java.lang

Class Float

java.lang.Object
    java.lang.Number
        java.lang.Float

All Implemented Interfaces:

Serializable, Comparable<Float>

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

The `Float` class wraps a value of primitive type `float` in an object. An object of type `Float` contains a single field whose type is `float`.

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

Since:

JDK1.0

See Also:

Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	**`MAX_EXPONENT`** Maximum exponent a finite `float` variable may have.
`static float`	**`MAX_VALUE`** A constant holding the largest positive finite value of type `float`, $(2-2^{-23})\cdot 2^{127}$.
`static int`	**`MIN_EXPONENT`** Minimum exponent a normalized `float` variable may have.
`static float`	**`MIN_NORMAL`** A constant holding the smallest positive normal value of type `float`, 2^{-126}.
`static float`	**`MIN_VALUE`** A constant holding the smallest positive nonzero value of type `float`, 2^{-149}.
`static float`	**`NaN`** A constant holding a Not-a-Number (NaN) value of type `float`.
`static float`	**`NEGATIVE_INFINITY`** A constant holding the negative infinity of type `float`.
`static float`	**`POSITIVE_INFINITY`** A constant holding the positive infinity of type `float`.
`static int`	**`SIZE`** The number of bits used to represent a `float` value.

static Class<Float>	**TYPE** The Class instance representing the primitive type float.

Constructor Summary

Constructors

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

Method Summary

Methods

Modifier and Type	Method and Description
byte	**byteValue**() Returns the value of this Float as a byte (by casting to a byte).
static int	**compare**(float f1, float f2) Compares the two specified float values.
int	**compareTo**(**Float** anotherFloat) Compares two Float objects numerically.
double	**doubleValue**() Returns the double value of this Float object.
boolean	**equals**(**Object** obj) Compares this object against the specified object.
static int	**floatToIntBits**(float value) Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout.
static int	**floatToRawIntBits**(float value) Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout, preserving Not-a-Number (NaN) values.
float	**floatValue**() Returns the float value of this Float object.
int	**hashCode**() Returns a hash code for this Float object.
static float	**intBitsToFloat**(int bits) Returns the float value corresponding to a given bit representation.
int	**intValue**() Returns the value of this Float as an int (by casting to type int).
boolean	**isInfinite**() Returns true if this Float value is infinitely large in magnitude, false otherwise.
static boolean	**isInfinite**(float v) Returns true if the specified number is infinitely large in magnitude, false otherwise.

boolean	**isNaN**() Returns true if this Float value is a Not-a-Number (NaN), false otherwise.
static boolean	**isNaN**(float v) Returns true if the specified number is a Not-a-Number (NaN) value, false otherwise.
long	**longValue**() Returns value of this Float as a long (by casting to type long).
static float	**parseFloat**(String s) Returns a new float initialized to the value represented by the specified String, as performed by the valueOf method of class Float.
short	**shortValue**() Returns the value of this Float as a short (by casting to a short).
static String	**toHexString**(float f) Returns a hexadecimal string representation of the float argument.
String	**toString**() Returns a string representation of this Float object.
static String	**toString**(float f) Returns a string representation of the float argument.
static Float	**valueOf**(float f) Returns a Float instance representing the specified float value.
static Float	**valueOf**(String s) Returns a Float object holding the float 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 float POSITIVE_INFINITY
```

A constant holding the positive infinity of type float. It is equal to the value returned by Float.intBitsToFloat(0x7f800000).

See Also:

Constant Field Values

NEGATIVE_INFINITY

```
public static final float NEGATIVE_INFINITY
```

A constant holding the negative infinity of type float. It is equal to the value returned by Float.intBitsToFloat(0xff800000).

See Also:

Constant Field Values

NaN

```
public static final float NaN
```

A constant holding a Not-a-Number (NaN) value of type `float`. It is equivalent to the value returned by `Float.intBitsToFloat(0x7fc00000)`.

See Also:

Constant Field Values

MAX_VALUE

`public static final float MAX_VALUE`

A constant holding the largest positive finite value of type `float`, $(2-2^{-23})\cdot 2^{127}$. It is equal to the hexadecimal floating-point literal `0x1.fffffeP+127f` and also equal to `Float.intBitsToFloat(0x7f7fffff)`.

See Also:

Constant Field Values

MIN_NORMAL

`public static final float MIN_NORMAL`

A constant holding the smallest positive normal value of type `float`, 2^{-126}. It is equal to the hexadecimal floating-point literal `0x1.0p-126f` and also equal to `Float.intBitsToFloat(0x00800000)`.

Since:

1.6

See Also:

Constant Field Values

MIN_VALUE

`public static final float MIN_VALUE`

A constant holding the smallest positive nonzero value of type `float`, 2^{-149}. It is equal to the hexadecimal floating-point literal `0x0.000002P-126f` and also equal to `Float.intBitsToFloat(0x1)`.

See Also:

Constant Field Values

MAX_EXPONENT

`public static final int MAX_EXPONENT`

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

Since:

1.6

See Also:

Constant Field Values

MIN_EXPONENT

