**Źródło w IDL:**

```
module MTestApp
{
 interface MInformator
 {
  long GetCallNumber();
  void AddToSum(in long i, out long sum);
 };
 };
```

**Kompilacja:**

```
javac -classpath "/home/tkubik/forte4j/sources/:.MTestApp/" *java MTestApp/*java
```

**Uruchomienie:**

```
tnameserv -ORBInitialPort 3200
java -cp "/home/tkubik/forte4j/sources/:." MTest -ORBInitialPort 3200 -ORBInitialHost localhost
```

**Implementacja:**

```
/*
 * JFrame.java
*/
/**
* @author  tkubik
 */

import MTestApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

class MInformatorImpl implements MInformatorOperations{ ///
    /// class MInformatorImpl extends MInformatorPOA{
  private int Sum=0;
  private int callNr=0;

  public  int GetCallNumber () {
       System.out.println("Server: callNr = "+ Integer.toString(callNr));
       return callNr;
  }

  public  void AddToSum (int i, org.omg.CORBA.IntHolder sum) {
      Sum += i;
      sum.value = Sum;
      callNr += 1;
      System.out.println("Server: sum = "+ Integer.toString(Sum));
  }
}


public class MTest extends javax.swing.JFrame {
    MInformator mInformatorClient;
    String clientName;

    MInformatorImpl mInformatorServer;
    String serverName;

    static NamingContextExt ncRef;
    static org.omg.CORBA.Object objRef;
    POA rootpoa;
    ORB orb=null;
```

```
static String argsI[];

class TalkingThread extends Thread {
public void run() {
   int i=1;
   int callNr;

   org.omg.CORBA.IntHolder s = new org.omg.CORBA.IntHolder();
   for(;;){
       try {
      this.sleep(100);
   } catch (InterruptedException e) {
   }

   mInformatorClient.AddToSum(i, s);
   System.out.println("Client: sum = "+ Integer.toString(s.value));

   callNr = mInformatorClient.GetCallNumber();
   System.out.println("Client: callNr = "+ Integer.toString(callNr));
       }
}
}
/** Creates new form JFrame */
public MTest() {
   initComponents();
}

private void initComponents() {
   jButtonRegister = new javax.swing.JButton();
   jButtonStart = new javax.swing.JButton();
   jButtonExit = new javax.swing.JButton();
   jScrollPane1 = new javax.swing.JScrollPane();
   jTextArea1 = new javax.swing.JTextArea();
   jLabel1 = new javax.swing.JLabel();
   jLabel2 = new javax.swing.JLabel();
   jTextFieldServer = new javax.swing.JTextField();
   jTextFieldClient = new javax.swing.JTextField();
   jButtonConnect = new javax.swing.JButton();

   getContentPane().setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout());

   addWindowListener(new java.awt.event.WindowAdapter() {
      public void windowClosing(java.awt.event.WindowEvent evt) {
         exitForm(evt);
      }
   });

   jButtonRegister.setText("Register");
   jButtonRegister.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent evt) {
         jButtonRegisterActionPerformed(evt);
      }
   });

   getContentPane().add(jButtonRegister, new org.netbeans.lib.awtextra.AbsoluteConstraints(0, 260, -1, 40));

   jButtonStart.setText("Start");
   jButtonStart.setEnabled(false);
   jButtonStart.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent evt) {
```

```
            jButtonStartActionPerformed(evt);
        }
    });

    getContentPane().add(jButtonStart, new org.netbeans.lib.awtextra.AbsoluteConstraints(198, 260, 100, 40));

    jButtonExit.setText("Exit");
    jButtonExit.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jButtonExitActionPerformed(evt);
        }
    });

    getContentPane().add(jButtonExit, new org.netbeans.lib.awtextra.AbsoluteConstraints(308, 260, 100, 40));

    jScrollPane1.setAutoscrolls(true);
    jScrollPane1.setViewportView(jTextArea1);

    getContentPane().add(jScrollPane1, new org.netbeans.lib.awtextra.AbsoluteConstraints(0, 0, 410, 220));

    jLabel1.setText("Server:");
    getContentPane().add(jLabel1, new org.netbeans.lib.awtextra.AbsoluteConstraints(10, 230, -1, -1));

    jLabel2.setText("Client:");
    getContentPane().add(jLabel2, new org.netbeans.lib.awtextra.AbsoluteConstraints(250, 230, -1, -1));

    jTextFieldServer.setText("MIserver1");
    getContentPane().add(jTextFieldServer, new org.netbeans.lib.awtextra.AbsoluteConstraints(60, 230, 90, -1));

    jTextFieldClient.setText("MIserver2");
    getContentPane().add(jTextFieldClient, new org.netbeans.lib.awtextra.AbsoluteConstraints(300, 230, 90, -1));

    jButtonConnect.setText("Connect");
    jButtonConnect.setEnabled(false);
    jButtonConnect.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jButtonConnectActionPerformed(evt);
