java.util

Class Formatter

java.lang.Object
  └ **java.util.Formatter**

All Implemented Interfaces:
Closeable, Flushable

```
public final class Formatter
extends Object
implements Closeable, Flushable
```

An interpreter for printf-style format strings. This class provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output. Common Java types such as `byte`, `BigDecimal`, and `Calendar` are supported. Limited formatting customization for arbitrary user types is provided through the `Formattable` interface.

Formatters are not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class.

Formatted printing for the Java language is heavily inspired by C's `printf`. Although the format strings are similar to C, some customizations have been made to accommodate the Java language and exploit some of its features. Also, Java formatting is more strict than C's; for example, if a conversion is incompatible with a flag, an exception will be thrown. In C inapplicable flags are silently ignored. The format strings are thus intended to be recognizable to C programmers but not necessarily completely compatible with those in C.

Examples of expected usage:

```
StringBuilder sb = new StringBuilder();
// Send all output to the Appendable object sb
Formatter formatter = new Formatter(sb, Locale.US);

// Explicit argument indices may be used to re-order output.
formatter.format("%4$2s %3$2s %2$2s %1$2s", "a", "b", "c", "d")
// -> " d  c  b  a"

// Optional locale as the first argument can be used to get
// locale-specific formatting of numbers.  The precision and width can be
// given to round and align the value.
formatter.format(Locale.FRANCE, "e = %+10.4f", Math.E);
// -> "e =    +2,7183"

// The '(' numeric flag may be used to format negative numbers with
// parentheses rather than a minus sign.  Group separators are
// automatically inserted.
formatter.format("Amount gained or lost since last statement: $ %(,.2f",
```

```
                    balanceDelta);
// -> "Amount gained or lost since last statement: $ (6,217.58)"
```

Convenience methods for common formatting requests exist as illustrated by the following invocations:

```
// Writes a formatted string to System.out.
System.out.format("Local time: %tT", Calendar.getInstance());
// -> "Local time: 13:34:18"

// Writes formatted output to System.err.
System.err.printf("Unable to open file '%1$s': %2$s",
                  fileName, exception.getMessage());
// -> "Unable to open file 'food': No such file or directory"
```

Like C's `sprintf(3)`, Strings may be formatted using the static method `String.format`:

```
// Format a string containing a date.
import java.util.Calendar;
import java.util.GregorianCalendar;
import static java.util.Calendar.*;

Calendar c = new GregorianCalendar(1995, MAY, 23);
String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
// -> s == "Duke's Birthday: May 23, 1995"
```

Organization

This specification is divided into two sections. The first section, Summary, covers the basic formatting concepts. This section is intended for users who want to get started quickly and are familiar with formatted printing in other programming languages. The second section, Details, covers the specific implementation details. It is intended for users who want more precise specification of formatting behavior.

Summary

This section is intended to provide a brief overview of formatting concepts. For precise behavioral details, refer to the Details section.

Format String Syntax

Every method which produces formatted output requires a *format string* and an *argument list*. The format string is a `String` which may contain fixed text and one or more embedded *format specifiers*. Consider the following example:

```
Calendar c = ...;
String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
```

This format string is the first argument to the `format` method. It contains three format specifiers "`%1$tm`", "`%1$te`", and "`%1$tY`" which indicate how the arguments should be processed and where they should be inserted in the text. The remaining portions of the format string are fixed text including "`Dukes Birthday:`

" and any other spaces or punctuation. The argument list consists of all arguments passed to the method after the format string. In the above example, the argument list is of size one and consists of the new Calendar object.

- The format specifiers for general, character, and numeric types have the following syntax:

```
%[argument_index$][flags][width][.precision]conversion
```

  The optional *argument_index* is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "1$", the second by "2$", etc.

  The optional *flags* is a set of characters that modify the output format. The set of valid flags depends on the conversion.

  The optional *width* is a non-negative decimal integer indicating the minimum number of characters to be written to the output.

  The optional *precision* is a non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.

  The required *conversion* is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type.

- The format specifiers for types which are used to represents dates and times have the following syntax:

```
%[argument_index$][flags][width]conversion
```

  The optional *argument_index*, *flags* and *width* are defined as above.

  The required *conversion* is a two character sequence. The first character is 't' or 'T'. The second character indicates the format to be used. These characters are similar to but not completely identical to those defined by GNU date and POSIX strftime(3c).

- The format specifiers which do not correspond to arguments have the following syntax:

```
%[flags][width]conversion
```

  The optional *flags* and *width* is defined as above.

  The required *conversion* is a character indicating content to be inserted in the output.

Conversions

Conversions are divided into the following categories:

1. **General** - may be applied to any argument type
2. **Character** - may be applied to basic types which represent Unicode characters: char, Character, byte, Byte, short, and Short. This conversion may also be applied to the types int and Integer when Character.isValidCodePoint(int) returns true

3. **Numeric**
    1. **Integral** - may be applied to Java integral types: `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, `Long`, and `BigInteger`
    2. **Floating Point** - may be applied to Java floating-point types: `float`, `Float`, `double`, `Double`, and `BigDecimal`
4. **Date/Time** - may be applied to Java types which are capable of encoding a date or time: `long`, `Long`, `Calendar`, and `Date`.
5. **Percent** - produces a literal `'%'` (`'\u0025'`)
6. **Line Separator** - produces the platform-specific line separator

The following table summarizes the supported conversions. Conversions denoted by an upper-case character (i.e. `'B'`, `'H'`, `'S'`, `'C'`, `'X'`, `'E'`, `'G'`, `'A'`, and `'T'`) are the same as those for the corresponding lower-case conversion characters except that the result is converted to upper case according to the rules of the prevailing `Locale`. The result is equivalent to the following invocation of `String.toUpperCase()`

```
out.toUpperCase()
```

Conversion	Argument Category	Description
`'b'`, `'B'`	general	If the argument *arg* is `null`, then the result is "`false`". If *arg* is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf()`. Otherwise, the result is "true".
`'h'`, `'H'`	general	If the argument *arg* is `null`, then the result is "`null`". Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`.
`'s'`, `'S'`	general	If the argument *arg* is `null`, then the result is "`null`". If *arg* implements `Formattable`, then `arg.formatTo` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.
`'c'`, `'C'`	character	The result is a Unicode character
`'d'`	integral	The result is formatted as a decimal integer
`'o'`	integral	The result is formatted as an octal integer
`'x'`, `'X'`	integral	The result is formatted as a hexadecimal integer
`'e'`, `'E'`	floating point	The result is formatted as a decimal number in computerized scientific notation
`'f'`	floating point	The result is formatted as a decimal number
`'g'`, `'G'`	floating point	The result is formatted using computerized scientific notation or decimal format, depending on the precision and the value after rounding.
`'a'`, `'A'`	floating point	The result is formatted as a hexadecimal floating-point number with a significand and an exponent
`'t'`, `'T'`	date/time	Prefix for date and time conversion characters. See Date/Time Conversions.
`'%'`	percent	The result is a literal `'%'` (`'\u0025'`)

'n'	line separator	The result is the platform-specific line separator

Any characters not explicitly defined as conversions are illegal and are reserved for future extensions.

