

CORBA and Java Compared: Object Model

Jan Kleindienst
kleindie@watson.ibm.com
<http://boss.uivt.cas.cz/~kleindie/homepage.html>

Dept. of Computer Science,
Faculty of Mathematic and Physic,
Charles University, Prague
&
Watson Research
IBM Prague
Murmanská 4/1475
100 00 Prague 10

Abstract: This paper compares Java and CORBA object models. By identifying main characteristics, the paper highlights similarities and differences of both models. Moreover, the key concepts of both object models are overviewed, including object definition, operations, requests, types, interfaces, classes, meta-classes, and inspection. A brief comparison of both models is presented at the end of the paper.

This paper has been written as an attempt to map and compare features and capabilities of two distributed object systems that seem to attract more and more interest these days — Java and CORBA. Each system is represented by its object model that defines what is the notion of object for this system, how the system treats objects in terms of lifetime, operation invocation, mutual communication, etc. Object models are distinguished based on the number of types for which a given operation can be defined. *Classical object models* require operations to be defined on a single type. If a model allows operation to be defined on zero or more types, it is denoted as a *generalized object model*. Both CORBA and Java are examples of classical object models.

The Java Object Model specifies what is *Java object* and describes its features. Java Object Model deals with non-distributed objects only. The Java Distributed Object Model extends the Java Object Model by defining *Java remote objects*. Section 1 overviews both the Java Object Model and the Java Distributed Object Model.

The main goal of the OMG Core Object Model aims at providing common ground for hundreds of vendors of the OMG technology to ensure compatibility between their implementations. The Core model provides a common language and terminology for all OMG-compliant products. The CORBA Object Model stems for the Core Object Model by translating the Core Object Model concepts into the CORBA technology language combined with the support of the Interface Definition Language (IDL). Section 2 overviews the CORBA Object Model, which is based on the OMG Core Object Model. Section 3 concludes this paper by a one-page comparison of both models.

1. Java Distributed Object Model

The following section overviews two Java object models: The Java Object Model (JOM) serves as a bottom line for the all Java-based technology. The second model denoted by JavaSoft as the Java Distributed Object Model (JDOM) [SUNJDOM] is younger and forms a theoretical background for the Remote Message Invocation, which is a pilot implementation of the Java Distributed Object System. This fact makes it more suitable (over the Java Object Model) for the comparison with the CORBA Object Model.

1.1 Basic Concepts of the Java Object Model

There is no publication dedicated to the Java Object Model. Basic concepts are described in [GJS96]. The importance of the Java Object Model increased rapidly with the introduction of the Java Distributed Object Model.

1.2 Basic Concepts of the Java Distributed Object Model

The document [SunDOM] highlights similarities and differences of the JOM and JDOM. Similarly to Java objects, remote Java objects can be also passed as an argument of any method invocation, can be cast to any of the set of remote interfaces, and keep the same semantics of the `instanceof` operator. There are however several changes to the invocation semantics of the remote objects contrasted to classical objects. These changes include passing non-remote arguments of remote methods by copy, passing remote arguments of remote methods by reference, specialized semantics of several methods of the `Object` class, and more complex exception handling for remote objects.

1.3 Objects

For the Java Object Model, an object is an instance of a class. The object data are called *instance variables* and they can be accessed for reading and writing if they are declared as public.

The Java Distributed Object Model introduces a notion of *remote object*. It is an object which methods can be invoked from another JVM potentially located on a different host. Such objects expose their methods via *remote interfaces*. Remote interfaces are specified as regular Java interfaces (using the Java interface keyword). Based on the previous definitions, *remote method invocation* is a process of invoking a method on a remote object. Remote invocations have the same syntax as calls on local methods.

Let us take a look how is the Java Distributed Object Model implemented at the language level. The `java.rmi.server.RemoteObject` class describes semantics of remote objects. `RemoteObject` extends the `Object`¹ class and overloads some of its methods such as `hashCode()`, `equals()`, and `toString()` to define a new semantics for remote objects. The `java.rmi.server.RemoteServer` class allows creation of remote objects and automates exporting their interfaces. The class `java.rmi.server.UnicastRemoteObject` is a subclass of `java.rmi.server.RemoteServer` and implements a TCP/IP-based object that is typically used as a starting point for building user servers. The inheritance hierarchy of these three server-side located classes is depicted on Figure 1.

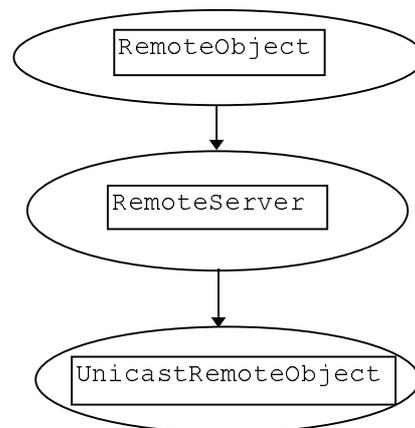


Figure 1: Inheritance hierarchy of RemoteObject, RemoteServer, and UnicastRemoteObject

The following code snippet (Code example 1) undertaken from [SunJDOM] illustrates how `java.rmi.server.UnicastRemoteObject` can be used for implementing user-defined remote object `BankAccountImpl` that implements the remote interface `BankAccount`.

¹ `Object` is a common superclass to all Java objects. It defines nine operations that each Java object implements. These operations are `clone()`, `equals()`, `finalize()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `toString()`, and `wait()`.

```

package my_package;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class BankAccountImpl
    extends UnicastRemoteObject
    implements BankAccount
{
    public void deposit (float amount) throws RemoteException {
        ...
    }
    public void withdraw (float amount) throws OverdrawnException,
        RemoteException {
        ...
    }
    public float balance() throws RemoteException {
        ...
    }
}

```

Code example 1: Extending the `java.rmi.server.UnicastRemoteObject` class

1.3.1 Operations

For the Java Distributed Object Model, methods defined by remote interfaces (i.e. methods that implement the `Remote` interface) must declare the `RemoteException` exception. Catching `RemoteException` guarantees that the method will be able to recover when that invocation fails. If a remote object is an argument or a return value of method, then the formal parameter must be declared as a remote interface rather than a class. Furthermore, local objects passed as arguments of remote calls are passed by copy and must implement the `Serializable` interface. Java runtime marshals (serializes, in the JavaSoft terminology) such objects, transfers their content to the server and unmarshals them in the environment of the remote JVM. The RMI programmer may choose to protect some local objects from leaving the space of the local JVM by not making them serializable. This is achieved by the objects' not implementing the `Serializable` interface. When the RMI runtime comes across such objects, marshalling fails and the runtime throws `NotSerializableException`. Passing of remote and local objects as arguments of a remote invocation is illustrated by Figure 2.

