Overview Package Class Use Tree Deprecated Index Help

Java™ Platform
Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class Double

java.lang.Object
    java.lang.Number
        java.lang.Double

All Implemented Interfaces:

Serializable, Comparable<Double>

```
public final class Double
extends Number
implements Comparable<Double>
```

The `Double` class wraps a value of the primitive type `double` in an object. An object of type `Double` contains a single field whose type is `double`.

In addition, this class provides several methods for converting a `double` to a `String` and a `String` to a `double`, as well as other constants and methods useful when dealing with a `double`.

Since:

JDK1.0

See Also:

Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	**MAX_EXPONENT** Maximum exponent a finite `double` variable may have.
`static double`	**MAX_VALUE** A constant holding the largest positive finite value of type `double`, $(2-2^{-52})\cdot2^{1023}$.
`static int`	**MIN_EXPONENT** Minimum exponent a normalized `double` variable may have.
`static double`	**MIN_NORMAL** A constant holding the smallest positive normal value of type `double`, 2^{-1022}.
`static double`	**MIN_VALUE** A constant holding the smallest positive nonzero value of type `double`, 2^{-1074}.
`static double`	**NaN** A constant holding a Not-a-Number (NaN) value of type `double`.
`static double`	**NEGATIVE_INFINITY** A constant holding the negative infinity of type `double`.
`static double`	**POSITIVE_INFINITY** A constant holding the positive infinity of type `double`.
`static int`	**SIZE** The number of bits used to represent a `double` value.
`static Class<Double>`	**TYPE** The `Class` instance representing the primitive type `double`.

Constructor Summary

Constructors

Constructor and Description
`Double(double value)` Constructs a newly allocated `Double` object that represents the primitive `double` argument.
`Double(String s)` Constructs a newly allocated `Double` object that represents the floating-point value of type `double` represented by the string.

Method Summary

Methods

Modifier and Type	Method and Description
`byte`	`byteValue()` Returns the value of this `Double` as a `byte` (by casting to a `byte`).
`static int`	`compare(double d1, double d2)` Compares the two specified `double` values.
`int`	`compareTo(Double anotherDouble)` Compares two `Double` objects numerically.
`static long`	`doubleToLongBits(double value)` Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout.
`static long`	`doubleToRawLongBits(double value)` Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout, preserving Not-a-Number (NaN) values.
`double`	`doubleValue()` Returns the `double` value of this `Double` object.
`boolean`	`equals(Object obj)` Compares this object against the specified object.
`float`	`floatValue()` Returns the `float` value of this `Double` object.
`int`	`hashCode()` Returns a hash code for this `Double` object.
`int`	`intValue()` Returns the value of this `Double` as an `int` (by casting to type `int`).
`boolean`	`isInfinite()` Returns `true` if this `Double` value is infinitely large in magnitude, `false` otherwise.
`static boolean`	`isInfinite(double v)` Returns `true` if the specified number is infinitely large in magnitude, `false` otherwise.
`boolean`	`isNaN()` Returns `true` if this `Double` value is a Not-a-Number (NaN), `false` otherwise.
`static boolean`	`isNaN(double v)` Returns `true` if the specified number is a Not-a-Number (NaN) value, `false` otherwise.
`static double`	`longBitsToDouble(long bits)` Returns the `double` value corresponding to a given bit representation.
`long`	`longValue()` Returns the value of this `Double` as a `long` (by casting to type `long`).
`static double`	`parseDouble(String s)` Returns a new `double` initialized to the value represented by the specified `String`, as performed by the `valueOf` method of class `Double`.
`short`	`shortValue()` Returns the value of this `Double` as a `short` (by casting to a `short`).
`static String`	`toHexString(double d)` Returns a hexadecimal string representation of the `double` argument.
`String`	`toString()` Returns a string representation of this `Double` object.
`static String`	`toString(double d)` Returns a string representation of the `double` argument.

static **Double**	**valueOf**(double d) Returns a `Double` instance representing the specified `double` value.
static **Double**	**valueOf**(**String** s) Returns a `Double` object holding the `double` value represented by the argument string `s`.

