Thread[ ] list);
        public int enumerate(Thread[ ] list, boolean recurse);
        public int enumerate(ThreadGroup[ ] list, boolean recurse);
        public final int getMaxPriority( );
        public final String getName( );
        public final ThreadGroup getParent( );
1.2     public final void interrupt( );
        public final boolean isDaemon( );
1.1     public boolean isDestroyed( );                  synchronized
        public void list( );
        public final boolean parentOf(ThreadGroup g);
        public final void setDaemon(boolean daemon);
        public final void setMaxPriority(int pri);
        public void uncaughtException(Thread t, Throwable e);
Implements:Thread.UncaughtExceptionHandler
// Methods Implementing Thread.UncaughtExceptionHandler
        public void uncaughtException(Thread t, Throwable e);
// Public Methods Overriding Object
        public String toString( );
// Deprecated Public Methods
1.1#    public boolean allowThreadSuspension(boolean b);
#       public final void resume( );
#       public final void stop( );
#       public final void suspend( );
}
```

Passed To

`SecurityManager.checkAccess( )`, `Thread.Thread( )`

Returned By

`SecurityManager.getThreadGroup( )`, `Thread.getThreadGroup( )`

ThreadLocal<T> java.lang

Java 1.2

This class provides a convenient way to create thread-local variables. When you declare a static field in a class, there is only one value for that field, shared by all objects of the class. When you declare a nonstatic instance field in a class, every object of the class has its own separate copy of that variable. `ThreadLocal` provides an option between these two extremes. If you declare a static field to hold a `ThreadLocal` object, that `ThreadLocal` holds a different value for each thread. Objects running in the same thread see the same value when they call the `get( )` method of the `ThreadLocal` object. Objects running in different threads obtain different values from `get( )`, however.

In Java 5.0, this class has been made generic and the type variable *`T`* represents the type of the object referenced by this `ThreadLocal`.

The `set( )` method sets the value held by the `ThreadLocal` object for the currently running thread. `get( )` returns the value held for the currently running thread. Note that there is no way to obtain the value of the `ThreadLocal` object for any thread other than

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

the one that calls `get( )`. To understand the `ThreadLocal` class, you may find it helpful to think of a `ThreadLocal` object as a hashtable or `java.util.Map` that maps from `Thread` objects to arbitrary values. Calling `set( )` creates an association between the current `Thread` (`Thread.currentThread( )`) and the specified value. Calling `get( )` first looks up the current thread, then uses the hashtable to look up the value associated with that current thread.

If a thread calls `get( )` for the first time without having first called `set( )` to establish a thread-local value, `get( )` calls the protected `initialValue( )` method to obtain the initial value to return. The default implementation of `initialValue( )` simply returns `null`, but subclasses can override this if they desire.

See also `InheritableThreadLocal`, which allows thread-local values to be inherited from parent threads by child threads.

```
public class ThreadLocal<T> {
// Public Constructors
     public ThreadLocal( );
// Public Instance Methods
     public T get( );
5.0  public void remove( );
     public void set(T value);
// Protected Instance Methods
     protected T initialValue( );                                    constant
}
```

Subclasses

`InheritableThreadLocal`

Throwable — java.lang

Java 1.0 — ***serializable***

This is the root class of the Java exception and error hierarchy. All exceptions and errors are subclasses of `Throwable`. The `getMessage( )` method retrieves any error message associated with the exception or error. The default implemenation of `getLocalizedMessage( )` simply calls `getMessage( )`, but subclasses may override this method to return an error message that has been localized for the default locale.

It is often the case that an `Exception` or `Error` is generated as a direct result of some other exception or error, perhaps one thrown by a lower-level API. As of Java 1.4 and later, all `Throwable` objects may have a "cause" which specifies the `Throwable` that caused this one. If there is a cause, pass it to the `Throwable( )` constructor, or to the

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`initCause( )` method. When you catch a `Throwable` object, you can obtain the `Throwable` that caused it, if any, with `getCause( )`.

Every `Throwable` object has information about the execution stack associated with it. This information is initialized when the `Throwable` object is created. If the object will be thrown somewhere other than where it was created, or if it caught and will be re-thrown, you can use `fillInStackTrace( )` to capture the current execution stack before throwing it. `printStackTrace( )` prints a textual representation of the stack to the specified `PrintWriter`, `PrintStream`, or to the `System.err` stream. In Java 1.4, you can also obtain this information with `getStackTrace( )` which returns an array of `StackTraceElement` objects describing the execution stack.

Figure 10-66. java.lang.Throwable


```
public class Throwable implements Serializable {
// Public Constructors
     public Throwable( );
     public Throwable(String message);
1.4  public Throwable(Throwable cause);
1.4  public Throwable(String message, Throwable cause);
// Public Instance Methods
     public Throwable fillInStackTrace( );                                native synchronized
1.4  public Throwable getCause( );                                      default:null
1.1  public String getLocalizedMessage( );                             default:null
     public String getMessage( );                                         default:null
1.4  public StackTraceElement[ ] getStackTrace( );
1.4  public Throwable initCause(Throwable cause);                    synchronized
     public void printStackTrace( );
     public void printStackTrace(java.io.PrintStream s);
1.1  public void printStackTrace(java.io.PrintWriter s);
1.4  public void setStackTrace(StackTraceElement[ ] stackTrace);
// Public Methods Overriding Object
     public String toString( );
}
```

Subclasses

`Error`, `Exception`

Passed To

Too many methods to list.

Returned By

`java.io.WriteAbortedException.getCause( )`, `ClassNotFoundException.{getCause( ), getException( )}`, `ExceptionInInitializerError.{getCause( ), getException( )}`, `java.lang.reflect.InvocationTargetException.{getCause( ), getTargetException( )}`, `java.lang.reflect.UndeclaredThrowableException.{getCause( ), getUndeclaredThrowable( )}`, `java.security.PrivilegedActionException.getCause( )`,

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

java.util.logging.LogRecord.getThrown( ),
javax.xml.transform.TransformerException.{getCause( ),
getException( ),initCause( )},
javax.xml.xpath.XPathException.getCause( )

Thrown By

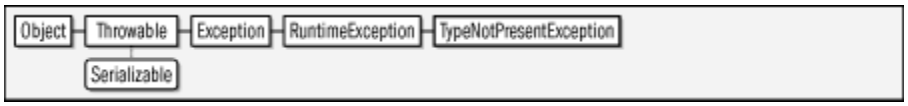
Object.finalize( ),java.lang.reflect.InvocationHandler.invoke( )

TypeNotPresentException java.lang

Java 5.0 ***serializable unchecked***

This unchecked exception signals that a class file associated with a `java.lang.reflect.Type` could not be found. It typically results when a class depends on a type that has changed or been removed and indicates version skew that requires recompilation or code refactoring. This is essentially the generic type version of `ClassNotFoundException`.

Figure 10-67. java.lang.TypeNotPresentException



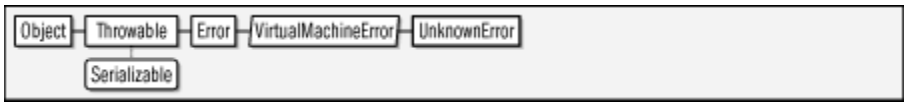

```
public class TypeNotPresentException extends RuntimeException {
// Public Constructors
     public TypeNotPresentException(String typeName, Throwable cause);
// Public Instance Methods
     public String typeName( );
}
```

UnknownError java.lang

Java 1.0 ***serializable error***

Signals that an unknown error has occurred at the level of the Java Virtual Machine.

Figure 10-68. java.lang.UnknownError



Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class UnknownError extends VirtualMachineError {
// Public Constructors
     public UnknownError( );
     public UnknownError(String s);
}
```

UnsatisfiedLinkError java.lang

Java 1.0 ***serializable error***

Signals that Java cannot satisfy all the links in a class that it has loaded.

Figure 10-69. java.lang.UnsatisfiedLinkError


```
public class UnsatisfiedLinkError extends LinkageError {
// Public Constructors
     public UnsatisfiedLinkError( );
     public UnsatisfiedLinkError(String s);
}
```

UnsupportedClassVersionError java.lang

Java 1.2 ***serializable error***

Every Java class file contains a version number that specifies the version of the class file format. This error is thrown when the Java Virtual Machine attempts to read a class file with a version number it does not support.

Figure 10-70. java.lang.UnsupportedClassVersionError


```
public class UnsupportedClassVersionError extends ClassFormatError {
// Public Constructors
     public UnsupportedClassVersionError( );
     public UnsupportedClassVersionError(String s);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

UnsupportedOperationException java.lang

Java 1.2 ***serializable unchecked***

Signals that a method you have called is not supported, and its implementation does not do anything (except throw this exception). This exception is used most often by the Java collection framework of `java.util`. Immutable or unmodifiable collections throw this exception when a modification method, such as `add( )` or `delete( )`, is called.

Figure 10-71. java.lang.UnsupportedOperationException


```
public class UnsupportedOperationException extends RuntimeException {
// Public Constructors
     public UnsupportedOperationException( );
5.0  public UnsupportedOperationException(Throwable cause);
     public UnsupportedOperationException(String message);
5.0  public UnsupportedOperationException(String message, Throwable cause);
}
```

Subclasses

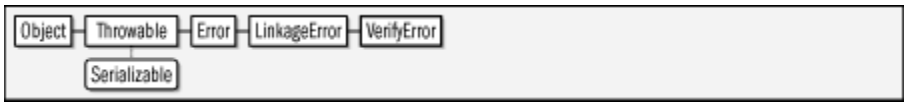
`java.nio.ReadOnlyBufferException`

VerifyError java.lang

Java 1.0 ***serializable error***

Signals that a class has not passed the byte-code verification procedures.

Figure 10-72. java.lang.VerifyError




```
public class VerifyError extends LinkageError {
// Public Constructors
     public VerifyError( );
     public VerifyError(String s);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

VirtualMachineError java.lang

Java 1.0 ***serializable error***

An abstract error type that serves as superclass for a group of errors related to the Java Virtual Machine. See `InternalError`, `UnknownError`, `OutOfMemoryError`, and `StackOverflowError`.

