

## CHAPTER 11

### *java.lang, java.lang.ref, and java.lang.reflect*

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 two specialized subpackages. `java.lang.ref` defines “reference” classes that refer to objects without preventing the garbage collector from reclaiming those objects. `java.lang.reflect` allows Java 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 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 1.4, both `String` and `StringBuffer` implement the `CharSequence` interface which allows instances of both 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.

`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.

*Interfaces:*

```
public interface CharSequence;  
public interface Cloneable;  
public interface Comparable;  
public interface Runnable;
```

*Classes:*

```
public class Object;  
    public final class Boolean implements Serializable;  
    public final class Character implements Comparable, Serializable;  
    public static class Character.Subset;  
        public static final class Character.UnicodeBlock extends Character.Subset;  
    public final class Class implements Serializable;  
    public abstract class ClassLoader;  
    public final class Compiler;  
    public final class Math;  
    public abstract class Number implements Serializable;  
        public final class Byte extends Number implements Comparable;  
        public final class Double extends Number implements Comparable;  
        public final class Float extends Number implements Comparable;  
        public final class Integer extends Number implements Comparable;  
        public final class Long extends Number implements Comparable;  
        public final class Short extends Number implements Comparable;  
    public class Package;  
    public abstract class Process;  
    public class Runtime;  
    public class SecurityManager;  
    public final class StackTraceElement implements Serializable;  
    public final class StrictMath;  
    public final class String implements CharSequence, Comparable, Serializable;  
    public final class StringBuffer implements CharSequence, Serializable;  
    public final class System;  
    public class Thread implements Runnable;  
    public class ThreadGroup;
```



*Package java.lang*

```
public class ThreadLocal;
    public class InheritableThreadLocal extends ThreadLocal;
    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 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 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;
```

## AbstractMethodError

Java 1.0

java.lang

*serializable error*

This error signals an attempt to invoke an abstract method.



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

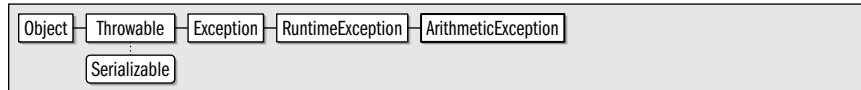
## ArithmeticException

Java 1.0

java.lang

*serializable unchecked*

This RuntimeException signals an exceptional arithmetic condition, such as integer division by zero.



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

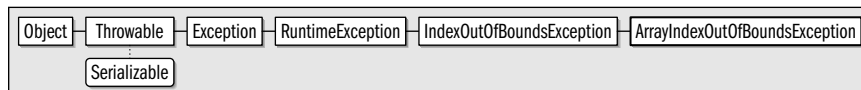
## ArrayIndexOutOfBoundsException

Java 1.0

java.lang

*serializable unchecked*

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



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

*Thrown By:* Too many methods to list.



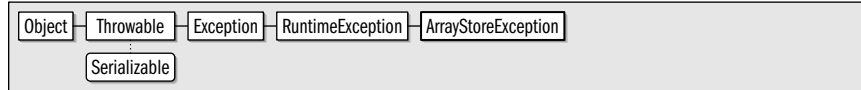
## ArrayStoreException

Java 1.0

java.lang

serializable unchecked

This exception signals an attempt to store the wrong type of object into an array.



```

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

## AssertionError

Java 1.4

java.lang

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.



```

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 1.0

java.lang

serializable

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 class method **valueOf()** parses a string and returns the **Boolean** object it represents. In Java 1.4, two new class methods convert primitive **boolean** values to **Boolean** and **String** objects.



```

public final class Boolean implements Serializable {
// 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 TYPE;
// Public Class Methods
public static boolean getBoolean(String name);
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();
// Public Methods Overriding Object
public boolean equals(Object obj);
public int hashCode();
public String toString();
}

```

*Passed To:* javax.swing.DefaultDesktopManager.setWaslcon()

*Returned By:* Boolean.valueOf(), javax.swing.filechooser.FileSystemView.isTraversable(),  
 javax.swing.filechooser.FileView.isTraversable(),  
 javax.swing.plaf.basic.BasicFileChooserUI.BasicFileView.isHidden()

*Type Of:* java.awt.font.TextAttribute.{RUN\_DIRECTION\_LTR, RUN\_DIRECTION\_RTL,  
 STRIKETHROUGH\_ON, SWAP\_COLORS\_ON}, Boolean.{FALSE, TRUE}

## Byte

Java 1.1

java.lang

serializable comparable

This class provides an 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.



```

public final class Byte extends Number implements Comparable {
// Public Constructors
public Byte(byte value);
public Byte(String s) throws NumberFormatException;
// Public Constants
public static final byte MAX_VALUE;
=127

```



## Byte

```
public static final byte MIN_VALUE;                                     =-128
public static final Class 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;
public static Byte valueOf(String s, int radix) throws NumberFormatException;
// Public Instance Methods
1.2 public int compareTo(Byte anotherByte);
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// Public Methods Overriding Number
public byte byteValue();
public double doubleValue();
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();
}
```

*Passed To:* Byte.compareTo()

*Returned By:* Byte.{decode(), valueOf()}

## Character

Java 1.0

java.lang

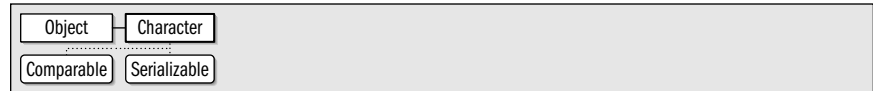
*serializable comparable*

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. (And some of the methods are useful only 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()` 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, for sufficiently large radix values, with 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 that represent numbers but



are not decimal digits, such as Roman numerals. Finally, the static `toString()` method returns a `String` of length 1 that contains the specified `char` value.



```

public final class Character implements Comparable, Serializable {
// 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
    1.1 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
    public static final int MAX_RADIX;                             =36
    public static final char MAX_VALUE;                           ='[bsol ]uFFFF'
    public static final int MIN_RADIX;                             =2
    public static final char MIN_VALUE;                           ='[bsol ]u0000'
    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
    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 TYPE;
    1.1 public static final byte UNASSIGNED;                       =0
    1.1 public static final byte UPPERCASE_LETTER;                 =1
    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
    1.4 public static final byte DIRECTIONALITY_OTHER_NEUTRALS;    =13

```



## Character

```
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
// Inner Classes
1.2 public static class Subset;
1.2 public static final class UnicodeBlock extends Character.Subset;
// Public Class Methods
    public static int digit(char ch, int radix);
    public static char forDigit(int digit, int radix);
1.4 public static byte getDirectionality(char c);
1.1 public static int getNumericValue(char ch);
1.1 public static int getType(char ch);
    public static boolean isDefined(char ch);
    public static boolean isDigit(char ch);
1.1 public static boolean isIdentifierIgnorable(char ch);
1.1 public static boolean isISOControl(char ch);
1.1 public static boolean isJavaIdentifierPart(char ch);
1.1 public static boolean isJavaIdentifierStart(char ch);
    public static boolean isLetter(char ch);
    public static boolean isLetterOrDigit(char ch);
    public static boolean isLowerCase(char ch);
1.4 public static boolean isMirrored(char c);
1.1 public static boolean isSpaceChar(char ch);
    public static boolean isTitleCase(char ch);
1.1 public static boolean isUnicodeIdentifierPart(char ch);
1.1 public static boolean isUnicodeIdentifierStart(char ch);
    public static boolean isUpperCase(char ch);
1.1 public static boolean isWhitespace(char ch);
    public static char toLowerCase(char ch);
1.4 public static String toString(char c);
    public static char toTitleCase(char ch);
    public static char toUpperCase(char ch);
// Public Instance Methods
    public char charValue();
1.2 public int compareTo(Character anotherCharacter);
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// 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);
}
```

*Passed To:* java.awt.AWTKeyStroke.getAWTKeyStroke(), Character.compareTo(),  
javax.swing.KeyStroke.getKeyStroke()



**Character.Subset**

Java 1.2

java.lang

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:** java.awt.im.InputSubset, Character.UnicodeBlock

**Passed To:** java.awt.im.InputContext.setCharacterSubsets(),  
java.awt.im.spi.InputMethod.setCharacterSubsets()

**Character.UnicodeBlock**

Java 1.2

java.lang

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 updates this class to a new version of the Unicode standard and defines a number of new block constants. The static method `of()` takes a character 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.

```
public static final class Character.UnicodeBlock extends Character.Subset {
    // No Constructor
    // Public Constants
    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;
    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;
    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;
    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;
```



