Complete Printf for Java Format String Specification

Lava's `printf` classes have been designed to offer Java programmers the same text- and data-formatting features that C programmers have enjoyed. Printf allows the programmer to specify textual representations of data using two parameters:

1. a **format string**, or "template," and
2. an **argument list**, or a vector of data to be formatted according to the template.

In C, printf output is specified using a "tuple" of arbitrary length,

> (format_string, data1, data2, data3, ...),

where the first parameter defines the format, and all additional parameters are the objects to be formatted.

Unfortunately, Java does not directly support variable argument lists. Instead, the Lava's printf classes simply accept a *pair* of arguments,

> (format_string, argument_vector),

where the first parameter defines the format (as before), and the second parameter is a *vector* of elements, specified as an arbitrary length `Object` array.

The format string is composed of *literal substrings* and *format specifiers*. When output is produced, literal substrings are copied verbatim, but format specifiers are replaced with string representations of elements in the argument list. The conversion of argument list elements into strings is governed by parameters embedded in the format specifiers. These parameters affect details such as

- the number of digits beyond the decimal point (for real numbers),
- the left- or right-alignment of text within a field,
- whether to pad short numbers with spaces or zeros, etc.

It is the versatility of these parameters that make printf formatting so powerful.

Lava's printf classes faithfully reproduce all of the formatting features available in the C version of printf, but also provide additional functionality not available in the C version. Lava extensions to standard C printf are indicated in **blue font**.

1. format specifiers

The general syntax of a format specifier is

> `%`[*flags*][*width*][*.precision*][*argsize*]*typechar*

Format specifiers begin with a percent character (%) and terminate with a "type character," which indicates the type of data (`int`, `float`, etc.) that will be converted the basic manner in which the data will be represented (decimal, hexadecimal, etc.) For example, the format string

```
"I_can_lift_%f_tons."
```

indicates that the formatted output will be the concatenation of the substring "`I_can_lift_`", a string representation of a `float` value, and the final substring "`_tons.`" The corresponding argument vector for this format string would contain one number, which would be interpreted as the `float`.

Depending on the conversion type, certain flags and formatting constraints can be inserted *between* the percent character and the type character. For example, changing the "`%f`" to "`%.3f`" in the above format string indicates that the `float` should be formatted with a precision of three decimal places after the decimal point.

The components of a format specifier are defined in the following sections.

1.1. conversion type character

typechar is a single character identifying the conversion type. The supported conversion types are listed below, along with their meanings, and the corresponding arguments expected in the argument vector:

type character	input	string result
`%d`	signed `int`	signed decimal integer
`%u`	unsigned `int`	unsigned decimal integer
`%o`	unsigned `int`	unsigned octal integer
`%x`, `%X`	unsigned `int`	unsigned hexadecimal integer, lowercase or uppercase
`%z[`*n*`]`, `%Z[`*n*`]`	unsigned `int`	unsigned integer base *n*, with *n* coded in decimal; include square brackets
`%f`	`float`	real number, standard notation

`%e`, `%E`	`float`	real number, scientific notation (lowercase or uppercase exponent marker)
`%g`, `%G`	`float`	same format as `%f` or `%e`, depending on the value. Scientific notation is used only if the exponent is greater than the precision or less than -4.
`%s`	`String`	string
`%c`	`char`	character
`%p`	`Object`	object identity hash code (i.e., pointer value), in unsigned hexadecimal
additional format specifiers that do not result in argument conversions		
`%\n`	*(none)*	platform-independent line separator (see §3)
`%n`	*(null)*	counts characters (see §4)

The conversion type not only indicates the type of string generated from the input, but also the type of input required.

Example: The format specifier "`%o`" means that an `int` argument will be converted to an unsigned octal string.

1.2. input size modifier

Each conversion type character indicates that an argument of a specific type is supplied. Failure to supply the correct data type for a format specifier results in an error, usually a `ClassCastException`. It is possible, however, to change the data type required for certain format specifiers. This is done by preceding the type character with an "input size modifier."

The following table lists the allowed input size modifiers:

default input size	input size modifier	actual input size used
`int`	*(none)*	`int`
`int`	`h`	`short`
`int`	`b`	`byte`
`int`	`B`	`BigInteger`
`int`	`l`	`long`
`float`	*(none)*	`float`
`float`	`l`	`double`
`float`	`B`	`BigDecimal`

Example: The format specifier "`%d`" menas that an `int` will be treated as a 32-bit signed decimal number. However, the format specifier "`%hd`" means that a `short` will be converted instead.

1.2.1. BigInteger formatting

The `%u`, `%x`, and `%o` format specifiers typically represent *unsigned* integer conversions. This is possible with primitive data types, such as `int`, because the sizes of the types are fixed and known, allowing every bit to be treated as a magnitude bit. `BigIntegers`, however, represent "infinitely long" bit patterns, so it is not possible to treat them as unsigned values. Because of this, the format specifiers `%Bx`, `%Bo`, and `%Bu` will produce *signed* results. (Note that it makes little sense to use `%Bu`, since it is the same as `%Bd`.)