Figure 10-73. java.lang.VirtualMachineError


```
public abstract class VirtualMachineError extends Error {
// Public Constructors
     public VirtualMachineError( );
     public VirtualMachineError(String s);
}
```

Subclasses

`InternalError`, `OutOfMemoryError`, `StackOverflowError`, `UnknownError`

Void java.lang

Java 1.1

The `Void` class cannot be instantiated and serves merely as a placeholder for its static `TYPE` field, which is a `Class` object constant that represents the `void` type.

```
public final class Void {
// No Constructor
// Public Constants
     public static final Class<Void> TYPE;
}
```

Package java.lang.annotation

Java 5.0

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This package defines the framework for annotations. It includes the base `Annotation` interface that all annotation types extend, meta-annotation types, their associated enumerated types, and exception and error classes related to annotations. The most important members of this package are the meta-annotation types: `Documented`, `Inherited`, `Retention`, and `Target`.

Interfaces

```
public interface Annotation;
```

Enumerated Types

```
public enum ElementType;
public enum RetentionPolicy;
```

Annotation Types

```
public @interface Documented;
public @interface Inherited;
public @interface Retention;
public @interface Target;
```

Exceptions

```
public class AnnotationTypeMismatchException extends RuntimeException;
public class IncompleteAnnotationException extends RuntimeException;
```

Errors

```
public class AnnotationFormatError extends Error;
```

Annotation java.lang.annotation

Java 5.0

A type declared with the `@interface` syntax is an annotation type that implicitly extends this interface. Note that the `Annotation` interface is not itself an annotation type. Furthermore, if you define an `interface` (rather than an `@interface`) that explicitly extends `Annotation`, the result is not an annotation type either. The only way to define an annotation type is with an `@interface` definition. When an annotation is queried with the `java.lang.reflect.AnnotatedElement` API, the object returned implements this interface as well as the interface defined by the specific annotation type.

This interface defines the `annotationType( )` method, which returns the `Class` of the annotation type for any annotation object. It also includes the `equals( )` and `hashCode( )` methods of `Object` to require an implementation to compare annotations by the values of their members rather than simply by using = =. Finally, `Annotation` also overrides the `toString( )` method to require implementations to provide some

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

meaningful string representation of an annotation. The format of the returned string is not specified, but you can expect implementations to produce a string using a syntax similar to that used to encode annotations in Java source code.

```
public interface Annotation {
// Public Instance Methods
     Class<? extends java.lang.annotation.Annotation> annotationType( );
     boolean equals(Object obj);
     int hashCode( );
     String toString( );
}
```

Implementations

`Deprecated`, `Override`, `SuppressWarnings`, `Documented`, `Inherited`, `Retention`, `Target`

Returned By

Too many methods to list.

AnnotationFormatError java.lang.annotation

Java 5.0 ***serializable error***

An error of this type indicates that a class file includes a malformed annotation.

Figure 10-74. java.lang.annotation.AnnotationFormatError


```
public class AnnotationFormatError extends Error {
// Public Constructors
     public AnnotationFormatError(Throwable cause);
     public AnnotationFormatError(String message);
     public AnnotationFormatError(String message, Throwable cause);
}
```

AnnotationTypeMismatchException java.lang.annotation

Java 5.0 ***serializable unchecked***

An exception of this type indicates version skew in an annotation type. It occurs when the Java VM attempts to read an annotation from a class file and discovers that the type of an

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

annotation member has changed since the class file (and the annotation it contains) was compiled.

Figure 10-75. java.lang.annotation.AnnotationTypeMismatchException


```
public class AnnotationTypeMismatchException extends RuntimeException {
// Public Constructors
     public AnnotationTypeMismatchException(java.lang.reflect.Method element, String foundType);
// Public Instance Methods
     public java.lang.reflect.Method element( );
     public String foundType( );
}
```

Documented **java.lang.annotation**

Java 5.0 ***@Documented @Retention(RUNTIME) @Target(ANNOTATION_TYPE) annotation***

A meta-annotation of this type indicates that the annotated type should be documented by Javadoc and similar documentation tools. If an annotation type is an `@Documented` annotation, then the presence of an annotation of that type is part of the public API of the annotated program element. `java.lang.Deprecated` is an `@Documented` annotation type, for example, and so are each of the meta-annotation types in this package.

It is recommended that any annotation type that is `@Documented` should also have runtime `@Retention` so that the presence of the annotation can be queried via reflection.

Figure 10-76. java.lang.annotation.Documented


```
public @interface Documented {
}
```

ElementType **java.lang.annotation**

Java 5.0 ***serializable comparable enum***

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The constants declared by this enumerated type represent the types of program elements that can be annotated. The `value` of an `@Target` annotation is an array of `ElementType` constants. Most of the constants have obvious meanings, but some require additional explanation. `TYPE` represents a class, interface, enumerated type, or annotation type. `ANNOTATION_TYPE` represents only annotation types and is used for meta-annotations. `FIELD` includes enumerated constants, and `PARAMETER` includes both method parameters and `catch` clause parameters. Note that the `METHOD` and `CONSTRUCTOR` are distinct constants.

Figure 10-77. java.lang.annotation.ElementType


```
public enum ElementType {
// Enumerated Constants
     TYPE,
     FIELD,
     METHOD,
     PARAMETER,
     CONSTRUCTOR,
     LOCAL_VARIABLE,
     ANNOTATION_TYPE,
     PACKAGE;
// Public Class Methods
     public static ElementType valueOf(String name);
     public static final ElementType[ ] values( );
}
```

Returned By

`Target.value( )`

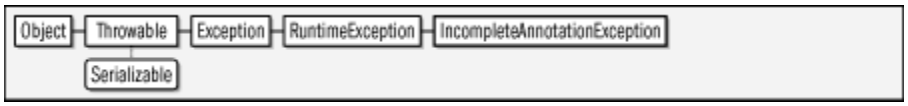
IncompleteAnnotationException java.lang.annotation

Java 5.0 ***serializable unchecked***

An exception of this type indicates version skew in an annotation type. It occurs when the Java VM attempts to read an annotation from a class file and discovers that the annotation type has added a new member since the class file was compiled. This means that the annotation compiled into the class file is incomplete since it does not define a value for all members of the annotation type. Note that this exception does not occur if a new member with a `default` clause is added to the annotation type.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-78. java.lang.annotation.IncompleteAnnotationException




```
public class IncompleteAnnotationException extends RuntimeException {
// Public Constructors
     public IncompleteAnnotationException(Class<? extends java.lang.annotation.Annotation> annotationType,
     String elementName);
// Public Instance Methods
     public Class<? extends java.lang.annotation.Annotation> annotationType( );
     public String elementName( );
}
```

Inherited **java.lang.annotation**

Java 5.0 ***@Documented @Retention(RUNTIME) @Target(ANNOTATION_TYPE) annotation***

When an annotation type that has an `@Inherited` meta-annotation is applied to a class, that annotation should be inherited by subclasses and descendants of the annotated class. The inheritance is only for classes and their subclasses. If an `@Inherited` annotation type is applied to a method or program element other than a class, no inheritance applies. If the `@Inherited` annotation type also has runtime `Retention`, reflective access to the annotation through `java.lang.reflect.AnnotatedElement` manages the inheritance of the annotation.

Figure 10-79. java.lang.annotation.Inherited


```
public @interface Inherited {
}
```

Retention **java.lang.annotation**

Java 5.0 ***@Documented @Retention(RUNTIME) @Target(ANNOTATION_TYPE) annotation***

A meta-annotation of this type specifies how long the annotated annotation type should be retained. The `value( )` of this annotation type is one of the three

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`RetentionPolicy` enumerated constants. See `RetentionPolicy` for details. If an annotation type does not have an `@Retention` meta-annotation, its default retention is `RetentionPolicy.CLASS`.

Figure 10-80. java.lang.annotation.Retention


```
public @interface Retention {
// Public Instance Methods
     RetentionPolicy value( );
}
```

RetentionPolicy **java.lang.annotation**

Java 5.0 ***serializable comparable enum***

The constants declared by the enumerated type specify the possible retention values for an `@Retention` meta-annotation. Annotations with `SOURCE` retention appear in Java source code only and are discarded by the compiler. Annotations with `CLASS` retention are compiled into the class file and are visible to tools that read class files but are not loaded by the Java VM at runtime. (This is the default retention for annotation types that do not have an `@Retention` meta-annotation.) Finally, annotations with `RUNTIME` retention are stored in the class file and loaded by the Java interpreter at runtime. These annotations are available for reflective access through `java.lang.reflect.AnnotatedElement`.

Figure 10-81. java.lang.annotation.RetentionPolicy


```
public enum RetentionPolicy {
// Enumerated Constants
     SOURCE,
     CLASS,
     RUNTIME;
// Public Class Methods
     public static RetentionPolicy valueOf(String name);
     public static final RetentionPolicy[ ] values( );
}
```

Returned By

`Retention.value( )`

Target — java.lang.annotation

Java 5.0 — *@Documented @Retention(RUNTIME) @Target(ANNOTATION_TYPE) annotation*

A meta-annotation of this type specifies what program elements the annotated annotation type can be applied to. The `value( )` of a `Target` annotation is an array of `ElementType` enumerated constants. See `ElementType` for details on the allowed values. If an annotation type does not have an `@Target` meta-annotation, it can be applied to any program element.

Figure 10-82. java.lang.annotation.Target


```
public @interface Target {
// Public Instance Methods
     ElementType[ ] value( );
}
```

Package java.lang.instrument

Java 5.0

This package defines the API for instrumenting a Java VM by transforming class files to add profiling support, code coverage testing, or other features.

The `-javaagent` command-line option to the Java interpreter provides a hook for running the `premain( )` method of a Java instrumentation *agent*. An `Instrumentation` object passed to the `premain( )` method provides an entry point into this package, and methods of `Instrumentation` allow loaded classes to be redefined and `ClassFileTransformer` objects to be registered for classes not yet loaded.

Interfaces

```
public interface ClassFileTransformer;
public interface Instrumentation;
```

Classes

```
public final class ClassDefinition;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Exceptions

```
public class IllegalClassFormatException extends Exception;
public class UnmodifiableClassException extends Exception;
```

ClassDefinition java.lang.instrument

Java 5.0

This class is a simple wrapper around a `Class` object and an array of bytes that represents a class file for that class. An array of `ClassDefinition` objects is passed to the `redefineClasses( )` method of the `Instrumentation` class. Class redefinitions are allowed to change method implementations, but not the members or inheritance of a class or the signature of the methods.