## *Character.UnicodeBlock*

```
1.4 public static final Character.UnicodeBlock CJK_UNIFIED_IDEOGRAPHS_EXTENSION_A;  
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;  
public static final Character.UnicodeBlock CYRILLIC;  
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;  
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;  
public static final Character.UnicodeBlock HEBREW;  
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;  
1.4 public static final Character.UnicodeBlock KHMER;  
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;  
public static final Character.UnicodeBlock MALAYALAM;  
public static final Character.UnicodeBlock MATHEMATICAL_OPERATORS;  
public static final Character.UnicodeBlock MISCELLANEOUS_SYMBOLS;  
public static final Character.UnicodeBlock MISCELLANEOUS_TECHNICAL;  
1.4 public static final Character.UnicodeBlock MONGOLIAN;  
1.4 public static final Character.UnicodeBlock MYANMAR;  
public static final Character.UnicodeBlock NUMBER_FORMS;  
1.4 public static final Character.UnicodeBlock OGHAM;  
public static final Character.UnicodeBlock OPTICAL_CHARACTER_RECOGNITION;  
public static final Character.UnicodeBlock ORIYA;  
public static final Character.UnicodeBlock PRIVATE_USE_AREA;  
1.4 public static final Character.UnicodeBlock RUNIC;  
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;
```



```

    public static final Character.UnicodeBlock SURROGATES_AREA;
1.4 public static final Character.UnicodeBlock SYRIAC;
    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;
1.4 public static final Character.UnicodeBlock UNIFIED_CANADIAN_ABORIGINAL_SYLLABICS;
1.4 public static final Character.UnicodeBlock YI_RADICALS;
1.4 public static final Character.UnicodeBlock YI_SYLLABLES;
// Public Class Methods
    public static Character.UnicodeBlock of(char c);
}

```

*Returned By:* Character.UnicodeBlock.of()

*Type Of:* Too many fields to list.

## CharSequence

Java 1.4

java.lang

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, 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
    public abstract char charAt(int index);
    public abstract int length();
    public abstract CharSequence subSequence(int start, int end);
    public abstract String toString();
}

```

*Implementations:* `String`, `StringBuffer`, `java.nio.CharBuffer`

*Passed To:* `java.nio.CharBuffer.wrap()`, `java.nio.charset.CharsetEncoder.canEncode()`, `java.util.regex.Matcher.reset()`, `java.util.regex.Pattern.{matcher(), matches(), split()}`

*Returned By:* `CharSequence.subSequence()`, `String.subSequence()`, `StringBuffer.subSequence()`, `java.nio.CharBuffer.subSequence()`

## Class

Java 1.0

java.lang

*serializable*

This class represents a Java class or interface, or, as of Java 1.1, any 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.



## Class

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.

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 only works 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.

`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`.

```
Object -- Class -- Serializable
public final class Class implements Serializable {
// 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;
// Property Accessor Methods (by property name)
    1.1 public boolean isArray(); native
    1.1 public Class[] getClasses();
        public ClassLoader getClassLoader();
    1.1 public Class getComponentType(); native
    1.1 public java.lang.reflect.Constructor[] getConstructors() throws SecurityException;
    1.1 public Class[] getDeclaredClasses() throws SecurityException;
    1.1 public java.lang.reflect.Constructor[] getDeclaredConstructors() throws SecurityException;
    1.1 public java.lang.reflect.Field[] getDeclaredFields() throws SecurityException;
    1.1 public java.lang.reflect.Method[] getDeclaredMethods() throws SecurityException;
    1.1 public Class getDeclaringClass(); native
    1.1 public java.lang.reflect.Field[] getFields() throws SecurityException;
        public boolean isInterface(); native
        public Class[] getInterfaces(); native
    1.1 public java.lang.reflect.Method[] getMethods() throws SecurityException;
    1.1 public int getModifiers(); native
        public String getName(); native
    1.2 public Package getPackage();
    1.1 public boolean isPrimitive(); native
    1.2 public java.security.ProtectionDomain getProtectionDomain();
    1.1 public Object[] getSigners(); native
        public Class getSuperclass(); native
// Public Instance Methods
    1.4 public boolean desiredAssertionStatus();
    1.1 public java.lang.reflect.Constructor getConstructor(Class[] parameterTypes) throws NoSuchMethodException,
        SecurityException;
    1.1 public java.lang.reflect.Constructor getDeclaredConstructor(Class[] parameterTypes)
        throws NoSuchMethodException, SecurityException;
```



## ClassCircularityError

```
1.1 public java.lang.reflect.Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException;
1.1 public java.lang.reflect.Method getDeclaredMethod(String name, Class[ ] parameterTypes)
    throws NoSuchMethodException, SecurityException;
1.1 public java.lang.reflect.Field getField(String name) throws NoSuchFieldException, SecurityException;
1.1 public java.lang.reflect.Method getMethod(String name, Class[ ] parameterTypes) throws NoSuchMethodException,
    SecurityException;
1.1 public java.net.URL getResource(String name);
1.1 public java.io.InputStream getResourceAsStream(String name);
1.1 public boolean isAssignableFrom(Class cls); native
1.1 public boolean isInstance(Object obj); native
    public Object newInstance() throws InstantiationException, IllegalAccessException;
// Public Methods Overriding Object
    public String toString();
}
```

*Passed To:* Too many methods to list.

*Returned By:* Too many methods to list.

*Type Of:* Too many fields to list.

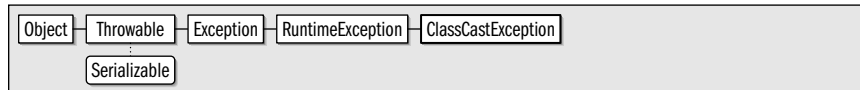
## ClassCastException

Java 1.0

java.lang

serializable unchecked

This exception signals an invalid cast of an object to a type of which it is not an instance.



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

*Thrown By:* javax.rmi.PortableRemoteObject.narrow(),  
javax.rmi.CORBA.PortableRemoteObjectDelegate.narrow(),  
org.xml.sax.helpers.ParserFactory.makeParser()

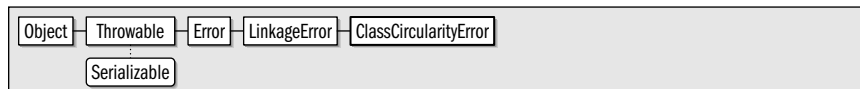
## ClassCircularityError

Java 1.0

java.lang

serializable error

This error signals that a circular dependency has been detected while performing initialization for a class.



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



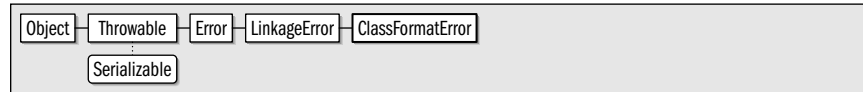
**ClassFormatError**

Java 1.0

java.lang

serializable error

This error signals an error in the binary format of a class file.



```

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

*Subclasses:* UnsupportedClassVersionError

*Thrown By:* ClassLoader.defineClass()

**ClassLoader**

Java 1.0

java.lang

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 getResource(String name);
    1.1 public static java.io.InputStream getResourceAsStream(String name);
}
  
```



```

1.2 public static java.util Enumeration getSystemResources(String name) throws java.io.IOException;
// Public Instance Methods
1.4 public void clearAssertionStatus(); synchronized
1.2 public final ClassLoader getParent();
1.1 public java.net.URL getResource(String name);
1.1 public java.io.InputStream getResourceAsStream(String name);
1.2 public final java.util Enumeration 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
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); native
1.2 protected java.net.URL findResource(String name); constant
1.2 protected java.util Enumeration 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:** Too many methods to list.

**Returned By:** Class.getClassLoader(), ClassLoader.{getParent(), getSystemClassLoader()},  
 SecurityManager.currentClassLoader(), Thread.getContextClassLoader(),  
 java.rmi.server.RMIClassLoader.getClassLoader(), java.rmi.server.RMIClassLoaderSpi.getClassLoader(),  
 java.security.ProtectionDomain.getClassLoader()

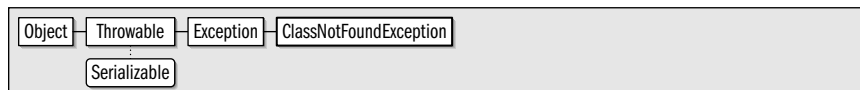
## ClassNotFoundException

Java 1.0

java.lang

serializable checked

This exception 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 exception with `getException()`, or with the newer, more general `getCause()`.



```

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

```



## *ClassNotFoundException*

```
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 1.0**

**java.lang**

*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.

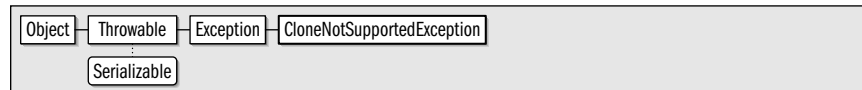
## **CloneNotSupportedException**

**Java 1.0**

**java.lang**

*serializable checked*

This exception signals that the **clone()** method has been called for an object of a class that does not implement the **Cloneable** interface.



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

*Subclasses:* **java.rmi.server.ServerCloneException**

*Thrown By:* Too many methods to list.

## **Comparable**

**Java 1.2**

**java.lang**

*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()**. As of Java 1.2, many of the key classes in the Java API have been modified to implement this interface.

The **compareTo()** object 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 {
    // Public Instance Methods
    public abstract int compareTo(Object o);
}
```

