JMultiComparibale

The template class JMultiComparibale resides in the name space JLANG and constitutes an auxiliary *base* class. Like the template class JComparibale, it implements the (not-)equal operators == and != as well as the comparison operators <, <=, > and >= of a derived class. In this case, the class can be derived from multiple other base classes, some of which providing for the policy method less. Here, a second template argument is used that corresponds to a so-called type list. The (not-)equal and comparison operators of the derived class corresponds to the (not-)equal and comparison operators of the base classes in the type list. In this, the order determines the comparison hierarchy. The type list can conveniently be specified with class JTYPELIST.

For example, the following classes A and B derive from JComparibale.

```
struct A :
  public JComparable<A>
{
  A() :
    value(0)
  {}

  A(const int value) :
    value(value)
  {}

  bool less(const A& object) const
  {
    return this->value < object.value;
  }

  int value;
};
```

```
struct B :
  public JComparable<B>
{
  B() :
    value(0)
  {}

  B(const int value) :
    value(value)
  {}

  bool less(const B& object) const
  {
    return this->value < object.value;
  }

  int value;
};
```

Here, the classes C and D derive from MultiJComparibale but with different type lists.

```
struct C :
  public A,
  public B,
  public JMultiComparable<C, JTYPELIST<A, B>::typelist>
{
  C(const int a, const int b) :
    A(a),
    B(b)
  {}
};
```

```
struct D :
  public A,
  public B,
  public JMultiComparable<D, JTYPELIST<B, A>::typelist>
{
  D(const int a, const int b) :
    A(a),
    B(b)
  {}
};
```

The following example

```
C c1(1, 0);
C c2(0, 1);

cout << (c1 == c2) << endl;
cout << (c1 != c2) << endl;
cout << (c1 <  c2) << endl;
cout << (c1 <= c2) << endl;
```

```
  cout << (c1 >  c2) << endl;
  cout << (c1 >= c2) << endl;
```

will produce

```
0
1
0
0
1
1
```

and

```
  D d1(1, 0);
  D d2(0, 1);

  cout << (d1 == d2) << endl;
  cout << (d1 != d2) << endl;
  cout << (d1 <  d2) << endl;
  cout << (d1 <= d2) << endl;
  cout << (d1 >  d2) << endl;
  cout << (d1 >= d2) << endl;
```

will produce

```
0
1
1
1
0
0
```

As can be seen, the comparison hierarchy of class `C` and `D` is inverted. Note that without `JMultiComparibale`, the compiler would have detected an error due to the ambiguity of the the (not-)equal operators.