```
public final class ClassDefinition {
// Public Constructors
     public ClassDefinition(Class<?> theClass, byte[ ] theClassFile);
// Public Instance Methods
     public Class<?> getDefinitionClass( );
     public byte[ ] getDefinitionClassFile( );
}
```

Passed To

`Instrumentation.redefineClasses( )`

ClassFileTransformer java.lang.instrument

Java 5.0

A `ClassFileTransformer` registered through an `Instrumentation` object is offered a chance to transform every class that is subsequently loaded or redefined. The final argument to `transform( )` is a byte array that contains the raw bytes of the class file (or bytes returned by a previously invoked `ClassFileTransformer`). If the `transform( )` method wishes to transform the class, it should return the transformed bytes in a newly allocated array. The array passed to `transform( )` should not be modified. If the `transform( )` method does not wish to transform a given class, it should return `null`.

```
public interface ClassFileTransformer {
// Public Instance Methods
     byte[ ] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
java.security.ProtectionDomain protectionDomain, byte[ ] classfileBuffer)
throws IllegalClassFormatException;
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Passed To

`Instrumentation.{addTransformer( ),removeTransformer( )}`

IllegalClassFormatException — java.lang.instrument

Java 5.0 — ***serializable checked***

A `ClassFileTransformer` should throw an exception of this type from its `transform( )` method if it believes that the class file bytes it has been passed are malformed (this could happen, for example, if a defective `ClassFileTransformer` had previously transformed a valid class file).

Figure 10-83. java.lang.instrument.IllegalClassFormatException


```
public class IllegalClassFormatException extends Exception {
// Public Constructors
     public IllegalClassFormatException( );
     public IllegalClassFormatException(String s);
}
```

Thrown By

`ClassFileTransformer.transform( )`

Instrumentation — java.lang.instrument

Java 5.0

This interface is the main entry point to the `java.lang.instrument` API. A Java instrumentation agent specified on the Java interpreter command line with the `-javaagent` argument must be a class that defines the following method:

```
public static void premain(String args, Instrumentation instr)
```

The Java interpreter invokes the `premain( )` method during startup before calling the `main( )` method of the program. Any arguments specified with the `-javaagent` command line are passed in the first `premain( )` argument, and an `Instrumentation` object is passed as the second argument.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The most powerful feature of the `Instrumentation` object is the ability to register `ClassFileTransformer` objects to augment or rewrite the byte code of Java class files as they are loaded into the interpreter. If `isRedefineClassesSupported( )` returns `true`, you can also redefine already-loaded classes on the fly with `redefineClasses( )`.

`getAllLoadedClasses( )` returns an array of all classes loaded into the VM, and `getInitiatedClasses( )` returns an array of classes loaded by a specified `ClassLoader`. `getObjectSize( )` returns an implementation-specific approximation of the amount of memory required by a specified object.

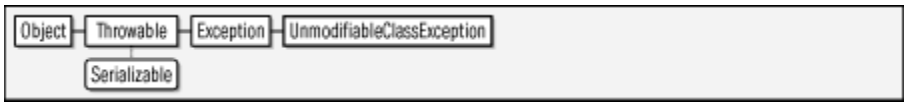
```
public interface Instrumentation {
// Public Instance Methods
     void addTransformer(ClassFileTransformer transformer);
     Class[ ] getAllLoadedClasses( );
     Class[ ] getInitiatedClasses(ClassLoader loader);
     long getObjectSize(Object objectToSize);
     boolean isRedefineClassesSupported( );
     void redefineClasses(ClassDefinition[ ] definitions) throws ClassNotFoundException,
     UnmodifiableClassException;
     boolean removeTransformer(ClassFileTransformer transformer);
}
```

UnmodifiableClassException — java.lang.instrument

Java 5.0 — ***serializable checked***

An exception of this type is thrown from `Instrumentation.redefineClasses( )` if a requested redefinition cannot be performed. This might occur, for example, if the redefinition attempts to add or remove members from the class.

Figure 10-84. java.lang.instrument.UnmodifiableClassException




```
public class UnmodifiableClassException extends Exception {
// Public Constructors
     public UnmodifiableClassException( );
     public UnmodifiableClassException(String s);
}
```

Thrown By

`Instrumentation.redefineClasses( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Package java.lang.management

Java 5.0

This package defines "management bean" or " MXBean" interfaces for managing and monitoring a running Java virtual machine. It relies on the JMX API of the `javax.management` package, which is not covered in this book. `ManagementFactory` is the main entry point to this API; it defines static factory methods for obtaining instances of the various management bean interfaces. These instances can then be queried for specific information about the Java VM. The *jconsole* tool shipped with the Java 5.0 JDK demonstrates the capabilites of this package.

Interfaces

```
public interface ClassLoadingMXBean;
public interface CompilationMXBean;
public interface GarbageCollectorMXBean extends MemoryManagerMXBean;
public interface MemoryManagerMXBean;
public interface MemoryMXBean;
public interface MemoryPoolMXBean;
public interface OperatingSystemMXBean;
public interface RuntimeMXBean;
public interface ThreadMXBean;
```

Enumerated Types

```
public enum MemoryType;
```

Classes

```
public class ManagementFactory;
public final class ManagementPermission extends java.security.BasicPermission;
public class MemoryNotificationInfo;
public class MemoryUsage;
public class ThreadInfo;
```

ClassLoadingMXBean java.lang.management

Java 5.0

This MXBean interface defines methods for determining how many classes are currently loaded in the Java VM, how many have ever been loaded, and how many have ever been unloaded. The `setVerbose( )` method turns verbose class loading output from the VM on or off.

```
public interface ClassLoadingMXBean {
// Public Instance Methods
     int getLoadedClassCount( );
     long getTotalLoadedClassCount( );
     long getUnloadedClassCount( );
     boolean isVerbose( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    void setVerbose(boolean value);
}
```

Returned By

`ManagementFactory.getClassLoadingMXBean( )`

CompilationMXBean java.lang.management

Java 5.0

This MXBean interface defines methods for querying the just-in-time compiler of the Java virtual machine. `getName( )` returns an identifying name for the compiler. If the implementation tracks compilation time, `getTotalCompilationTime( )` returns the approximate total compilation time in milliseconds.

```
public interface CompilationMXBean {
// Public Instance Methods
    String getName( );
    long getTotalCompilationTime( );
    boolean isCompilationTimeMonitoringSupported( );
}
```

Returned By

`ManagementFactory.getCompilationMXBean( )`

GarbageCollectorMXBean java.lang.management

Java 5.0

This MXBean interface allows monitoring of the number of garbage collections that have occurred and the approximate time they consumed in milliseconds. The methods return -1 to indicate that the garbage collector does not maintain those statistics. Note that VM implementations commonly have more than one garbage collector and use different collection strategies for new objects and old objects. Note also that this is a subinterface of `MemoryManagerMXBean`.

Figure 10-85. java.lang.management.GarbageCollectorMXBean


```
public interface GarbageCollectorMXBean extends MemoryManagerMXBean {
// Public Instance Methods
    long getCollectionCount( );
    long getCollectionTime( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

ManagementFactory java.lang.management

Java 5.0

This class provides the main entry point into the `java.lang.management` API. The static factory methods provide a convenient way to obtain instances of the various MXBean interfaces for the currently running Java virtual machine. The returned instances can then be queried to monitor memory usage, class loading, and other details of virtual machine performance.

To obtain an MXBean for a Java virtual machine running in another process, use the `newPlatformMXBeanProxy( )` method, specifying a `javax.management.MBeanServerConnection` as well as the name and type of the desired MXBean. The constant fields of this class define the names of the available beans. Note that the `javax.management` package is beyond the scope of this quick reference.

```
public class ManagementFactory {
// No Constructor
// Public Constants
     public static final String CLASS_LOADING_MXBEAN_NAME;         ="java.lang:type=ClassLoading"
     public static final String COMPILATION_MXBEAN_NAME;           ="java.lang:type=Compilation"
     public static final String GARBAGE_COLLECTOR_MXBEAN_DOMAIN_TYPE; ="java.lang:type=GarbageCollector"
     public static final String MEMORY_MANAGER_MXBEAN_DOMAIN_TYPE; ="java.lang:type=MemoryManager"
     public static final String MEMORY_MXBEAN_NAME;                ="java.lang:type=Memory"
     public static final String MEMORY_POOL_MXBEAN_DOMAIN_TYPE;    ="java.lang:type=MemoryPool"
     public static final String OPERATING_SYSTEM_MXBEAN_NAME;      ="java.lang:type=OperatingSystem"
     public static final String RUNTIME_MXBEAN_NAME;               ="java.lang:type=Runtime"
     public static final String THREAD_MXBEAN_NAME;                ="java.lang:type=Threading"
// Public Class Methods
     public static ClassLoadingMXBean getClassLoadingMXBean( );
     public static CompilationMXBean getCompilationMXBean( );
     public static java.util.List<GarbageCollectorMXBean> getGarbageCollectorMXBeans( );
     public static java.util.List<MemoryManagerMXBean> getMemoryManagerMXBeans( );
     public static MemoryMXBean getMemoryMXBean( );
     public static java.util.List<MemoryPoolMXBean> getMemoryPoolMXBeans( );
     public static OperatingSystemMXBean getOperatingSystemMXBean( );
     public static javax.management.MBeanServer getPlatformMBeanServer( );     synchronized
     public static RuntimeMXBean getRuntimeMXBean( );
     public static ThreadMXBean getThreadMXBean( );
     public static <T> T newPlatformMXBeanProxy(javax.management.
MBeanServerConnection connection, String mxbeanName, Class<T> mxbeanInterface)
throws java.io.IOException;
}
```

ManagementPermission java.lang.management

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 5.0 ***serializable permission***

This `java.security.Permission` subclass governs access to the Java VM monitoring and management capabilities of this package. The two defined targets for this permission are `control`, which grants permission to manage the VM, and `monitor`, which grants permission to monitor VM state. Fine-grained control over individual MXBeans is not supported.

Figure 10-86. java.lang.management.ManagementPermission


```
public final class ManagementPermission extends java.security.BasicPermission {
// Public Constructors
     public ManagementPermission(String name);
     public ManagementPermission(String name, String actions) throws IllegalArgumentException;
}
```

MemoryManagerMXBean **java.lang.management**

Java 5.0

This MXBean interface allows monitoring of a single memory manager (such as a garbage collector) in a Java VM. A VM implementation typically has more than one memory manager, and the `ManagementFactory` method `getMemoryManagerMXBeans( )` returns a `List` of objects of this type. Some or all of the objects in the returned list will also implement the `GarbageCollectorMXBean` subinterface.

Each memory manager may manage one or more memory pools, and `getMemoryPoolNames( )` returns the names of these pools. See also `ManagementFactory.getMemoryPoolMXBeans( )` and `MemoryPoolMXBean`.