Date/Time Conversions

The following date and time conversion suffix characters are defined for the 't' and 'T' conversions. The types are similar to but not completely identical to those defined by GNU date and POSIX strftime(3c). Additional conversion types are provided to access Java-specific functionality (e.g. 'L' for milliseconds within the second).

The following conversion characters are used for formatting times:

- 'H' Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary i.e. 00 - 23.
- 'I' Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary, i.e. 01 - 12.
- 'k' Hour of the day for the 24-hour clock, i.e. 0 - 23.
- 'l' Hour for the 12-hour clock, i.e. 1 - 12.
- 'M' Minute within the hour formatted as two digits with a leading zero as necessary, i.e. 00 - 59.
- 'S' Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. 00 - 60 ("60" is a special value required to support leap seconds).
- 'L' Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. 000 - 999.
- 'N' Nanosecond within the second, formatted as nine digits with leading zeros as necessary, i.e. 000000000 - 999999999.
- 'p' Locale-specific morning or afternoon marker in lower case, e.g."am" or "pm". Use of the conversion prefix 'T' forces this output to upper case.
- 'z' RFC 822 style numeric time zone offset from GMT, e.g. -0800.
- 'Z' A string representing the abbreviation for the time zone. The Formatter's locale will supersede the locale of the argument (if any).
- 's' Seconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE/1000 to Long.MAX_VALUE/1000.
- 'Q' Milliseconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE to Long.MAX_VALUE.

The following conversion characters are used for formatting dates:

- 'B' Locale-specific full month name, e.g. "January", "February".
- 'b' Locale-specific abbreviated month name, e.g. "Jan", "Feb".

'h' Same as 'b'.

'A' Locale-specific full name of the day of the week, e.g. "Sunday", "Monday"

'a' Locale-specific short name of the day of the week, e.g. "Sun", "Mon"

'C' Four-digit year divided by 100, formatted as two digits with leading zero as necessary, i.e. 00 - 99

'Y' Year, formatted as at least four digits with leading zeros as necessary, e.g. 0092 equals 92 CE for the Gregorian calendar.

'y' Last two digits of the year, formatted with leading zeros as necessary, i.e. 00 - 99.

'j' Day of year, formatted as three digits with leading zeros as necessary, e.g. 001 - 366 for the Gregorian calendar.

'm' Month, formatted as two digits with leading zeros as necessary, i.e. 01 - 13.

'd' Day of month, formatted as two digits with leading zeros as necessary, i.e. 01 - 31

'e' Day of month, formatted as two digits, i.e. 1 - 31.

The following conversion characters are used for formatting common date/time compositions.

'R' Time formatted for the 24-hour clock as "%tH:%tM"

'T' Time formatted for the 24-hour clock as "%tH:%tM:%tS".

'r' Time formatted for the 12-hour clock as "%tI:%tM:%tS %Tp". The location of the morning or afternoon marker ('%Tp') may be locale-dependent.

'D' Date formatted as "%tm/%td/%ty".

'F' ISO 8601 complete date formatted as "%tY-%tm-%td".

'c' Date and time formatted as "%ta %tb %td %tT %tZ %tY", e.g. "Sun Jul 20 16:17:00 EDT 1969".

Any characters not explicitly defined as date/time conversion suffixes are illegal and are reserved for future extensions.

Flags

The following table summarizes the supported flags. *y* means the flag is supported for the indicated argument types.

Flag	General	Character	Integral	Floating Point	Date/Time	Description
'-'	y	y	y	y	y	The result will be left-justified.
'#'	y[1]	-	y[3]	y	-	The result should use a conversion-dependent alternate form
'+'	-	-	y[4]	y	-	The result will always include a sign

' '	-	-	y[4]	y	-	The result will include a leading space for positive values
'0'	-	-	y	y	-	The result will be zero-padded
','	-	-	y[2]	y[5]	-	The result will include locale-specific grouping separators
'('	-	-	y[4]	y[5]	-	The result will enclose negative numbers in parentheses

[1] Depends on the definition of `Formattable`.

[2] For `'d'` conversion only.

[3] For `'o'`, `'x'`, and `'X'` conversions only.

[4] For `'d'`, `'o'`, `'x'`, and `'X'` conversions applied to `BigInteger` or `'d'` applied to `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, and `Long`.

[5] For `'e'`, `'E'`, `'f'`, `'g'`, and `'G'` conversions only.

Any characters not explicitly defined as flags are illegal and are reserved for future extensions.

Width

The width is the minimum number of characters to be written to the output. For the line separator conversion, width is not applicable; if it is provided, an exception will be thrown.

Precision

For general argument types, the precision is the maximum number of characters to be written to the output.

For the floating-point conversions `'e'`, `'E'`, and `'f'` the precision is the number of digits after the decimal separator. If the conversion is `'g'` or `'G'`, then the precision is the total number of digits in the resulting magnitude after rounding. If the conversion is `'a'` or `'A'`, then the precision must not be specified.

For character, integral, and date/time argument types and the percent and line separator conversions, the precision is not applicable; if a precision is provided, an exception will be thrown.

Argument Index

The argument index is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "`1$`", the second by "`2$`", etc.

Another way to reference arguments by position is to use the `'<'` (`'\u003c'`) flag, which causes the argument for the previous format specifier to be re-used. For example, the following two statements would produce identical strings:

```
Calendar c = ...;
String s1 = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);

String s2 = String.format("Duke's Birthday: %1$tm %<$te,%<$tY", c);
```

Details

This section is intended to provide behavioral details for formatting, including conditions and exceptions, supported data types, localization, and interactions between flags, conversions, and data types. For an overview of formatting concepts, refer to the Summary

Any characters not explicitly defined as conversions, date/time conversion suffixes, or flags are illegal and are reserved for future extensions. Use of such a character in a format string will cause an `UnknownFormatConversionException` or `UnknownFormatFlagsException` to be thrown.