1.3.2 Requests

In contrast to CORBA, Java offers no notion of request and request form (Section 2.3.2). The invocation of a method on a remote object is called the *remote method invocation* in Java.

1.3.3 Exceptions

For the Java Distributed Object Model, there is one common base class for all RMI-related exceptions — `java.rmi.RemoteException`. Since Java requires all methods of remote interfaces to declare `RemoteException` in their throw clause, the RMI runtime is able to catch all remote exception which adds to robustness of the system.

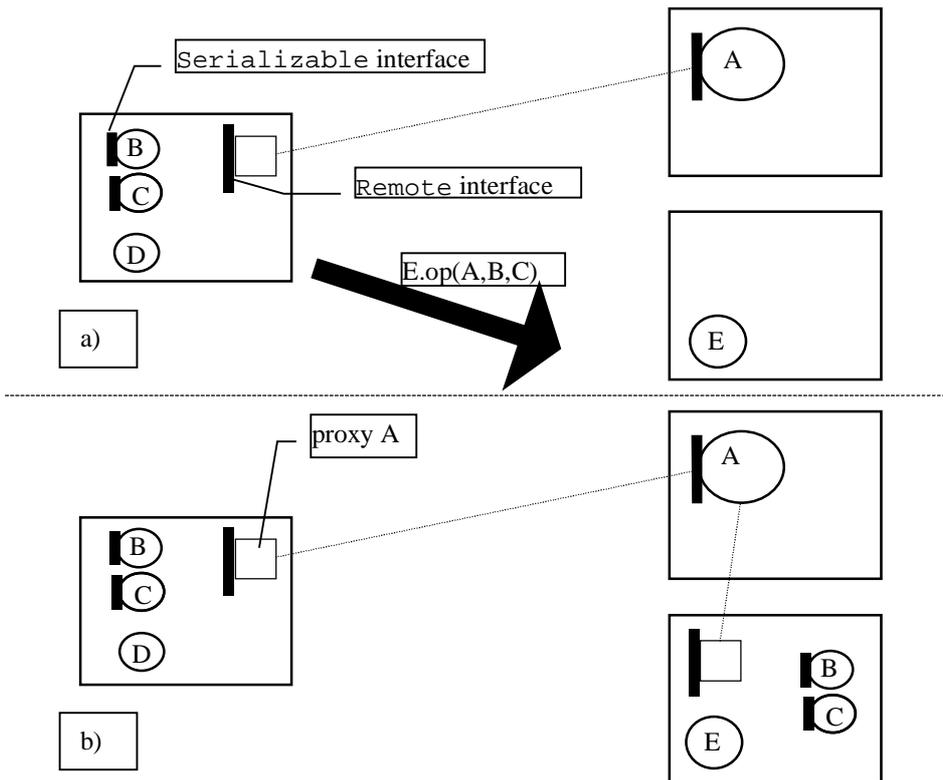


Figure 2: When the operation `op` is invoked on `E` with one remote object `A` and two remote objects `B` and `C` as parameters (a), `A` is passed by reference and `B` and `C` as copy (b). If `D` were passed the call would fail since `D` does not implement the `Serializable` interface.

1.3.4 Specification of behavioral semantics

Java does not define how the incoming invocations will be processed. Behavior in Java is defined by the execution order of class methods.

1.3.5 State

An object state is defined by object attributes. Changing attributes means changing the state.

1.3.6 Object Lifetime

Java objects are typically created by special objects called classloaders that are responsible for creating objects image on the system heap. New classloader object can be created by deriving from the `java.lang.ClassLoader`. For example, the `java.applet` package defines `AppletClassLoader`, while the `java.rmi` package introduces `RMIClassLoader`. Objects cease to exist when the Java automatic background garbage collection detects they are no longer used and reclaim them from the system heap. Object can implement the `finalize()` method where they can implement their „final wish“. This method is called by the garbage collector just before such object is deleted from the memory. For remote objects (i.e. superclasses of `RemoteObject`) `finalize` can be used for example to deactivate the object server that is responsible for maintaining such remote objects.

1.3.7 Communication Model

The Java communication model is synchronous. When event source (Section 1.3.8) calls the listener, the call is carried on in the source's thread. Also, the RMI communication model is synchronous. Clients wait in the invocation thread for the server's results. In contrast, the CORBA event model implemented by the OMG Event Service [OMGEvents] allows asynchronous event delivery.

1.3.8 Events

The Java *event delegation mechanism* (first described in the JavaBeans specification [SunJavaBeans]) decouples event *sources* and event *listeners*. This model is called *delegation*, since it allows the programmer to delegate authority for event handling to any object that implements the appropriate listener interface. Events are mostly used in the AWT package for changing object state information by delivering mouse clicks, keyboard actions, and window component updates. One-to-many relationship is defined by having one source provide AWT events to many listeners (Figure 3) by calling methods on their `EventListener` interface. Listeners must register with the event source. Events themselves are defined as immutable objects derived from `java.util.EventObject`. Special objects called adapters (see Section 1.6) can be interposed between the source and the listeners that augment event delivery with additional semantics. The main idea is to provide means for event queuing, filtering, and demultiplexing. Adapters may for example implement the concept of event channel (Section 2.3.10) defined by the OMG Event Service (ES). Further comparison to the OMG Event Service reveals that the Java events implement only the push model semantics (Section 2.3.10) of the OMG Event Service by pushing events from sources to listeners. Listeners cannot poll servers for events as in the Event Service pull model.

Code example 3 illustrates using the event delegation model in JDK1.1 for delivering button click events to the registered listener. The `mybutton` button in the `EventSource` applet generates the `ActionEvent`. This event is delivered to the `ClickHandler` class that needs to implement the `ActionListener` interface. `ClickHandler` first must register itself with `mybutton` by calling the `addActionListener()` method. The method adds reference to the `ClickHandler` to the list of already registered listeners (kept as a private field within the `mybutton` object). When a mouse click occurs, `mybutton` iterates over this list and calls the `actionPerformed()` method on each listener.

JDK 1.1 currently defines the event delegation mechanism as a recommended solution for delivering events from event sources to event listeners. The event model used in earlier versions of JDK suffered a huge inefficiency when delivering fired events from source objects to consumer objects. Here is the description of the old-style event model as described in [SunJFC].