```
public interface MemoryManagerMXBean {
// Public Instance Methods
     String[ ] getMemoryPoolNames( );
     String getName( );
     boolean isValid( );
}
```

Implementations

`GarbageCollectorMXBean`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

MemoryMXBean java.lang.management

Java 5.0

This MXBean interface allows monitoring of current memory usage information for heap memory (allocated objects) and nonheap memory (loaded classes and libraries). It also allows the garbage collector to be explicitly invoked and verbose garbage-collection related output to be turned on or off.

See `MemoryUsage` for details on how memory usage information is returned. See also `MemoryPoolMXBean` for a way to obtain both current and peak memory usage for individual memory pools.

```
public interface MemoryMXBean {
// Public Instance Methods
     void gc( );
     MemoryUsage getHeapMemoryUsage( );
     MemoryUsage getNonHeapMemoryUsage( );
     int getObjectPendingFinalizationCount( );
     boolean isVerbose( );
     void setVerbose(boolean value);
}
```

Returned By

`ManagementFactory.getMemoryMXBean( )`

MemoryNotificationInfo java.lang.management

Java 5.0

This class holds information about memory usage in a given memory pool and is generated when that usage crosses a threshold specified by a `MemoryPoolMXBean`. Use the `from( )` method to construct a `MemoryNotificationInfo` object from the user data of a `javax.management.Notification` object. Notifications and the `javax.management` package are beyond the scope of this book.

```
public class MemoryNotificationInfo {
// Public Constructors
     public MemoryNotificationInfo(String poolName, MemoryUsage usage, long count);
// Public Constants
     public static final String MEMORY_COLLECTION_THRESHOLD_EXCEEDED;
     ="java.management.memory.collection.threshold.exceeded"
     public static final String MEMORY_THRESHOLD_EXCEEDED;
     ="java.management.memory.threshold.exceeded"
// Public Class Methods
     public static MemoryNotificationInfo from(javax.management.openmbean.CompositeData cd);
// Public Instance Methods
     public long getCount( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public String getPoolName( );
    public MemoryUsage getUsage( );
}
```

MemoryPoolMXBean java.lang.management

Java 5.0

This MXBean interface allows monitoring of the current and peak memory usage for a single memory pool. Typical Java VM implementations segregate garbage-collected heap memory into two or more memory pools based on the age of the objects. Obtain a `List` of `MemoryPoolMXBean` instances with `ManagementFactory.getMemoryPoolMXBeans( ).getName( )` and `getType( )` return the name and type of each pool. `getUsage( )` and `getPeakUsage( )` return the current and peak memory usage for the pool in the form of a `MemoryUsage` object.

If `isUsageThresholdSupported( )` returns `true`, you can use `setUsageThreshold( )` to define a memory usage threshold. The `MemoryPoolMXBean` then keeps track of threshold crossings and issues notifications through the `javax.management.NotificationEmitter` API. You can register a `javax.management.NotificationListener` to receive these notifications. (Note that the `javax.management` package is not covered in this book.) Use `setCollectionUsageThreshold( )` instead to receive notifications when memory usage exceeds a specified threshold after a garbage collection pass.

```
public interface MemoryPoolMXBean {
// Public Instance Methods
    MemoryUsage getCollectionUsage( );
    long getCollectionUsageThreshold( );
    long getCollectionUsageThresholdCount( );
    String[ ] getMemoryManagerNames( );
    String getName( );
    MemoryUsage getPeakUsage( );
    MemoryType getType( );
    MemoryUsage getUsage( );
    long getUsageThreshold( );
    long getUsageThresholdCount( );
    boolean isCollectionUsageThresholdExceeded( );
    boolean isCollectionUsageThresholdSupported( );
    boolean isUsageThresholdExceeded( );
    boolean isUsageThresholdSupported( );
    boolean isValid( );
    void resetPeakUsage( );
    void setCollectionUsageThreshold(long threhsold);
    void setUsageThreshold(long threshold);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

MemoryType java.lang.management

Java 5.0 ***serializable comparable enum***

The constants defined by this enumerated type define the type of a memory pool as either heap or nonheap memory. See `MemoryPoolMXBean.getType( )`.

Figure 10-87. java.lang.management.MemoryType


```
public enum MemoryType {
// Enumerated Constants
     HEAP,
     NON_HEAP;
// Public Class Methods
     public static MemoryType valueOf(String name);
     public static final MemoryType[ ] values( );
// Public Methods Overriding Enum
     public String toString( );
}
```

Returned By

`MemoryPoolMXBean.getType( )`

MemoryUsage java.lang.management

Java 5.0

A `MemoryUsage` object represents a snapshot of memory usage for a specified type or pool of memory. Memory usage is measured as four `long` values, each of which represents a number of bytes. `getInit( )` returns the initial amount of memory that the Java VM requests from the operating system. `getUsed( )` returns the actual number of bytes used. `getCommitted( )` returns the number of bytes that the operating system has committed to the Java VM for this pool. These bytes may not all be in use, but they are not available to other processes running on the system. `getMax( )` returns the maximum amount of memory that the Java VM requests for this pool. `getMax( )` returns `-1` if there is no defined maximum value.

```
public class MemoryUsage {
// Public Constructors
     public MemoryUsage(long init, long used, long committed, long max);
// Public Class Methods
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    public static MemoryUsage from(javax.management.openmbean.CompositeData cd);
// Public Instance Methods
    public long getCommitted( );
    public long getInit( );
    public long getMax( );
    public long getUsed( );
// Public Methods Overriding Object
    public String toString( );
}
```

Passed To

`MemoryNotificationInfo.MemoryNotificationInfo( )`

Returned By

`MemoryMXBean.{getHeapMemoryUsage( ),getNonHeapMemoryUsage( )}, MemoryNotificationInfo.getUsage( ),MemoryPoolMXBean.{getCollectionUsage( ),getPeakUsage( ),getUsage( )}`

OperatingSystemMXBean java.lang.management

Java 5.0

This MXBean interface allows queries of the operating system name, version, and CPU architecture as well as the number of available CPUs.

```
public interface OperatingSystemMXBean {
// Public Instance Methods
    String getArch( );
    int getAvailableProcessors( );
    String getName( );
    String getVersion( );
}
```

Returned By

`ManagementFactory.getOperatingSystemMXBean( )`

RuntimeMXBean java.lang.management

Java 5.0

This MXBean interface provides access to the runtime configuration of the Java virtual machine, including system properties, command-line arguments, class path, virtual machine vendor and version, and so on. `getUptime( )` returns the uptime of the virtual machine in milliseconds.

```
public interface RuntimeMXBean {
// Public Instance Methods
    String getBootClassPath( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    String getClassPath( );
    java.util.List<String> getInputArguments( );
    String getLibraryPath( );
    String getManagementSpecVersion( );
    String getName( );
    String getSpecName( );
    String getSpecVendor( );
    String getSpecVersion( );
    long getStartTime( );
    java.util.Map<String,String> getSystemProperties( );
    long getUptime( );
    String getVmName( );
    String getVmVendor( );
    String getVmVersion( );
    boolean isBootClassPathSupported( );
}
```

Returned By

`ManagementFactory.getRuntimeMXBean( )`

ThreadInfo java.lang.management

Java 5.0

This class represents information about a thread from a `ThreadMXBean`. Some information, such as thread name, id, state, and stack trace are also available through the `java.lang.Thread` object. Other more useful information includes the object upon which a thread is waiting and the owner of the lock that the thread is trying to acquire. If `ThreadMXBean` indicates that thread contention monitoring is supported and enabled, the `ThreadInfo` methods `getBlockedCount( )` and `getBlockedTime( )` return the number of times the thread has blocked or waited and the amount of time it has spent in the blocked and waiting states.

```
public class ThreadInfo {
// No Constructor
// Public Class Methods
    public static ThreadInfo from(javax.management.openmbean.CompositeData cd);
// Public Instance Methods
    public long getBlockedCount( );
    public long getBlockedTime( );
    public String getLockName( );
    public long getLockOwnerId( );
    public String getLockOwnerName( );
    public StackTraceElement[ ] getStackTrace( );
    public long getThreadId( );
    public String getThreadName( );
    public Thread.State getThreadState( );
    public long getWaitedCount( );
    public long getWaitedTime( );
    public boolean isInNative( );
    public boolean isSuspended( );
// Public Methods Overriding Object
    public String toString( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Returned By

`ThreadMXBean.getThreadInfo( )`

ThreadMXBean java.lang.management

Java 5.0

This MXBean interface allows monitoring of thread usage in a Java VM. A number of methods, such as `getThreadCount( )` and `getPeakThreadCount( )`, return information about all running threads. Other methods return information about individual threads. Threads are identified by their thread id, which is a `long` integer. `getAllThreadIds( )` returns all ids as an array of `long`. Complete information, including stack trace, about a thread or set of threads can be obtained with the `getThreadInfo( )` methods, which return `ThreadInfo` objects.

If `isThreadCpuTimeSupported( )` returns `true`, you can enable thread timing with `setThreadCpuTimeEnabled( )` and query the runtime of a specific thread with `getThreadCpuTime( )` and `getThreadUserTime( )`. The values returned by these methods are measured in nanoseconds.

One of the potentially most useful methods of this interface is `findMonitorDeadlockedThreads( )`. It looks for cycles of threads that are deadlocked waiting to lock objects whose locks are held by other threads in the cycle.

