CHAPTER FOURTEEN

The Sun way – Java, JavaBeans, EJB, and Java 2 editions

Java is a true phenomenon in the industry. It is one of the very few success stories of the 1990s – a new programming language, which, together with its own view of the world, really made an impact. Sun released Java in alpha form in early 1995 and, although it was hardly known and very little used before early 1996, it is now one of the most commonly used buzzwords. It seems that the time was ripe for the message that there is more to come in the way of programming languages than was at first commonly believed. However, it was not the language that attracted attention originally. Other, and perhaps better, languages had failed before. The real attractor was the concept of applets, mini applications that function within a web page.

This chapter presents a detailed account of the Java language, the Java component models that have been formed so far (JavaBeans, Servlets, Enterprise JavaBeans in four flavors, and J2EE Application Components), and a selection of the large number of relevant Java services. In line with the arguments of component safety by construction, the safety features of Java and their interaction with the Java security mechanisms are explored in detail. The introduction to the Java language is important because much of each Java component model directly builds on the properties of the language.

14.1 Overview and history of Java component technologies

As observed above, the applet was the initial factor of attraction and breakthrough for Java. Indeed, much of early Java was designed specifically to allow an untrusted downloaded applet to execute in the same process as a client's web browser without posing an unacceptable security threat. For this purpose, language Java was designed to allow a compiler to check an applet's code for safety. The idea is that an applet that passes the compiler's checks cannot be a security threat. As the compiled code can still be tampered with, it is again checked ("verified") at load-time. The verified applet is then known to be safe

and can thus be subjected to strong security policies. None of this would be possible with most of the established programming languages, including C++ and Object Pascal. Security policies can, of course, be enforced for languages that are usually interpreted, such as Smalltalk or Visual Basic. Java, however, was designed to allow for the compilation to efficient executables at the target site. This is done by so-called just-in-time (JIT) compilers.

As an aside, it is possible to base security policies on trust in applet vendors. Authentication techniques can be used to make sure that a received applet has not been tampered with and, indeed, comes from the announced vendor. If that vendor is trusted, the applet can be loaded, even if it has come in binary form. This is largely the approach taken by Microsoft's ActiveX ("Authenticode"). An obvious disadvantage is that no one will ever trust the large number of small developers providing the web community with applets. A combination of the two approaches is best and is now also supported by Java, known as "signed applets." If a downloaded applet is signed and authenticated as coming from a trusted source, it can be given more privileges than an applet that does not pass this test. The set of trusted sources is under the user's control.

The second winning aspect of Java is the Java virtual machine (JVM). Java compilers normally compile Java into Java byte code, a format that is understood by the JVM. By implementing the JVM on all relevant platforms, Java packages compiled to byte code are platform independent. This is a major advantage when downloading applets from the internet. The true advantage is not so much the JVM but the Java class-file and JAR archive formats (see section 17.3).

None of the advantages of Java are technically new. Safe languages existed before, including efficiently compilable ones (Reiser and Wirth, 1992). Virtual machines with byte code instruction sets go back to the early times of the Pascal p-machine (Nori *et al.*, 1981) and the Smalltalk 80 virtual machine (Krasner, 1983). Even the concept of object integration into the web had been demonstrated before. The principal achievement with Java was to pull it all together and release it in a very timely fashion.

14.1.1 Java versus Java 2

While the initial Java specification suite was much under the influence of the early applet idea, the Java 2 platform (introduced in late 1998) breaks free and relegates applets to a marginal side role. Java 2 introduced the notion of platform editions, which are selections of Java specifications that together address the concern of a particular class of Java users. Figure 14.1 shows the organization of the Java 2 space; more information on the Java 2 editions can be found in the following sections.

The flagship platform edition – J2EE (Java 2 platform, enterprise edition), first released end 1999 – is the most successful. With the specification of Enterprise JavaBeans (EJBs) at its heart, J2EE is the suite of specifications underlying a large