The event model in AWT 1.0 was quite simple. All events were passed up the inheritance hierarchy forcing objects to catch and process events by subclassing and overriding either the `handleEvent()` or `action()` methods. Complex if-then-else conditional logic was required in the top level object to determine which object triggered an event. This was not scalable and was ill-suited to high-performance distributed applications.

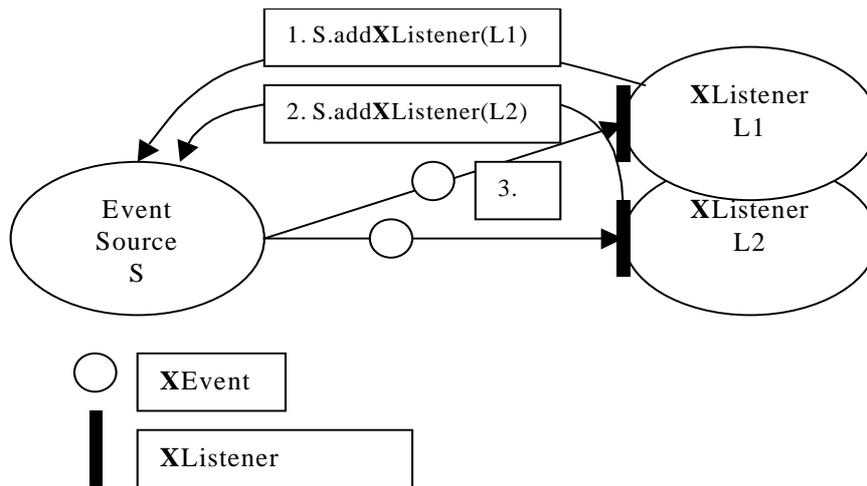


Figure 3: The basics of the event delegation model: Let X be an event, for example the Action event. Listeners L1 and L2 first have to register themselves with the event source S by calling addActionListener() [1.,2.]. When Action occurs S sends the ActionEvent object to both registered listeners [3.]. The act of sending the ActionEvent is translated to the call on a method of the ActionListener interface with the ActionEvent object as parameter.

Thus in the old-style event model, a simple mouse click event could bubble up through the whole hierarchy of GUI application classes till it reached its consumer. The efficiency of the event model was the key motivation for introducing the new event delegation model. The event delegation model delivers events directly to registered listeners without having them to wander in the AWT class hierarchy and waiting for object that will have consumed them.

1.4 Binding

For a client to be able to invoke a remote operation on an object, it must first obtain a reference to this object. The RMI provides a simple bootstrapping mechanism that relies on the Java Naming (or at least its first implementation, further version of Java Naming are in pipe as documented by [SunJNDI]). The remote object have to register itself using the `java.rmi.Naming.bind()` method. This call causes that the pair `<object_name, object_reference>` is registered with the Java Naming. The client calls the `java.rmi.Naming.lookup()` method with a human readable `object_name` passed as an argument, and the method returns a stub object, that mediates invocation to the actual remote object. Code example 2 undertaken from [SunJDOM] documents the point.

```

BankAccount acct = new BankAccountImpl();
String url = „rmi://java.sun.com/account“;
// Bind url to remote object
java.rmi.Naming.bind(url,acct);
...
// Lookup account in the client code
BankAccount acct = (BankAccount) java.rmi.Naming.lookup(url);

```

Code example 2: Binding and looking up object reference

```

import java.applet.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

// ActionEvent producer
public class EventSource extends Applet {
    Button mybutton;

    public EventSource() {
        setLayout(new BorderLayout());
        mybutton = new Button("Click Me");
        add(mybutton);

        mybutton.addActionListener( new ClickHandler() );
    }
}

// ActionEvent consumer
class ClickHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked! Event:" + e);
    }
}

```

Code example 3: Example of using the event delegation model

1.5 Polymorphism

All methods in Java are "virtual" in the C++ sense; that is, method invocations are resolved dynamically. Also, Java has a notion of an abstract class that can declare abstract methods. These can be later implemented by subclasses. Generally, Java supports polymorphism by allowing methods defined by a superclass to be overwritten (overloaded) by subclasses and based on the actual object type properly resolved at runtime.

1.6 Encapsulation

Fields and methods encapsulated in classes can be tagged with `public`, `protected`, and `private` modifiers, which affect their visibility to other Java classes. By provision of packages, Java conveys the idea of encapsulation to somewhat higher level of abstraction. Modifiers applied to classes assess their visibility outside the package, which encapsulate them. Using `protected` modifier lack sense this respect. A field declared without the modifier is considered as a package-private. Such field is visible to all classes inside the package but it not visible from non-package classes.

Before JDK1.1 had arrived, top-level classes were the only design option in Java. *Top-level classes* are those contained in packages². The design decisions to deploy the delegated event model in the new AWT combined with the pressing needs to provide an alternative to multiple inheritance caused introduction of inner classes. *Inner classes* are classes that may be declared inside another class or even inside a class method. JavaSoft has the following on inner classes:

Inner classes result from the combination of block structures with class-based programming, which was pioneered by the programming language Beta.

² If the class does not define the package where it belongs, Java assigns it to the *default* package. Thus all classes are contained in a package by definition.

To confuse the enemy, inner classes declared as `static` are considered top-level classes. According to JavaSoft [SunInnerCls], the key difference between the top-level and inner classes is that inner classes can make direct use not only of their instant variables but also of the fields of the class declaring the inner class.

Class	Member of	Direct Use of
Inner	<ul style="list-style-type: none"> • Class • Method 	<ul style="list-style-type: none"> • inner class instant variables • instant variables of class where the inner class is declared
Top-level	<ul style="list-style-type: none"> • package • class, but declared <code>static</code> 	<ul style="list-style-type: none"> • the top-level class instant variables only

The following paragraphs show how inner classes help Java to cope with lack of multiple inheritance. The event delegation model (that is closely described in Section 1.3.8) of the AWT in JKD 1.1 defines event sources, which fire events to event listeners. The AWT event listeners must implement special interfaces, which may in some cases represent overkill for the application. Suppose a simple application displays a button and needs to monitor mouse clicks on this button. The code of this application is presented in Code example 4. To facilitate catching mouse clicks, the `Clicker` applet must implement the `MouseListener` interface. Although `Clicker` is interested in being notified only when someone clicks the button, it has to implement the rest of `MouseListener` methods just to get the code compiled. This may be annoying for the programmer and the inconvenience grows with larger applications.