1.3. width specifier

An optional "width specifier," if present, indicates the field with, or the minimum number of characters in the output that the formatted argument will span. If the string representation of the value does not fill the minimum length, the field will be left-padded with spaces. If the converted value *exceeds* the minimum length, however, the converted result will not be truncated.

Example: If the format specifier is "`%6d`", and the supplied `int` argument is 52, then the output will be "`    52`" (four spaces on the left).

If no width is specified, then there is no minimum size for the converted result. The field will be only as large as necessary to display the result.

1.3.1. variable width

It is possible to specify a *variable* field width, where one of the arguments in the argument vector is used to determine the field's width. This is indicated in the format string by using an asterisk character ('`*`') in place of the actual width value. The corresponding value in the argument list must be an `int`, and must precede the actual argument being formatted.

Example: If the format specifier is "`%*hd`", then there are two arguments in the argument list corresponding to this format specifier. The first argument is an `int`, indicating the field width, and the second argument is converted as a signed `short`.

1.4. precision specifier

An optional "precision specifier" may be included in a format specifier to indicate the precision with which to convert the data. The meaning of the word "precision" depends on the type of conversion being performed:

conversion type	meaning of "precision"	default value (if omitted)
`%d`, `%o`, `%u`, `%x`, `%X`, `%z[n]`, `%Z[n]` (integer conversions)	minimum number of digits. The converted value will be prepended with zeros if necessary. Note that if the precision is 0 and the value is zero, nothing will be printed, or the entire field will be nothing but padding.	1
`%f`, `%e`, `%E`, `%g`, `%G` (real conversions)	number of fractional digits after the decimal point. The converted value will be rounded if necessary.	6
`%s` (strings)	the maximum number of characters. If the string is too long, it will be truncated.	infinity

If no precision is specified, the default precision will be used. Precision cannot be specified for conversion types other than those listed above.

1.4.1. variable precision

Like the width field, the precision field may also be made *variable* replacing the number with an asterisk. The expected argument is an int. (See the section on "variable width" for details.) If both variable width and variable precision are used in the same format specifier, the width argument appears first in the argument list.

1.5. flags

Flags are single characters that indicate exceptions to the conversion type's default behavior. A format specifier may have multiple flags, but some flags are mutually exclusive. Multiple flags can appear in any order. The following table lists the formatting flags supported by Lava's printf classes:

flag	meaning	conversion types applicable
`'-'`	Result is left-aligned in the field. This flag is meaningless if no mandatory field width is specified.	`%d`, `%u`, `%o`, `%x`, `%X`, `%z[n]`, `%Z[n]`, `%f`, `%e`, `%E`, `%g`, `%G`, `%s`, `%c`, `%p`
`'^'`	Result is centered in the field. This flag is meaningless if no mandatory field width is specified.	
`'+'`	Non-negative values begin with a plus character ('+').	`%d`, `%f`
`' '`	Non-negative values begin with a space character (' '). This flag is only useful for signed conversion results (%d and %f).	
`'#'`	Argument is represented in an "alternate form." This depends on the conversion type: `%o` — Non-negative octal values are prepended with a zero ('0'). `%x`, `%X` — Hexadecimal values are prepended with the prefix "0x" or "0X". `%e`, `%E`, `%f` — The integer portion of the result always ends with a decimal point ('.'), even if the fractional portion is zero. `%g`, `%G` — The fractional portion always appears, even if it is zero. `%c` — If the character is special or unprintable, it is output in an escaped form. The output can be surrounded by single quotes to form a syntactically valid Java character literal. There is no alternate form for `%s`, `%d`, `%u`, `%z[n]`, and `%Z[n]`.	

2. argument vectors

An argument vector is a list of data elements which will be converted into strings and inserted in the printf output, according to formatting criteria specified in the format string. Each format specifier in the format string requires a specific number and type of corresponding arguments in the list. Because Java does allow methods with variable-length or variable-type argument lists, all of the arguments must be encapsulated in a generic `Object` array, which is then passed as a single parameter.

Unfortunately, `Object` arrays cannot hold primitive data types, such as `int` or `long`. Thus, to include primitives in an argument list, `Object` encapsulations must be used instead. The following table lists the input types required by the format specifiers, and the classes that should be used to include them in the argument list:

input type	recommended object type	required object type	method called to obtain input
`byte`	`java.lang.Byte`	`java.lang.Number`	`byteValue()`
`short`	`java.lang.Short`	`java.lang.Number`	`shortValue()`
`int`	`java.lang.Integer`	`java.lang.Number`	`intValue()`
`long`	`java.lang.Long`	`java.lang.Number`	`longValue()`
`float`	`java.lang.Float`	`java.lang.Number`	`floatValue()`
`double`	`java.lang.Double`	`java.lang.Number`	`doubleValue()`
`char`	`java.lang.Character`	`java.lang.Character`	`charValue()`
`java.lang.BigInteger`	`java.lang.BigInteger`	`java.lang.BigInteger`	n/a

`java.lang.BigDecimal`	`java.lang.BigDecimal`	`java.lang.BigDecimal`	n/a
`java.lang.String`	`java.lang.String`	`java.lang.Object`	`toString()`

Example: If the format specifier is "`%f`", then the corresponding argument in the `Object` array must be a `Float` object.