**Implementations:** Too many classes to list.

**Passed To:** `javax.imageio.metadata.IIOMetadataFormatImpl.addObjectValue()`,  
`javax.swing.SpinnerDateModel.setEnd()`, `setStart()`, `SpinnerDateModel()`,  
`javax.swing.SpinnerNumberModel.setMaximum()`, `setMinimum()`, `SpinnerNumberModel()`,  
`javax.swing.text.InternationalFormatter.setMaximum()`, `setMinimum()`

**Returned By:** `javax.imageio.metadata.IIOMetadataFormat.getObjectMaxValue()`,  
`getObjectMinValue()`, `javax.imageio.metadata.IIOMetadataFormatImpl.getObjectMaxValue()`,  
`getObjectMinValue()`, `javax.swing.SpinnerDateModel.getEnd()`, `getStart()`,  
`javax.swing.SpinnerNumberModel.setMaximum()`, `getMinimum()`,  
`javax.swing.text.InternationalFormatter.setMaximum()`, `getMinimum()`

## Compiler

Java 1.0

### java.lang

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);
    public static boolean compileClass(Class clazz);
    public static boolean compileClasses(String string);
    public static void disable();
    public static void enable();
}
```

## Double

Java 1.0

### java.lang

*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.



## Double

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 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 equals 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 the `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 on objects and is useful when ordering and sorting arrays of double values.

`doubleToLongBits()`, `doubleToRawLongBits()`, 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).



```
public final class Double extends Number implements Comparable {
// 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
1.1 public static final Class 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;
    public static String toString(double d);
    public static Double valueOf(String s) throws NumberFormatException;
// Public Instance Methods
1.2 public int compareTo(Double anotherDouble);
    public boolean isInfinite();
    public boolean isNaN();
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// 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();
}

```

*Passed To:* Double.compareTo()

*Returned By:* Double.valueOf()

## Error

Java 1.0

java.lang

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.



```

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:* java.awt.AWTError, AssertionError, LinkageError, ThreadDeath, VirtualMachineError, java.nio.charset.CoderMalfunctionError, javax.xml.parsers.FactoryConfigurationError, javax.xml.transform.TransformerFactoryConfigurationError

*Passed To:* java.rmi.ServerError.ServerError()

## Exception

Java 1.0

java.lang

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.



```

public class Exception extends Throwable {
// Public Constructors

```



## Exception

```
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:* Too many methods to list.

*Returned By:* java.awt.event.InvocationEvent.getException(),  
java.security.PrivilegedActionException.getException(),  
javax.xml.parsers.FactoryConfigurationError.getException(),  
javax.xml.transform.TransformerFactoryConfigurationError.getException(),  
org.omg.CORBA.Environment.exception(), org.xml.sax.SAXException.getException()

*Thrown By:* java.awt.im.spi.InputMethodDescriptor.createInputMethod(),  
java.beans.Expression.getValue(), java.beans.Statement.execute(),  
java.rmi.server.RemoteCall.executeCall(), java.rmi.server.RemoteRef.invoke(),  
java.rmi.server.Skeleton.dispatch(), java.security.PrivilegedExceptionAction.run(),  
javax.naming.spi.DirectoryManager.getObjectInstance(),  
javax.naming.spi.DirObjectFactory.getObjectInstance(),  
javax.naming.spi.NamingManager.getObjectInstance(),  
javax.naming.spi.ObjectFactory.getObjectInstance()

*Type Of:* java.io.WriteAbortedException.detail, java.rmi.server.ServerCloneException.detail

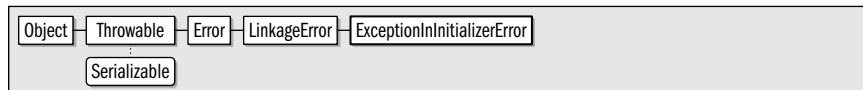
## ExceptionInInitializerError

Java 1.1

java.lang

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.



```
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  
}
```

## Float

Java 1.0

java.lang

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 sim-



ilar to `Double`, and defines the same set of useful methods and constants as that class does. See `Double` for details.



```

public final class Float extends Number implements Comparable {
// Public Constructors
    public Float(String s) throws NumberFormatException;
    public Float(float value);
    public Float(double 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
1.1 public static final Class 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;
    public static String toString(float f);
    public static Float valueOf(String s) throws NumberFormatException;
// Public Instance Methods
1.2 public int compareTo(Float anotherFloat);
    public boolean isInfinite();
    public boolean isNaN();
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// 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();
}
  
```

*Passed To:* `Float.compareTo()`

*Returned By:* `Float.valueOf()`

*Type Of:* Too many fields to list.



## *IllegalAccessError*

### **IllegalAccessError**

**Java 1.0**

**java.lang**

**serializable error**

This error signals an attempted use of a class, method, or field that is not accessible.



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

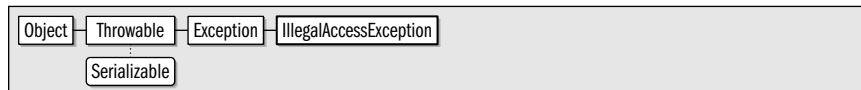
### **IllegalAccessException**

**Java 1.0**

**java.lang**

**serializable checked**

This exception signals that a class or initializer is not accessible. It is thrown by `Class.newInstance()`.



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

*Thrown By:* Too many methods to list.

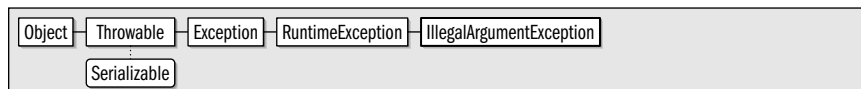
### **IllegalArgumentException**

**Java 1.0**

**java.lang**

**serializable unchecked**

This exception signals an illegal argument to a method. See subclasses `IllegalThreadStateException` and `NumberFormatException`.



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

*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.regex.PatternSyntaxException`

*Thrown By:* Too many methods to list.



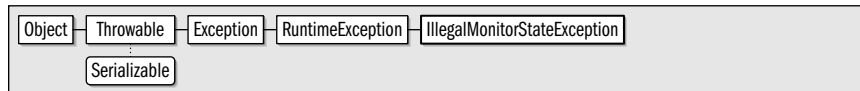
## IllegalMonitorStateException

Java 1.0

java.lang

serializable unchecked

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



```

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

## IllegalStateException

Java 1.1

java.lang

serializable unchecked

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



```

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

**Subclasses:** `java.awt.IllegalComponentStateException`, `java.awt.dnd.InvalidDnOperationException`, `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`

**Thrown By:** Too many methods to list.

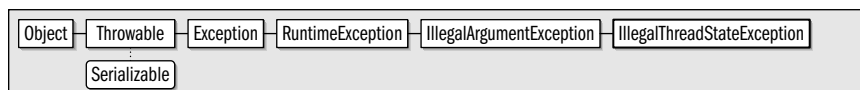
## IllegalThreadStateException

Java 1.0

java.lang

serializable unchecked

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



```

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



## *IncompatibleClassChangeError*

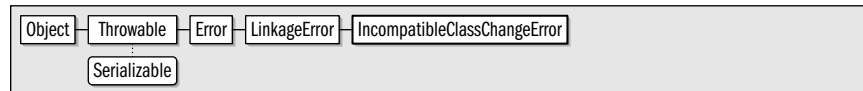
### **IncompatibleClassChangeError**

**Java 1.0**

**java.lang**

**serializable error**

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



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

**Subclasses:** AbstractMethodError, IllegalAccessException, InstantiationException, NoSuchFieldError, NoSuchMethodError

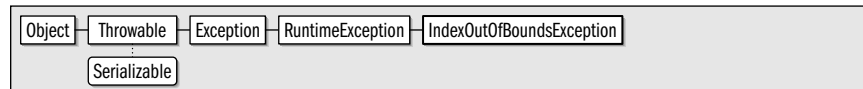
### **IndexOutOfBoundsException**

**Java 1.0**

**java.lang**

**serializable unchecked**

This exception signals that an index is out of bounds. See the subclasses `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException`.



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

**Subclasses:** `ArrayIndexOutOfBoundsException`, `StringIndexOutOfBoundsException`

**Thrown By:** `java.awt.Toolkit.createCustomCursor()`, `java.awt.print.Book.getPageFormat()`, `getPrintable()`, `setPage()`, `java.awt.print.Pageable.getPageFormat()`, `getPrintable()`

### **InheritableThreadLocal**

**Java 1.2**

**java.lang**

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 superclass-to-subclass inheritance; instead, it is parent-thread-to-child-thread inheritance.