```
public interface ThreadMXBean {
// Public Instance Methods
     long[ ] findMonitorDeadlockedThreads( );
     long[ ] getAllThreadIds( );
     long getCurrentThreadCpuTime( );
     long getCurrentThreadUserTime( );
     int getDaemonThreadCount( );
     int getPeakThreadCount( );
     int getThreadCount( );
     long getThreadCpuTime(long id);
     ThreadInfo getThreadInfo(long id);
     ThreadInfo[ ] getThreadInfo(long[ ] ids);
     ThreadInfo[ ] getThreadInfo(long[ ] ids, int maxDepth);
     ThreadInfo getThreadInfo(long id, int maxDepth);
     long getThreadUserTime(long id);
     long getTotalStartedThreadCount( );
     boolean isCurrentThreadCpuTimeSupported( );
     boolean isThreadContentionMonitoringEnabled( );
     boolean isThreadContentionMonitoringSupported( );
     boolean isThreadCpuTimeEnabled( );
     boolean isThreadCpuTimeSupported( );
     void resetPeakThreadCount( );
     void setThreadContentionMonitoringEnabled(boolean enable);
     void setThreadCpuTimeEnabled(boolean enable);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Returned By

`ManagementFactory.getThreadMXBean( )`

Package java.lang.ref

Java 1.2

The `java.lang.ref` package defines classes that allow Java programs to interact with the Java garbage collector. A `Reference` represents an indirect reference to an arbitrary object, known as the *referent*. `SoftReference`, `WeakReference`, and `PhantomReference` are three concrete subclasses of `Reference` that interact with the garbage collector in different ways, as explained in the individual class descriptions that follow. `ReferenceQueue` represents a linked list of `Reference` objects. Any `Reference` object may have a `ReferenceQueue` associated with it. A `Reference` object is *enqueued* on its `ReferenceQueue` at some point after the garbage collector determines that the referent object has become appropriately unreachable. (The exact level of unreachability depends on the type of `Reference` being used.) An application can monitor a `ReferenceQueue` to determine when referent objects enter a new reachability status.

Using the mechanisms defined in this package, you can implement a cache that grows and shrinks in size according to the amount of available system memory. Or, you can implement a hashtable that associates auxiliary information with arbitrary objects, but does not prevent those objects from being garbage-collected if they are otherwise unused. The mechanisms provided by this package are low-level ones, however, and typical applications do not use `java.lang.ref` directly. Instead, they rely on higher-level utilities built on top of the package. See `java.util.WeakHashMap` for one example.

In Java 5.0, the classes in this package have all been made into generic types. The type variable *`T`* represents the type of the object that is referred to.

Classes

```
public abstract class Reference<T>;
   public class PhantomReference<T> extends Reference<T>;
   public class SoftReference<T> extends Reference<T>;
   public class WeakReference<T> extends Reference<T>;
public class ReferenceQueue<T>;
```

PhantomReference<T> java.lang.ref

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Java 1.2

This class represents a reference to an object that does not prevent the referent object from being finalized by the garbage collector. When (or at some point after) the garbage collector determines that there are no more hard (direct) references to the referent object, that there are no `SoftReference` or `WeakReference` objects that refer to the referent, and that the referent has been finalized, it enqueues the `PhantomReference` object on the `ReferenceQueue` specified when the `PhantomReference` was created. This serves as notification that the object has been finalized and provides one last opportunity for any required cleanup code to be run.

To prevent a `PhantomReference` object from resurrecting its referent object, its `get( )` method always returns `null`, both before and after the `PhantomReference` is enqueued. Nevertheless, a `PhantomReference` is not automatically cleared when it is enqueued, so when you remove a `PhantomReference` from a `ReferenceQueue`, you must call its `clear( )` method or allow the `PhantomReference` object itself to be garbage-collected.

This class provides a more flexible mechanism for object cleanup than the `finalize( )` method does. Note that in order to take advantage of it, it is necessary to subclass `PhantomReference` and define a method to perform the desired cleanup. Furthermore, since the `get( )` method of a `PhantomReference` always returns `null`, such a subclass must also store whatever data is required for the cleanup operation.

Figure 10-88. java.lang.ref.PhantomReference<T>


```
public class PhantomReference<T> extends Reference<T> {
// Public Constructors
     public PhantomReference(T referent, ReferenceQueue<? super T> q);
// Public Methods Overriding Reference
     public T get( );                                                    constant
}
```

Reference<T> **java.lang.ref**

Java 1.2

This abstract class represents some type of indirect reference to a referent. `get( )` returns the referent if the reference has not been explicitly cleared by the `clear( )` method or

implicitly cleared by the garbage collector. There are three concrete subclasses of `Reference`. The garbage collector handles these subclasses differently and clears their references under different circumstances.

Each of the subclasses of `Reference` defines a constructor that allows a `ReferenceQueue` to be associated with the `Reference` object. The garbage collector places `Reference` objects onto their associated `ReferenceQueue` objects to provide notification about the state of the referent object. `isEnqueued( )` tests whether a `Reference` has been placed on the associated queue, and `enqueue( )` explicitly places it on the queue. `enqueue( )` returns `false` if the `Reference` object does not have an associated `ReferenceQueue`, or if it has already been enqueued.

```
public abstract class Reference<T> {
// No Constructor
// Public Instance Methods
     public void clear( );
     public boolean enqueue( );
     public T get( );
     public boolean isEnqueued( );
}
```

Subclasses

`PhantomReference`, `SoftReference`, `WeakReference`

Returned By

`ReferenceQueue.{poll( ), remove( )}`

ReferenceQueue<T> java.lang.ref

Java 1.2

This class represents a queue (or linked list) of `Reference` objects that have been enqueued because the garbage collector has determined that the referent objects to which they refer are no longer adequately reachable. It serves as a notification system for object-reachability changes. Use `poll( )` to return the first `Reference` object on the queue; the method returns `null` if the queue is empty. Use `remove( )` to return the first element on the queue, or, if the queue is empty, to wait for a `Reference` object to be enqueued. You can create as many `ReferenceQueue` objects as needed. Specify a `ReferenceQueue` for a `Reference` object by passing it to the `SoftReference( )`, `WeakReference( )`, or `PhantomReference( )` constructor.

A `ReferenceQueue` is required to use `PhantomReference` objects. It is optional with `SoftReference` and `WeakReference` objects; for these classes, the `get( )` method returns `null` if the referent object is no longer adequately reachable.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class ReferenceQueue<T> {
// Public Constructors
     public ReferenceQueue( );
// Public Instance Methods
     public Reference<? extends T> poll( );
     public Reference<? extends T> remove( ) throws InterruptedException;
     public Reference<? extends T> remove(long timeout) throws IllegalArgumentException,
InterruptedException;
}
```

Passed To

`PhantomReference.PhantomReference( ), SoftReference.SoftReference( ),WeakReference.WeakReference( )`

SoftReference<T> java.lang.ref

Java 1.2

This class represents a soft reference to an object. A `SoftReference` is not cleared while there are any remaining hard (direct) references to the referent. Once the referent is no longer in use (i.e., there are no remaining hard references to it), the garbage collector may clear the `SoftReference` to the referent at any time. However, the garbage collector does not clear a `SoftReference` until it determines that system memory is running low. In particular, the Java VM never throws an `OutOfMemoryError` without first clearing all soft references and reclaiming the memory of the referents. The VM may (but is not required to) clear soft references according to a least-recently-used ordering.

If a `SoftReference` has an associated `ReferenceQueue`, the garbage collector enqueues the `SoftReference` at some time after it clears the reference.

`SoftReference` is particularly useful for implementing object-caching systems that do not have a fixed size, but grow and shrink as available memory allows.

Figure 10-89. java.lang.ref.SoftReference<T>


```
public class SoftReference<T> extends Reference<T> {
// Public Constructors
     public SoftReference(T referent);
     public SoftReference(T referent, ReferenceQueue<? super T> q);
// Public Methods Overriding Reference
     public T get( );
}
```

WeakReference<T> java.lang.ref

Java 1.2

This class refers to an object in a way that does not prevent that referent object from being finalized and reclaimed by the garbage collector. When the garbage collector determines that there are no more hard (direct) references to the object, and that there are no `SoftReference` objects that refer to the object, it clears the `WeakReference` and marks the referent object for finalization. At some point after this, it also enqueues the `WeakReference` on its associated `ReferenceQueue`, if there is one, in order to provide notification that the referent has been reclaimed.

`WeakReference` is used by `java.util.WeakHashMap` to implement a hashtable that does not prevent the hashtable key object from being garbage-collected. `WeakHashMap` is useful when you want to associate auxiliary information with an object but do not want to prevent the object from being reclaimed.

Figure 10-90. java.lang.ref.WeakReference<T>


```
public class WeakReference<T> extends Reference<T> {
// Public Constructors
     public WeakReference(T referent);
     public WeakReference(T referent, ReferenceQueue<? super T> q);

}
```

Package java.lang.reflect

Java 1.1

The `java.lang.reflect` package contains the classes and interfaces that, along with `java.lang.Class`, comprise the Java Reflection API.

The `Constructor`, `Field`, and `Method` classes represent the constructors, fields, and methods of a class. Because these types all represent members of a class, they each implement the `Member` interface, which defines a simple set of methods that can be invoked for any class member. These classes allow information about the class members to be obtained, methods and constructors to be invoked, and fields to be queried and set.

Class member modifiers are represented as integers that specify a number of bit flags. The `Modifier` class defines static methods that help interpret the meanings of these flags. The `Array` class defines static methods for creating arrays and reading and writing array elements.

As of Java 1.3, the `Proxy` class allows the dynamic creation of new Java classes that implement a specified set of interfaces. When an interface method is invoked on an instance of such a proxy class, the invocation is delegated to an `InvocationHandler` object.

There have been a number of changes to this package to support the new language features of Java 5.0. The most important changes are support for querying the generic signature of classes, methods, constructors, and fields. `Class`, `Method` and `Constructor` implement the new `GenericDeclaration` interface, which provides access to the `TypeVariable` declarations of generic classes, methods, and constructors. In general, the package has been modified to add new generic versions of methods like `Field.getType( )` and `Method.getParameterTypes( )`. Instead of returning `Class` objects, the new generic methods, like `Field.getGenericType( )` and `Method.getGenericParameterTypes( )`, return `Type` objects. The `Type` interface is new in Java 5.0, and represents any kind of generic or nongeneric type. `Class` implements `Type`, so a `Type` object may simply be an ordinary `Class`. `Type` is also the super-interface for four other new interfaces: `ParameterizedType`, `TypeVariable`, `WildcardType` and `GenericArrayType`. A `Type` object that is not a `Class` should be an instance of one of these other interfaces, representing a generic type of some sort.

Support for reflection on annotations is provided by the `AnnotatedElement` interface which is implemented by `Class`, `Package`, `Method`, `Constructor` and `Field`. `Method` and `Constructor` also have new `getParameterAnnotations( )` for querying annotations on method parameters. Other, more minor changes in Java 5.0 include the `isEnumConstant( )` method of `Field` and the `isVarArgs( )` method of `Method` and `Constructor`.

Interfaces