```
public static final int MIN_EXPONENT
```

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

Since:

1.6

See Also:

Constant Field Values

SIZE

```
public static final int SIZE
```

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

Since:

1.5

See Also:

Constant Field Values

TYPE

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

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

Since:

JDK1.1

Constructor Detail

Float

```
public Float(float value)
```

Constructs a newly allocated `Float` object that represents the primitive `float` argument.

Parameters:

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

Float

```
public Float(double value)
```

Constructs a newly allocated `Float` object that represents the argument converted to type `float`.

Parameters:

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

Float

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

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

Parameters:

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

Throws:

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

See Also:

`valueOf(java.lang.String)`

Method Detail

toString

```
public static String toString(float f)
```

Returns a string representation of the `float` 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 $10^{n} \le m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^{n} so that $1 \le a < 10$. 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 `float`. That is, suppose that *x* is the exact mathematical value represented by the decimal representation produced by this method for a finite nonzero argument *f*. Then *f* must be the `float` value nearest to *x*; or, if two `float` values are equally close to *x*, then *f* must be one of them and the least significant bit of the significand of *f* must be `0`.

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

Parameters:

`f` - the float to be converted.

Returns:

a string representation of the argument.

toHexString

```
public static String toHexString(float f)
```

Returns a hexadecimal string representation of the `float` 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 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 `float` 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 `float` 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-126"`. 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`
`Float.MAX_VALUE`	`0x1.fffffep127`
`Minimum Normal Value`	`0x1.0p-126`
`Maximum Subnormal Value`	`0x0.fffffep-126`
`Float.MIN_VALUE`	`0x0.000002p-126`

Parameters:

`f` - the `float` to be converted.

Returns:

a hex string representation of the argument.

Since:

1.5

valueOf

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

Returns a `Float` object holding the `float` 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 `float` 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 `float` 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 `Float` object representing this `float` 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. In general, the two-step sequence of conversions, string to `double` followed by `double` to `float`, is *not* equivalent to converting a string directly to `float`. For example, if first converted to an intermediate `double` and then to `float`, the string
`"1.00000017881393421514957253748434595763683319091796875001d"`
results in the `float` value `1.0000002f`; if the string is converted directly to `float`, `1.0000001f` results.

To avoid calling this method on an invalid string and having a `NumberFormatException` be thrown, the documentation for `Double.valueOf` lists a regular expression which can be used to screen the input.

Parameters:

`s` - the string to be parsed.

Returns:

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

Throws:

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

valueOf

```
public static Float valueOf(float f)
```

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

Parameters:

`f` - a float value.

Returns:

a `Float` instance representing `f`.

Since:

1.5

parseFloat

```
public static float parseFloat(String s)
                        throws NumberFormatException
```

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

Parameters:

`s` - the string to be parsed.

Returns:

the `float` value represented by the string argument.

Throws:

`NullPointerException` - if the string is null

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

Since:

1.2

See Also:

`valueOf(String)`

isNaN

```
public static boolean isNaN(float 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 argument is NaN; `false` otherwise.

isInfinite

```
public static boolean isInfinite(float 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 argument is positive infinity or negative infinity; `false` otherwise.

isNaN

```
public boolean isNaN()
```

Returns `true` if this `Float` 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 `Float` 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 `Float` object. The primitive `float` 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(float)`

byteValue

```
public byte byteValue()
```

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

Overrides:

`byteValue` in class `Number`

Returns:

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

shortValue

```
public short shortValue()
```

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

Overrides:

`shortValue` in class `Number`

Returns:

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

Since:

JDK1.1

intValue

```
public int intValue()
```

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

Specified by:

`intValue` in class `Number`

Returns:

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

longValue

```
public long longValue()
```

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

Specified by:

`longValue` in class `Number`

Returns:

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

floatValue

```
public float floatValue()
```

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

Specified by:

`floatValue` in class `Number`

Returns:

the `float` value represented by this object

doubleValue

```
public double doubleValue()
```

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

Specified by:

`doubleValue` in class `Number`

Returns:

the `float` value represented by this object is converted to type `double` and the result of the conversion is returned.

hashCode