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

```

Object -- ThreadLocal -- InheritableThreadLocal

public class InheritableThreadLocal extends ThreadLocal {
// Public Constructors
    public InheritableThreadLocal();
// Protected Instance Methods
    protected Object childValue(Object parentValue);
}

```

## InstantiationError

Java 1.0

java.lang

serializable error

This error signals an attempt to instantiate an interface or abstract class.

```

Object -- Throwable -- Error -- LinkageError -- IncompatibleClassChangeError -- InstantiationError
      |
      +-- Serializable

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

```

## InstantiationException

Java 1.0

java.lang

serializable checked

This exception signals an attempt to instantiate an interface or an abstract class.

```

Object -- Throwable -- Exception -- InstantiationException
      |
      +-- Serializable

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

```

*Thrown By:* `Class.newInstance()`, `java.lang.reflect.Constructor.newInstance()`, `javax.swing.UIManager.setLookAndFeel()`, `org.xml.sax.helpers.ParserFactory.makeParser()`

## Integer

Java 1.0

java.lang

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



## Integer

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.



```
public final class Integer extends Number implements Comparable {
    // Public Constructors
    public Integer(int value);
    public Integer(String s) throws NumberFormatException;
    // Public Constants
    public static final int MAX_VALUE;                                     =2147483647
    public static final int MIN_VALUE;                                   =-2147483648
    1.1 public static final Class TYPE;
    // Public Class Methods
    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);
    public static int parseInt(String s) throws NumberFormatException;
    public static int parseInt(String s, int radix) throws NumberFormatException;
    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);
    public static Integer valueOf(String s) throws NumberFormatException;
    public static Integer valueOf(String s, int radix) throws NumberFormatException;
    // Public Instance Methods
    1.2 public int compareTo(Integer anotherInteger);
    // Methods Implementing Comparable
    1.2 public int compareTo(Object o);
    // 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();
}
```

**Passed To:** `Integer.compareTo()`, `Integer.getInteger()`, `javax.swing.JInternalFrame.setLayer()`

**Returned By:** `Integer.decode()`, `Integer.getInteger()`, `Integer.valueOf()`,  
`javax.swing.JLayeredPane.getObjectForLayer()`

**Type Of:** `java.awt.font.TextAttribute`.{`SUPERSCRIPT_SUB`, `SUPERSCRIPT_SUPER`,  
`UNDERLINE_LOW_DASHED`, `UNDERLINE_LOW_DOTTED`, `UNDERLINE_LOW_GRAY`,  
`UNDERLINE_LOW_ONE_PIXEL`, `UNDERLINE_LOW_TWO_PIXEL`, `UNDERLINE_ON`},  
`javax.swing.JLayeredPane`.{`DEFAULT_LAYER`, `DRAW_LAYER`, `FRAME_CONTENT_LAYER`, `MODAL_LAYER`,  
`PALETTE_LAYER`, `POPUP_LAYER`}



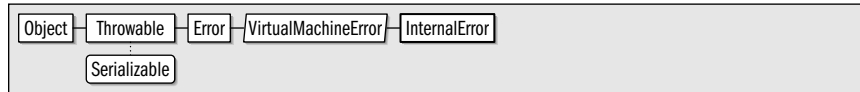
**InternalError**

Java 1.0

java.lang

serializable error

This error signals an internal error in the Java interpreter.



```

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

**InterruptedException**

Java 1.0

java.lang

serializable checked

This exception signals that the thread has been interrupted.



```

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

*Thrown By:* Too many methods to list.

**LinkageError**

Java 1.0

java.lang

serializable error

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



```

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

*Subclasses:* ClassCircularityError, ClassFormatError, ExceptionInInitializerError, IncompatibleClassChangeError, NoClassDefFoundError, UnsatisfiedLinkError, VerifyError

**Long**

Java 1.0

java.lang

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.



*Long*

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.



```

public final class Long extends Number implements Comparable {
// 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
1.1 public static final Class TYPE;
// Public Class Methods
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);
    public static long parseLong(String s) throws NumberFormatException;
    public static long parseLong(String s, int radix) throws NumberFormatException;
    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);
    public static Long valueOf(String s) throws NumberFormatException;
    public static Long valueOf(String s, int radix) throws NumberFormatException;
// Public Instance Methods
1.2 public int compareTo(Long anotherLong);
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// 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();
}

```

*Passed To:* Long.{compareTo(), getLong()}

**Returned By:** Long.{decode(), getLong(), valueOf()}



**Math**

Java 1.0

**java.lang**

This class defines constants for the mathematical values  $e$  and  $\pi$  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 pseudo-random 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. In Java 1.3, applications that require strict platform-independence of results should use **StrictMath** instead.

```
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);                                strictfp
    public static long abs(long a);                             strictfp
    public static float abs(float a);                           strictfp
    public static double abs(double a);                         strictfp
    public static double acos(double a);                       strictfp
    public static double asin(double a);                       strictfp
    public static double atan(double a);                       strictfp
    public static double atan2(double y, double x);            strictfp
    public static double ceil(double a);                       strictfp
    public static double cos(double a);                        strictfp
    public static double exp(double a);                        strictfp
    public static double floor(double a);                      strictfp
    public static double IEEERemainder(double f1, double f2);  strictfp
    public static double log(double a);                        strictfp
    public static int max(int a, int b);                         strictfp
    public static long max(long a, long b);                     strictfp
    public static float max(float a, float b);                 strictfp
    public static double max(double a, double b);              strictfp
    public static int min(int a, int b);                         strictfp
    public static long min(long a, long b);                     strictfp
    public static float min(float a, float b);                 strictfp
    public static double min(double a, double b);              strictfp
    public static double pow(double a, double b);              strictfp
    public static double random();                             strictfp
    public static double rint(double a);                       strictfp
    public static int round(float a);                           strictfp
    public static long round(double a);                         strictfp
    public static double sin(double a);                         strictfp
    public static double sqrt(double a);                       strictfp
    public static double tan(double a);                         strictfp
    1.2 public static double toDegrees(double angdeg);         strictfp
    1.2 public static double toRadians(double angdeg);         strictfp
}
```



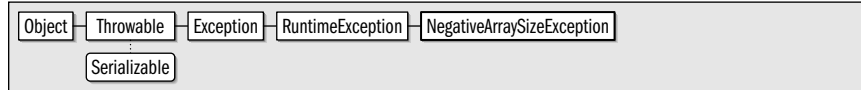
## NegativeArraySizeException

Java 1.0

java.lang

serializable unchecked

This exception signals an attempt to allocate an array with fewer than zero elements.



```

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

*Thrown By:* java.lang.reflect.Array.newInstance()

## NoClassDefFoundError

Java 1.0

java.lang

serializable error

This error signals that the definition of a specified class cannot be found.



```

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

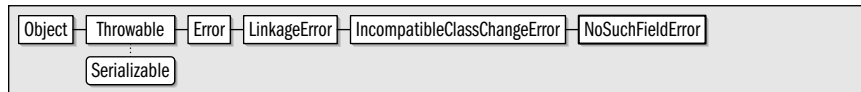
## NoSuchFieldError

Java 1.0

java.lang

serializable error

This error signals that a specified field cannot be found.



```

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

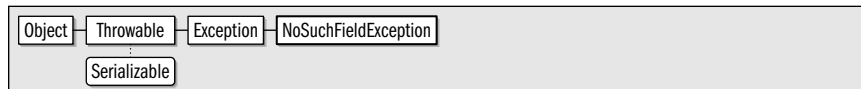
## NoSuchFieldException

Java 1.1

java.lang

serializable checked

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





## *NullPointerException*

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

*Thrown By:* Class.{getDeclaredField(), getField()}

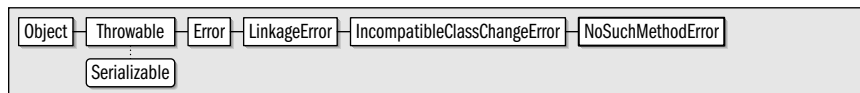
## **NoSuchMethodError**

**Java 1.0**

java.lang

*serializable error*

This error signals that a specified method cannot be found.



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

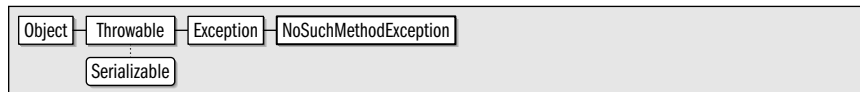
## **NoSuchMethodException**

**Java 1.0**

java.lang

*serializable checked*

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



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

*Thrown By:* Class.{getConstructor(), getDeclaredConstructor(), getDeclaredMethod(), getMethod()}

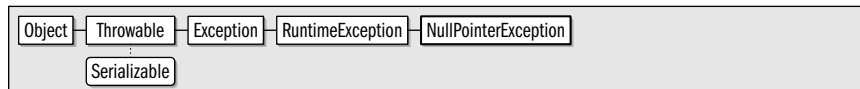
## **NullPointerException**

**Java 1.0**

java.lang

*serializable unchecked*

This exception signals an attempt to access a field or invoke a method of a null object.



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

*Thrown By:* java.awt.print.PrinterJob.setPageable(), org.xml.sax.helpers.ParserFactory.makeParser()



## Number

### Number

Java 1.0

java.lang

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.

```
Object -- Number -- Serializable

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`

*Passed To:* Too many methods to list.

*Returned By:* Too many methods to list.

### NumberFormatException

Java 1.0

java.lang

serializable unchecked

This exception signals an illegal number format.