        }
    });

    getContentPane().add(jButtonConnect, new org.netbeans.lib.awtextra.AbsoluteConstraints(90, 260, 100, 40));

    pack();
}
```

```
 private void jButtonConnectActionPerformed(java.awt.event.ActionEvent evt) {
        // Add your handling code here:
        try {
        // resolve the Object Reference in Naming
        System.out.println("Trying connect to sever: " + clientName);
        mInformatorClient = MInformatorHelper.narrow(ncRef.resolve_str(clientName));
        System.out.println("Client connected to sever: " + clientName);
//        System.out.println("Obtained a handle on server object: " + mInformatorClient);
        jButtonStart.setEnabled(true);
        jButtonConnect.setEnabled(false);

        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
          e.printStackTrace(System.out); }

    }

    private void jButtonExitActionPerformed(java.awt.event.ActionEvent evt) {
        // Add your handling code here:
        if(orb!=null)
        orb.shutdown(false);
        System.exit(0);
    }

    private void jButtonStartActionPerformed(java.awt.event.ActionEvent evt) {
        // Add your handling code here:
        jButtonStart.setEnabled(false);
        TalkingThread t = new MTest.TalkingThread();
        t.start();
    }

    private void jButtonRegisterActionPerformed(java.awt.event.ActionEvent evt) {
        // Add your handling code here:
        clientName = jTextFieldClient.getText();
        serverName = jTextFieldServer.getText();
        System.out.println("client name:-" + clientName + "-");
        System.out.println("server name:-" + serverName + "-");

        try {
    // create and initialize the ORB
        this.orb = ORB.init(argsI, null);
    // get the root naming context
        objRef = this.orb.resolve_initial_references("NameService");

    // get reference to rootpoa & activate the POAManager
        rootpoa = POAHelper.narrow(this.orb.resolve_initial_references("RootPOA"));
        rootpoa.the_POAManager().activate();

    // Use NamingContextExt instead of NamingContext. This is
    // part of the Interoperable naming Service.
        ncRef = NamingContextExtHelper.narrow(objRef);

    // create servant and register it with the ORB
    // create servant
        mInformatorServer = new MInformatorImpl();

    // create a tie, with servant being the delegate.
            MInformatorPOATie tie = new MInformatorPOATie(mInformatorServer, rootpoa); ///
    // obtain the objectRef for the tie, this step also implicitly activates the object
            MInformator href = tie._this(this.orb); ///
```

```
    // get object reference from the servant
    ///    org.omg.CORBA.Object ref = rootpoa.servant_to_reference(mInformatorServer);
    ///    MInformator href = MInformatorHelper.narrow(ref);

       String ior = orb.object_to_string(href);
       System.out.println("Server reference: "+ior);

       // get the root naming context
       // NameService invokes the name service

       // bind the Object Reference in Naming String name = serverName;
       NameComponent path[] = ncRef.to_name( serverName );
       ncRef.rebind(path, href);
       System.out.println("Server: " + serverName + " ready and waiting ...");

       // wait for invocations from clients
       jButtonConnect.setEnabled(true);
       jButtonRegister.setEnabled(false);
        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
          e.printStackTrace(System.out); }

    }

    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {
        if(orb!=null)
            orb.shutdown(false);
        System.exit(0);
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        argsI=args;
        new MTest().show();
    }


    // Variables declaration - do not modify
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JButton jButtonConnect;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JTextField jTextFieldClient;
    private javax.swing.JTextArea jTextArea1;
    private javax.swing.JTextField jTextFieldServer;
    private javax.swing.JButton jButtonStart;
    private javax.swing.JButton jButtonExit;
    private javax.swing.JButton jButtonRegister;
    // End of variables declaration

}
```

```
// Set up full path example
NameComponent comp1 = new NameComponent("servers", "");
NameComponent comp2 = new NameComponent("ThisOrThat", "");
NameComponent serverName = new NameComponent("server1", "");
NameComponent objPath[] = { comp1, comp2, serverName };
```