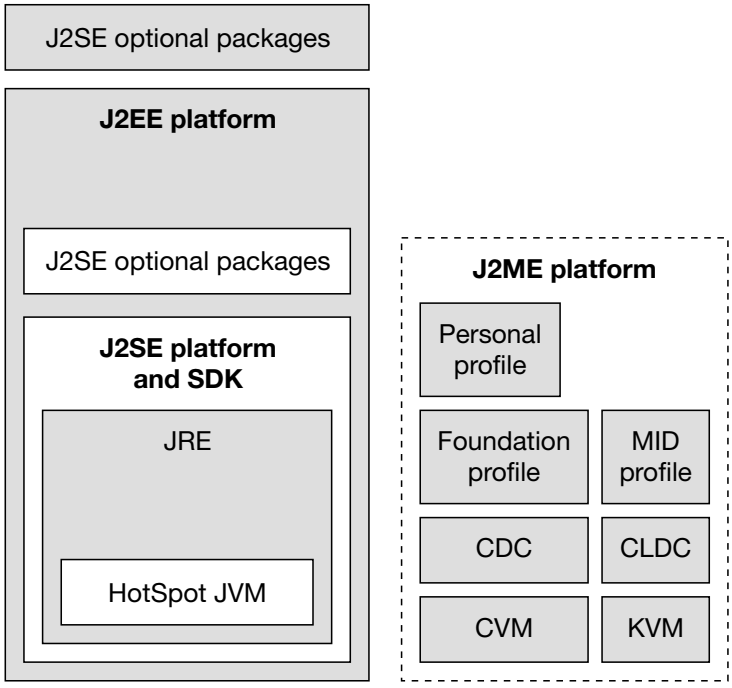

Figure 14.1 Organization of the Java 2 space.

number of application servers from a large number of different vendors. (The two largest such vendors are IBM, with its WebSphere products, and BEA, with its WebLogic products, but, by the end of 2001, there were almost 40 vendors listed on the Flashline.com comparative matrix, with prices running anywhere from free/open source to $75,000 per CPU.) The micro edition is also fairly successful, especially in the mobile phone sector. However, it is the enterprise edition that establishes a rich environment for component software, making it the most relevant with regard to the scope of this book.

Besides editions, Java 2 also formalized the notions of runtime environment (RE), software development kit (SDK), and reference implementation. Runtime environments are implementations of the JVM and the mandatory J2SE API specifications. An RE is typically paired with an SDK of a matching version that contributes development tools, including compiler and debugger. Confusingly, the 1.x numbering of the older "Java 1" specifications is continued for runtime environments and SDKs in Java 2. So, for example, a particular RE might be referred to as "Java 2 Runtime Environment, Standard Edition, v1.4."

14.1.2 Runtime environments and reference implementations

The Java runtime environment (JRE) is included in the J2SE platform, which itself is a subset of the J2EE platform. The JRE includes the runtime, core libraries, and browser plugin. Sun's reference implementation of the JRE 1.4 builds on its HotSpot runtime and HotSpot JIT compilers, which perform online re-optimization of JIT-compiled binary code. Separate HotSpot compilers are

available for client and server environments, respectively. They differ in their optimization target function with differing tradeoffs for memory footprint, startup-time, throughput, and latency. The Java SDK 1.4 includes JRE 1.4, as well as the Java compiler, debugger, platform debugger architecture APIs (JPDA), and a tool to generate documentation (javadoc). Figure 14.2 outlines the structure of the J2SE platform as of v1.4.

There is no standard RE or SDK for J2EE (beyond the included J2SE and J2SE options) as J2EE is meant to be a specification to be implemented by many vendors. Instead, Sun provides a reference implementation as a proof of concept and to clarify specifications. They are available in source code at a danger of making the reference implementation's exact semantics be the standard. Sun refers to this as the reference implementation serving as an