```
Object -- Throwable -- Exception -- RuntimeException -- IllegalArgumentException -- NumberFormatException
                                           |
                                           +-- Serializable

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

*Thrown By:* Too many methods to list.

### Object

Java 1.0

java.lang

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.

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 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:* Too many fields to list.

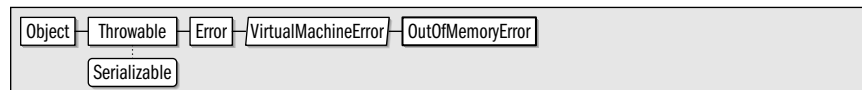
## OutOfMemoryError

Java 1.0

java.lang

serializable error

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



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

## Package

Java 1.2

java.lang

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`



## Package

object is not defined unless at least one class 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.

```
public class Package {
    // No Constructor
    // Public Class Methods
    public static Package getPackage(String name);
    public static Package[] getPackages();
    // Property Accessor Methods (by property name)
    public String getImplementationTitle();
    public String getImplementationVendor();
    public String getImplementationVersion();
    public String getName();
    public boolean isSealed();
    public boolean isSealed(java.net.URL url);
    public String getSpecificationTitle();
    public String getSpecificationVendor();
    public String getSpecificationVersion();
    // Public Instance Methods
    public boolean isCompatibleWith(String desired) throws NumberFormatException;
    // Public Methods Overriding Object
    public int hashCode();
    public String toString();
}
```

*Returned By:* `Class.getPackage()`, `ClassLoader.{definePackage(), getPackage(), getPackages()}`, `Package.{getPackage(), getPackages()}`, `java.net.URLClassLoader.definePackage()`

## Process

Java 1.0

### java.lang

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



## Runtime

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();  
    // Property Accessor Methods (by property name)  
    public abstract java.io.InputStream getErrorStream();  
    public abstract java.io.InputStream getInputStream();  
    public abstract java.io.OutputStream getOutputStream();  
    // Public Instance Methods  
    public abstract void destroy();  
    public abstract int exitValue();  
    public abstract int waitFor() throws InterruptedException;  
}
```

*Returned By:* `Runtime.exec()`

## Runnable

Java 1.0

`java.lang`

*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  
    public abstract void run();  
}
```

*Implementations:* `java.awt.image.renderable.RenderableImageProducer`, `Thread`,  
`java.util.TimerTask`, `javax.swing.text.AsyncBoxView.ChildState`

*Passed To:* `java.awt.EventQueue.{invokeAndWait(), invokeLater()}`,  
`java.awt.event.InvocationEvent.InvocationEvent()`, `Thread.Thread()`,  
`javax.swing.SwingUtilities.{invokeAndWait(), invokeLater()}`, `javax.swing.text.AbstractDocument.render()`,  
`javax.swing.text.Document.render()`, `javax.swing.text.LayoutQueue.addTask()`

*Returned By:* `javax.swing.text.LayoutQueue.waitForWork()`

*Type Of:* `java.awt.event.InvocationEvent.runnable`

## Runtime

Java 1.0

`java.lang`

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()`.



## Runtime

`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);
    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 cmd, 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);
}
```

*Returned By:* `Runtime.getRuntime()`

## RuntimeException

Java 1.0

`java.lang`

*serializable unchecked*

This exception type is not used directly, but serves as a superclass of a group of runtime exceptions that need not be declared in the `throws` clause of a method definition.



## RuntimePermission

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.



```
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);
}
```

*Subclasses:* Too many classes to list.

## RuntimePermission

Java 1.2

java.lang

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.

Supported `RuntimePermission` names are: “accessClassInPackage.package”, “accessDeclaredMembers”, “createClassLoader”, “createSecurityManager”, “defineClassInPackage.package”, “exitVM”, “getClassLoader”, “getProtectionDomain”, “loadLibrary.library\_name”, “modifyThread”, “modifyThreadGroup”, “queuePrintJob”, “readFileDescriptor”, “setContextClassLoader”, “setFactory”, “setIO”, “setSecurityManager”, “stopThread”, and “writeFileDescriptor”.

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 `RuntimePermission` directly.



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



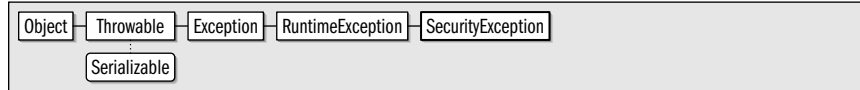
## SecurityException

Java 1.0

java.lang

serializable unchecked

This exception signals that an operation is not permitted for security reasons.



```

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

*Subclasses:* java.rmi.RMISecurityException, java.security.AccessControlException

*Thrown By:* Too many methods to list.

## SecurityManager

Java 1.0

java.lang

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 a need to subclass it. If so, the method returns silently; if not, it throws a **SecurityException**. **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();
// Property Accessor Methods (by property name)
    public Object getSecurityContext(); default:AccessControlContext
1.1 public ThreadGroup getThreadGroup();
// 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);
// 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;
}

```

**Subclasses:** java.rmi.RMISecurityManager

**Passed To:** System.setSecurityManager()

**Returned By:** System.getSecurityManager()

## Short

Java 1.1

java.lang

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.

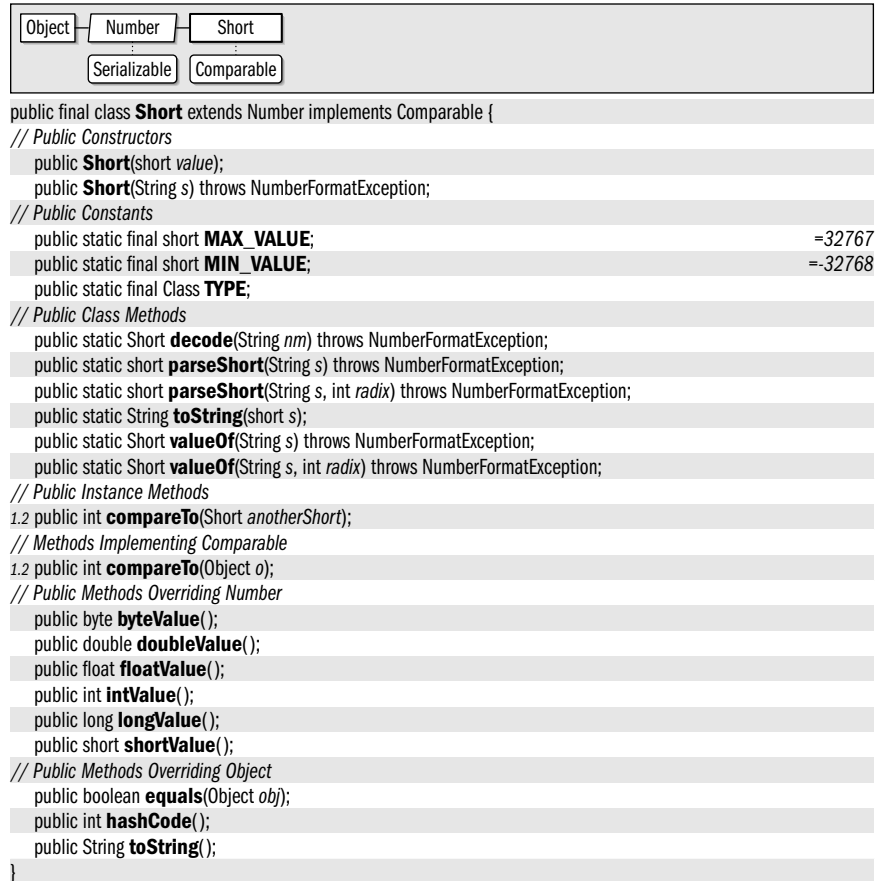
Most of the static methods of this class can convert a **String** to a **Short** object or a **short** value; the four **parseShort()** and **valueOf()** methods parse a number from the specified string using an optionally specified radix and return it in one of these two forms. The **decode()** method 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

Short object to a string. Most of the remaining methods convert a Short to various primitive numeric types.



*Passed To:* Short.compareTo()

*Returned By:* Short.{decode(), valueOf()}

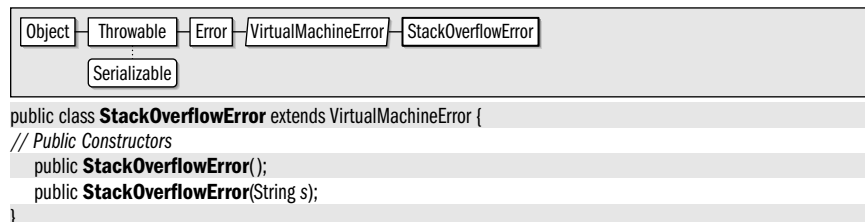
## StackOverflowError

Java 1.0

java.lang

serializable error

This error signals that a stack overflow has occurred within the Java interpreter.





**StackTraceElement**

Java 1.4

java.lang

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).