Methods inherited from class java.lang.Object

`clone, finalize, getClass, notify, notifyAll, wait, wait, wait`

Field Detail

POSITIVE_INFINITY

`public static final double POSITIVE_INFINITY`

A constant holding the positive infinity of type `double`. It is equal to the value returned by `Double.longBitsToDouble(0x7ff0000000000000L)`.

See Also:

Constant Field Values

NEGATIVE_INFINITY

`public static final double NEGATIVE_INFINITY`

A constant holding the negative infinity of type `double`. It is equal to the value returned by `Double.longBitsToDouble(0xfff0000000000000L)`.

See Also:

Constant Field Values

NaN

`public static final double NaN`

A constant holding a Not-a-Number (NaN) value of type `double`. It is equivalent to the value returned by `Double.longBitsToDouble(0x7ff8000000000000L)`.

See Also:

Constant Field Values

MAX_VALUE

`public static final double MAX_VALUE`

A constant holding the largest positive finite value of type `double`, $(2-2^{-52})\cdot 2^{1023}$. It is equal to the hexadecimal floating-point literal `0x1.fffffffffffffP+1023` and also equal to `Double.longBitsToDouble(0x7fefffffffffffffL)`.

See Also:

Constant Field Values

MIN_NORMAL

`public static final double MIN_NORMAL`

A constant holding the smallest positive normal value of type `double`, 2^{-1022}. It is equal to the hexadecimal floating-point literal `0x1.0p-1022` and also equal to `Double.longBitsToDouble(0x0010000000000000L)`.

Since:

1.6

See Also:

Constant Field Values

MIN_VALUE

```
public static final double MIN_VALUE
```

A constant holding the smallest positive nonzero value of type `double`, 2^{-1074}. It is equal to the hexadecimal floating-point literal `0x0.0000000000001P-1022` and also equal to `Double.longBitsToDouble(0x1L)`.

See Also:

Constant Field Values

MAX_EXPONENT

```
public static final int MAX_EXPONENT
```

Maximum exponent a finite `double` variable may have. It is equal to the value returned by `Math.getExponent(Double.MAX_VALUE)`.

Since:

1.6

See Also:

Constant Field Values

MIN_EXPONENT

```
public static final int MIN_EXPONENT
```

Minimum exponent a normalized `double` variable may have. It is equal to the value returned by `Math.getExponent(Double.MIN_NORMAL)`.

Since:

1.6

See Also:

Constant Field Values

SIZE

```
public static final int SIZE
```

The number of bits used to represent a `double` value.

Since:

1.5

See Also:

Constant Field Values

TYPE

```
public static final Class<Double> TYPE
```

The `Class` instance representing the primitive type `double`.

Since:

JDK1.1

Constructor Detail

Double

```
public Double(double value)
```

Constructs a newly allocated `Double` object that represents the primitive `double` argument.

Parameters:

`value` - the value to be represented by the `Double`.

Double

```
public Double(String s)
       throws NumberFormatException
```

Constructs a newly allocated `Double` object that represents the floating-point value of type `double` represented by the string. The string is converted to a `double` value as if by the `valueOf` method.

Parameters:

`s` - a string to be converted to a `Double`.

Throws:

`NumberFormatException` - if the string does not contain a parsable number.

See Also:

`valueOf(java.lang.String)`

Method Detail

toString

```
public static String toString(double d)
```

Returns a string representation of the `double` argument. All characters mentioned below are ASCII characters.