**Using multiple naming services**

If each independent Naming Service has its own ORB behind it, you can simply get a reference to each ORB and ask it for a reference to its Naming Service:

```
String host1 = "orbhost1.net";
int port1 = 1234;
String host2 = "orghost2.net";
int port2 = 2345;

// Initialize the first ORB reference
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialHost", host1);
props.put("org.omg.CORBA.ORBInitialPort", String.valueOf(port1));
ORB orb1 = ORB.init((String[])null, props);

// Initialize another ORB reference
props.put("org.omg.CORBA.ORBInitialHost", host2);
props.put("org.omg.CORBA.ORBInitialPort", String.valueOf(port2));
ORB orb2 = ORB.init((String[])null, props);

// Get references to the Naming Services
org.omg.CORBA.Object nc1Ref =
  orb1.resolve_initial_references("NameService");
org.omg.CORBA.Object nc2Ref =
  orb2.resolve_initial_references("NameService");

// Narrow the Naming Service references to NamingContexts and use them
...
```

Sun's Java implementation of the CORBA standard !!!!

Another option is to have one Naming Service hold references to other Naming Services located elsewhere on the network. You first have to run some code on the server that is going to act as the bridge. This code gets a reference to the local Naming Service and stores references to remote Naming Services in the local directory:

```
// Get the local ORB and main NamingContext
ORB myORB = ORB.init(...);
org.omg.CORBA.Object ncRef =
  orb.resolve_initial_references("NameService");
NamingContext localNC = NamingContextHelper.narrow(ncRef);

// Create a new subcontext to hold the remote contexts
NameComponent nodeName = new NameComponent("RemoteContexts", "");
NameComponent path[] = {nodeName};
NamingContext ncNode = localNC.bind_new_context(path);

// Get a reference to a remote Naming Service
// using Sun's non-standard ORB properties
Properties remoteORBProps = new Properties();
remoteORBProps.put("org.omg.CORBA.ORBInitialHost", "remote.orb.com");
ORB remoteORB = ORB.init((String[])null, remoteORBProps);
org.omg.CORBA.Object remoteNCRef =
  remoteORB.resolve_initial_references("NameService");
NamingContext remoteNC = NamingContextHelper.narrow(remoteNCRef);

// Store the remote reference in the local context
NameComponent sub = new NameComponent("Naming1", "");
NameComponent path2[] = {nodeName, sub};
localNC.bind(path2, remoteNC);
```

With this done, a remote client can get a reference to the main Naming Service directory and then look up other remote directories within the bridge directory:

```
public class NamingClient {
  public static void main(String argv[]) {
    ORB orb = ORB.init(argv, null);
    org.omg.CORBA.Object ref = null;
    try {
      ref = orb.resolve_initial_references("NameService");
    }
    catch (InvalidName invN) {
      System.out.println("No primary NameService available.");
      System.exit(1);
    }
    NamingContext nameContext = NamingContextHelper.narrow(ref);
    NameComponent topNC = new NameComponent("RemoteContexts", "");
    NameComponent subNC = new NameComponent("Naming1", "");
    NameComponent path[] = {topNC, subNC };
    try {
      org.omg.CORBA.Object ref2 = nameContext.resolve(path);
      NamingContext nameContext2 = NamingContextHelper.narrow(ref2);
      System.out.println("Got secondary naming context...");
    }
    catch (Exception e) {
      System.out.println("Failed to resolve secondary NameService:");
      e.printStackTrace();
    }
  }
}
```

### Modules

The IDL module construct is mapped to a Java package and the package is given the same name as the IDL module. For all IDL types within the module that are mapped to Java classes or Java interfaces, the corresponding Java class or interface is declared inside the Java package that is generated. Figure 5-1 shows the Java code generated for an enumeration declared within an IDL module.

```
/* From Example.idl: */
module Example {
    enum EnumType { .... };
};
```

```
// Generated to Example/EnumType.java:
package Example;
final public class EnumType {
    ....

}
```

*Figure 5-1 Mapping an IDL module to a Java package*

### Global Scope

Any IDL construct declared in the global scope will be placed in a java package which is named IDL_GLOBAL. This name can be changed to a user defined one through the use of compiler options described in idl2java.