If the format specifier contains a width or precision with an invalid value or which is otherwise unsupported, then a `IllegalFormatWidthException` or `IllegalFormatPrecisionException` respectively will be thrown.

If a format specifier contains a conversion character that is not applicable to the corresponding argument, then an `IllegalFormatConversionException` will be thrown.

All specified exceptions may be thrown by any of the `format` methods of `Formatter` as well as by any `format` convenience methods such as `String.format` and `PrintStream.printf`.

Conversions denoted by an upper-case character (i.e. `'B'`, `'H'`, `'S'`, `'C'`, `'X'`, `'E'`, `'G'`, `'A'`, and `'T'`) are the same as those for the corresponding lower-case conversion characters except that the result is converted to upper case according to the rules of the prevailing `Locale`. The result is equivalent to the following invocation of `String.toUpperCase()`

```
out.toUpperCase()
```

General

The following general conversions may be applied to any argument type:

`'b'`	`'\u0062'`	Produces either "`true`" or "`false`" as returned by `Boolean.toString(boolean)`. If the argument is `null`, then the result is "`false`". If the argument is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf()`. Otherwise, the result is "`true`". If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.
`'B'`	`'\u0042'`	The upper-case variant of `'b'`.
`'h'`	`'\u0068'`	Produces a string representing the hash code value of the object.

If the argument, *arg* is `null`, then the result is "`null`". Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`.

If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.

`'H'` `'\u0048'` The upper-case variant of `'h'`.

`'s'` `'\u0073'` Produces a string.

If the argument is `null`, then the result is "`null`". If the argument implements `Formattable`, then its `formatTo` method is invoked. Otherwise, the result is obtained by invoking the argument's `toString()` method.

If the `'#'` flag is given and the argument is not a `Formattable` , then a `FormatFlagsConversionMismatchException` will be thrown.

`'S'` `'\u0053'` The upper-case variant of `'s'`.

The following flags apply to general conversions:

`'-'` `'\u002d'` Left justifies the output. Spaces (`'\u0020'`) will be added at the end of the converted value as required to fill the minimum width of the field. If the width is not provided, then a `MissingFormatWidthException` will be thrown. If this flag is not given then the output will be right-justified.

`'#'` `'\u0023'` Requires the output use an alternate form. The definition of the form is specified by the conversion.

The width is the minimum number of characters to be written to the output. If the length of the converted value is less than the width then the output will be padded by `' '` (`\u0020'`) until the total number of characters equals the width. The padding is on the left by default. If the `'-'` flag is given, then the padding will be on the right. If the width is not specified then there is no minimum.

The precision is the maximum number of characters to be written to the output. The precision is applied before the width, thus the output will be truncated to `precision` characters even if the width is greater than the precision. If the precision is not specified then there is no explicit limit on the number of characters.

Character

This conversion may be applied to `char`, `Character`, `byte`, `Byte`, `short`, and `Short`. This conversion may also be applied to the types `int` and `Integer` when `Character.isValidCodePoint(int)` returns `true`. If it returns `false` then an `IllegalFormatCodePointException` will be thrown.

`'c'` `'\u0063'` Formats the argument as a Unicode character as described in Unicode Character Representation. This may be more than one 16-bit `char` in the case where the argument represents a supplementary character.

If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be

thrown.

`'C'`	`'\u0043'`	The upper-case variant of `'c'`.

The `'-'` flag defined for General conversions applies. If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.

The width is defined as for General conversions.

The precision is not applicable. If the precision is specified then an `IllegalFormatPrecisionException` will be thrown.

Numeric

Numeric conversions are divided into the following categories:

1. **Byte, Short, Integer, and Long**
2. **BigInteger**
3. **Float and Double**
4. **BigDecimal**

Numeric types will be formatted according to the following algorithm:

Number Localization Algorithm

After digits are obtained for the integer part, fractional part, and exponent (as appropriate for the data type), the following transformation is applied:

1. Each digit character *d* in the string is replaced by a locale-specific digit computed relative to the current locale's zero digit *z*; that is *d* `-` `'0'` + *z*.
2. If a decimal separator is present, a locale-specific decimal separator is substituted.
3. If the `','` (`'\u002c'`) flag is given, then the locale-specific grouping separator is inserted by scanning the integer part of the string from least significant to most significant digits and inserting a separator at intervals defined by the locale's grouping size.
4. If the `'0'` flag is given, then the locale-specific zero digits are inserted after the sign character, if any, and before the first non-zero digit, until the length of the string is equal to the requested field width.
5. If the value is negative and the `'('` flag is given, then a `'('` (`'\u0028'`) is prepended and a `')'` (`'\u0029'`) is appended.
6. If the value is negative (or floating-point negative zero) and `'('` flag is not given, then a `'-'` (`'\u002d'`) is prepended.
7. If the `'+'` flag is given and the value is positive or zero (or floating-point positive zero), then a `'+'` (`'\u002b'`) will be prepended.

If the value is NaN or positive infinity the literal strings "NaN" or "Infinity" respectively, will be output. If the value is negative infinity, then the output will be "(Infinity)" if the `'('` flag is given otherwise the output will be "-Infinity". These values are not localized.

Byte, Short, Integer, and Long

The following conversions may be applied to `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, and `Long`.

`'d'`	`'\u0054'`	Formats the argument as a decimal integer. The localization algorithm is applied. If the `'0'` flag is given and the value is negative, then the zero padding will occur after the sign. If the `'#'` flag is given then a `FormatFlagsConversionMismatchException` will be thrown.
`'o'`	`'\u006f'`	Formats the argument as an integer in base eight. No localization is applied. If *x* is negative then the result will be an unsigned value generated by adding 2^n to the value where `n` is the number of bits in the type as returned by the static `SIZE` field in the Byte, Short, Integer, or Long classes as appropriate. If the `'#'` flag is given then the output will always begin with the radix indicator `'0'`. If the `'0'` flag is given then the output will be padded with leading zeros to the field width following any indication of sign. If `'('`, `'+'`, `' '`, or `','` flags are given then a `FormatFlagsConversionMismatchException` will be thrown.
`'x'`	`'\u0078'`	Formats the argument as an integer in base sixteen. No localization is applied. If *x* is negative then the result will be an unsigned value generated by adding 2^n to the value where `n` is the number of bits in the type as returned by the static `SIZE` field in the Byte, Short, Integer, or Long classes as appropriate. If the `'#'` flag is given then the output will always begin with the radix indicator `"0x"`. If the `'0'` flag is given then the output will be padded to the field width with leading zeros after the radix indicator or sign (if present). If `'('`, `' '`, `'+'`, or `','` flags are given then a `FormatFlagsConversionMismatchException` will be thrown.
`'X'`	`'\u0058'`	The upper-case variant of `'x'`. The entire string representing the number will be converted to upper case including the `'x'` (if any) and all hexadecimal digits `'a'` - `'f'` (`'\u0061'` - `'\u0066'`).