- If the argument is NaN, the result is the string "`NaN`".
- Otherwise, the result is a string that represents the sign and magnitude (absolute value) of the argument. If the sign is negative, the first character of the result is '`-`' (`'\u002D'`); if the sign is positive, no sign character appears in the result. As for the magnitude *m*:
  - If *m* is infinity, it is represented by the characters `"Infinity"`; thus, positive infinity produces the result `"Infinity"` and negative infinity produces the result `"-Infinity"`.
  - If *m* is zero, it is represented by the characters `"0.0"`; thus, negative zero produces the result `"-0.0"` and positive zero produces the result `"0.0"`.
  - If *m* is greater than or equal to 10^{-3} but less than 10^{7}, then it is represented as the integer part of *m*, in decimal form with no leading zeroes, followed by `'.'` (`'\u002E'`), followed by one or more decimal digits representing the fractional part of *m*.
  - If *m* is less than 10^{-3} or greater than or equal to 10^{7}, then it is represented in so-called "computerized scientific notation." Let *n* be the unique integer such that $10^{n} \leq m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^{n} so that $1 \leq a < 10$. The magnitude is then represented as the integer part of *a*, as a single decimal digit, followed by `'.'` (`'\u002E'`), followed by decimal digits representing the fractional part of *a*, followed by the letter '`E`' (`'\u0045'`), followed by a representation of *n* as a decimal integer, as produced by the method `Integer.toString(int)`.

How many digits must be printed for the fractional part of *m* or *a*? There must be at least one digit to represent the fractional part, and beyond that as many, but only as many, more digits as are needed to uniquely distinguish the argument value from adjacent values of type `double`. That is, suppose that *x* is the exact mathematical value represented by the decimal representation produced by this method for a finite nonzero argument *d*. Then *d* must be the `double` value nearest to *x*; or if two `double` values are equally close to *x*, then *d* must be one of them and the least significant bit of the significand of *d* must be `0`.

To create localized string representations of a floating-point value, use subclasses of `NumberFormat`.

Parameters:

`d` - the `double` to be converted.

Returns:

a string representation of the argument.

toHexString

```
public static String toHexString(double d)
```

Returns a hexadecimal string representation of the `double` argument. All characters mentioned below are ASCII characters.

- If the argument is NaN, the result is the string "`NaN`".
- Otherwise, the result is a string that represents the sign and magnitude of the argument. If the sign is negative, the first character of the result is '`-`' (`'\u002D'`); if the sign is positive, no sign character appears in the result. As for the magnitude *m*:
  - If *m* is infinity, it is represented by the string `"Infinity"`; thus, positive infinity produces the result `"Infinity"` and negative infinity produces the result `"-Infinity"`.
  - If *m* is zero, it is represented by the string `"0x0.0p0"`; thus, negative zero produces the result `"-0x0.0p0"` and positive zero produces the result `"0x0.0p0"`.
  - If *m* is a `double` value with a normalized representation, substrings are used to represent the significand and exponent fields. The significand is represented by the characters `"0x1."` followed by a lowercase hexadecimal representation of the rest of the significand as a fraction. Trailing zeros in the hexadecimal representation are removed unless all the digits are zero, in which case a single zero is used. Next, the exponent is represented by `"p"` followed by a decimal string of the unbiased exponent as if produced by a call to `Integer.toString` on the exponent value.
  - If *m* is a `double` value with a subnormal representation, the significand is represented by the characters `"0x0."` followed by a hexadecimal representation of the rest of the significand as a fraction. Trailing zeros in the hexadecimal representation are removed. Next, the exponent is represented by `"p-1022"`. Note that there must be at least one nonzero digit in a subnormal significand.

Examples

Floating-point Value	Hexadecimal String
`1.0`	`0x1.0p0`
`-1.0`	`-0x1.0p0`
`2.0`	`0x1.0p1`
`3.0`	`0x1.8p1`
`0.5`	`0x1.0p-1`
`0.25`	`0x1.0p-2`
`Double.MAX_VALUE`	`0x1.fffffffffffffp1023`
`Minimum Normal Value`	`0x1.0p-1022`
`Maximum Subnormal Value`	`0x0.fffffffffffffp-1022`
`Double.MIN_VALUE`	`0x0.0000000000001p-1022`

Parameters:

`d` - the `double` to be converted.

Returns:

a hex string representation of the argument.

Since:

1.5

valueOf

```
public static Double valueOf(String s)
                      throws NumberFormatException
```

Returns a `Double` object holding the `double` value represented by the argument string `s`.

If `s` is `null`, then a `NullPointerException` is thrown.

