

# 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`.

<pre> struct A :   public JComparibale&lt;A&gt; {   A() :     value(0)   {}    A(const int value) :     value(value)   {}    bool less(const A&amp; object) const   {     return this-&gt;value &lt; object.value;   }    int value; }; </pre>	<pre> struct B :   public JComparibale&lt;B&gt; {   B() :     value(0)   {}    B(const int value) :     value(value)   {}    bool less(const B&amp; object) const   {     return this-&gt;value &lt; object.value;   }    int value; }; </pre>
--	--

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

<pre> struct C :   public A,   public B,   public JMultiComparibale&lt;C, JTYPELIST&lt;A, B&gt;::typelist&gt; {   C(const int a, const int b) :     A(a),     B(b)   {} }; </pre>	<pre> struct D :   public A,   public B,   public JMultiComparibale&lt;D, JTYPELIST&lt;B, A&gt;::typelist&gt; {   D(const int a, const int b) :     A(a),     B(b)   {} }; </pre>
---	---

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.