If the conversion is `'o'`, `'x'`, or `'X'` and both the `'#'` and the `'0'` flags are given, then result will contain the radix indicator (`'0'` for octal and `"0x"` or `"0X"` for hexadecimal), some number of zeros (based on the width), and the value.

If the `'-'` flag is not given, then the space padding will occur before the sign.

The following flags apply to numeric integral conversions:

'+'	'\u002b'	Requires the output to include a positive sign for all positive numbers. If this flag is not given then only negative values will include a sign. If both the '+' and ' ' flags are given then an IllegalFormatFlagsException will be thrown.
' '	'\u0020'	Requires the output to include a single extra space ('\u0020') for non-negative values. If both the '+' and ' ' flags are given then an IllegalFormatFlagsException will be thrown.
'0'	'\u0030'	Requires the output to be padded with leading zeros to the minimum field width following any sign or radix indicator except when converting NaN or infinity. If the width is not provided, then a MissingFormatWidthException will be thrown. If both the '-' and '0' flags are given then an IllegalFormatFlagsException will be thrown.
','	'\u002c'	Requires the output to include the locale-specific group separators as described in the "group" section of the localization algorithm.
'('	'\u0028'	Requires the output to prepend a '(' ('\u0028') and append a ')' ('\u0029') to negative values.

If no flags are given the default formatting is as follows:

- The output is right-justified within the width
- Negative numbers begin with a '-' ('\u002d')
- Positive numbers and zero do not include a sign or extra leading space
- No grouping separators are included

The width is the minimum number of characters to be written to the output. This includes any signs, digits, grouping separators, radix indicator, and parentheses. If the length of the converted value is less than the width then the output will be padded by spaces ('\u0020') until the total number of characters equals width. The padding is on the left by default. If '-' flag is given then the padding will be on the right. If width is not specified then there is no minimum.

The precision is not applicable. If precision is specified then an IllegalFormatPrecisionException will be thrown.

BigInteger

The following conversions may be applied to BigInteger.

'd'	'\u0054'	Requires the output to be formatted as a decimal integer. The localization algorithm is applied.

If the '#' flag is given FormatFlagsConversionMismatchException will be thrown.

'o' '\u006f' Requires the output to be formatted as an integer in base eight. No localization is applied.

If *x* is negative then the result will be a signed value beginning with '-' ('\u002d'). Signed output is allowed for this type because unlike the primitive types it is not possible to create an unsigned equivalent without assuming an explicit data-type size.

If *x* is positive or zero and the '+' flag is given then the result will begin with '+' ('\u002b').

If the '#' flag is given then the output will always begin with '0' prefix.

If the '0' flag is given then the output will be padded with leading zeros to the field width following any indication of sign.

If the ',' flag is given then a FormatFlagsConversionMismatchException will be thrown.

'x' '\u0078' Requires the output to be formatted as an integer in base sixteen. No localization is applied.

If *x* is negative then the result will be a signed value beginning with '-' ('\u002d'). Signed output is allowed for this type because unlike the primitive types it is not possible to create an unsigned equivalent without assuming an explicit data-type size.

If *x* is positive or zero and the '+' flag is given then the result will begin with '+' ('\u002b').

If the '#' flag is given then the output will always begin with the radix indicator "0x".

If the '0' flag is given then the output will be padded to the field width with leading zeros after the radix indicator or sign (if present).

If the ',' flag is given then a FormatFlagsConversionMismatchException will be thrown.

'X' '\u0058' The upper-case variant of 'x'. The entire string representing the number will be converted to upper case including the 'x' (if any) and all hexadecimal digits 'a' - 'f' ('\u0061' - '\u0066').

If the conversion is 'o', 'x', or 'X' and both the '#' and the '0' flags are given, then result will contain the base indicator ('0' for octal and "0x" or "0X" for hexadecimal), some number of zeros (based on the width), and the value.

If the '0' flag is given and the value is negative, then the zero padding will occur after the sign.

If the '-' flag is not given, then the space padding will occur before the sign.

All flags defined for Byte, Short, Integer, and Long apply. The default behavior when no flags are given is the same as for Byte, Short, Integer, and Long.

The specification of width is the same as defined for Byte, Short, Integer, and Long.

The precision is not applicable. If precision is specified then an `IllegalFormatPrecisionException` will be thrown.

Float and Double

The following conversions may be applied to `float`, `Float`, `double` and `Double`.

`'e'` `'\u0065'` Requires the output to be formatted using computerized scientific notation. The localization algorithm is applied.

The formatting of the magnitude *m* depends upon its value.

If *m* is NaN or infinite, the literal strings "NaN" or "Infinity", respectively, will be output. These values are not localized.

If *m* is positive-zero or negative-zero, then the exponent will be `"+00"`.

Otherwise, the result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value.

Let *n* be the unique integer such that $10^n <= m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^n so that $1 <= a < 10$. The magnitude is then represented as the integer part of *a*, as a single decimal digit, followed by the decimal separator followed by decimal digits representing the fractional part of *a*, followed by the exponent symbol `'e'` (`'\u0065'`), followed by the sign of the exponent, followed by a representation of *n* as a decimal integer, as produced by the method `Long.toString(long, int)`, and zero-padded to include at least two digits.

The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is `6`. If the precision is less than the number of digits which would appear after the decimal point in the string returned by `Float.toString(float)` or `Double.toString(double)` respectively, then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value, use `Float.toString(float)` or `Double.toString(double)` as appropriate.

If the `','` flag is given, then an `FormatFlagsConversionMismatchException` will be thrown.