Leading and trailing whitespace characters in `s` are ignored. Whitespace is removed as if by the `String.trim()` method; that is, both ASCII space and control characters are removed. The rest of `s` should constitute a *FloatValue* as described by the lexical syntax rules:

> ***FloatValue:***
>
> *Sign*$_{opt}$ `NaN`
>
> *Sign*$_{opt}$ `Infinity`
>
> *Sign*$_{opt}$ *FloatingPointLiteral*
>
> *Sign*$_{opt}$ *HexFloatingPointLiteral*
>
> *SignedInteger*
>
> ***HexFloatingPointLiteral*:**
>
> *HexSignificand BinaryExponent FloatTypeSuffix*$_{opt}$
>
> ***HexSignificand:***
>
> *HexNumeral*
>
> *HexNumeral* `.`
>
> `0x` *HexDigits*$_{opt}$ `.` *HexDigits*
>
> `0X` *HexDigits*$_{opt}$ `.` *HexDigits*
>
> ***BinaryExponent:***
>
> *BinaryExponentIndicator SignedInteger*
>
> ***BinaryExponentIndicator:***
>
> `p`
>
> `P`

where *Sign*, *FloatingPointLiteral*, *HexNumeral*, *HexDigits*, *SignedInteger* and *FloatTypeSuffix* are as defined in the lexical structure sections of *The Java™ Language Specification*, except that underscores are not accepted between digits. If `s` does not have the form of a *FloatValue*, then a `NumberFormatException` is thrown. Otherwise, `s` is regarded as representing an exact decimal value in the usual "computerized scientific notation" or as an exact hexadecimal value; this exact numerical value is then conceptually converted to an "infinitely precise" binary value that is then rounded to type `double` by the usual round-to-nearest rule of IEEE 754 floating-point arithmetic, which includes preserving the sign of a zero value. Note that the round-to-nearest rule also implies overflow and underflow behaviour; if the exact value of `s` is large enough in magnitude (greater than or equal to (`MAX_VALUE` + `ulp(MAX_VALUE)`/2), rounding to `double` will result in an infinity and if the exact value of `s` is small enough in magnitude (less than or equal to `MIN_VALUE`/2), rounding to float will result in a zero. Finally, after rounding a `Double` object representing this `double` value is returned.

To interpret localized string representations of a floating-point value, use subclasses of `NumberFormat`.

Note that trailing format specifiers, specifiers that determine the type of a floating-point literal (`1.0f` is a `float` value; `1.0d` is a `double` value), do *not* influence the results of this method. In other words, the numerical value of the input string is converted directly to the target floating-point type. The two-step sequence of conversions, string to `float` followed by `float` to `double`, is *not* equivalent to converting a string directly to `double`. For example, the `float` literal `0.1f` is equal to the `double` value `0.10000000149011612`; the `float` literal `0.1f` represents a different numerical value than the `double` literal `0.1`. (The numerical value 0.1 cannot be exactly represented in a binary floating-point number.)

To avoid calling this method on an invalid string and having a `NumberFormatException` be thrown, the regular expression below can be used to screen the input string:

```
final String Digits     = "(\\p{Digit}+)";
final String HexDigits  = "(\\p{XDigit}+)";
// an exponent is 'e' or 'E' followed by an optionally
// signed decimal integer.
final String Exp        = "[eE][+-]?"+Digits;
final String fpRegex    =
    ("[\\x00-\\x20]*"+   // Optional leading "whitespace"
     "[+-]?(" + // Optional sign character
     "NaN|" +           // "NaN" string
     "Infinity|" +      // "Infinity" string

     // A decimal floating-point string representing a finite positive
     // number without a leading sign has at most five basic pieces:
     // Digits . Digits ExponentPart FloatTypeSuffix
     //
     // Since this method allows integer-only strings as input
     // in addition to strings of floating-point literals, the
     // two sub-patterns below are simplifications of the grammar
     // productions from section 3.10.2 of
     // The Java™ Language Specification.

     // Digits ._opt Digits_opt ExponentPart_opt FloatTypeSuffix_opt
     "((("+Digits+"(\\.)?("+Digits+"?)("+Exp+")?)|"+

     // . Digits ExponentPart_opt FloatTypeSuffix_opt
     "(\\.("+Digits+")("+Exp+")?)|"+

     // Hexadecimal strings
     "((" +
      // 0[xX] HexDigits ._opt BinaryExponent FloatTypeSuffix_opt
      "(0[xX]" + HexDigits + "(\\.)?)|" +

      // 0[xX] HexDigits_opt . HexDigits BinaryExponent FloatTypeSuffix_opt
      "(0[xX]" + HexDigits + "?(\\.)" + HexDigits + ")" +

      ")[pP][+-]?" + Digits + "))" +
     "[fFdD]?))" +
     "[\\x00-\\x20]*");// Optional trailing "whitespace"

if (Pattern.matches(fpRegex, myString))
    Double.valueOf(myString); // Will not throw NumberFormatException
else {
    // Perform suitable alternative action
}
```