To alleviate this inconvenience, JKD 1.1 introduced *adapter classes*, which “pre-implements” the listener interfaces. A typical example of an adapter class would be the `MouseAdapter` class that implements empty bodies of the `MouseListener` interface. Here is the code:

```
class MouseAdaptor implements MouseListener {
    public void mouseClicked( MouseEvent e ) {
    public void mouseEntered( MouseEvent e ) {}
    public void mouseExited( MouseEvent e ) {}
    public void mousePressed( MouseEvent e ) {}
    public void mouseReleased( MouseEvent e ) {}
}
```

The code from Code example 4 can be rewritten as shown in Code example 6. The programmer now shakes off the burden of implementing empty bodies of `MouseListener`. Instead, he or she may use the `MouseAdaptor` class implemented by a third party and make thus the code more readable. Although this code may now seem elegant, it suffers one fundamental drawback: it would not compile! The reason is Java does not support multiple inheritance of classes, and since both `Applet` and `MouseAdaptor` are classes, the following statement from Code example 6 is syntactically wrong:

```
public class Clicker extends Applet, MouseAdapter {
```

So we have this nice third-party adapter class but we cannot use it due to the absence of class multiple inheritance in Java. Time is now ripe for employing inner classes.

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Clicker extends Applet implements MouseListener {
    Button mybutton;

    // Display the button
    public Clicker() {
        top.setLayout(new BorderLayout());
        mybutton = new Button("Click Me");
        add(mybutton);

        mybutton.addMouseListener( this );
    }

    // Originally abstract methods of MouseListener interface
    public void mouseClicked( MouseEvent e ) { // I need this one
        System.out.println("mouse clicked");
    }
    public void mouseEntered(MouseEvent e) {} // But I do not
    public void mouseExited(MouseEvent e) {} // need the rest
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
}

```

Code example 4: Inconvenient use of the MouseListener interface

The inner class ClickCatcher in Code example 5 bypasses the necessity to use multiple inheritance. As depicted at Figure 4, the multiple inheritance translates to a pair of single inheritance relations by utilizing the inner class. Note that an instance of the inner class is

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Clicker extends Applet {
    Button mybutton;

    // Display the button
    public Clicker() {
        top.setLayout(new BorderLayout());
        mybutton = new Button("Click Me");
        add(mybutton);

        mybutton.addMouseListener( new ClickCatcher() );
    }

    // The inner class implementing the MouseListener interface
    class ClickCatcher extends MouseAdapter {
        public void mouseClicked( MouseEvent e ) {
            System.out.println("mouse clicked");
        }
    }

    // Originally abstract methods of MouseListener interface
    public void mouseClicked( MouseEvent e ) { // I need this one
        System.out.println("mouse clicked");
    }
}

```

Code example 5: Deploying inner classes

registered with the mybutton object when calling the addMouseListener() function in Code example 5.

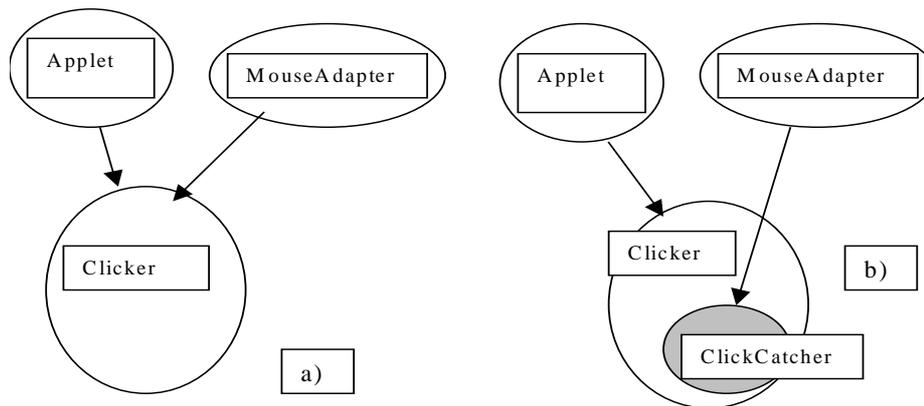


Figure 4: Multiple inheritance (a) translates to two single inheritance relationships (b): one for Clicker, one for the inner class ClickCatcher

There are more interesting things to inner classes. They can be declared inside methods such as this

```

Foo myFooGenerator (int seed)
{
    class Foo {
        private int startValue;
        public Foo (int seed) { startValue = seed; }
    }
    Foo genFoo = new Foo(seed);
    return genFoo;
}

```

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Clicker extends Applet, MouseAdapter {
    Button mybutton;

    // Display the button
    public Clicker() {
        top.setLayout(new BorderLayout());
        mybutton = new Button("Click Me");
        add(mybutton);

        mybutton.addMouseListener( this );
    }

    // Originally abstract methods of MouseListener interface
    public void mouseClicked( MouseEvent e ) { // I need this one
        System.out.println("mouse clicked");
    }
}

```

Code example 6: Oops, we do not have multiple inheritance of classes in Java

or even declared in time when they are going to be used (called anonymous inner classes in such case):

```
Foo myFooGenerator (int seed)
{
    return new class Foo(seed) {
        private int startValue;
        public Foo (int seed) { startValue = seed; }
    }
}
```

The programs using inner classes are backward compatible with the older releases of JDK at the level of class files. The bottom line is that inner classes are transparently compiled to top-level classes, using the special dollar (\$) naming convention. For example the inner class from Code example 5 would be known to the Java Virtual Machine as `Clicker$ClickCatcher`. The reader can find more on inner classes online in [\[SunInnerCls\]](#).

1.7 Identity, Equality, Copy

Java objects can be tested for equivalence using the `equals(Object obj)` method declared in the `java.lang.Object`. JDK online documentation says:

The `equals()` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x==y` has the value `true`).

`java.lang.Object` also provides the `clone()` method that serves as a “copy constructor” for creating the same copies of the source class. The cloning itself is “automagically” handled by the Java runtime without the necessity to provide a copy constructor as it would be required in C++.

Both methods can be overwritten by subclasses in case more suitable semantics is required. Classes that overwrite `clone()` must also implement the `Cloneable` interface to express their real interest of being cloned. Otherwise, `CloneNotSupportedException` is thrown by the Java runtime at the time of cloning.

1.8 Types, Interfaces, and Classes

Classes are “templates” for creating objects. Classes are not objects. Several objects can share class static attributes and methods. The Java Distributed Object Model introduces a notion of *remote interfaces* that expose semantics of remote objects. Technically, all remote interfaces directly or indirectly inherits from the `Remote` interface. Classes may implement any number of remote interfaces. Class methods not specified by remote interfaces can used only locally and are not visible in remote stubs. One thing worth to emphasize is that the stub type corresponds to the type of its implementing class.

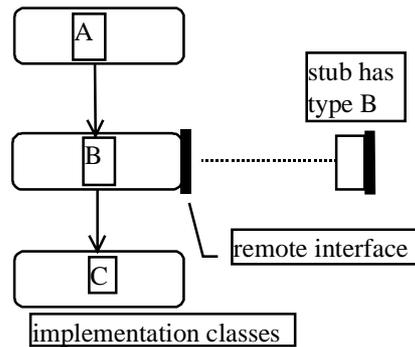


Figure 5: Type equivalency of remote objects with stubs. The stub type is equivalent to the type of B.

1.9 Inheritance and Delegation

Java class relationships are defined in terms of inheritance. Hierarchical structure of the inheritance tree always begins with the upper-most class `Object`. Subclasses inherit data and methods from their superclasses. Abstract classes cannot be instantiated.

1.10 Metaclasses and Metaobject Protocol

Java designers rejected the idea of having a special metalanguage such as CORBA IDL describing the fields and methods of Java classes. Instead, they implemented metalanguage support in Java and integrated reflection APIs into the Java Core classes. The methods `getFields()`, `getMethods()`, `getConstructors()`, `getModifiers()`, `getInterfaces()`, etc. of the `Class` object are here to give a handle on performing thorough introspection of Java classes.

1.11 Introspection

The `java.lang.reflect` package implements five classes: `Array`, `Constructor`, `Field`, `Method`, and `Modifier` and specifies the `Member` interface. That provides enough artillery for manipulating object states at the run-time. For example, the following code snippet (Code example 7) shows how to print out all method names preceding with the substring `get` in a class `MyFoo` and its superclasses.