`'E'` `'\u0045'` The upper-case variant of `'e'`. The exponent symbol will be `'E'` (`'\u0045'`).

`'g'` `'\u0067'` Requires the output to be formatted in general scientific notation as described below. The localization algorithm is applied.

After rounding for the precision, the formatting of the resulting magnitude *m* depends

on its value.

If *m* is greater than or equal to 10^{-4} but less than $10^{precision}$ then it is represented in *decimal format*.

If *m* is less than 10^{-4} or greater than or equal to $10^{precision}$, then it is represented in *computerized scientific notation*.

The total number of significant digits in *m* is equal to the precision. If the precision is not specified, then the default value is `6`. If the precision is `0`, then it is taken to be `1`.

If the `'#'` flag is given then an `FormatFlagsConversionMismatchException` will be thrown.

`'G'` `'\u0047'` The upper-case variant of `'g'`.

`'f'` `'\u0066'` Requires the output to be formatted using decimal format. The localization algorithm is applied.

The result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value.

If *m* NaN or infinite, the literal strings "NaN" or "Infinity", respectively, will be output. These values are not localized.

The magnitude is formatted as the integer part of *m*, with no leading zeroes, followed by the decimal separator followed by one or more decimal digits representing the fractional part of *m*.

The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is `6`. If the precision is less than the number of digits which would appear after the decimal point in the string returned by `Float.toString(float)` or `Double.toString(double)` respectively, then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value,use `Float.toString(float)` or `Double.toString(double)` as appropriate.

`'a'` `'\u0061'` Requires the output to be formatted in hexadecimal exponential form. No localization is applied.

The result is a string that represents the sign and magnitude (absolute value) of the argument *x*.

If *x* is negative or a negative-zero value then the result will begin with `'-'` (`'\u002d'`).

If *x* is positive or a positive-zero value and the `'+'` flag is given then the result will begin with `'+'` (`'\u002b'`).

The formatting of the magnitude *m* depends upon its value.

- If the value is NaN or infinite, the literal strings "NaN" or "Infinity", respectively, will be output.
- If *m* is zero then it is represented by the string `"0x0.0p0"`.
- If *m* is a `double` value with a normalized representation then substrings are used to represent the significand and exponent fields. The significand is represented by the characters `"0x1."` followed by the hexadecimal representation of the rest of the significand as a fraction. The exponent is represented by `'p'` (`'\u0070'`) followed by a decimal string of the unbiased exponent as if produced by invoking `Integer.toString` on the exponent value.
- If *m* is a `double` value with a subnormal representation then the significand is represented by the characters `'0x0.'` followed by the hexadecimal representation of the rest of the significand as a fraction. The exponent is represented by `'p-1022'`. Note that there must be at least one nonzero digit in a subnormal significand.

If the `'('` or `','` flags are given, then a `FormatFlagsConversionMismatchException` will be thrown.

`'A'` `'\u0041'` The upper-case variant of `'a'`. The entire string representing the number will be converted to upper case including the `'x'` (`'\u0078'`) and `'p'` (`'\u0070'` and all hexadecimal digits `'a'` - `'f'` (`'\u0061'` - `'\u0066'`).

All flags defined for Byte, Short, Integer, and Long apply.

If the `'#'` flag is given, then the decimal separator will always be present.

If no flags are given the default formatting is as follows:

- The output is right-justified within the `width`
- Negative numbers begin with a `'-'`
- Positive numbers and positive zero do not include a sign or extra leading space
- No grouping separators are included
- The decimal separator will only appear if a digit follows it

The width is the minimum number of characters to be written to the output. This includes any signs, digits, grouping separators, decimal separators, exponential symbol, radix indicator, parentheses, and strings representing infinity and NaN as applicable. If the length of the converted value is less than the width then the output will be padded by spaces (`'\u0020'`) until the total number of characters equals width. The padding is on the left by default. If the `'-'` flag is given then the padding will be on the right. If width is not specified then there is no minimum.

If the conversion is `'e'`, `'E'` or `'f'`, then the precision is the number of digits after the decimal separator. If the precision is not specified, then it is assumed to be `6`.

If the conversion is `'g'` or `'G'`, then the precision is the total number of significant digits in the resulting magnitude after rounding. If the precision is not specified, then the default value is `6`. If the precision is `0`, then it is taken to be `1`.

If the conversion is `'a'` or `'A'`, then the precision is the number of hexadecimal digits after the decimal separator. If the precision is not provided, then all of the digits as returned by

Double.toHexString(double) will be output.

BigDecimal

The following conversions may be applied BigDecimal.

'e'	'\u0065'	Requires the output to be formatted using computerized scientific notation. The localization algorithm is applied. The formatting of the magnitude *m* depends upon its value. If *m* is positive-zero or negative-zero, then the exponent will be "+00". Otherwise, the result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value. Let *n* be the unique integer such that $10^n <= m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^n so that $1 <= a < 10$. The magnitude is then represented as the integer part of *a*, as a single decimal digit, followed by the decimal separator followed by decimal digits representing the fractional part of *a*, followed by the exponent symbol 'e' ('\u0065'), followed by the sign of the exponent, followed by a representation of *n* as a decimal integer, as produced by the method Long.toString(long, int), and zero-padded to include at least two digits. The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is 6. If the precision is less than the number of digits which would appear after the decimal point in the string returned by Float.toString(float) or Double.toString(double) respectively, then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value, use BigDecimal.toString(). If the ',' flag is given, then an FormatFlagsConversionMismatchException will be thrown.
'E'	'\u0045'	The upper-case variant of 'e'. The exponent symbol will be 'E' ('\u0045').
'g'	'\u0067'	Requires the output to be formatted in general scientific notation as described below. The localization algorithm is applied. After rounding for the precision, the formatting of the resulting magnitude *m* depends on its value. If *m* is greater than or equal to 10^{-4} but less than $10^{precision}$ then it is represented in *decimal format*. If *m* is less than 10^{-4} or greater than or equal to $10^{precision}$, then it is represented in *computerized scientific notation*.

The total number of significant digits in *m* is equal to the precision. If the precision is not specified, then the default value is `6`. If the precision is `0`, then it is taken to be `1`.

If the `'#'` flag is given then an `FormatFlagsConversionMismatchException` will be thrown.