```

public final class StackTraceElement implements Serializable {
    // No Constructor
    // Property Accessor Methods (by property name)
    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:* `Throwable.getStackTrace()`

**StrictMath**

Java 1.3

java.lang

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, down to the least significant bit, for all possible arguments. 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);                  strictfp
    public static long abs(long a);                strictfp
    public static float abs(float a);              strictfp
    public static double abs(double a);            strictfp
    public static double acos(double a);          native strictfp
    public static double asin(double a);          native strictfp
    public static double atan(double a);          native strictfp
    public static double atan2(double y, double x); native strictfp
    public static double ceil(double a);          native strictfp
}

```



## StrictMath

public static double <b>cos</b> (double a);	native strictfp
public static double <b>exp</b> (double a);	native strictfp
public static double <b>floor</b> (double a);	native strictfp
public static double <b>IEEERemainder</b> (double f1, double f2);	native strictfp
public static double <b>log</b> (double a);	native strictfp
public static int <b>max</b> (int a, int b);	strictfp
public static long <b>max</b> (long a, long b);	strictfp
public static float <b>max</b> (float a, float b);	strictfp
public static double <b>max</b> (double a, double b);	strictfp
public static int <b>min</b> (int a, int b);	strictfp
public static long <b>min</b> (long a, long b);	strictfp
public static float <b>min</b> (float a, float b);	strictfp
public static double <b>min</b> (double a, double b);	strictfp
public static double <b>pow</b> (double a, double b);	native strictfp
public static double <b>random</b> ();	strictfp
public static double <b>rint</b> (double a);	native strictfp
public static int <b>round</b> (float a);	strictfp
public static long <b>round</b> (double a);	strictfp
public static double <b>sin</b> (double a);	native strictfp
public static double <b>sqrt</b> (double a);	native strictfp
public static double <b>tan</b> (double a);	native strictfp
public static double <b>toDegrees</b> (double angdeg);	strictfp
public static double <b>toRadians</b> (double angdeg);	strictfp

}

## String

Java 1.0

java.lang

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; typically, this method of creation is 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 a `char` array or subarray. All three variants perform an identical function. You can also use the `String()` constructor to create a `String` object from an array or subarray 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.)

`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 only a selected region of the string into an existing array. Use `getBytes()` 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. In 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. There is a two-argument version of `startsWith()` that allows you to specify a position within this string where the prefix comparison will 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 the string. The five-argument version of this method allows you to perform this comparison ignoring the cases of the characters being compared. The final string comparison method is `matches()`, which, as described later, 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 than 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 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.

`substring()` returns a string that consists of the characters from (and including) the specified start position to (but not including) the specified end position. There is also a one-argument version that returns all characters from (and including) the specified start position to the end of the string. In Java 1.4, the `String` class implements the `CharSequence` interface and defines the `subSequence()` methods, 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. More general methods, `replaceAll()` and `replaceFirst()`, use regular expression pattern matching; they are described later. `toUpperCase()` and `toLowerCase()` return a new string in which all characters have been converted to upper- or lowercase. 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.

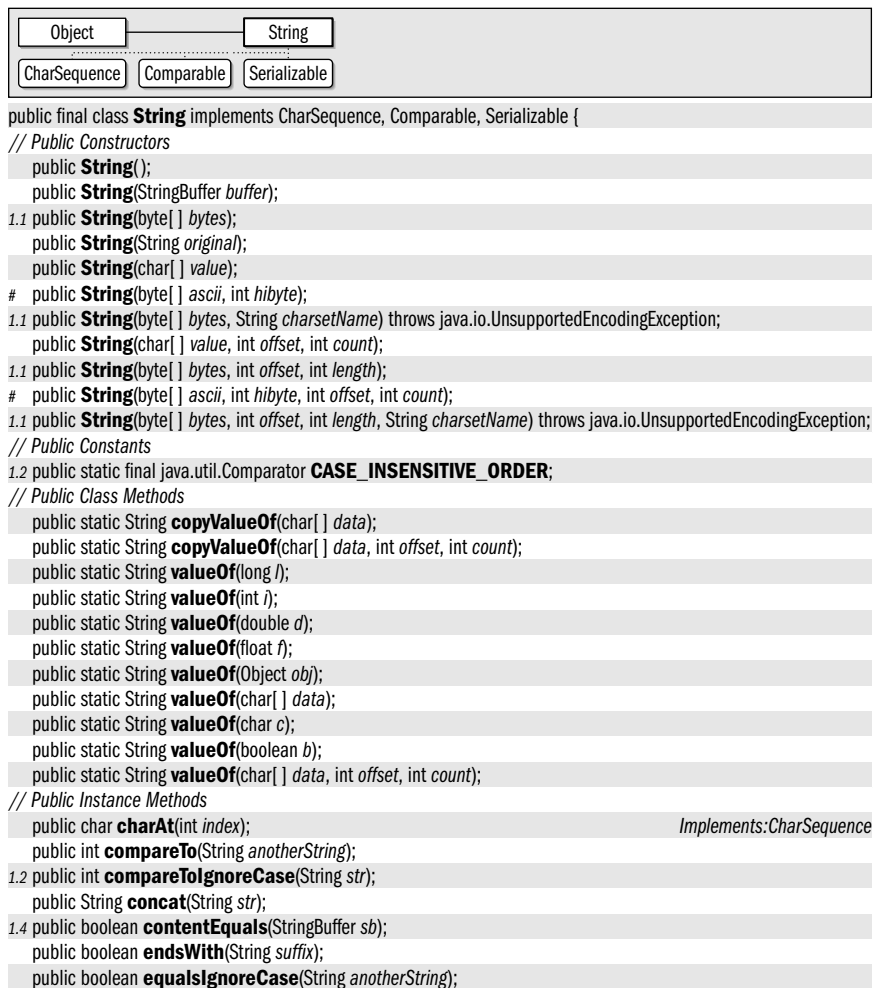
Note that `String` objects are immutable; there is no `setCharAt()` method to change the contents. The methods that return a `String` do not modify the string on which they are invoked but instead return a new `String` object that holds a modified copy of the text of the original. Use a `StringBuffer` 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 1.4 introduces 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. And 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.



## String

Many programs use strings as often as they use Java primitive values. Because the `String` type is an object rather than a primitive value, however, you cannot generally use the `==` operator to compare two strings for equality. Even though strings are immutable, you must instead 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 ensures 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 `!=`.





```

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(String str);
    public int indexOf(int ch);
    public int indexOf(String str, int fromIndex);
    public int indexOf(int ch, 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);
    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);
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);
    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); Implements:CharSequence
    public String toString();
    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
1.2 public int compareTo(Object o);
// 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 1.0

java.lang

serializable

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



## StringBuffer

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.

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 `setLength()` to set the length of the buffer by 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 will need 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, and character arrays, 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`.

Call `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`. In Java 1.4, `StringBuffer` implements `CharSequence`, and therefore defines a `subSequence()` method that is like `substring()` but returns its value as a `CharSequence`. Also new in Java 1.4 is the addition of `indexOf()` and `lastIndexOf()` methods, which 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 same-named methods of `String` after which these methods are modeled.

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.

String concatenation in Java is performed with the `+` operator and implemented using the `append()` method of a `StringBuffer`. 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.

Object	StringBuffer
CharSequence	Serializable

```
public final class StringBuffer implements CharSequence, Serializable {  
    // Public Constructors  
    public StringBuffer();  
    public StringBuffer(int length);  
    public StringBuffer(String str);  
    // Public Instance Methods  
    public StringBuffer append(Object obj); synchronized  
    public StringBuffer append(boolean b);  
    public StringBuffer append(char c); synchronized  
    public StringBuffer append(char[] str); synchronized  
    1.4 public StringBuffer append(StringBuffer sb); synchronized  
    public StringBuffer append(String str); synchronized
```



## StringIndexOutOfBoundsException

```
public StringBuffer append(float f);
public StringBuffer append(long l);
public StringBuffer append(int i);
public StringBuffer append(double d);
public StringBuffer append(char[] str, int offset, int len); synchronized
public int capacity(); 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 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
public StringBuffer insert(int offset, boolean b);
public StringBuffer insert(int offset, char[] str); synchronized
public StringBuffer insert(int offset, Object obj); synchronized
public StringBuffer insert(int offset, String str); synchronized
public StringBuffer insert(int offset, char c); synchronized
public StringBuffer insert(int offset, float f);
public StringBuffer insert(int offset, double d);
public StringBuffer insert(int offset, int i);
public StringBuffer insert(int offset, long l);
1.2 public StringBuffer insert(int index, char[] str, int offset, int len); synchronized
1.4 public int lastIndexOf(String str); synchronized
1.4 public int lastIndexOf(String str, int fromIndex); synchronized
public int length(); Implements:CharSequence synchronized
1.2 public StringBuffer replace(int start, int end, String str); synchronized
public StringBuffer reverse(); 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
public String toString(); Implements:CharSequence
// Methods Implementing CharSequence
public char charAt(int index); synchronized
public int length(); synchronized
1.4 public CharSequence subSequence(int start, int end);
public String toString();
}
```

*Passed To:* Too many methods to list.

*Returned By:* Too many methods to list.