Example: If the format string is

```
"My friend %s is %d years old."
```

then the corresponding argument list must consist of a `String` and an `int`. Assuming that `name` is a `String` and `age` is an `int`, then the corresponding argument list might be created using the expression

```
new Object [] {name, new Integer (age)}
```

In this expression, the first element in the `Object` array is a `String`, which corresponds to the first format specifier, "`%s`". The second element in the array is an `Integer`, which corresponds to the second format specifier, "`%d`".

2.1. abstract numerical types

Despite the recommendations given in the above section regarding primitive type encapsulation, in reality it is legal to use any instance of `java.lang.Number` wherever the input types

`int`, `long`, `short`, `byte`, `float`, or `double`

are expected. This is because the formatting engine will correctly call the appropriate abstract method,

`intValue()`, `longValue()`, `shortValue()`, `byteValue()`, `floatValue()`, or `doubleValue()`,

respectively, to obtain the necessary data. This behavior provides a measure of flexibility to be enjoyed by advanced programmers, including the ability to invent and format one's own numerical data types.

2.2. automatic string conversion

Wherever an instance of `String` is needed in the argument list, it is actually safe to use *any* object type. This is because the formatting engine will automatically call the object's `toString()` method (which, for a `String` object, simply returns a self-reference). This makes it easy to format string representations of all types of objects. It also makes it legal to supply a `StringBuffer`, rather than a `String`, as the corresponding paramter for `%s`.

3. platform-independent line separators

In situations where Java software will be run on more than one platform, it is wise to avoid using system-dependent control characters, such as '\n'. Lava's printf classes support the notion of a "platform-independent line separator," represented in format strings by the "`%\n`" format specifier. Wherever `%\n` occurs in a format string, the printf classes will automatically insert the host system's native line separator.

Note that if cross-platform compatibility is not an issue, and the software will run primarily on Unix- and Windows-based hosts, using the traditional newline character '`\n`' will probably produce acceptable results.

4. character counting

A special format specifier not yet mentioned, "`%n`", does not require any data from the argument list, nor does it produce any output. Instead, when this specifier appears in the format string, the number of characters formatted up to the point where it appears is computed, and the resulting `Integer` value is written into the supplied argument vector. Thus, when building the argument list for a format string containing this specifier, an extra slot should be reserved for the result. This is the only instance in which the argument vector will be modified by a printf call.

Example: This program prints the string "`My friend Sharky is 27 years old.`" It uses `%n` to obtain character positions so that it can underline the number 27.

```
// These are the parameters:
String name = "Sharky";
int age = 27;

// This is the argument vector.  Observe that we're leaving
// slots for the %n format specifier to place its results:
Object[] args = new Object[] {name, null, new Integer (age), null};

// Now we'll produce the first line of output:
Stdio.printf ("My friend %s is %n%d%n years old.\n", args);

// Next, we'll use the results from %n to form an underline string:
int start = ((Integer) args [1]) . intValue ();
int stop  = ((Integer) args [3]) . intValue ();
String underline_string =
        StringToolbox.repeat (' ', start       ) +
        StringToolbox.repeat ('-', stop - start);
```

```
// Lastly, we print the underline string.  Notice that, unlike the
// above formatting operation, this time we form the argument vector
// inline.  This is because we won't need to access its contents after
// we've used it, as we did above:
Stdio.printf ("%s\n", new Object[] {underline_string});
```

Results:

```
My friend Sharky is 27 years old.
                    --
```

5. precompiled format strings

Whenever printf argument formatting is performed, the format string must be interpreted to determine what kind of output to produce. For one-shot deals, this is not a great concern. In many situations, however, printf formatting takes place within a loop, and it is a great waste of time to interpret the same format string over and over again.

To increase the efficiency of printf formatting within a loop, Lava supports "precompiled format strings." This makes it possible to interpret a format string just once, and save the interpreted results for use in multiple printf method calls. Many programmers have found it convenient to store the precompiled format strings as static members of the classes in which they are used. For more details, see the `PrintfFormatString` class.

6. additional resources

The following classes and methods support the printf specification described in this document:

- class lava.clib.stdio.Printf
- class lava.clib.stdio.PrintfFormatString
- class lava.clib.stdio.PrintfFormatException
- methods lava.clib.Stdio.printf
- methods lava.clib.Stdio.fprintf
- methods lava.clib.Stdio.sprintf

The following documents contain additional information about Printf for Java:

- Introduction to Printf for Java
- Optimizing Printf for Java