All the IDL examples used in this chapter are from the IDL module Example. Therefore, the generated Java declarations will all be placed inside the Java package Example.

### Const Values

IDL constant values are mapped to public static final instances in Java. The name of the IDL constant is used as the name of a Java class. The Java class will have an instance variable value, which contains the value associated with the constant. Figure 5-2 shows a constant named aLong and the resulting aLong class in Java.

```
/* From Example.idl: */
module Example {
    const long aLong = -12345;
};
```

```
// Generated to Example/aLong.java:
package Example;
public final class aLong {
    final public static int value = (int)-12345;
}
```

*Figure 5-2 Mapping an IDL constant to a Java class.*

A client of Example::aLong would access it as:

```
Example.aLong.value
```

**Basic Types**

**Boolean**

The IDL type boolean is mapped to the Java type boolean. The IDL constants TRUE and FALSE are mapped to the Java constants true and false.

```
/* From Example.idl: */
module Example {
    const boolean truth = TRUE;
};
```

```
// Generated to Example/truth.java:
package Example;
final public class truth{
    final public static boolean value = true;
}
```

*Figure 5-3 Mapping an IDL boolean to a Java boolean.*

**Char**

The IDL type char is mapped to the Java type char.

```
/* From Example.idl: */
module Example {
    const char aChar = 'A';
};
```

```
// Generated to Example/aChar.java:
package Example;
final public class aChar{
    final public static char value = (char)'A';
}
```

*Figure 5-4 Mapping an IDL char to a Java char.*

**Octet**

The IDL type octet is mapped to the Java type byte. IDL 2.0 does not allow octet constants so the example shows an interface with a method that accepts an octet as a parameter.

```
/* From Example.idl: */
interface anOctet {
    void foo(in octet x);
};
```

```
// Generated to Example/anOctet.java:
```

```
package Example;
public interface anOctet extends CORBA.Object {
    public void foo(byte x) throws CORBA.SystemException;
}
```

*Figure 5-5 Mapping an IDL octet to a Java byte.*

**String**

The IDL type string is mapped to the Java type java.lang.String.

```
/* From Example.idl: */
module Example {
    const string aString = "Visigenic";
};
```

```
// Generated to Example/aString.java:
package Example;
final public class aString{
    final public static String value = "Visigenic";
}
```

*Figure 5-6 Mapping an IDL string to a java.lang.String.*

**Integer Types**

IDL four integer data types; short, long, unsigned short, and unsigned long. In contrast, Java has only three integer data types; short, int, and long.

```
/* From Example.idl: */
module Example {
    const short aShort = -1;
    const unsigned short anUnsignedShort = 15907;
    const long aLong = -12345;
    const unsigned long anUnsignedLong = 901008;
};
```

```
// Generated to Example/aShort.java:
package Example;
final public class aShort {
    final public static short value = (short) (-1L);
}
// Generated to Example/anUnsignedShort.java:
package Example;
final public class anUnsignedShort{
     final public static short value = (short)15907;
}
// Generated to Example/aLong.java:
package Example;
final public class aLong{

    final public static int value = (int)-12345;
}
// Generated to Example/anUnsignedLong.java:
package Example;
```

```
final public class anUnsignedLong{
    final public static int value = (int)901008;
}
```

*Figure 5-7 Mapping IDL integer data types to Java.*

**Floating Point Types**

The IDL floating point types float and double map to a Java class containing the corresponding data type.

```
/* From Example.idl: */
module Example {
    const float aFloat = 3.14159;
    const double aDouble = 2.718281828459045;
};
```

```
// Generated to Example/aFloat.java:
package Example;
final public class aFloat{

    final public static float value = (float)3.14159;
}
// Generated to Example/aDouble.java:
package Example;
final public class aDouble{
    final public static double value = (double)2.718281828459045;
}
```

*Figure 5-8 Mapping IDL float and double types to Java.*

**Constructed Types**

IDL constructed types include enum, struct, union, sequence and array. The types sequence and array are both mapped to the Java array type. The IDL constructed types enum, struct, and union are mapped to a Java class that implements the semantics of the IDL type. The Java class generated will have the same name as the original IDL type.