## StringIndexOutOfBoundsException

Java 1.0

java.lang

serializable unchecked

This exception signals that the index used to access a character of a String or StringBuffer is less than zero or is too large.



```
public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException {
// Public Constructors
public StringIndexOutOfBoundsException();
```



## *StringIndexOutOfBoundsException*

```
public StringIndexOutOfBoundsException(int index);  
public StringIndexOutOfBoundsException(String s);  
}
```

### **System**

**Java 1.0**

#### **java.lang**

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. 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.

<i>Property name</i>	<i>Description</i>
<code>java.home</code>	The directory Java is installed in
<code>java.class.path</code>	Where classes are loaded from
<code>java.specification.version</code>	Version of the Java API specification (Java 1.2)
<code>java.specification.vendor</code>	Vendor of the Java API specification (Java 1.2)
<code>java.specification.name</code>	Name of the Java API specification (Java 1.2)
<code>java.version</code>	Version of the Java API implementation
<code>java.vendor</code>	Vendor of this Java API implementation
<code>java.vendor.url</code>	URL of the vendor of this Java API implementation
<code>java.vm.specification.version</code>	Version of the Java VM specification (Java 1.2)
<code>java.vm.specification.vendor</code>	Vendor of the Java VM specification (Java 1.2)
<code>java.vm.specification.name</code>	Name of the Java VM specification (Java 1.2)
<code>java.vm.version</code>	Version of the Java VM implementation (Java 1.2)
<code>java.vm.vendor</code>	Vendor of the Java VM implementation (Java 1.2)
<code>java.vm.name</code>	Name of the Java VM implementation (Java 1.2)
<code>java.class.version</code>	Version of the Java class file format
<code>os.name</code>	Name of the host operating system
<code>os.arch</code>	Host operating system architecture
<code>os.version</code>	Version of the host operating system
<code>file.separator</code>	Platform directory separator character
<code>path.separator</code>	Platform path separator character
<code>line.separator</code>	Platform line separator character(s)
<code>user.name</code>	Current user's account name
<code>user.home</code>	Home directory of current user
<code>user.dir</code>	The current working directory



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. `gc()` requests that the garbage collector perform a thorough garbage-collection pass, and `runFinalization()` requests that the garbage collector finalize all objects that are ready for finalization. Applications do not typically need to call these garbage-collection methods, but they can be useful when benchmarking code with `currentTimeMillis()`. `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. `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
    public static long currentTimeMillis();                                                         native
    public static void exit(int status);
    public static void gc();
    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
    public static void load(String filename);
    public static void loadLibrary(String libname);
    1.2 public static String mapLibraryName(String libname);                                         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
    # public static String getenv(String name);
    1.1# public static void runFinalizersOnExit(boolean value);
}
```

## Thread

Java 1.0

java.lang

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



## Thread

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-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 non-daemon 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 reduce more deeply before running out of stack space. And 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.

Object	Thread	Runnable
--------	--------	----------

```
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
    public static final int MIN_PRIORITY;                              =1
    public static final int NORM_PRIORITY;                            =5
// Public Class Methods
    public static int activeCount();
    public static Thread currentThread();                             native
    public static void dumpStack();
    public static int enumerate(Thread[ ] tarray);
1.4 public static boolean holdsLock(Object obj);                       native
    public static boolean interrupted();
    public static void sleep(long millis) throws InterruptedException;   native
    public static void sleep(long millis, int nanos) throws InterruptedException;
    public static void yield();                                         native
// Property Accessor Methods (by property name)
    public final boolean isAlive();                                     native default:false
1.2 public ClassLoader getContextClassLoader();
1.2 public void setContextClassLoader(ClassLoader cl);
    public final boolean isDaemon();                                   default:false
    public final void setDaemon(boolean on);
    public boolean isInterrupted();                                   default:false
    public final String getName();                                     default: [quot ] Thread-1 [quot ]
    public final void setName(String name);
    public final int getPriority();                                    default:5
    public final void setPriority(int newPriority);
    public final ThreadGroup getThreadGroup();
// Public Instance Methods
    public final void checkAccess();
    public void destroy();
    public void interrupt();
    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
    public void start();                                               native synchronized
// Methods Implementing Runnable
    public void run();
// Public Methods Overriding Object
    public String toString();
// Deprecated Public Methods
#    public int countStackFrames();                                   native
#    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.enumerate(), ThreadGroup.enumerate(), uncaughtException()

**Returned By:** Thread.currentThread(), javax.swing.text.AbstractDocument.getCurrentWriter()



**ThreadDeath**

Java 1.0

java.lang

serializable error

This error signals that a thread should terminate. It 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.



```

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

```

**ThreadGroup**

Java 1.0

java.lang

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`.

```

public class ThreadGroup {
    // Public Constructors
    public ThreadGroup(String name);
    public ThreadGroup(ThreadGroup parent, String name);
    // Property Accessor Methods (by property name)
    public final boolean isDaemon();
    public final void setDaemon(boolean daemon);
    1.1 public boolean isDestroyed(); synchronized
    public final int getMaxPriority();
    public final void setMaxPriority(int pri);
    public final String getName();
    public final ThreadGroup getParent();
    // Public Instance Methods
    public int activeCount();
    public int activeGroupCount();
    public final void checkAccess();
    public final void destroy();
    public int enumerate(ThreadGroup[] list);
    public int enumerate(Thread[] list);
}

```



```

    public int enumerate(Thread[] list, boolean recurse);
    public int enumerate(ThreadGroup[] list, boolean recurse);
1.2 public final void interrupt();
    public void list();
    public final boolean parentOf(ThreadGroup g);
    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()`, `ThreadGroup.enumerate()`, `parentOf()`, `ThreadGroup()`

**Returned By:** `SecurityManager.getThreadGroup()`, `Thread.getThreadGroup()`, `ThreadGroup.getParent()`

## ThreadLocal

Java 1.2

java.lang

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.

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 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 {
// Public Constructors
    public ThreadLocal();
// Public Instance Methods
    public Object get();
    public void set(Object value);
}

```



## ThreadLocal

```
// Protected Instance Methods
protected Object initialValue(); constant
}
```

*Subclasses:* InheritableThreadLocal

## Throwable

Java 1.0

java.lang

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 implementation 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. In Java 1.4 and later, all **Throwable** objects may have a “cause” object that specifies the **Throwable** that caused them. If there is a cause, pass it to the `Throwable()` constructor or to the `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 was 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.

Object	Throwable	Serializable
--------	-----------	--------------

```
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);
// Property Accessor Methods (by property name)
    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 void setStackTrace(StackTraceElement[] stackTrace);
// Public Instance Methods
    public Throwable fillInStackTrace(); native synchronized
    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);
// Public Methods Overriding Object
    public String toString();
}
```

*Subclasses:* Error, Exception



## UnsupportedClassVersionError

**Passed To:** Too many methods to list.

**Returned By:** Too many methods to list.

**Thrown By:** java.awt.Cursor.finalize(), java.awt.Font.finalize(), java.awt.Frame.finalize(), java.awt.Window.finalize(), Object.finalize(), java.lang.reflect.InvocationHandler.invoke(), javax.imageio.spi.ServiceRegistry.finalize(), javax.imageio.stream.ImageInputStreamImpl.finalize(), javax.swing.text.AbstractDocument.AbstractElement.finalize()

**Type Of:** java.rmi.RemoteException.detail, java.rmi.activation.ActivationException.detail, javax.naming.NamingException.rootException, org.omg.CORBA.portable.UnknownException.originalEx

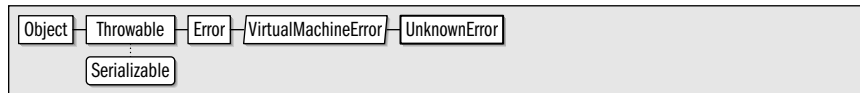
### UnknownError

Java 1.0

java.lang

serializable error

This error signals that an unknown error has occurred at the level of the Java VM.



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

### UnsatisfiedLinkError

Java 1.0

java.lang

serializable error

This error signals that Java cannot satisfy all the links in a class that it has loaded.



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

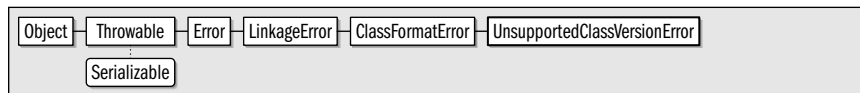
### UnsupportedClassVersionError

Java 1.2

java.lang

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 VM attempts to read a class file with a version number it does not support.



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



## UnsupportedOperationException

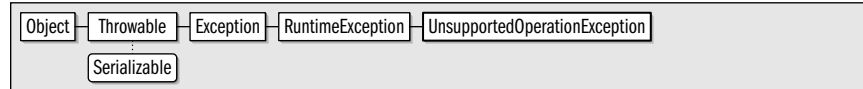
### UnsupportedOperationException

Java 1.2

java.lang

serializable unchecked

This exception 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.



