Vala String Sample

```
void println (string str) {
    stdout.printf ("%s\n", str);
}

void main () {

    /* Strings are of data type 'string' and can be
concatenated with the plus
     * operator resulting in a new string:
     */

    string a = "Concatenated ";
    string b = "string";
    string c = a + b;
    println (c);

    /* If you want to have a mutable string you should use
StringBuilder.
     * With its help you are able to build strings ad
libitum by prepending,
     * appending, inserting or removing parts. It's faster
than multiple
     * concatenations. In order to obtain the final product
you access the
     * field '.str'.
     */

    var builder = new StringBuilder ();
    builder.append ("built ");
```

```
   builder.prepend ("String ");
   builder.append ("StringBuilder");
   builder.append_unichar ('.');
   builder.insert (13, "by ");
   println (builder.str);       // => "String built by
StringBuilder."

   /* You can create a new string according to a format
string by calling the
    * method 'printf' on it. Format strings follow the
usual rules, known from
    * C and similar programming languages.
    */

   string formatted = "PI %s equals %g.".printf
("approximately", Math.PI);
   println (formatted);

   /* Strings prefixed with '@' are string templates. They
can evaluate
    * embedded variables and expressions prefixed with '$'.
    * Since Vala 0.7.8.
    */

   string name = "Dave";
   println (@"Good morning, $name!");
   println (@"4 + 3 = $(4 + 3)");

   /* The equality operator compares the content of two
strings, contrary to
    * Java's behaviour which in this case would check for
referential equality.
    */
   a = "foo";
   b = "foo";
   if (a == b) {
      println ("String == operator compares content, not 
reference.");
   } else {
      assert_not_reached ();
   }

   /* You can compare strings lexicographically with the < 
and > operators: */

   if ("blue" < "red" && "orange" > "green") {
      // That's correct
   }

   // Switch statement

   string pl = "vala";
   switch (pl) {
   case "java":
      assert_not_reached ();
   case "vala":
      println ("Switch statement works fine with 
strings.");
      break;
   case "ruby":
      assert_not_reached ();
   }

   /* Vala offers a feature called verbatim strings. These 
are strings in
```

```
     * which escape sequences (such as \n) won't be
interpreted, line breaks
     * will be preserved and quotation marks don't have to
be masked. They are
     * enclosed with triple double quotation marks. Possible
indentations
     * after a line break are part of the string as well.
Note that syntax
     * highlighting in this Wiki is not aware of verbatim
strings.
     */

   string verbatim = """This is a so-called "verbatim
string".
Verbatim strings don't process escape sequences, such as \
n, \t, \\, etc.
They may contain quotes and may span multiple lines.""";
   println (verbatim);

   /* You can apply various operations on strings. Here's a
small selection: */

   println ("from lower case to upper case".up ());
   println ("reversed string".reverse ());
   println ("...substring...".substring (3, 9));

   /* The 'in' keyword is syntactic sugar for checking if
one string contains
     * another string. The following expression is identical
to
     * "swordfish".contains ("word")
     */
```

```
    if ("word" in "swordfish") {
        // ...
    }

    // Regular expressions

    try {
        var regex = new Regex ("(jaguar|tiger|leopard)");
        string animals = "wolf, tiger, eagle, jaguar, 
leopard, bear";
        println (regex.replace (animals, -1, 0, "kitty"));
    } catch (RegexError e) {
        warning ("%s", e.message);
    }
}
```

Compile and Run

```
$ valac stringsample.vala
$ ./stringsample
```

String Length

Before Vala 0.11 .length returned the number of Unicode code points of a string and .size() returned the length in bytes:

```
string dessert = "crème brûlée";
assert (dessert.length == 12);
assert (dessert.size () == 15);
```

Starting with Vala 0.11 .length returns the length in bytes and index access is byte-based as well:

```
string dessert = "crème brûlée";
assert (dessert.length == 15);
uint8 a_byte = dessert[3];
assert (a_byte == 0xA8);
```

Note that strings in Vala are still UTF-8, nothing changed in that regard. The difference is that indices/offsets/lengths are measured in code units (bytes as we are talking about UTF-8) instead of code points.

This is in line with string APIs in many libraries (Java, .NET, Qt, Go) and makes it possible to improve performance in string handling without resorting to pointers all the time while still properly supporting Unicode strings.

The unit of a Unicode code point is not very meaningful in most contexts as it's not the same as a glyph. See this paper for the difference between Unicode units, code points and glyphs.

C# and Java developers should pay attention, because Java's String.length and .NET's String.Length counts the UTF-16 code units, each taking 2 bytes, while Vala counts the UTF-8 code units, each taking 1 bytes. For example string "𐌉𐌕 𐌋𐌏𐌏𐌊𐌔 𐌋𐌉𐌊𐌄 𐌓𐌖𐌍𐌄𐌔" contains 20 code points (what most programmers call 'characters') while is has length 35 in .Net/Java and 67 in Vala, taking 70 and 67 bytes respectively. As most used code points take 1 UTF-16 code unit it leads to misconception that C# and Java strings count 'characters'.

UTF-8 Characters Iteration

UTF-8 is variable-length encoding, so characters can not be directly accessed via their indexes. To avoid unnecessary overhead in character iteration, it is better not to code like this:

```
string str = "吃饭了";
for (int i = 0; i < str.char_count (); i++) {
   str.get_char (str.index_of_nth_char (i));
}
```

This is also inefficient:

```
for (var i = str; i.get_char () != 0; i = i.next_char ())
{
   stdout.printf ("%s\n", i.get_char ().to_string ());
}
/*
```

```
get_char() returns the first character when given no
parameter or 0.
But next_char() actually creates a new string by
duplication without the first character.
*/
```

You can use the new API added in Vala 0.12.0 like this:

```
unichar c;
for (int i = 0; str.get_next_char (ref i, out c);) {
   stdout.printf ("%d, %s\n", i, c.to_string ());
}
/*
get_next_char will start the iteration from offset (not
index) i of str.
It outputs the current character to c and writes the
offset of the next character to i repeatedly.
*/
```

Vala/Examples