**Enum**

Each IDL enum type is mapped to a Java class that defines a final int with the value of each enum member. The narrow method is provided to check whether or not the supplied Java int is in the range defined by the IDL enum. If the int is out of range, a CORBA.BAD_PARAM exception is raised.

```
/* From Example.idl: */

module Example {

    enum EnumType { none, first, second, third, fourth };
};
```

```
// Generated to Example/EnumType.java:
package Example;


final public class EnumType {
```

```
    public static final int none = 0;
    public static final int first = 1;
    public static final int second = 2;
    public static final int third = 3;
    public static final int fourth = 4;
    public static final int narrow(int i) throws CORBA.BAD_PARAM { ... };
};
```

*Figure 5-9 Mapping an IDL enum to a Java enum.*

**Struct**

An IDL struct is mapped to a Java class that provides instance variables for the fields and a constructor from values. A default constructor is also provided that allows the structure's fields to be initialized later.

```
/* From Example.idl: */
module Example {
    struct StructType {
        long these;
        string those;
    };
};
```

```
// Generated to Example/StructType.java:
package Example;


final public class StructType {
// data member
    public int      these;
    public String   those;
//constructors
    public StructType(int _these,String _those) { ... }
    public StructType()  { ... };
}
```

*Figure 5-10 Mapping an IDL struct to Java.*

**Union**

An IDL union is mapped to a Java class that provides a default constructor, an accessor method for the union's discriminator, a method for setting each of the union's branches, and a method for retrieving each of the union's branches. The Java class is given the same name as the IDL union.

The methods for setting and retrieving a union branch have the same name but differ by their signature. The constructor creates the union without initializing it. The union can be initialized using the methods provided.

```
/* From Example.idl: */
module Example {
    union UnionType switch (EnumType) {
        case first:     long win;
        case second:    long place;
        case third:     long show;
        case fourth:
```

```
        default:        long other;
    };
};



// Generated to Example/UnionType.java:
package Example;


final public class UnionType {
    public UnionType() {};
    public int discriminator() { ... }
    public int       win() { ... }
    public void      win(int discriminator, int _val) { ... }
    public int       place() { ... }
    public void      place(int discriminator, int _val) { ... }
    public int       show() { ... }
    public void      show(int discriminator, int _val) { ... }
    public int       other() { ... }
    public void      other(int discriminator, int _val) { ... }
}
```

*Figure 5-11 Mapping an IDL union to Java.*

**Sequence**

An IDL sequence is mapped to a Java array. Anywhere the sequence type is needed, an array of the mapped type of element is used.

```
/* From Example.idl: */
module Example {
    typedef sequence< StructType > StructSequence;
    typedef sequence< StructType, 42 > StructSequence42;
    struct SequenceContainer {
         StructSequence contained;
         StructSequence42 contained42;
    };
};



// Generated to Example/SequenceContainer.java:
package Example;


final public class SequenceContainer {
    // data members
    public Example.StructType[]     contained;
    public Example.StructType[]     contained42;


    // constructors
    public SequenceContainer(Example.StructType[]
                        _contained,Example.StructType[] _contained42) { ... }
    public SequenceContainer()  { ... }
}
```

*Figure 5-12 Mapping an IDL sequence to Java.*

**Array**

An IDL array is mapped to a Java array. Anywhere the array type is needed, an array of the mapped type of element is used.