```

Class cls = MyFoo.class;
Method methods[] = cls.getMethods();
for (int i = 0; i < methods.length; i++) {
    Method method = methods[i];
    if (method.getName().startsWith("get"))
        System.out.println(method.getName());
}

```

Code example 7: Using introspection for getting class method names

Given a method name, a target object and the requested parameters, it is possible to invoke the method represented by the `Method` object on the specified object with the specified parameters. Code example 8 illustrates how it can be done. First, the method `getBar()` of the `MyFoo` class

is extracted to the `getter` class. Since `getBar()` takes no parameters, `margs[]` is initialized to the empty array. Finally, `getBar()` is invoked on the class `target` with `margs` containing parameters. If `target` does not implement `getBar()`, `res. parameters have bad format`, `res. an access violation is encountered`, the following exceptions are thrown: `InvocationTargetException`, `res. IllegalArgumentException`, `res. IllegalAccessException`.

```
Class cls = MyFoo.class;
Method getter = cls.getMethod("getBar",cls);

Object margs[] = {};
try {
    Bar mybar = getter.invoke(target, margs);
} catch (Exception ex) { System.err.println(ex); }
```

Code example 8: Invoking the `getBar()` method using the Java reflection API

Among others, the class `Field` provides the `set()` and `get()` methods for manipulating objects' attributes at run-time. Code example 9 documents how the value of the `BarHolder` attribute of the `MyFoo` class can be obtained and subsequently changed.

The `Modifier` class provides static methods and constants to decode class and member access modifiers. `Modifier` is useful for discovering whether a target class is abstract, a requested method is private, a given field is static, an encountered interface is public, etc.

```
Class cls = MyFoo.class;
Field bar = cls.getField("BarHolder",cls);

Object margs[] = {};
try {
    Bar mybar = bar.get(target);
    System.out.println("Value of BarHolder is:" + mybar);