`'G'`	`'\u0047'`	The upper-case variant of `'g'`.
`'f'`	`'\u0066'`	Requires the output to be formatted using decimal format. The localization algorithm is applied. The result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value. The magnitude is formatted as the integer part of *m*, with no leading zeroes, followed by the decimal separator followed by one or more decimal digits representing the fractional part of *m*. The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is `6`. If the precision is less than the number of digits which would appear after the decimal point in the string returned by `Float.toString(float)` or `Double.toString(double)` respectively, then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value, use `BigDecimal.toString()`.

All flags defined for Byte, Short, Integer, and Long apply.

If the `'#'` flag is given, then the decimal separator will always be present.

The default behavior when no flags are given is the same as for Float and Double.

The specification of width and precision is the same as defined for Float and Double.

Date/Time

This conversion may be applied to `long`, `Long`, `Calendar`, and `Date`.

`'t'`	`'\u0074'`	Prefix for date and time conversion characters.
`'T'`	`'\u0054'`	The upper-case variant of `'t'`.

The following date and time conversion character suffixes are defined for the `'t'` and `'T'` conversions. The types are similar to but not completely identical to those defined by GNU `date` and POSIX `strftime(3c)`. Additional conversion types are provided to access Java-specific functionality (e.g. `'L'` for milliseconds within the second).

The following conversion characters are used for formatting times:

'H'	'\u0048'	Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary i.e. 00 - 23. 00 corresponds to midnight.
'I'	'\u0049'	Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary, i.e. 01 - 12. 01 corresponds to one o'clock (either morning or afternoon).
'k'	'\u006b'	Hour of the day for the 24-hour clock, i.e. 0 - 23. 0 corresponds to midnight.
'l'	'\u006c'	Hour for the 12-hour clock, i.e. 1 - 12. 1 corresponds to one o'clock (either morning or afternoon).
'M'	'\u004d'	Minute within the hour formatted as two digits with a leading zero as necessary, i.e. 00 - 59.
'S'	'\u0053'	Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. 00 - 60 ("60" is a special value required to support leap seconds).
'L'	'\u004c'	Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. 000 - 999.
'N'	'\u004e'	Nanosecond within the second, formatted as nine digits with leading zeros as necessary, i.e. 000000000 - 999999999. The precision of this value is limited by the resolution of the underlying operating system or hardware.
'p'	'\u0070'	Locale-specific morning or afternoon marker in lower case, e.g."am" or "pm". Use of the conversion prefix 'T' forces this output to upper case. (Note that 'p' produces lower-case output. This is different from GNU date and POSIX strftime(3c) which produce upper-case output.)
'z'	'\u007a'	RFC 822 style numeric time zone offset from GMT, e.g. -0800.
'Z'	'\u005a'	A string representing the abbreviation for the time zone.
's'	'\u0073'	Seconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE/1000 to Long.MAX_VALUE/1000.
'Q'	'\u004f'	Milliseconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE to Long.MAX_VALUE. The precision of this value is limited by the resolution of the underlying operating system or hardware.

The following conversion characters are used for formatting dates:

'B'	'\u0042'	Locale-specific full month name, e.g. "January", "February".
'b'	'\u0062'	Locale-specific abbreviated month name, e.g. "Jan", "Feb".
'h'	'\u0068'	Same as 'b'.
'A'	'\u0041'	Locale-specific full name of the day of the week, e.g. "Sunday", "Monday"
'a'	'\u0061'	Locale-specific short name of the day of the week, e.g. "Sun", "Mon"
'C'	'\u0043'	Four-digit year divided by 100, formatted as two digits with leading zero as necessary, i.e. 00 - 99

'Y'	'\u0059'	Year, formatted to at least four digits with leading zeros as necessary, e.g. 0092 equals 92 CE for the Gregorian calendar.
'y'	'\u0079'	Last two digits of the year, formatted with leading zeros as necessary, i.e. 00 - 99.
'j'	'\u006a'	Day of year, formatted as three digits with leading zeros as necessary, e.g. 001 - 366 for the Gregorian calendar. 001 corresponds to the first day of the year.
'm'	'\u006d'	Month, formatted as two digits with leading zeros as necessary, i.e. 01 - 13, where "01" is the first month of the year and ("13" is a special value required to support lunar calendars).
'd'	'\u0064'	Day of month, formatted as two digits with leading zeros as necessary, i.e. 01 - 31, where "01" is the first day of the month.
'e'	'\u0065'	Day of month, formatted as two digits, i.e. 1 - 31 where "1" is the first day of the month.

The following conversion characters are used for formatting common date/time compositions.

'R'	'\u0052'	Time formatted for the 24-hour clock as "%tH:%tM"
'T'	'\u0054'	Time formatted for the 24-hour clock as "%tH:%tM:%tS".
'r'	'\u0072'	Time formatted for the 12-hour clock as "%tI:%tM:%tS %Tp". The location of the morning or afternoon marker ('%Tp') may be locale-dependent.
'D'	'\u0044'	Date formatted as "%tm/%td/%ty".
'F'	'\u0046'	ISO 8601 complete date formatted as "%tY-%tm-%td".
'c'	'\u0063'	Date and time formatted as "%ta %tb %td %tT %tZ %tY", e.g. "Sun Jul 20 16:17:00 EDT 1969".

The '-' flag defined for General conversions applies. If the '#' flag is given, then a FormatFlagsConversionMismatchException will be thrown.

The width is the minimum number of characters to be written to the output. If the length of the converted value is less than the width then the output will be padded by spaces ('\u0020') until the total number of characters equals width. The padding is on the left by default. If the '-' flag is given then the padding will be on the right. If width is not specified then there is no minimum.

The precision is not applicable. If the precision is specified then an IllegalFormatPrecisionException will be thrown.

Percent

The conversion does not correspond to any argument.

'%'	The result is a literal '%' ('\u0025')

The width is the minimum number of characters to be written to the output including the '%'. If the length of the converted value is less than the width then the output will be padded by spaces

('\u0020') until the total number of characters equals width. The padding is on the left. If width is not specified then just the '%' is output.

The '-' flag defined for General conversions applies. If any other flags are provided, then a FormatFlagsConversionMismatchException will be thrown.

The precision is not applicable. If the precision is specified an IllegalFormatPrecisionException will be thrown.

Line Separator

The conversion does not correspond to any argument.

'n' the platform-specific line separator as returned by System.getProperty("line.separator").

Flags, width, and precision are not applicable. If any are provided an IllegalFormatFlagsException, IllegalFormatWidthException, and IllegalFormatPrecisionException, respectively will be thrown.