Parameters:

`s` - the string to be parsed.

Returns:

a `Double` object holding the value represented by the `String` argument.

Throws:

`NumberFormatException` - if the string does not contain a parsable number.

valueOf

```
public static Double valueOf(double d)
```

Returns a `Double` instance representing the specified `double` value. If a new `Double` instance is not required, this method should generally be used in preference to the constructor `Double(double)`, as this method is likely to yield significantly better space and time performance by caching frequently requested values.

Parameters:

`d` - a double value.

Returns:

a `Double` instance representing `d`.

Since:

1.5

parseDouble

```
public static double parseDouble(String s)
                          throws NumberFormatException
```

Returns a new `double` initialized to the value represented by the specified `String`, as performed by the `valueOf` method of class `Double`.

Parameters:

`s` - the string to be parsed.

Returns:

the `double` value represented by the string argument.

Throws:

`NullPointerException` - if the string is null

`NumberFormatException` - if the string does not contain a parsable `double`.

Since:

1.2

See Also:

`valueOf(String)`

isNaN

```
public static boolean isNaN(double v)
```

Returns `true` if the specified number is a Not-a-Number (NaN) value, `false` otherwise.

Parameters:

`v` - the value to be tested.

Returns:

`true` if the value of the argument is NaN; `false` otherwise.

isInfinite

```
public static boolean isInfinite(double v)
```

Returns `true` if the specified number is infinitely large in magnitude, `false` otherwise.

Parameters:

`v` - the value to be tested.

Returns:

`true` if the value of the argument is positive infinity or negative infinity; `false` otherwise.

isNaN

```
public boolean isNaN()
```

Returns `true` if this `Double` value is a Not-a-Number (NaN), `false` otherwise.

Returns:

`true` if the value represented by this object is NaN; `false` otherwise.

isInfinite

```
public boolean isInfinite()
```

Returns `true` if this `Double` value is infinitely large in magnitude, `false` otherwise.

Returns:

`true` if the value represented by this object is positive infinity or negative infinity; `false` otherwise.

toString

```
public String toString()
```

Returns a string representation of this `Double` object. The primitive `double` value represented by this object is converted to a string exactly as if by the method `toString` of one argument.

Overrides:

`toString` in class `Object`

Returns:

a `String` representation of this object.

See Also:

`toString(double)`

byteValue

```
public byte byteValue()
```

Returns the value of this `Double` as a `byte` (by casting to a `byte`).

Overrides:

`byteValue` in class `Number`

Returns:

the `double` value represented by this object converted to type `byte`

Since:

JDK1.1

shortValue

```
public short shortValue()
```

Returns the value of this `Double` as a `short` (by casting to a `short`).

Overrides:

`shortValue` in class `Number`

Returns:

the `double` value represented by this object converted to type `short`

Since:

JDK1.1

intValue

```
public int intValue()
```

Returns the value of this `Double` as an `int` (by casting to type `int`).

Specified by:

`intValue` in class `Number`

Returns:

the `double` value represented by this object converted to type `int`

longValue

```
public long longValue()
```

Returns the value of this `Double` as a `long` (by casting to type `long`).