    mybar = ...; // set a new value
    bar.set(target, mybar);
    System.out.println("Value of BarHolder has been changed to:" +
bar.get(target));
} catch (Exception ex) { System.err.println(ex); }
```

Code example 9: Getting and setting value of `BarHolder`

The `Constructor` class provides information about, and access to, a single constructor for a class. For example, it provides the `newInstance()` method that create a new instance of a class given an array of parameters.

The `Array` class provides static methods to dynamically create and access Java arrays. This class provides setter and getter methods that conveniently manipulate arrays such as `getFloat()` and `setChar()`.

Except `Modifier` and `Array`, the classes `Constructor`, `Method`, and `Field` implement the interface `Member` that represents their role of containees in the class container. The `Method` interface specifies only the following three methods: `getDeclaringClass()`, `getModifiers()`, and `getName()` with obvious semantics.

The capability of `java.lang.reflect` package to introspect Java classes is heavily used by the `JavaBeans` [[SunJavaBeans](#)], the component software environment implemented by `JavaSoft`. `JavaBeans` interconnect two or more Java classes (called beans in this context) by

linking event bean sources with the event listeners. This so-called event delegation model is described in Section 1.3.8. To accomplish this task, JavaBeans must be able to introspect cooperating beans. JavaBeans define two approaches to discover what methods, events, and properties a given bean exports.

- The low-level reflection mechanism relies on the special syntax extensions that based on the well-defined method names allow guessing what properties a given class maintains. For example, `Bar.getFoo()` and `Bar.setFoo()` methods hints to the reflection mechanism that `Bar` has the property `FOO`. This first approach relies on the programmer's discipline to follow the set of required syntax extensions (denoted a bit confusingly as *design patterns* by JavaSoft).
- The `BeanInfo` class represents a higher level approach. If a matching `BeanInfo` class is found for an analyzed Java class, the `java.beans.Introspector` class responsible for inspecting Java classes analyzes the information contained in the `BeanInfo` class to figure out what methods, events, and properties this class exposes. If the corresponding `BeanInfo` is not found, the introspection is carried on by the low-level reflection mechanism described by the previous bullet.

It is possible to combine both approaches. For example, `MyFooBeanInfo` may describe only methods and events of the `MyFoo` class, suggesting to `java.beans.Introspector` to find the information on `MyFoo` properties via the low-level reflection mechanism. A subset of the JavaBeans API — the `java.beans` package — has become part of the Java Core packages.

2. CORBA Object Model

CORBA Object Model derives from the Core Object Model. The Core Object Model is denoted as *abstract object model*. The adjective *abstract* suggests that this model is not implemented by any particular technology (similar as abstract classes are not instantiated by any particular program) but serves as a starting point for to-be-implemented object models. The following sections describe the CORBA Object Model and its “base class” — the Core Object Model.

2.1 Basic Concepts of the Core Object Model

This is what OMG has to say about its Core Object Model [\[OMGExecutiveOverview\]](#):

The OMG Object Model is based on a small number of basic concepts: objects, operations, and types and subtyping. The OMG Object Model defines a core set of requirements, based on the above mentioned basic concepts, that must be supported in any system that complies with the Object Model standard. While the Core Model serves as the common ground, the OMG Object Model also allows for extensions to the Core to enable even greater commonality within different technology domains. The concepts, known as Components and Profiles, are supported by the OMA and are discussed at length in the OMA Guide.

OMG's understanding of the object-oriented paradigm is reflected in the Core 92 Object Model (Core92) [\[OMA95\]](#). Core92 defines such concepts as object, inheritance, subtyping, operations, signatures, etc. Additional concepts can be added to Core92 to create an extension (*component*). A component should not replace, duplicate, and remove concepts. Components should be orthogonal to each other. A *profile* is a combination of Core92 and one or more

components. Typical examples of profiles are the CORBA profile, the Common Object Model, the ODP Reference Model, and the Core 95 Object Model [OMGCore95].

2.2 Basic Concepts of the CORBA Object Model

The concrete CORBA Object Model is built from its underlying abstract Core Object Model by the following means:

- **elaboration**, e. g. by defining form of request parameters, or the language used for specifying types
- **population**, e.g. by introducing specific objects, operations, and types based on existing components
- **restriction**, e.g. by eliminating entities and placing additional restriction on their use

Elaboration, population, and restriction are the only transformations allowed. It is worth to note that a concrete object model cannot *extend* the abstract object model by adding new object, operation, and types that are not derived from the existing components. Each concrete model must behave inside a box set by the abstract object model. Basically, abstract object model is a „base class“ of all the concrete models that have been created by any of the three operations defined above.

The CORBA Object Model is a classical object model, where a message uniquely identifies an object, operation, and parameters. Based on the operation and the object type, a method that implements this operation on this object is selected. Behind the scope of the CORBA Object Model are concepts more properly related to architectural reference model such as compound objects, links, copying of objects, change management, transaction, model of control and execution. The CORBA Object Model defines concepts for both the client side of the communication as well as for the implementation side located on the server (The CORBA Object Model defines a *client* of a service as any entity capable of requesting the service).

- **Client side:** Concepts on the client side deal with issues of object creation, object identity, requests and operation, types and signatures, etc.
- **Implementation side:** Object implementation side describes concepts related to methods, execution engines, and activation.

The following section deals with various object model characteristics. Each section is divided into two parts. The first part list definitions specified by the Core Object Model, while the second part show how the CORBA Object Model elaborates, populates, or restricts the Core Object Model.

2.3 Objects

The Core Object Model defines objects as instances of types. For the CORBA Object Model, an object is an identifiable, encapsulated entity providing one or more services that can be requested by a client.

2.3.1 Operations

Operation describes actions that can be applied to arguments. Each operation has a signature. The signature consists of the operation's name, list of arguments, and list of returned values, if any. Formally, the operation O has the signature

$$O:(p_1:t_1, p_2:t_2, \dots, p_n:t_n) \rightarrow (r_1:s_1, r_2:s_2, \dots, r_m:s_m),$$

where O is the name of the operation. The signature specifies $n \geq 1$ parameters with names p_i and types t_i , and $m \geq 0$ results with names r_i and types s_i .

Operations are always specified with a *controlling parameter*. Operation is defined on the type of its (just one) controlling parameter. The Core Object Model is thus a classical model. All operations on the particular type have distinct names. Operations can only be defined on object types. For the Core Object Model, the concept of "operations being applied to objects" is equivalent with "sending requests to objects". Operations (definitions of signatures) are not objects.

For the CORBA Object Model, an operation is an identifiable entity denoting a requested service. It is not a value. Operations are identified by *operation identifiers*. In addition to the Core Object Model, the CORBA Object Model augments the operation *signature* with additional components:

- specification of exception that may be raised,
- specification of additional contextual information and
- indication of the execution semantics the client should expect from a request for the operation.

To summarize, the general form of the operation signature for the CORBA Object Model is

[oneway] <return_type> <op_name> (p_1, p_2, \dots, p_k) [raises (e_1, e_2, \dots, e_l)] [context(n_1, n_2, \dots, n_3)],

where *oneway* indicates that the client do not want to wait for the result value, p_i are parameters (i.e. , *in*, *out*, and *inout*), e_i are allowed extensions, and n_i indicates what contextual information are required to be transported with the request.

2.3.2 Requests

An operation invocation is called *request*. A request indicates an operation and possibly lists some arguments to which the operation will be applied. Request is send as a event on behalf of requester. The outcome of sending request can be the following:

- returned result(s)
- state change with no immediate result(s) returned
- exception

