

JMultiEquals

The template class JMultiEquals resides in the name space JLANG and constitutes an auxiliary *base* class. Like the template class JEquals, it implements the (not-)equal 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 equals. Here, a second template argument is used that corresponds to a so-called type list. The (not-)equal operators of the derived class corresponds to the (not-)equal operators of the base classes in the type list. The type list can conveniently be specified with class JTYPELIST.

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

<pre>struct A : { public JEquals<A> { A(int value) : value(value) {} bool equals(const A& object) const { return this->value == object.value; } int value; };</pre>	<pre>struct B : { public JEquals { B(int value) : value(value) {} bool equals(const B& object) const { return this->value == object.value; } int value; };</pre>
--	--

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

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

The following example

```
C c1(1,1);
C c2(0,1);
C c3(1,0);

cout << (c1 == c1) << endl;
cout << (c1 == c2) << endl;
cout << (c1 == c3) << endl;
```

will produce

```
1
0
1
```

and

```
D d1(1,1);  
D d2(0,1);  
D d3(1,0);  
  
cout << (d1 == d1) << endl;  
cout << (d1 == d2) << endl;  
cout << (d1 == d3) << endl;
```

will produce

```
1  
0  
0
```

Note that without `JMultiEquals`, the compiler would have detected an error due to the ambiguity of the the (not-)equal operators.