```
public interface AnnotatedElement;
public interface GenericArrayType extends Type;
public interface GenericDeclaration;
public interface InvocationHandler;
public interface Member;
public interface ParameterizedType extends Type;
public interface Type;
public interface TypeVariable<D extends GenericDeclaration> extends Type;
public interface WildcardType extends Type;
```

Classes

```
public class AccessibleObject implements AnnotatedElement;
   public final class Constructor<T> extends AccessibleObject implements
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
GenericDeclaration, Member;
   public final class Field extends AccessibleObject implements Member;
   public final class Method extends AccessibleObject implements
GenericDeclaration, Member;
public final class Array;
public class Modifier;
public class Proxy implements Serializable;
public final class ReflectPermission extends java.security.BasicPermission;
```

Exceptions

```
public class InvocationTargetException extends Exception;
public class MalformedParameterizedTypeException extends RuntimeException;
public class UndeclaredThrowableException extends RuntimeException;
```

Errors

```
public class GenericSignatureFormatError extends ClassFormatError;
```

AccessibleObject java.lang.reflect

Java 1.2

This class is the superclass of the `Method`, `Constructor`, and `Field` classes; its methods provide a mechanism for trusted applications to work with `private`, `protected`, and default visibility members that would otherwise not be accessible through the Reflection API. This class is new as of Java 1.2; in Java 1.1, the `Method`, `Constructor`, and `Field` classes extended `Object` directly.

To use the `java.lang.reflect` package to access a member to which your code would not normally have access, pass `true` to the `setAccessible( )` method. If your code has an appropriate `ReflectPermission` (such as "suppressAccessChecks"), this allows access to the member as if it were declared `public`. The static version of `setAccessible( )` is a convenience method that sets the accessible flag for an array of members but performs only a single security check.

Figure 10-91. java.lang.reflect.AccessibleObject


```
public class AccessibleObject implements AnnotatedElement {
// Protected Constructors
     protected AccessibleObject( );
// Public Class Methods
     public static void setAccessible(AccessibleObject[ ] array, boolean flag)
     throws SecurityException;
// Public Instance Methods
     public boolean isAccessible( );
     public void setAccessible(boolean flag) throws SecurityException;
// Methods Implementing AnnotatedElement
5.0  public <T extends java.lang.annotation.Annotation> T getAnnotation(Class<T>
annotationClass);
5.0  public java.lang.annotation.Annotation[ ] getAnnotations( );
5.0  public java.lang.annotation.Annotation[ ] getDeclaredAnnotations( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
5.0  public boolean isAnnotationPresent(Class<? extends java.lang.annotation.
Annotation> annotationClass);
}
```

Subclasses

`Constructor`, `Field`, `Method`

AnnotatedElement java.lang.reflect

Java 5.0

This interface is implemented by the classes representing program elements that can be annotated in Java 5.0: `java.lang.Class`, `java.lang.Package`, `Method`, `Constructor`, and `Field`. The methods of this interface allow you to test for the presence of a specific annotation, query an annotation object of a specific type, or query all annotations present on an annotated element. `getDeclaredAnnotations( )` differs from `getAnnotations( )` in that it does not include inherited annotations. (See the `java.lang.annotation.Inherited` meta-annotation.) If no annotations are present, `getAnnotations( )` and `getDeclaredAnnotations( )` return an array of length zero rather than `null`. It is safe to modify the arrays returned by these methods.

See also the `getParameterAnnotations( )` methods of `Method` and `Constructor`, which provide access to annotations on method parameters.

```
public interface AnnotatedElement {
// Public Instance Methods
     <T extends java.lang.annotation.Annotation> T getAnnotation(Class<T> annotationType);
     java.lang.annotation.Annotation[ ] getAnnotations( );
     java.lang.annotation.Annotation[ ] getDeclaredAnnotations( );
     boolean isAnnotationPresent(Class<? extends java.lang.annotation.Annotation>
     annotationType);
}
```

Implementations

`Class`, `Package`, `AccessibleObject`

Array java.lang.reflect

Java 1.1

This class contains methods that allow you to set and query the values of array elements, to determine the length of an array, and to create new instances of arrays. Note that the `Array` class can manipulate only array values, not array types; Java data types, including

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

array types, are represented by `java.lang.Class`. Since the `Array` class represents a Java value, unlike the `Field`, `Method`, and `Constructor` classes, which represent class members, the `Array` class is significantly different (despite some surface similarities) from those other classes in this package. Most notably, all the methods of `Array` are static and apply to all array values, not just a specific field, method, or constructor.

The `get( )` method returns the value of the specified element of the specified array as an `Object`. If the array elements are of a primitive type, the value is converted to a wrapper object before being returned. You can also use `getInt( )` and related methods to query array elements and return them as specific primitive types. The `set( )` method and its primitive type variants perform the opposite operation. Also, the `getLength( )` method returns the length of the array.

The `newInstance( )` methods create new arrays. One version of this method is passed the number of elements in the array and the type of those elements. The other version of this method creates multidimensional arrays. Besides specifying the component type of the array, it is passed an array of numbers. The length of this array specifies the number of dimensions for the array to be created, and the values of each of the array elements specify the size of each dimension of the created array.

```
public final class Array {
// No Constructor
// Public Class Methods
     public static Object get(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static boolean getBoolean(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static byte getByte(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static char getChar(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static double getDouble(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static float getFloat(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static int getInt(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static int getLength(Object array)
throws IllegalArgumentException;     native
     public static long getLong(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static short getShort(Object array, int index)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static Object newInstance(Class<?> componentType, int length)
throws NegativeArraySizeException;
     public static Object newInstance(Class<?> componentType, int[ ] dimensions)
throws IllegalArgumentException, NegativeArraySizeException;
     public static void set(Object array, int index, Object value)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setBoolean(Object array, int index, boolean z)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setByte(Object array, int index, byte b)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setChar(Object array, int index, char c)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setDouble(Object array, int index, double d)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setFloat(Object array, int index, float f)
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setInt(Object array, int index, int i)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setLong(Object array, int index, long l)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
     public static void setShort(Object array, int index, short s)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;     native
}
```

Constructor<T> java.lang.reflect

Java 1.1

This class represents a constructor method of a class. Instances of `Constructor` are obtained by calling `getConstructor( )` and related methods of `java.lang.Class`. `Constructor` implements the `Member` interface, so you can use the methods of that interface to obtain the constructor name, modifiers, and declaring class. In addition, `getParameterTypes( )` and `getExceptionTypes( )` also return important information about the represented constructor.

In addition to these methods that return information about the constructor, the `newInstance( )` method allows the constructor to be invoked with an array of arguments in order to create a new instance of the class that declares the constructor. If any of the arguments to the constructor are of primitive types, they must be converted to their corresponding wrapper object types to be passed to `newInstance( )`. If the constructor causes an exception, the `Throwable` object it throws is wrapped within the `InvocationTargetException` that is thrown by `newInstance( )`. Note that `newInstance( )` is much more useful than the `newInstance( )` method of `java.lang.Class` because it can pass arguments to the constructor.

`Constructor` has been modified in Java 5.0 to support generics, annotations, and varargs. The changes are the same as the Java 5.0 changes to the `Method` class. Additionally, `Constructor` has been made a generic type in Java 5.0. The type variable *`T`* represents the type that the constructor constructs, and is used as the return type of the `newInstance( )` method.

Figure 10-92. java.lang.reflect.Constructor<T>


```
public final class Constructor<T> extends AccessibleObject implements
GenericDeclaration, Member {
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// No Constructor
// Public Instance Methods
     public Class<?>[ ] getExceptionTypes( );
5.0  public Type[ ] getGenericExceptionTypes( );
5.0  public Type[ ] getGenericParameterTypes( );
5.0  public java.lang.annotation.Annotation[ ][ ] getParameterAnnotations( );
     public Class<?>[ ] getParameterTypes( );
5.0  public boolean isVarArgs( );
     public T newInstance(Object ... initargs)
throws InstantiationException, IllegalAccessException, IllegalArgumentException,
InvocationTargetException;
5.0  public String toGenericString( );
// Methods Implementing GenericDeclaration
5.0  public TypeVariable<Constructor<T>>[ ] getTypeParameters( );
// Methods Implementing Member
     public Class<T> getDeclaringClass( );
     public int getModifiers( );
     public String getName( );
5.0  public boolean isSynthetic( );
// Public Methods Overriding AccessibleObject
5.0  public <T extends java.lang.annotation.Annotation> T getAnnotation
(Class<T> annotationClass);
5.0  public java.lang.annotation.Annotation[ ] getDeclaredAnnotations( );
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```

Returned By

`Class.{getConstructor( ),getConstructors( ), getDeclaredConstructor( ),getDeclaredConstructors( ), getEnclosingConstructor( )}`

Field java.lang.reflect

Java 1.1

This class represents a field of a class. Instances of `Field` are obtained by calling the `getField( )` and related methods of `java.lang.Class`. `Field` implements the `Member` interface, so once you have obtained a `Field` object, you can use `getName( )`, `getModifiers( )`, and `getDeclaringClass( )` to determine the name, modifiers, and class of the field. Additionally, `getType( )` returns the type of the field.

The `set( )` method sets the value of the represented field for a specified object. (If the represented field is `static`, no object need be specified, of course.) If the field is of a primitive type, its value can be specified using a wrapper object of type `Boolean`, `Integer`, and so on, or it can be set using the `setBoolean( )`, `setInt( )`, and related methods. Similarly, the `get( )` method queries the value of the represented field for a specified object and returns the field value as an `Object`. Various other methods query the field value and return it as various primitive types.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

In Java 5.0, `Field` implements `AnnotatedElement` to support reflection on field annotations. The new `getGenericType( )` method supports reflection on the generic type of fields, and `isEnumConstant( )` supports fields of `enum` types.