The Core Object Model does not consider request to be objects. The CORBA Object Model is more precise on what information requests convey. The information consists of an operation, a target object, zero or more parameters, and an optional request context. A procedure of preparing the request in a standardized manner is called *request form*. Request form depends on the language binding used and varies for static and dynamic invocation. In the requests, an

actual parameter is represented by *value*. In addition to the IDL types, a value can also identify an object. Such value is called *object name*. If a object name identifies the object *reliably* (this rather diplomatic formulation is an outcome of OMG not having defined the comparison operation on OIDs, otherwise *uniquely* would be the right word), CORBA Object Model denoted it as an object reference.

On top of the return semantic defined by the Core Object Model, the CORBA Object Model augments the semantic two statements:

- any output parameters are undefined if an exception is returned and
- values that can be returned in an input-output parameter may be constrained by the value that was input.

2.3.3 Interfaces

The CORBA Object Model defines an object's *interface* as a description of operation that the client can possibly request on the object. An interface type is a type that is satisfied by any object that satisfies a particular interface. Interfaces are specified in OMG IDL. Objects can support multiple interfaces. The CORBA Object Model also defines a concept of the principal interface, which consists of all operations in the transitive closure of the interface inheritance graph.

2.3.4 Exceptions

For the CORBA Object Model, an exception is an indication that an operation request was not performed successfully. Each exception may be accompanied by an additional information that provides more details on the exception.

2.3.5 Specification of behavioral semantics

The Core Object Model does not specify the execution order of operations. The Core Object Model does not care whether the client issues request sequentially or concurrently. Moreover, formal specification of the operation semantic is not required. It is however a recommended practice to include a comment that specifies the semantics of the operation, preconditions to be met, postconditions to be guaranteed, and invariants to be preserved.

The CORBA Object Model defines two classes of execution semantics [OMGObjectModel]:

- *At-most-once*: if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most - once
- *Best-effort*: a best-effort operation is a request-only operation, i.e. it cannot return any results and the requester never synchronizes with the completion, if any, of the request

To avoid cases where the client would choose a two different semantics for one object implementation, the execution semantic is associated with the operation. Best-effort semantic is identified by the *oneway* keyword.

2.3.6 Execution and Construction

The execution model describes how services are performed. It defines the notion of a *method* as a code that executes requested operation (the execution itself is denoted as *method activation*), the notion of a *method format* defining a set of execution engines that can interpret the method, where an *execution engine* is an abstract machine that can only interpret methods of a certain format. The mechanism of the invocation is as follows: Upon receiving the client's request, the appropriate method is called. It takes input parameters delivered by the request and passes the return value back to the client. Before the method activation, it may be necessary to first copy an object state into the execution context; this action is called *activation*.

The construction model describes definition of services. Construction stems from the concept of *object implementation*, that implements the object state, define object methods, and other things that must ensure seamless object creation, invocation, and deletion.

2.3.7 State

State is important for capturing side effects of operation invocations. State also affects the results of operations. There is a good example in [ANSIX3H7] illustrating the concept of state:

For example, the operation *marry* takes two person objects as input arguments and produces side effects on these two objects. State captures these side effects, and, presumably, a subsequent application of the *spouse* function to either object will now yield a different result from what it did before.

Formally, the Core Object Model uses object operations to model an interface through which the object state is accessed. The modeling is based on *attributes* and *relationships*.

2.3.8 Object Lifetime

The Core Object Model does not address object creation and deletion.

From the client's perspective, the CORBA Object Model defines that objects are created and destroyed as an outcome of issuing requests. The existence of a new object is revealed to the client by receiving its object reference.

On the server side, lifetime of object can be maintained by the Lifecycle Service [OMGCORBA20]. The Lifecycle Service provides its client with means to create, delete, copy, and move objects. Objects are created by *object factories*. These are simply other objects capable of creating and returning an object as a result of some sort of *create()* request. As the parameters required to create an object may vary among different object types, the factory interface is not standardized. A *generic factory* can be used to dispatch or coordinate calls to several object factories. Generic factories can be hierarchically organized. As it is not possible to dispatch calls to factories with potentially different proprietary interfaces (provided by different vendors), the generic factory can only dispatch calls to factories that inherit the standardized *GenericFactory* interface. Additional request parameters are passed in a form of *criteria* list, each parameter being stored as a named value.

An object is deleted by issuing a *remove()* request on its *LifeCycleObject* interface. Objects can make themselves unremovable; this property is signaled by returning an appropriate exception as a result of the *remove()* request. An object can be moved or copied to another location using the *move()* or *copy()* requests of its *LifeCycleObject* interface. To find a target

location of the operation, the Life Cycle Service uses a *factory finder*. By issuing the *find_factories()* request, the client can ask a factory finder to return a set of factories capable of creating a new copy of the object in a target location. The specification does not standardize any mechanism for transferring the object state from one location to another.

2.3.9 Communication Model

The standard CORBA communication model is synchronous. The stub waits in the invocation call till the results come back from the server. There is also semi-synchronous model available by using the keyword *oneway*. Under this semantics, the stub returns from the invocation code immediately without waiting for the server's results. The client can register a special method that will get called as soon as the results arrive from the server.

Asynchronous communication model can be employed via the OMG Event Service (Section 2.3.10). The Event Service is not part of the CORBA bare system and CORBA assumes them optional.

2.3.10 Events

CORBA addresses event notification via the Event Service. This service defines notions of event suppliers and receivers. Two approaches are defined depending upon who commences the communication. In the push model, supplier calls its consumer to deliver data, while in the pull model, consumer calls its supplier to request data. Although in principal possible, neither supplier nor consumer is expected to call the other communicating object directly. Instead, all events are passed through an *event channel*. From the supplier's point of view, the event channel acts as a consumer; from the consumer's point of view, the event channel acts as a supplier. The event channel also provides operations to register both suppliers and consumers. Subject to the quality of service, the event channel can provide one-to-one, one-to-many, or many-to-many communication, and other additional features.

Communication among objects can be either *generic* or *typed*. Generic communication relies on suppliers and consumers having a standardized interface capable of passing an object of the class *any* as event data. With typed communication, suppliers and consumers are expected to agree on a proprietary one-way operation to pass event data in a suitable format.

2.4 Binding

When the request is delivered, a specific method is selected for the operation execution. This selection is called *operation dispatching*. The Core Object Model bases operation dispatching on the type of the object passed as the controlling argument. This selection process invokes method that has the same name as the operation. This method is executed on the object that has the immediate type of the controlling argument. This techniques can be either compile-time or run-time based.

2.5 Polymorphism

The Core Object Model supports polymorphism based on subtyping, while allowing other kind of polymorphism to take place [OMAG].

2.6 Encapsulation

In the Core Object Model, a type exports all operations that are defined on it. The specification of IDL does not define the access control modifiers such as `public`, `private`, and `protected`.

2.7 Identity, Equality, Copy