Argument Index

Format specifiers can reference arguments in three ways:

- *Explicit indexing* is used when the format specifier contains an argument index. The argument index is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "1$", the second by "2$", etc. An argument may be referenced more than once.

  For example:

```
formatter.format("%4$s %3$s %2$s %1$s %4$s %3$s %2$s %1$s",
                 "a", "b", "c", "d")
// -> "d c b a d c b a"
```

- *Relative indexing* is used when the format specifier contains a '<' ('\u003c') flag which causes the argument for the previous format specifier to be re-used. If there is no previous argument, then a MissingFormatArgumentException is thrown.

```
formatter.format("%s %s %<s %<s", "a", "b", "c", "d")
// -> "a b b b"
// "c" and "d" are ignored because they are not referenced
```

- *Ordinary indexing* is used when the format specifier contains neither an argument index nor a '<' flag. Each format specifier which uses ordinary indexing is assigned a sequential implicit index into argument list which is independent of the indices used by explicit or relative indexing.

```
formatter.format("%s %s %s %s", "a", "b", "c", "d")
// -> "a b c d"
```

It is possible to have a format string which uses all forms of indexing, for example:

```
formatter.format("%2$s %s %<s %s", "a", "b", "c", "d")
// -> "b a a b"
// "c" and "d" are ignored because they are not referenced
```

The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the Java Virtual Machine Specification. If the argument index is does not correspond to an available argument, then a `MissingFormatArgumentException` is thrown.

If there are more arguments than format specifiers, the extra arguments are ignored.

Unless otherwise specified, passing a `null` argument to any method or constructor in this class will cause a `NullPointerException` to be thrown.

Since:

1.5

Nested Class Summary

static class	**Formatter.BigDecimalLayoutForm**

Constructor Summary

Formatter() Constructs a new formatter.
Formatter(Appendable a) Constructs a new formatter with the specified destination.
Formatter(Appendable a, Locale l) Constructs a new formatter with the specified destination and locale.
Formatter(File file) Constructs a new formatter with the specified file.
Formatter(File file, String csn) Constructs a new formatter with the specified file and charset.
Formatter(File file, String csn, Locale l) Constructs a new formatter with the specified file, charset, and locale.
Formatter(Locale l) Constructs a new formatter with the specified locale.
Formatter(OutputStream os) Constructs a new formatter with the specified output stream.
Formatter(OutputStream os, String csn) Constructs a new formatter with the specified output stream and charset.
Formatter(OutputStream os, String csn, Locale l) Constructs a new formatter with the specified output stream, charset, and locale.

Formatter(PrintStream ps) Constructs a new formatter with the specified print stream.
Formatter(String fileName) Constructs a new formatter with the specified file name.
Formatter(String fileName, String csn) Constructs a new formatter with the specified file name and charset.
Formatter(String fileName, String csn, Locale l) Constructs a new formatter with the specified file name, charset, and locale.

Method Summary

void	**close**() Closes this formatter.
void	**flush**() Flushes this formatter.
Formatter	**format**(Locale l, String format, Object... args) Writes a formatted string to this object's destination using the specified locale, format string, and arguments.
Formatter	**format**(String format, Object... args) Writes a formatted string to this object's destination using the specified format string and arguments.
IOException	**ioException**() Returns the IOException last thrown by this formatter's Appendable.
Locale	**locale**() Returns the locale set by the construction of this formatter.
Appendable	**out**() Returns the destination for the output.
String	**toString**() Returns the result of invoking toString() on the destination for the output.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Formatter

```
public Formatter()
```

Constructs a new formatter.

The destination of the formatted output is a StringBuilder which may be retrieved by invoking out() and whose current content may be converted into a string by invoking toString(). The

locale used is the default locale for this instance of the Java virtual machine.

Formatter

```
public Formatter(Appendable a)
```

Constructs a new formatter with the specified destination.

The locale used is the default locale for this instance of the Java virtual machine.

Parameters:

`a` - Destination for the formatted output. If `a` is `null` then a `StringBuilder` will be created.

Formatter

```
public Formatter(Locale l)
```

Constructs a new formatter with the specified locale.

The destination of the formatted output is a `StringBuilder` which may be retrieved by invoking `out()` and whose current content may be converted into a string by invoking `toString()`.

Parameters:

`l` - The locale to apply during formatting. If `l` is `null` then no localization is applied.

Formatter

```
public Formatter(Appendable a,
                 Locale l)
```

Constructs a new formatter with the specified destination and locale.

Parameters:

`a` - Destination for the formatted output. If `a` is `null` then a `StringBuilder` will be created.
`l` - The locale to apply during formatting. If `l` is `null` then no localization is applied.

Formatter

```
public Formatter(String fileName)
          throws FileNotFoundException
```

Constructs a new formatter with the specified file name.

The charset used is the default charset for this instance of the Java virtual machine.

The locale used is the default locale for this instance of the Java virtual machine.

Parameters:

`fileName` - The name of the file to use as the destination of this formatter. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:

`SecurityException` - If a security manager is present and `checkWrite(fileName)` denies write access to the file

`FileNotFoundException` - If the given file name does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

---

Formatter

```
public Formatter(String fileName,
                 String csn)
          throws FileNotFoundException,
                 UnsupportedEncodingException
```

Constructs a new formatter with the specified file name and charset.

The locale used is the default locale for this instance of the Java virtual machine.

Parameters:

`fileName` - The name of the file to use as the destination of this formatter. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

`csn` - The name of a supported charset

Throws:

`FileNotFoundException` - If the given file name does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

`SecurityException` - If a security manager is present and `checkWrite(fileName)` denies write access to the file

`UnsupportedEncodingException` - If the named charset is not supported

Formatter

```
public Formatter(String fileName,
                 String csn,
                 Locale l)
          throws FileNotFoundException,
                 UnsupportedEncodingException
```

Constructs a new formatter with the specified file name, charset, and locale.

Parameters:

`fileName` - The name of the file to use as the destination of this formatter. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

`csn` - The name of a supported charset

`l` - The locale to apply during formatting. If `l` is `null` then no localization is applied.

Throws:

FileNotFoundException - If the given file name does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

SecurityException - If a security manager is present and checkWrite(fileName) denies write access to the file

UnsupportedEncodingException - If the named charset is not supported

---

Formatter

```
public Formatter(File file)
          throws FileNotFoundException
```

Constructs a new formatter with the specified file.