Figure 10-93. java.lang.reflect.Field


```
public final class Field extends AccessibleObject implements Member {
// No Constructor
// Public Instance Methods
     public Object get(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public boolean getBoolean(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public byte getByte(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public char getChar(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public double getDouble(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public float getFloat(Object obj)
throws IllegalArgumentException, IllegalAccessException;
5.0  public Type getGenericType( );
     public int getInt(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public long getLong(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public short getShort(Object obj)
throws IllegalArgumentException, IllegalAccessException;
     public Class<?> getType( );
5.0  public boolean isEnumConstant( );
     public void set(Object obj, Object value)
throws IllegalArgumentException, IllegalAccessException;
     public void setBoolean(Object obj, boolean z)
throws IllegalArgumentException, IllegalAccessException;
     public void setByte(Object obj, byte b)
throws IllegalArgumentException, IllegalAccessException;
     public void setChar(Object obj, char c)
throws IllegalArgumentException, IllegalAccessException;
     public void setDouble(Object obj, double d)
throws IllegalArgumentException, IllegalAccessException;
     public void setFloat(Object obj, float f)
throws IllegalArgumentException, IllegalAccessException;
     public void setInt(Object obj, int i)
throws IllegalArgumentException, IllegalAccessException;
     public void setLong(Object obj, long l)
throws IllegalArgumentException, IllegalAccessException;
     public void setShort(Object obj, short s)
throws IllegalArgumentException, IllegalAccessException;
5.0  public String toGenericString( );
// Methods Implementing Member
     public Class<?> getDeclaringClass( );
     public int getModifiers( );
     public String getName( );
5.0  public boolean isSynthetic( );
// Public Methods Overriding AccessibleObject
5.0  public <T extends java.lang.annotation.Annotation> T getAnnotation
(Class<T> annotationClass);
5.0  public java.lang.annotation.Annotation[ ] getDeclaredAnnotations( );
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Returned By

`Class.{getDeclaredField( ),getDeclaredFields( ),getField( ), getFields( )}`

GenericArrayType java.lang.reflect

Java 5.0

This interface extends `Type` and represents a one-dimensional array of some element `Type`. Note that in the case of multidimensional arrays, the `Type` returned by `getGenericComponentType( )` is itself a `GenericArrayType`.

Figure 10-94. java.lang.reflect.GenericArrayType


```
public interface GenericArrayType extends Type {
// Public Instance Methods
     Type getGenericComponentType( );
}
```

GenericDeclaration java.lang.reflect

Java 5.0

This interface is implemented by the classes that represent program elements that can be made generic: `java.lang.Class` as well as `Method` and `Constructor`. It provides access to the type variables declared by the generic type, method, or constructor. `getTypeParameters( )` never returns `null`: if there are no declared type variables, it returns a zero-length array.

```
public interface GenericDeclaration {
// Public Instance Methods
     TypeVariable<?>[ ] getTypeParameters( );
}
```

Implementations

`Class`, `Constructor`, `Method`

Returned By

`TypeVariable.getGenericDeclaration( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

GenericSignatureFormatError — java.lang.reflect

Java 5.0 — *serializable error*

An error of this type is thrown if the Java interpreter tries to load a class file that contains malformed generic signature information.

Figure 10-95. java.lang.reflect.GenericSignatureFormatError


```
public class GenericSignatureFormatError extends ClassFormatError {
// Public Constructors
     public GenericSignatureFormatError( );
}
```

InvocationHandler — java.lang.reflect

Java 1.3

This interface defines a single `invoke( )` method that is called whenever a method is invoked on a dynamically created `Proxy` object. Every `Proxy` object has an associated `InvocationHandler` object that is specified when the `Proxy` is instantiated. All method invocations on the proxy object are translated into calls to the `invoke( )` method of the `InvocationHandler`.

The first argument to `invoke( )` is the `Proxy` object through which the method was invoked. The second argument is a `Method` object that represents the method that was invoked. Call the `getDeclaringClass( )` method of this `Method` object to determine the interface in which the method was declared. This may be a superinterface of one of the specified interfaces or even `java.lang.Object` when the method invoked is `toString( )`, `hashCode( )`, or one of the other `Object` methods. The third argument to `invoke( )` is the array of method arguments. Any primitive type arguments are wrapped in their corresponding object wrappers (e.g., `Boolean`, `Integer`, `Double`).

The value returned by `invoke( )` becomes the return value of the proxy object method invocation and must be of an appropriate type. If the proxy object method returns a

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

primitive type, `invoke( )` should return an instance of the corresponding wrapper class. `invoke( )` can throw any unchecked (i.e., runtime) exceptions or any checked exceptions declared by the proxy object method. If `invoke( )` throws a checked exception that is not declared by the proxy object, that exception is wrapped within an unchecked `UndeclaredThrowableException` that is thrown in its place.

```
public interface InvocationHandler {
// Public Instance Methods
     Object invoke(Object proxy, Method method, Object[ ] args) throws Throwable;
}
```

Passed To

`java.lang.reflect.Proxy.{newProxyInstance( ),Proxy( )}`

Returned By

`java.lang.reflect.Proxy.getInvocationHandler( )`

Type Of

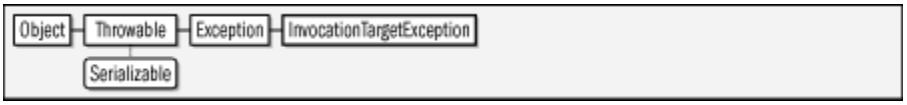
`java.lang.reflect.Proxy.h`

InvocationTargetException java.lang.reflect

Java 1.1 ***serializable checked***

An object of this class is thrown by `Method.invoke( )` and `Constructor.newInstance( )` when an exception is thrown by the method or constructor invoked through those methods. The `InvocationTargetException` class serves as a wrapper around the object that was thrown; that object can be retrieved with the `getTargetException( )` method. In Java 1.4 and later, all exceptions can be "chained" in this way, and `getTargetException( )` is superseded by the more general `getCause( )` method.

Figure 10-96. java.lang.reflect.InvocationTargetException




```
public class InvocationTargetException extends Exception {
// Public Constructors
     public InvocationTargetException(Throwable target);
     public InvocationTargetException(Throwable target, String s);
// Protected Constructors
     protected InvocationTargetException( );
// Public Instance Methods
     public Throwable getTargetException( );
// Public Methods Overriding Throwable
1.4  public Throwable getCause( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Thrown By

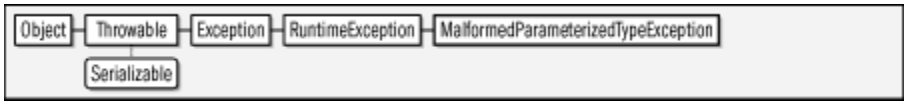
`Constructor.newInstance( ),Method.invoke( )`

MalformedParameterizedTypeException java.lang.reflect

Java 5.0 ***serializable unchecked***

An exception of this type is thrown during reflection if the generic type information contained in a class file is syntactically correct but semantically wrong. An example would be if the number of type parameters in a `ParameterizedType` differs from the number of type variables declared by the generic type. See also `GenericSignatureFormatError`. Although this type is not an `Error`, it does indicate a malformed class file and should not arise in common practice.

Figure 10-97. java.lang.reflect.MalformedParameterizedTypeException




```
public class MalformedParameterizedTypeException extends RuntimeException {
// Public Constructors
     public MalformedParameterizedTypeException( );
}
```

Member java.lang.reflect

Java 1.1

This interface defines the methods shared by all members (fields, methods, and constructors) of a class. `getName( )` returns the name of the member, `getModifiers( )` returns its modifiers, and `getDeclaringClass( )` returns the `Class` object that represents the class of which the member is a part. `isSynthetic( )` returns `true` if the member is one that does not appear in the source code but was introduced by the compiler.

```
public interface Member {
// Public Constants
     public static final int DECLARED;     =1
     public static final int PUBLIC;       =0
// Public Instance Methods
     Class getDeclaringClass( );
     int getModifiers( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    String getName( );
5.0 boolean isSynthetic( );
}
```

Implementations

`Constructor`, `Field`, `Method`

Method java.lang.reflect

Java 1.1

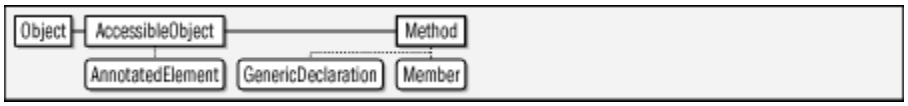
This class represents a method. Instances of `Method` are obtained by calling the `getMethod( )` and related methods of `java.lang.Class`. `Method` implements the `Member` interface, so you can use the methods of that interface to obtain the method name, modifiers, and declaring class. In addition, `getReturnType( )`, `getParameterTypes( )`, and `getExceptionTypes( )` also return important information about the represented method.

Perhaps most importantly, the `invoke( )` method allows the method represented by the `Method` object to be invoked with a specified array of argument values. If any of the arguments are of primitive types, they must be converted to their corresponding wrapper object types in order to be passed to `invoke( )`. If the represented method is an instance method (i.e., if it is not `static`), the instance on which it should be invoked must also be passed to `invoke( )`. The return value of the represented method is returned by `invoke( )`. If the return value is a primitive value, it is first converted to the corresponding wrapper type. If the invoked method causes an exception, the `Throwable` object it throws is wrapped within the `InvocationTargetException` that is thrown by `invoke( )`.