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

**Subclasses:** `java.awt.HeadlessException`, `java.nio.ReadOnlyBufferException`

**Thrown By:** `java.awt.Toolkit.{getLockingKeyState(), setLockingKeyState()}`,  
`javax.imageio.ImageReadParam.setSourceRenderSize()`

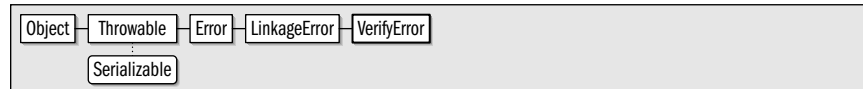
### VerifyError

Java 1.0

java.lang

serializable error

This error signals that a class has not passed the byte-code verification procedures.



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

### VirtualMachineError

Java 1.0

java.lang

serializable error

This is 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`.



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

**Subclasses:** `InternalError`, `OutOfMemoryError`, `StackOverflowError`, `UnknownError`



**Void**

Java 1.1

**java.lang**

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 TYPE;
}
```

**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.

*Classes:*

```
public abstract class Reference;
    public class PhantomReference extends Reference;
    public class SoftReference extends Reference;
    public class WeakReference extends Reference;
public class ReferenceQueue;
```

**PhantomReference**

Java 1.2

**java.lang.ref**

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.



## PhantomReference

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.

Object	Reference	PhantomReference
--------	-----------	------------------

```
public class PhantomReference extends java.lang.ref.Reference {  
    // Public Constructors  
    public PhantomReference(Object referent, ReferenceQueue q);  
    // Public Methods Overriding Reference  
    public Object get();  
}  
constant
```

## Reference

Java 1.2

java.lang.ref

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 {  
    // No Constructor  
    // Public Instance Methods  
    public void clear();  
    public boolean enqueue();  
    public Object get();  
    public boolean isEnqueued();  
}
```

*Subclasses:* `PhantomReference`, `SoftReference`, `WeakReference`

*Returned By:* `ReferenceQueue.poll()`, `remove()`

## ReferenceQueue

Java 1.2

java.lang.ref

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.

```
public class ReferenceQueue {
    // Public Constructors
    public ReferenceQueue();
    // Public Instance Methods
    public java.lang.ref.Reference poll();
    public java.lang.ref.Reference remove() throws InterruptedException;
    public java.lang.ref.Reference remove(long timeout) throws IllegalArgumentException, InterruptedException;
}
```

*Passed To:* `PhantomReference.PantomReference()`, `SoftReference.SoftReference()`, `WeakReference.WeakReference()`

## SoftReference

Java 1.2

`java.lang.ref`

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.



```
public class SoftReference extends java.lang.ref.Reference {
    // Public Constructors
    public SoftReference(Object referent);
    public SoftReference(Object referent, ReferenceQueue q);
    // Public Methods Overriding Reference
    public Object get();
}
```

## WeakReference

Java 1.2

`java.lang.ref`

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

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.

```
Object — Reference — WeakReference
```

```
public class WeakReference extends java.lang.ref.Reference {  
    // Public Constructors  
    public WeakReference(Object referent);  
    public WeakReference(Object referent, ReferenceQueue 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.

In 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.

### Interfaces:

```
public interface InvocationHandler;  
public interface Member;
```

### Classes:

```
public class AccessibleObject;  
    public final class Constructor extends AccessibleObject implements Member;  
    public final class Field extends AccessibleObject implements Member;  
    public final class Method extends AccessibleObject implements 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 UndeclaredThrowableException extends RuntimeException;
```



**AccessibleObject****Java 1.2****java.lang.reflect**

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** ("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.

```
public class AccessibleObject {
    // 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;
}
```

**Subclasses:** **Constructor**, **Field**, **Method**

**Passed To:** **AccessibleObject.setAccessible()**

**Array****Java 1.1****java.lang.reflect**

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 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
```



## Array

```
// Public Class Methods
public static Object get(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static boolean getBoolean(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static byte getByte(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static char getChar(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static double getDouble(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static float getFloat(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static int getInt(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static int getLength(Object array) throws IllegalArgumentException; native
public static long getLong(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static short getShort(Object array, int index) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
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, native
    ArrayIndexOutOfBoundsException;
public static void setBoolean(Object array, int index, boolean z) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static void setByte(Object array, int index, byte b) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static void setChar(Object array, int index, char c) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static void setDouble(Object array, int index, double d) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static void setFloat(Object array, int index, float f) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static void setInt(Object array, int index, int i) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static void setLong(Object array, int index, long l) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
public static void setShort(Object array, int index, short s) throws IllegalArgumentException, native
    ArrayIndexOutOfBoundsException;
}
```

## Constructor

Java 1.1

### java.lang.reflect

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.



```

public final class Constructor extends AccessibleObject implements Member {
    // No Constructor
    // Public Instance Methods
    public Class[] getExceptionTypes();
    public Class[] getParameterTypes();
    public Object newInstance(Object[] initargs) throws InstantiationException,
        IllegalAccessException,
        IllegalArgumentException, InvocationTargetException;
    // Methods Implementing Member
    public Class getDeclaringClass();
    public int getModifiers();
    public String getName();
    // Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}

```

**Returned By:** `Class.{getConstructor(), getConstructors(), getDeclaredConstructor(), getDeclaredConstructors()}`

## Field

Java 1.1

### java.lang.reflect

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.



```

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;

```



## Field

```
public double getDouble(Object obj) throws IllegalArgumentException, IllegalAccessException;
public float getFloat(Object obj) throws IllegalArgumentException, IllegalAccessException;
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();
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;
// Methods Implementing Member
public Class getDeclaringClass();
public int getModifiers();
public String getName();
// Public Methods Overriding Object
public boolean equals(Object obj);
public int hashCode();
public String toString();
}
```

*Returned By:* Class.{getDeclaredField(), getDeclaredFields(), getField(), getFields()}

## InvocationHandler

Java 1.3

java.lang.reflect

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 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
    public abstract Object invoke(Object proxy, Method method, Object[ ] args) throws Throwable;
}
```

*Implementations:* java.beans.EventHandler

*Passed To:* Proxy.{newProxyInstance(), Proxy()}

*Returned By:* Proxy.getInvocationHandler()

*Type Of:* Proxy.h

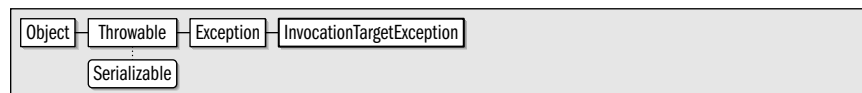
## InvocationTargetException

Java 1.1

java.lang.reflect

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.



```
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();
}
```

*Thrown By:* java.awt.EventQueue.invokeAndWait(), Constructor.newInstance(), Method.invoke(), javax.swing.SwingUtilities.invokeAndWait()

## Member

Java 1.1

java.lang.reflect

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.

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



## Member

```
    public abstract String getName();  
}
```

*Implementations:* Constructor, Field, Method

## Method

Java 1.1

### java.lang.reflect

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()**.



```
public final class Method extends AccessibleObject implements Member {  
    // No Constructor  
    // Public Instance Methods  
    public Class[] getExceptionTypes();  
    public Class[] getParameterTypes();  
    public Class getReturnType();  
    public Object invoke(Object obj, Object[] args) throws IllegalAccessException, IllegalArgumentException,  
        InvocationTargetException;  
    // Methods Implementing Member  
    public Class getDeclaringClass();  
    public int getModifiers();  
    public String getName();  
    // Public Methods Overriding Object  
    public boolean equals(Object obj);  
    public int hashCode();  
    public String toString();  
}
```

*Passed To:* Too many methods to list.

*Returned By:* java.beans.EventSetDescriptor.{getAddListenerMethod(), getGetListenerMethod(),  
getListenerMethods(), getRemoveListenerMethod()},  
java.beans.IndexedPropertyDescriptor.{getIndexedReadMethod(), getIndexedWriteMethod()},  
java.beans.MethodDescriptor.getMethod(), java.beans.PropertyDescriptor.{getReadMethod(),  
getWriteMethod()}, Class.{getDeclaredMethod(), getDeclaredMethods(), getMethod(), getMethods()})



**Modifier****Java 1.1****java.lang.reflect**

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 `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);
}
```

**Proxy****Java 1.3****java.lang.reflect****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()`



## Proxy

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.



```
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 c);  
    public static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)  
        throws IllegalArgumentException;  
    // Protected Instance Fields  
    protected InvocationHandler h;  
}
```

## ReflectPermission

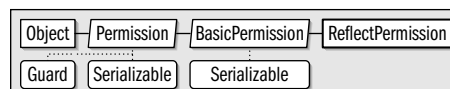
Java 1.2

`java.lang.reflect`

*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.



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

## UndeclaredThrowableException

Java 1.3

`java.lang.reflect`

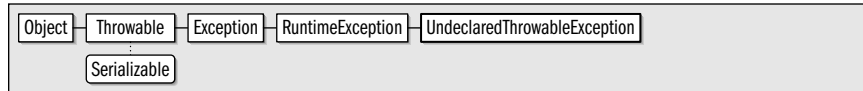
*serializable unchecked*

This exception is 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



## *UndeclaredThrowableException*

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.



```
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();  
}
```