The charset used is the default charset for this instance of the Java virtual machine.

The locale used is the default locale for this instance of the Java virtual machine.

Parameters:

file - The file to use as the destination of this formatter. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

Throws:

SecurityException - If a security manager is present and checkWrite(file.getPath()) denies write access to the file

FileNotFoundException - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

---

Formatter

```
public Formatter(File file,
                 String csn)
          throws FileNotFoundException,
                 UnsupportedEncodingException
```

Constructs a new formatter with the specified file and charset.

The locale used is the default locale for this instance of the Java virtual machine.

Parameters:

file - The file to use as the destination of this formatter. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.

csn - The name of a supported charset

Throws:

FileNotFoundException - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

SecurityException - If a security manager is present and checkWrite(file.getPath()) denies write access to the file
UnsupportedEncodingException - If the named charset is not supported

---

Formatter

```
public Formatter(File file,
                 String csn,
                 Locale l)
          throws FileNotFoundException,
                 UnsupportedEncodingException
```

Constructs a new formatter with the specified file, charset, and locale.

Parameters:

file - The file to use as the destination of this formatter. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.
csn - The name of a supported charset
l - The locale to apply during formatting. If l is null then no localization is applied.

Throws:

FileNotFoundException - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file
SecurityException - If a security manager is present and checkWrite(file.getPath()) denies write access to the file
UnsupportedEncodingException - If the named charset is not supported

---

Formatter

```
public Formatter(PrintStream ps)
```

Constructs a new formatter with the specified print stream.

The locale used is the default locale for this instance of the Java virtual machine.

Characters are written to the given PrintStream object and are therefore encoded using that object's charset.

Parameters:

ps - The stream to use as the destination of this formatter.

Formatter

```
public Formatter(OutputStream os)
```

Constructs a new formatter with the specified output stream.

The charset used is the default charset for this instance of the Java virtual machine.

The locale used is the default locale for this instance of the Java virtual machine.

Parameters:

`os` - The output stream to use as the destination of this formatter. The output will be buffered.

Formatter

```
public Formatter(OutputStream os,
                 String csn)
          throws UnsupportedEncodingException
```

Constructs a new formatter with the specified output stream and charset.

The locale used is the default locale for this instance of the Java virtual machine.

Parameters:

`os` - The output stream to use as the destination of this formatter. The output will be buffered.

`csn` - The name of a supported charset

Throws:

`UnsupportedEncodingException` - If the named charset is not supported

Formatter

```
public Formatter(OutputStream os,
                 String csn,
                 Locale l)
          throws UnsupportedEncodingException
```

Constructs a new formatter with the specified output stream, charset, and locale.

Parameters:

`os` - The output stream to use as the destination of this formatter. The output will be buffered.

`csn` - The name of a supported charset

`l` - The locale to apply during formatting. If `l` is `null` then no localization is applied.

Throws:

`UnsupportedEncodingException` - If the named charset is not supported

Method Detail

locale

```
public Locale locale()
```

Returns the locale set by the construction of this formatter.

The `format` method for this object which has a locale argument does not change this value.

Returns:

`null` if no localization is applied, otherwise a locale

Throws:
FormatterClosedException - If this formatter has been closed by invoking its close() method

---

out

```
public Appendable out()
```

Returns the destination for the output.

Returns:
The destination for the output
Throws:
FormatterClosedException - If this formatter has been closed by invoking its close() method

---

toString

```
public String toString()
```

Returns the result of invoking `toString()` on the destination for the output. For example, the following code formats text into a StringBuilder then retrieves the resultant string:

```
Formatter f = new Formatter();
f.format("Last reboot at %tc", lastRebootDate);
String s = f.toString();
// -> s == "Last reboot at Sat Jan 01 00:00:00 PST 2000"
```

An invocation of this method behaves in exactly the same way as the invocation

```
out().toString()
```

Depending on the specification of `toString` for the Appendable, the returned string may or may not contain the characters written to the destination. For instance, buffers typically return their contents in `toString()`, but streams cannot since the data is discarded.

Overrides:
toString in class Object

Returns:
The result of invoking `toString()` on the destination for the output
Throws:
FormatterClosedException - If this formatter has been closed by invoking its close() method

---

flush

```
public void flush()
```

Flushes this formatter. If the destination implements the Flushable interface, its `flush` method will be invoked.

Flushing a formatter writes any buffered output in the destination to the underlying stream.

Specified by:
`flush` in interface `Flushable`

Throws:
`FormatterClosedException` - If this formatter has been closed by invoking its `close()` method

---

close

```
public void close()
```

Closes this formatter. If the destination implements the `Closeable` interface, its `close` method will be invoked.

Closing a formatter allows it to release resources it may be holding (such as open files). If the formatter is already closed, then invoking this method has no effect.

Attempting to invoke any methods except `ioException()` in this formatter after it has been closed will result in a `FormatterClosedException`.

Specified by:
`close` in interface `Closeable`

---

ioException

```
public IOException ioException()
```

Returns the `IOException` last thrown by this formatter's `Appendable`.

If the destination's `append()` method never throws `IOException`, then this method will always return `null`.

Returns:
The last exception thrown by the Appendable or `null` if no such exception exists.

---

format

```
public Formatter format(String format,
                        Object... args)
```

Writes a formatted string to this object's destination using the specified format string and arguments. The locale used is the one defined during the construction of this formatter.

Parameters:
`format` - A format string as described in Format string syntax.
`args` - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The maximum number of

arguments is limited by the maximum dimension of a Java array as defined by the Java Virtual Machine Specification.

Returns:

This formatter

Throws:

`IllegalFormatException` - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the Details section of the formatter class specification.

`FormatterClosedException` - If this formatter has been closed by invoking its `close()` method

---

format

```
public Formatter format(Locale l,
                        String format,
                        Object... args)
```

Writes a formatted string to this object's destination using the specified locale, format string, and arguments.

Parameters:

`l` - The locale to apply during formatting. If `l` is `null` then no localization is applied. This does not change this object's locale that was set during construction.

`format` - A format string as described in Format string syntax

`args` - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by the Java Virtual Machine Specification

Returns:

This formatter

Throws:

`IllegalFormatException` - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the Details section of the formatter class specification.

`FormatterClosedException` - If this formatter has been closed by invoking its `close()` method

---