Specified by:

`longValue` in class `Number`

Returns:

the `double` value represented by this object converted to type `long`

floatValue

```
public float floatValue()
```

Returns the `float` value of this `Double` object.

Specified by:

`floatValue` in class `Number`

Returns:

the `double` value represented by this object converted to type `float`

Since:

JDK1.0

doubleValue

```
public double doubleValue()
```

Returns the `double` value of this `Double` object.

Specified by:

`doubleValue` in class `Number`

Returns:

the `double` value represented by this object

hashCode

```
public int hashCode()
```

Returns a hash code for this `Double` object. The result is the exclusive OR of the two halves of the `long` integer bit representation, exactly as produced by the method `doubleToLongBits(double)`, of the primitive `double` value represented by this `Double` object. That is, the hash code is the value of the expression:

```
    (int)(v^(v>>>32))
```

where `v` is defined by:

```
    long v = Double.doubleToLongBits(this.doubleValue());
```

Overrides:

`hashCode` in class `Object`

Returns:

a `hash code` value for this object.

See Also:

`Object.equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

equals

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

Compares this object against the specified object. The result is `true` if and only if the argument is not `null` and is a `Double` object that represents a `double` that has the same value as the `double` represented by this object. For this purpose, two `double` values are considered to be the same if and only if the method `doubleToLongBits(double)` returns the identical `long` value when applied to each.

Note that in most cases, for two instances of class `Double`, `d1` and `d2`, the value of `d1.equals(d2)` is `true` if and only if

```
     d1.doubleValue() == d2.doubleValue()
```

also has the value `true`. However, there are two exceptions:

- If `d1` and `d2` both represent `Double.NaN`, then the `equals` method returns `true`, even though `Double.NaN==Double.NaN` has the value `false`.
- If `d1` represents `+0.0` while `d2` represents `-0.0`, or vice versa, the `equal` test has the value `false`, even though `+0.0==-0.0` has the value `true`.

This definition allows hash tables to operate properly.

Overrides:

`equals` in class `Object`

Parameters:

`obj` - the object to compare with.

Returns:

`true` if the objects are the same; `false` otherwise.

See Also:

`doubleToLongBits(double)`

doubleToLongBits

```
public static long doubleToLongBits(double value)
```

Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout.

Bit 63 (the bit that is selected by the mask `0x8000000000000000L`) represents the sign of the floating-point number. Bits 62-52 (the bits that are selected by the mask `0x7ff0000000000000L`) represent the exponent. Bits 51-0 (the bits that are selected by the mask `0x000fffffffffffffL`) represent the significand (sometimes called the mantissa) of the floating-point number.

If the argument is positive infinity, the result is `0x7ff0000000000000L`.

If the argument is negative infinity, the result is `0xfff0000000000000L`.

If the argument is NaN, the result is `0x7ff8000000000000L`.

In all cases, the result is a `long` integer that, when given to the `longBitsToDouble(long)` method, will produce a floating-point value the same as the argument to `doubleToLongBits` (except all NaN values are collapsed to a single "canonical" NaN value).

Parameters:

`value` - a `double` precision floating-point number.

Returns:

the bits that represent the floating-point number.

doubleToRawLongBits

```
public static long doubleToRawLongBits(double value)
```

Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout, preserving Not-a-Number (NaN) values.

Bit 63 (the bit that is selected by the mask `0x8000000000000000L`) represents the sign of the floating-point number. Bits 62-52 (the bits that are selected by the mask `0x7ff0000000000000L`) represent the exponent. Bits 51-0 (the bits that are selected by the mask `0x000fffffffffffffL`) represent the significand (sometimes called the mantissa) of the floating-point number.

If the argument is positive infinity, the result is `0x7ff0000000000000L`.

If the argument is negative infinity, the result is `0xfff0000000000000L`.

If the argument is NaN, the result is the `long` integer representing the actual NaN value. Unlike the `doubleToLongBits` method, `doubleToRawLongBits` does not collapse all the bit patterns encoding a NaN to a single "canonical" NaN value.