```
public int hashCode()
```

Returns a hash code for this `Float` object. The result is the integer bit representation, exactly as produced by the method `floatToIntBits(float)`, of the primitive `float` value represented by this `Float` object.

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 `Float` object that represents a `float` with the same value as the `float` represented by this object. For this purpose, two `float` values are considered to be the same if and only if the method `floatToIntBits(float)` returns the identical `int` value when applied to each.

Note that in most cases, for two instances of class `Float`, `f1` and `f2`, the value of `f1.equals(f2)` is `true` if and only if

```
f1.floatValue() == f2.floatValue()
```

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

- If `f1` and `f2` both represent `Float.NaN`, then the `equals` method returns `true`, even though `Float.NaN==Float.NaN` has the value `false`.
- If `f1` represents `+0.0f` while `f2` represents `-0.0f`, or vice versa, the `equal` test has the value `false`, even though `0.0f==-0.0f` has the value `true`.

This definition allows hash tables to operate properly.

Overrides:

`equals` in class `Object`

Parameters:

`obj` - the object to be compared

Returns:

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

See Also:

`floatToIntBits(float)`

floatToIntBits

```
public static int floatToIntBits(float value)
```

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

Bit 31 (the bit that is selected by the mask `0x80000000`) represents the sign of the floating-point number. Bits 30-23 (the bits that are selected by the mask `0x7f800000`) represent the exponent. Bits 22-0 (the bits that are selected by the mask `0x007fffff`) represent the significand (sometimes called the mantissa) of the floating-point number.

If the argument is positive infinity, the result is `0x7f800000`.

If the argument is negative infinity, the result is `0xff800000`.

If the argument is NaN, the result is `0x7fc00000`.

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

Parameters:

`value` - a floating-point number.

Returns:

the bits that represent the floating-point number.

floatToRawIntBits

```
public static int floatToRawIntBits(float value)
```

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

Bit 31 (the bit that is selected by the mask `0x80000000`) represents the sign of the floating-point number. Bits 30-23 (the bits that are selected by the mask `0x7f800000`) represent the exponent. Bits 22-0 (the bits that are selected by the mask `0x007fffff`) represent the significand (sometimes called the mantissa) of the floating-point number.

If the argument is positive infinity, the result is `0x7f800000`.

If the argument is negative infinity, the result is `0xff800000`.

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

In all cases, the result is an integer that, when given to the `intBitsToFloat(int)` method, will produce a floating-point value the same as the argument to `floatToRawIntBits`.

Parameters:

`value` - a floating-point number.

Returns:

the bits that represent the floating-point number.

Since:

1.3

intBitsToFloat

```
public static float intBitsToFloat(int bits)
```

Returns the `float` 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 "single format" bit layout.

If the argument is `0x7f800000`, the result is positive infinity.

If the argument is `0xff800000`, the result is negative infinity.

If the argument is any value in the range `0x7f800001` through `0x7fffffff` or in the range `0xff800001` through `0xffffffff`, 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 `Float.floatToRawIntBits` method.

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

```
int s = ((bits >> 31) == 0) ? 1 : -1;
int e = ((bits >> 23) & 0xff);
int m = (e == 0) ?
                (bits & 0x7fffff) << 1 :
                (bits & 0x7fffff) | 0x800000;
```

Then the floating-point result equals the value of the mathematical expression $s \cdot m \cdot 2^{e-150}$.

Note that this method may not be able to return a `float` NaN with exactly same bit pattern as the `int` 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 `intBitsToFloat` may not be able to return a `float` with a signaling NaN bit pattern. Consequently, for some `int` values, `floatToRawIntBits(intBitsToFloat(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` - an integer.

Returns:

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

compareTo

`public int compareTo(Float anotherFloat)`

Compares two `Float` 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 `float` values:

- `Float.NaN` is considered by this method to be equal to itself and greater than all other `float` values (including `Float.POSITIVE_INFINITY`).
- `0.0f` is considered by this method to be greater than `-0.0f`.

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

Specified by:

`compareTo` in interface `Comparable<Float>`

Parameters:

`anotherFloat` - the `Float` to be compared.

Returns:

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

Since:

1.2

See Also:

`Comparable.compareTo(Object)`

compare

```
public static int compare(float f1,
                          float f2)
```

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

```
new Float(f1).compareTo(new Float(f2))
```

Parameters:

`f1` - the first `float` to compare.

`f2` - the second `float` to compare.

Returns:

the value `0` if `f1` is numerically equal to `f2`; a value less than `0` if `f1` is numerically less than `f2`; and a value greater than `0` if `f1` is numerically greater than `f2`.

Since:

1.4