Optimizations

1. Optimizations

Automatically generated code tends to become unnecessarily complex and unreadable. This page summarizes some optimization concepts offered by JaxMeJS and possible strategies to use them.

2. The DirectAccessible interface

In general JaxMeJS deals with objects, in the most general sense. An object can be almost everything: An array or collection containing a piece of code, a variable, method, or class name or whatever else. While this is fine for the most situations, it can become difficult in the following example:

```
    Object value;
    return new Object[]{"((", value, ") * (", value, "))"};
```

The example computes the square of the given "value". This example is obviously fine, if "value" contains a simple variable name or something similar. But how about that:

```
    value = "Math.sin(x)";
```

The computation of a sine can become relatively expensive, in particular, if the same computation is performed twice. However, at least the example works, which is no longer the case here:

```
    value = "i++";
```

Filling this value in the above example would yield

```
    ((i++) * (i++))
```

This is simply wrong, because the variable "i" is incremented twice. The obvious solution is the use of a local Java field, as in the following example:

```
    Object value;
    JavaQName type;
```

```
    LocalJavaField v = method.newJavaField(type);
    v.setFinal(true);
    v.addLine(value);
    return new Object[]{"((", v, ") * (", v, "))"};
```

This is obviously better, as it handles both the performance problem caused by the sine computation and the logical problem caused by the increment. However, if you start using such local fields, your code will soon start to look like the following:

```
    String val;
    String _1 = val;
    String _2 = _1;
```

Again, this is something we do not actually like to see. JaxMeJS suggests the use of the DirectAccessible interface. This is an interface implemented by objects, which are accessible immediately and fast: Class and instance fields, method parameters, and local fields. This allows the following final example:

```
    Object value;
    JavaQName type;
    if (!(value instanceof DirectAccessible)) {
      LocalJavaField v = method.newJavaField(type);
      v.setFinal(true);
      v.addLine(value);
      value = v;
    }
    return new Object[]{"((", value, ") * (", value, "))"};
```

3. Avoiding unnecessary checks for null

The DirectAccessible interface can also be used to avoid situations like this:

```
public void doThis(JavaMethod pMethod, Object pValue) {
  pMethod.addIf(pValue, " == null");
  pMethod.addThrowNew(NullPointerException.class, "value must not be null");
  pMethod.addEndIf();
  addLine("// Do this here ...");
  doThat(pMethod, pValue);
}
public void doThat(JavaMethod pMethod, Object pValue) {
  pMethod.addIf(pValue, " == null");
  pMethod.addThrowNew(NullPointerException.class, "value must not be null");
  pMethod.addEndIf();
  addLine("// Do that here ...");
}
```

If we invoke the method `doThis(method, someVariable)`, it is creates the following

code:

```
if (someVariable == null) {
  throw new NullPointerException("value must not be null");
}
// Do this here ...
if (someVariable == null) {
  throw new NullPointerException("value must not be null");
}
// Do that here ...
```

A better approach would be:

```
public void doThis(JavaMethod pMethod, Object pValue, JavaQName pType) {
  DirectAccessible value;
  if (pValue instanceof DirectAccessible) {
    value = pValue;
  } else {
    LocalJavaField foo = pMethod.newJavaField(pType);
    foo.addLine(pValue);
    value = foo;
  }
  if (value.isNullable()) {
    pMethod.addIf(pValue, " == null");
    pMethod.addThrowNew(NullPointerException.class, "value must not be null");
    pMethod.addEndIf();
    value.setNullable(false);
  }
  addLine("// Do this here ...");
  doThat(pMethod, pValue, pType);
}
public void doThat(JavaMethod pMethod, Object pValue, JavaQName pType) {
  DirectAccessible value;
  if (pValue instanceof DirectAccessible) {
    value = pValue;
  } else {
    LocalJavaField foo = pMethod.newJavaField(pType);
    foo.addLine(pValue);
    value = foo;
  }
  if (value.isNullable()) {
    pMethod.addIf(pValue, " == null");
    pMethod.addThrowNew(NullPointerException.class, "value must not be null");
    pMethod.addEndIf();
    value.setNullable(false);
  }
  addLine("// Do that here ...");
}
```

In the worst case, this will create the following code:

```
if (someVariable == null) {
  throw new NullPointerException("value must not be null");
}
// Do this here ...
// Do that here ...
```

Note, that loop variables, as generated by `addForArray()`, `addForIterator`, or `addForList`, will never be nullable. In other words, the following code sequence will emit no checks for null at all:

```
DirectAccessible loopVar = pMethod.addForList(list);
pMethod.doThis();
pMethod.addEndFor();
```