In all cases, the result is a `long` integer that, when given to the `longBitsToDouble(long)` method, will produce a floating-point value the same as the argument to `doubleToRawLongBits`.

Parameters:

`value` - a `double` precision floating-point number.

Returns:

the bits that represent the floating-point number.

Since:

1.3

longBitsToDouble

```
public static double longBitsToDouble(long bits)
```

Returns the `double` value corresponding to a given bit representation. The argument is considered to be a representation of a floating-point value according to the IEEE 754 floating-point "double format" bit layout.

If the argument is `0x7ff0000000000000L`, the result is positive infinity.

If the argument is `0xfff0000000000000L`, the result is negative infinity.

If the argument is any value in the range `0x7ff0000000000001L` through `0x7fffffffffffffffL` or in the range `0xfff0000000000001L` through `0xffffffffffffffffL`, the result is a NaN. No IEEE 754 floating-point operation provided by Java can distinguish between two NaN values of the same type with different bit patterns. Distinct values of NaN are only distinguishable by use of the `Double.doubleToRawLongBits` method.

In all other cases, let *s*, *e*, and *m* be three values that can be computed from the argument:

```
        int s = ((bits >> 63) == 0) ? 1 : -1;
        int e = (int)((bits >> 52) & 0x7ffL);
        long m = (e == 0) ?
                        (bits & 0xfffffffffffffL) << 1 :
                        (bits & 0xfffffffffffffL) | 0x10000000000000L;
```

Then the floating-point result equals the value of the mathematical expression $s \cdot m \cdot 2^{e-1075}$.

Note that this method may not be able to return a `double` NaN with exactly same bit pattern as the `long` argument. IEEE 754 distinguishes between two kinds of NaNs, quiet NaNs and *signaling NaNs*. The differences between the two kinds of NaN are generally not visible in Java. Arithmetic operations on signaling NaNs turn them into quiet NaNs with a different, but often similar, bit pattern. However, on some processors merely copying a signaling NaN also performs that conversion. In particular, copying a signaling NaN to return it to the calling method may perform this conversion. So `longBitsToDouble` may not be able to return a `double` with a signaling NaN bit pattern. Consequently, for some `long` values, `doubleToRawLongBits(longBitsToDouble(start))` may *not* equal `start`. Moreover, which particular bit patterns represent signaling NaNs is platform dependent; although all NaN bit patterns, quiet or signaling, must be in the NaN range identified above.

Parameters:

`bits` - any `long` integer.

Returns:

the `double` floating-point value with the same bit pattern.

compareTo

```
public int compareTo(Double anotherDouble)
```

Compares two `Double` objects numerically. There are two ways in which comparisons performed by this method differ from those performed by the Java language numerical comparison operators (`<, <=, ==, >=, >`) when applied to primitive `double` values:

- `Double.NaN` is considered by this method to be equal to itself and greater than all other `double` values (including

`Double.POSITIVE_INFINITY`).
- `0.0d` is considered by this method to be greater than `-0.0d`.

This ensures that the *natural ordering* of `Double` objects imposed by this method is *consistent with equals*.

Specified by:

`compareTo` in interface `Comparable<Double>`

Parameters:

`anotherDouble` - the `Double` to be compared.

Returns:

the value `0` if `anotherDouble` is numerically equal to this `Double`; a value less than `0` if this `Double` is numerically less than `anotherDouble`; and a value greater than `0` if this `Double` is numerically greater than `anotherDouble`.

Since:

1.2

compare

```
public static int compare(double d1,
          double d2)
```

Compares the two specified `double` values. The sign of the integer value returned is the same as that of the integer that would be returned by the call:

```
    new Double(d1).compareTo(new Double(d2))
```

Parameters:

`d1` - the first `double` to compare

`d2` - the second `double` to compare

Returns:

the value `0` if `d1` is numerically equal to `d2`; a value less than `0` if `d1` is numerically less than `d2`; and a value greater than `0` if `d1` is numerically greater than `d2`.

Since:

1.4

Overview Package Class Use Tree Deprecated Index Help

Java™ Platform
Standard Ed. 7

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.