Each object has a unique identity in the Core Object Model. This identity is supposed to be independent of any object characteristics. Even though object characteristics can change, object identity remains the same. The identifier that denotes object identity is called OID.

The Core Object Model does not require a comparison operation for OIDs. The reason is hidden in the personal story about the formal CORBA guru Mike Powel³. Unfortunately, the inability to compare object identity causes severe inconveniences when implementing some CORBA Services, as for example the OMG Relationship Service [KPT96].

2.8 Types, Interfaces, and Classes

The Core Object Model defines objects as instance of types. The following claims are equivalent: “object is and instance of type T” and “object is of type T”.

OMG defines types as follows [OMAExecutiveOverview]:

Type can be viewed as a template for object creation. A type characterizes the behavior of its instances by describing the operations that can be applied to those objects. There can exist a relationship between types. The relationships between types are known as supertypes/subtypes.

Types can be hierarchically arranged (directed acyclic graph). The root of such graph is represented by the type *Object*. Every new type is introduced by subtyping from *Object*. The concept of one common root has one significant advantage: *Object* can represent an object of any type.

Set of operation signatures defined on a type is called type’s *interface* (it also includes signatures inherited from its supertypes). Object type is independent on its interface and instances. Both can change, but object type remains the same.

The combination of a type specification and one of the implementation is denoted as *class*. In conformance with this concept, every object is an instance of a class. The Core Object Model does not require the object to retain implementation that was initially assigned to it.

The CORBA Object Model defines type as an identifiable entity with an associated predicate. This predicate is used for detection whether a given value satisfies a type, i.e. the object predicate is true on that value. Such value is called a *member of the type*. The *extension of a type* is the set of values to satisfy the type at any particular time. An *object type* is a type whose members are objects. In contrast to the Core Object Model, the CORBA Object Model

³ According to Dough Lee, the former CORBA pioneer Mike Powel strongly opposed the idea of CORBA objects having unique and immutable OIDs. Instead, M. Powel allegedly pursued the rest of the OMG gurus to view CORBA objects as some sort of channels that change in time and thus have no identity.

is designed to be implemented. Hence, it must deal with the “low-level” type related issues. It defines a set of eight basic types (e.g., 32-bit and 64-bit IEEE floating point numbers, or the type “any” that can represent any possible basic or component type) and seven constructed types such as record, array, and interface.

2.9 Inheritance and Delegation

The Core Object Model distinguishes subtyping and inheritance in the following manner:

Subtyping is a relationship between types based on their interfaces. It defines the rules by which objects of one type are determined to be acceptable in contexts expecting another type. *Inheritance* is a mechanism for reuse. It allows a type to be defined in terms of another type.

The Core Object Model supports subtyping of object types. It restricts objects to be direct instances of exactly one type. This type is called immediate type and there is no mechanism defined how to change an object’s immediate type once this has been assigned. While subtyping models relationship between interfaces/types, inheritance applies to implementation as well. The Core Object Model relates subtyping and multiple inheritance, which allow a subtype to have a multiple supertypes.

2.10 Metaclasses and Metaobject Protocol

A *metaobject* is an object that represents the type, operations, class, methods, or other Object Model entities of its corresponding object. The Core Object Model does not require existence of metaobjects to analyze object methods and attributes at runtime. In contrast, Java in its package `java.lang.reflect` provides such introspection classes (i.e. `Class`, `Method`, `Field`, etc., see Section 1.11). To solicit further standardization in this area, OMG recently issued the Meta-Object Facility RFP.

2.11 Introspection

CORBA allows CORBA objects to runtime analyze the methods, fields and inheritance hierarchy of IDL interfaces. In contrast to Java reflection, this mechanism does not inherently allow to check (and change) the state of a specific CORBA object (see Code example 9 for comparison).

3. Conclusion

This paper is based on the ANSI X3H7 Object Model Features Matrix [[ANSIX3H7](#)]. This matrix overviews several single as well as distributed object systems such as C++, Smalltalk, Eiffel, ActiveX, and CORBA. We use this feature matrix as a source of categories that we deploy for classifying and comparing both object models.

Before starting to conclude, it is worth to emphasize that roots and mission of CORBA and Java differ as an outcome of historical conditions and perhaps marketing strategy. Java is one-language system that only recently augmented its ability to process distributed code with the possibility to encompass distributed objects. CORBA, on the other hand, is built on the distributed object paradigm since its very childhood dating back to 1990. With its mission to provide a cross-language platform CORBA comes with the OMG IDL language that describe the syntax of remote services. Java does not need the notion of an interface language since Java remote invocations originate in one JVM and head to another. The invocation mechanism can

be thus fully based on the Java language definition as documented by specifying interfaces via the `interface` keyword.

Java is a one-language system and it hence makes complete sense to integrate the meta-language support by using the language itself. The Java reflection mechanism implemented by the `java.lang.reflect` package provides an elegant and self-contained solution. CORBA on the other hand must reflect its cross-language mission and offer a “hyper language” solution. Concepts of IDL and Interface Repository give the CORBA designer capability to browse through methods and attributes of CORBA objects stored in the Interface Repository and based on these information dynamically construct and send requests to these objects. To solicit integration of Java and CORBA, there are commercial tools that allow generating IDL files from Java class files [[VisiBroker](#)]. The Java reflection mechanism introspects all class fields and methods, including its superclasses. Given a Java object the reflection APIs may even excavate a value of a requested field. IDL idea is to provide only the language independent description of object interfaces and value. Given a CORBA object implemented in C++, there is no way to introspect values of its attributes and discover names of method and attributes that were used in object implementation in addition to the ones described in IDL. The Java reflection is thus more powerful, yet limited to Java applications only.

Generally, the CORBA object model is cleanly defined and stated; its taxonomy is clear and consistent. For Java, it is hard to identify a book or a document that would introduce the terminology of the Java Object Model with respect to categories listed in this paper. The Java Distributed Object Model is introduced in the [[SunJDOM](#)] document. However, this document is far away from providing the precise definition of the Java Distributed Object Model.

References

Object model:

[ANSIX3H7] **ANSI X3H7 Object Model Features Matrix**,
<http://info.gte.com/ftp/doc/activities/x3h7>

CORBA:

[Ben95] Ben-Nathan, R.: **A Guide to Common Object Request Broker Architecture**, McGraw-Hill, 1995

[MZ95] Mowbray, T.J., Zahavi R.: **The Essential CORBA**, J. Wiley & Sons, 1995

[Sie96] Siegel, J.: **CORBA: Fundamentals and Programming**, J. Wiley & Sons, 1996

[OMG] **OMG Homepage**, <http://www.omg.org>

[OMGOMA] **Object Management Architecture Guide**, 3rd Edition, R.M. Soley (Editor), J. Wiley & Sons, 1990

[OMGCORBA20]

Common Object Request Broker Architecture and Specification, Revision 2.0, OMG 96-3-4, 1995

[OMA95] **Object Management Architecture Guide**, 3rd Edition, R.M. Soley (Editor), John Wiley & Sons, 1990.

[OMGCore95] **Object Models**, Draft 0.3, OMG 95-1-13, 1995.

[OMAExecutiveOverview]

OMA Executive Overview, <http://www.omg.org/omaov.htm>

[OMGEvents] **Event Notification Model**, OMG 97-02-09,
<http://www.omg.org/corba/sectrans.htm>

[OMGObjectModel]

OMG Object Model, <http://www.omg.org/corba2/obmod2.htm>

Java:

[GJS96] Gosling, J., Joy B., Steele G.: **The Java Language Specification**, Addison-Wesley 1996.

[GM95] Gosling, J., McGilton, H.: **The Java Language Environment: A White Paper**, October 95, <http://www.javasoft.com/whitePaper/javawhitepaper.html>

[SunJavaBeans]

JavaBeans 1.0, Jul 97, <http://java.sun.com/beans>

[SunInnerCls] **Inner Classes Specification**, May 1997, <http://java.sun.com/innerclasses.doc>

[SunJDOM] **Java Distributed Object Model**,
<http://java.sun.com/products/JDK/1.1/docs/guide/rmi/rmi-objmodel.doc.html>

[SunJFC] **Java Foundation Classes: Now And The Future**,
http://java.sun.com/marketing/collateral/foundation_classes.html

[SunJNDI] **JNDI: Java Naming and Directory Services**, Verison 1.00-A,
<http://java.sun.com/products/jndi/index.html>

[VisiBroker] **Visigenic Home Page**, <http://www.pomoco.com>