In Java 5.0, `Method` implements `GenericDeclaration` to support reflection on the type variables defined by generic methods and `AnnotatedElement` to support reflection on method annotations. Additionally, `getParameterAnnotations( )` supports reflection on method parameter annotations. The new methods `getGenericReturnType( )`, `getGenericParameterTypes( )`, and `getGenericExceptionTypes( )` support reflection on generic method signatures. Finally, the new `isVarArgs( )` method returns true if the method was declared using Java 5.0 varargs syntax.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-98. java.lang.reflect.Method




```
public final class Method extends AccessibleObject implements GenericDeclaration, Member {
// No Constructor
// Public Instance Methods
5.0  public Object getDefaultValue( );
     public Class<?>[ ] getExceptionTypes( );
5.0  public Type[ ] getGenericExceptionTypes( );
5.0  public Type[ ] getGenericParameterTypes( );
5.0  public Type getGenericReturnType( );
5.0  public java.lang.annotation.Annotation[ ][ ] getParameterAnnotations( );
     public Class<?>[ ] getParameterTypes( );
     public Class<?> getReturnType( );
     public Object invoke(Object obj, Object... args)
throws IllegalAccessException, IllegalArgumentException, InvocationTargetException;
5.0  public boolean isBridge( );
5.0  public boolean isVarArgs( );
5.0  public String toGenericString( );
// Methods Implementing GenericDeclaration
5.0  public TypeVariable<Method>[ ] getTypeParameters( );
// Methods Implementing Member
     public Class<?> getDeclaringClass( );
     public int getModifiers( );
     public String getName( );
5.0  public boolean isSynthetic( );
// Public Methods Overriding AccessibleObject
5.0  public <T extends java.lang.annotation.Annotation> T getAnnotation
(Class<T> annotationClass);
5.0  public java.lang.annotation.Annotation[ ] getDeclaredAnnotations( );
// Public Methods Overriding Object
     public boolean equals(Object obj);
     public int hashCode( );
     public String toString( );
}
```

Passed To

`java.lang.annotation.AnnotationTypeMismatchException.AnnotationTypeMismatchException( ), InvocationHandler.invoke( )`

Returned By

`Class.{getDeclaredMethod( ), getDeclaredMethods( ), getEnclosingMethod( ), getMethod( ), getMethods( )}, java.lang.annotation.AnnotationTypeMismatchException.element( )`

Modifier java.lang.reflect

Java 1.1

This class defines a number of constants and static methods that can interpret the integer values returned by the `getModifiers( )` methods of the `Field`, `Method`, and `Constructor` classes. The `isPublic( )`, `isAbstract( )`, and related methods return `true` if the modifier value includes the specified modifier; otherwise, they return

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`false`. The constants defined by this class specify the various bit flags used in the modifiers value. You can use these constants to test for modifiers if you want to perform your own boolean algebra.

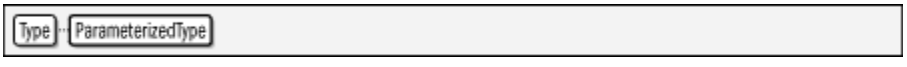
```
public class Modifier {
// Public Constructors
     public Modifier( );
// Public Constants
     public static final int ABSTRACT;     =1024
     public static final int FINAL;        =16
     public static final int INTERFACE;    =512
     public static final int NATIVE;       =256
     public static final int PRIVATE;      =2
     public static final int PROTECTED;    =4
     public static final int PUBLIC;       =1
     public static final int STATIC;       =8
1.2  public static final int STRICT;   =2048
     public static final int SYNCHRONIZED; =32
     public static final int TRANSIENT;    =128
     public static final int VOLATILE;     =64
// Public Class Methods
     public static boolean isAbstract(int mod);
     public static boolean isFinal(int mod);
     public static boolean isInterface(int mod);
     public static boolean isNative(int mod);
     public static boolean isPrivate(int mod);
     public static boolean isProtected(int mod);
     public static boolean isPublic(int mod);
     public static boolean isStatic(int mod);
1.2  public static boolean isStrict(int mod);
     public static boolean isSynchronized(int mod);
     public static boolean isTransient(int mod);
     public static boolean isVolatile(int mod);
     public static String toString(int mod);
}
```

ParameterizedType java.lang.reflect

Java 5.0

This subinterface of `Type` represents a parameterized type. `getRawType( )` returns the base type that has been parameterized. `getActualTypeArguments( )` returns the type parameters as a `Type[ ]`. Note that these parameters may themselves be `ParameterizedType` objects. `getOwnerType( )` is used with parameterized types that are also nested types: it returns the generic type of the containing type.

Figure 10-99. java.lang.reflect.ParameterizedType




```
public interface ParameterizedType extends Type {
// Public Instance Methods
     Type[ ] getActualTypeArguments( );
     Type getOwnerType( );
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    Type getRawType( );
}
```

Proxy **java.lang.reflect**

Java 1.3 ***serializable***

This class defines a simple but powerful API for dynamically generating a *proxy class*. A proxy class implements a specified list of interfaces and delegates invocations of the methods defined by those interfaces to a separate invocation handler object.

The static `getProxyClass( )` method dynamically creates a new `Class` object that implements each of the interfaces specified in the supplied `Class[ ]` array. The newly created class is defined in the context of the specified `ClassLoader`. The `Class` returned by `getProxyClass( )` is a subclass of `Proxy`. Every class that is dynamically generated by `getProxyClass( )` has a single public constructor, which expects a single argument of type `InvocationHandler`. You can create an instance of the dynamic proxy class by using the `Constructor` class to invoke this constructor. Or, more simply, you can combine the call to `getProxyClass( )` with the constructor call by calling the static `newProxyInstance( )` method, which both defines and instantiates a proxy class.

Every instance of a dynamic proxy class has an associated `InvocationHandler` object. All method calls made on a proxy class are translated into calls to the `invoke( )` method of this `InvocationHandler` object, which can handle the call in any way it sees fit. The static `getInvocationHandler( )` method returns the `InvocationHandler` object for a given proxy object. The static `isProxyClass( )` method returns `true` if a specified `Class` object is a dynamically generated proxy class.

Figure 10-100. java.lang.reflect.Proxy


```
public class Proxy implements Serializable {
// Protected Constructors
     protected Proxy(InvocationHandler h);
// Public Class Methods
     public static InvocationHandler getInvocationHandler(Object proxy)
throws IllegalArgumentException;
     public static Class<?> getProxyClass(ClassLoader loader, Class<?>
... interfaces) throws IllegalArgumentException;
     public static boolean isProxyClass(Class<?> cl);
     public static Object newProxyInstance(ClassLoader loader, Class<?>[ ]
interfaces, InvocationHandler h) throws IllegalArgumentException;
// Protected Instance Fields
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    protected InvocationHandler h;
}
```

ReflectPermission java.lang.reflect

Java 1.2 ***serializable permission***

This class is a `java.security.Permission` that governs access to `private`, `protected`, and default-visibility methods, constructors, and fields through the Java Reflection API. In Java 1.2, the only defined name, or target, for `ReflectPermission` is "suppressAccessChecks". This permission is required to call the `setAccessible( )` method of `AccessibleObject`. Unlike some `Permission` subclasses, `ReflectPermission` does not use a list of actions. See also `AccessibleObject`.

System administrators configuring security policies should be familiar with this class, but application programmers should never need to use it directly.

Figure 10-101. java.lang.reflect.ReflectPermission


```
public final class ReflectPermission extends java.security.BasicPermission {
// Public Constructors
    public ReflectPermission(String name);
    public ReflectPermission(String name, String actions);
}
```

Type java.lang.reflect

Java 5.0

This interface has no members but is implemented or extended by any type that represents a generic or nongeneric type. `java.lang.Class` implements this interface. `Type` is also extended by four interfaces that represent four specific kinds of generic types: `ParameterizedType`, `TypeVariable`, `WildcardType`, and `GenericArrayType`.

```
public interface Type {
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Implementations

`Class`, `GenericArrayType`, `ParameterizedType`, `TypeVariable`, `WildcardType`

Returned By

`Class.{getGenericInterfaces( ), getGenericSuperclass( )}`, `Constructor.{getGenericExceptionTypes( ), getGenericParameterTypes( )}`, `Field.getGenericType( )`, `GenericArrayType.getGenericComponentType( )`, `Method.{getGenericExceptionTypes( ), getGenericParameterTypes( ), getGenericReturnType( )}`, `ParameterizedType.{getActualTypeArguments( ), getOwnerType( ), getRawType( )}`, `TypeVariable.getBounds( )`, `WildcardType.{getLowerBounds( ), getUpperBounds( )}`

TypeVariable<D extends GenericDeclaration> java.lang.reflect

Java 5.0

This interface extends `Type` and represents the generic type represented by a type variable. `getName( )` returns the name of the type variable, as it was declared in Java source code. `getBounds( )` returns an array of `Type` objects that serve as the upper bounds for the variable. The returned array is never empty: if the type variable has no bounds declared, the single element of the array is `Object.class`. The `getGenericDeclaration( )` method returns the `Class`, `Method`, or `Constructor` that declared this type variable (each of these classes implements the `GenericDeclaration` interface). Note that `TypeVariable` is itself a generic type and is parameterized with the kind of `GenericDeclaration` that declared the variable.

Figure 10-102. java.lang.reflect.TypeVariable<D extends GenericDeclaration>


```
public interface TypeVariable<D extends GenericDeclaration> extends Type {
// Public Instance Methods
     Type[ ] getBounds( );
     D getGenericDeclaration( );
     String getName( );
}
```

Returned By

`Class.getTypeParameters( )`, `Constructor.getTypeParameters( )`, `GenericDeclaration.getTypeParameters( )`, `Method.getTypeParameters( )`

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

UndeclaredThrowableException java.lang.reflect

Java 1.3 ***serializable unchecked***

Thrown by a method of a `Proxy` object if the `invoke( )` method of the proxy's `InvocationHandler` throws a checked exception not declared by the original method. This class serves as an unchecked exception wrapper around the checked exception. Use `getUndeclaredThrowable( )` to obtain the checked exception thrown by `invoke( )`. In Java 1.4 and later, all exceptions can be "chained" in this way, and `getUndeclaredThrowable( )` is superseded by the more general `getCause( )` method.

Figure 10-103. java.lang.reflect.UndeclaredThrowableException


```
public class UndeclaredThrowableException extends RuntimeException {
// Public Constructors
     public UndeclaredThrowableException(Throwable undeclaredThrowable);
     public UndeclaredThrowableException(Throwable undeclaredThrowable, String s);
// Public Instance Methods
     public Throwable getUndeclaredThrowable( );
// Public Methods Overriding Throwable
1.4  public Throwable getCause( );
}
```

WildcardType java.lang.reflect

Java 5.0

This interface extends `Type` and represents a generic type declared with a bounded or unbounded wildcard. `getUpperBounds( )` returns the upper bounds of the wildcard. The returned array always includes at least one element. If no upper bound is declared, `Object.class` is the implicit upper bound. `getLowerBounds( )` returns the lower bounds of the wildcard. If no lower bound is declared, this method returns an empty array .

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Figure 10-104. java.lang.reflect.WildcardType


```
public interface WildcardType extends Type {
// Public Instance Methods
     Type[ ] getLowerBounds( );
     Type[ ] getUpperBounds( );
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01