**Exceptions**

A user-defined exception in IDL is mapped in much the same way that an IDL struct is mapped. A Java class is generated that provides instance variables for each of the fields of the exception. A default constructor is provided for the exception, along with a constructor that initializes each field. All user-defined IDL exceptions extend the class CORBA.UserException which, in turn, extends the class java.lang.Exception. This mapping makes it possible to catch specific user-defined exceptions by catching CORBA.UserException or to catch all user-defined exceptions and system exceptions by simply catching CORBA.Exception.

```
/* From Example.idl: */
module Example {
    exception That {
        string reason;
    };
};
```

```
// Generated to Example/That.java:
package Example;
```

```
public class That extends CORBA.UserException {
    public String   reason;
    public That(String _reason) { ... }
    public That()  { ... }
}
```

*Figure 5-13 Mapping an IDL exception to Java.*

**Interfaces**

IDL interfaces are mapped to java interfaces and given the same name. The Java interface that is generated will always inherit from CORBA.Object

```
/* From Example.idl: */
module Example {
    interface Foo { };
};
```

```
// Generated to Example/Foo.java:
package Example;
public interface Foo extends CORBA.Object { ... }
```

*Figure 5-14 Mapping an IDL interface to Java.*

**Passing Parameters**

IDL defines three parameter passing modes:

- in
- out
- inout

The in parameter are passed by value so IDL in parameters are supplied by passing the actual parameter when a method is invoked. The results of an IDL operations are returned as the results of the corresponding Java method.

Both out and inout parameters are passed by reference: The client supplies an actual parameter that is changed as a result of the invocation. Since Java only supports parameter passing by value, an additional mechanism is required to support out and inout parameters.

The IDL-to-Java mapping defines variable classes for all the IDL basic and user-defined types. A client supplies an instance of the appropriate variable class that is passed, by value, for each IDL out or inout parameter. The contents of the variable, but not the variable itself, are modified by the invocation. The client extracts the changed contents once the invocation returns. Each variable class has a constructor from an instance and a default constructor as well as a public instance member that represents the typed value. The variable classes for the IDL basic types are listed below and are described in CORBA Classes."

- Any_var
- Object_var
- String_var
- boolean_var
- byte_var
- char_var
- double_var
- float_var
- int_var
- short_var

The IDL-to-Java compiler generates variable classes for all user-defined types. Each generated variable class is identified by the name of the IDL type for which it can be assigned, with the suffix _var.

**Attributes**

IDL interfaces may have attributes which control access to their typed fields through *get* and *set* methods. If an IDL attribute is designated as readonly, then only a *get* method will be generated. When these interfaces are mapped to Java, both the *get* and *set* mehtods have the same name but their signatures will differ.

Figure 5-15 shows two attributes named assignable and fetchable. The attribute named assignable can be set or retrieved while the attribute named fetchable is readonly and can only be retrieved. When the Java code is generated from this IDL definition, two methods are provided for the attribute assignable; one for retrieving and one for setting the attribute. Only a *set* method will be generated for a readonly attribute fetchable.

```
/* From Example.idl: */
module Example {
    interface Attributes {
        attribute long assignable;
        readonly attribute long fetchable;
    };
};
```

```
// Generated to Example/AttributesOperations.java:
package Example;
public interface Attributes extends CORBA.Object {


    public int assignable() throws CORBA.SystemException;
    public void assignable(int val) throws CORBA.SystemException;
```

```
    public int fetchable() throws CORBA.SystemException;
}
```

*Figure 5-15 Mapping IDL attributes to Java.*

**Nil Object References**

CORBA nil object references are mapped to the Java null object. Nil object references that are mapped as null are valid parameters and can be used anywhere a CORBA object reference is required.

**Interface Scope**

Java does not allow declarations to be nested within an interface scope nor does it allow packages and interfaces to have the same name. Accordingly, interface scope is mapped to a package with the same name with an underscore suffix.

**CORBA Object**

The CORBA::Object type is mapped to the Java interface CORBA.Object. However, the duplicate and release methods are not mapped because they are required by the C and C++ programming languages' explicit memory management characteristics.

```
package CORBA;


public interface Object {
    boolean _is_a(String repId) throws CORBA.SystemException;
    boolean _is_equivalent(CORBA.Object other_object)
                throws CORBA.SystemException;
    boolean _non_existent() throws CORBA.SystemException;
    int _hash(int maximum) throws CORBA.SystemException;
...
}
```

*Figure 5-19 Mapping the CORBA Object to Java.*

**The Any Type**

The IDL type Any is mapped to the Java class CORBA.Any, which provides *to* and *from* conversions methods for all the defined IDL types. The Any class has just a default constructor which creates an empty object. The *from* methods can be used to initialize and update the Any. This class is discussed in detail in the topic called Any.

```
short x = new Any().from_short(4);

x.from_short(6);
```

*Figure 5-20 Initializing and updating an Any.*

**User-defined Types**

The Java mapping for a user-defined IDL type provides a pair of methods for converting the type to and from an Any.

**Structures, unions and enumerations**

The Java classes that map structures, unions, and enumerations have two overloaded static methods called any, shown in Figure 5-21. One of these methods can produce an Any type from the user-defined type. The other method can produce the user-defined type, given an Any.

```
/* From Example.idl: */
module Example {
    struct StructType {
        long these;
        string those;
    };
};


// Generated to Example/StructType.java:
package Example;

final public class StructType {
    ...
    public CORBA.Any any() throws CORBA.SystemException {...}
    public StructType(CORBA.Any x) throws CORBA.SystemException {...}
}
```

*Figure 5-21 Converting user-defined types.*

**Arrays and Sequences**

IDL arrays and sequences are not directly mapped to a Java class, their _var classes are used to support the Any conversion.

```
/* From Example.idl: */
module Example {
    typedef sequence< StructType > StructSequence;
}




// Generated to Example/StructSequence_var.java:
package Example;


final public class StructSequence_var {


public static final CORBA.Any any(Example.StructType[]value )
                    throws CORBA.SystemException {...}


public static Example.StructType[] any(CORBA.Any x)
                    throws CORBA.SystemException {...}
...>
}
```

*Figure 5-22 Converting sequences and arrays.*

**Interfaces**

Interface mappings provide an any method that maps the object to an Any. The reverse operation can be accomplished using the Any.from_object, which is applicable to any interface type.

```
/* From Example.idl: */
module Example {
    interface Account {
    };
};

// Generated to Example/Account_var.java:
package Example;

final public class Account_var {
    ...
    public static Example.Account any(CORBA.Any x)
                                      throws CORBA.SystemException {...}
}
```

*Figure 5-23 Converting interfaces to Any.*

**CORBA TypeCode**

The CORBA TypeCode is a pseudo interface type used to represent a type for and is mapped according to the rules used for interfaces. All defined IDL type codes are accessible as static data members using the TCKind class, described in the TCKind topic.

The IDL-to-Java mapping defines a typecode method for obtaining the type associated with user-defined types. For interfaces, arrays and sequences, the corresponding _var class provides the typecode method.

```
/* From Example.idl: */
module Example {
    struct StructType {
        ...
    };
};
```

```
// Generated to Example/StructType.java:
package Example;

final public class StructType {
...
public static CORBA.TypeCode typecode() throws CORBA.SystemException {...}

}
```

*Figure 5-24 The typecode method generated for a union.*

```
/* From Example.idl: */
module Example {
    interface Account {
    };
};
```

```
// Generated to Example/Account_var.java:
package Example;


final public class Account_var {
    ...

    public static CORBA.TypeCode typecode() throws CORBA.SystemException {...}
}
```

*Figure 5-25 The typecode method generated within a _var class for an interface.*

**Standard Exceptions**

The mappings for all IDL defined exceptions inherit from the Java CORBA.Exception class. The Java class CORBA.SystemException is used to represent standard system exceptions and CORBA.UserException is used to represent user-defined exceptions. Clients may simply catch the type CORBA.Exception and then narrow the reference to determine the specific type of exception that has occurred. For a the complete list of system exceptions, see "List of System Exceptions".

```
package CORBA;
abstract public class Exception extends java.lang.Exception {
...
}
```

*Figure 5-26 The Java CORBA.Exception class.*

Standard system exceptions are mapped to Java classes that provide methods for accessing the minor code and completion status.

- The reason string contains a description of the failure, if one is available.
- The exception value may contain the actual Java exception. If a java.io.IOException is raised, it will be put into the exception field of a COMM_FAILURE exception.
- The completed value indicates whether or not the requested operation completed successfully.

*Note: The minor code is not used at present because OMG has not specified their values. Also note that the completed value may not always be reliable.*

```
package CORBA;
public class UNKNOWN extends CORBA.SystemException {};
```