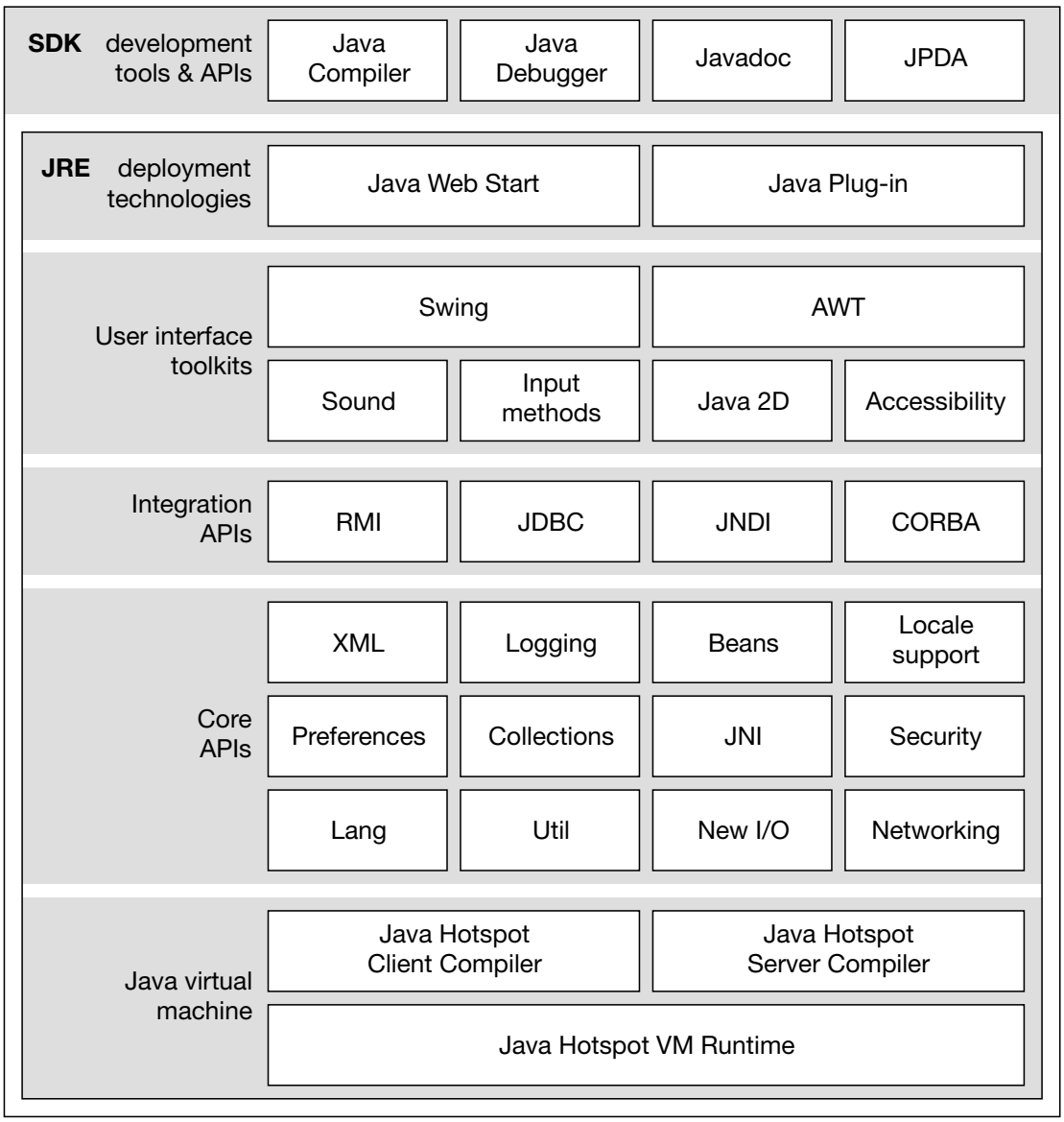

Figure 14.2 Organization of the Java 2 platform, Standard Edition v1.4. (From **java.sun.com**)

"operational specification." Reference implementations are not meant to be used as such – for example, tradeoffs are made in favor of clarity instead of performance. Instead, reference implementations ought to be used to verify that a proposed new implementation is indeed compliant.

The reference implementations are also useful to provide a complete environment for a new implementation of some Java standard. Compatibility test suites are provided to help validate implementations and these test suites themselves have been validated against the corresponding reference implementations. Java BluePrints is a collection of design guidelines and patterns that can be used to map certain classes of enterprise applications to the space of J2EE technologies.

14.1.3 Spectrum of editions – Micro, Standard, and Enterprise

Originally, Sun announced special packagings of Java to address the demands of various markets, such as Personal Java, Enterprise Java, Embedded Java, or Smartcard Java. With the arrival of the Java 2 restructuring of the overall organization of the Java specifications, this space has also been tidied up, so, as of mid 1999, Java 2 defines three packagings – the micro, standard, and enterprise editions. (The Java Smartcard space remained, catering for the particularly resource-constraint nature of these devices.) The micro edition is a family of specifications that aim at a spectrum of small and embedded devices (see Figure 14.1). At the foundation are a lightweight implementation of the full JVM (called CVM) and a trimmed-down JVM (called KVM – "K" stands for kilobyte, an indication of aiming for severely constrained devices). APIs on top of these two VMs are organized into configurations and profiles – cut-down sets of classes and interfaces appropriate for different device configurations. At the core is either the connected devices configuration (CDC) or the configuration for limited devices (CLDC). Device-specific profiles, such as the mobile information device profile (MIDP) augment the core configurations.

The standard and enterprise editions target the client-only and the multitier solution space, respectively. The standard edition is a proper subset of the enterprise edition. The micro edition does not establish any such upwards compatibility to other editions, acknowledging that most components will not usefully scale from such special small devices to more generic computing platforms. (JavaCard is addressing even smaller devices – smartcards – but it does not form a formal tier in the Java 2 classification.)

Java 2 Standard Edition

The size of J2SE has grown substantially over the first five years since its introduction in 1997. From 212 classes and interfaces in v1.0 (1997) on to 504, 1781, 2130, and now 3395 (in over 130 packages) in v1.4, released early in 2002. Included in v1.4 are 578 classes and interfaces defined by the Apache

group, W3C (DOM interfaces), and by the SAX project. J2SE 1.4 comprises 2606 classes and 789 interfaces, clustered into the following packages, grouped by functional areas.

- java.lang, java.lang.ref, java.lang.reflect Basic types of Java language, foundation classes for class loading and runtime reflection, support for various special references to partially control garbage collection, and extended reflection support.
- java.applet Applet base class and support types.
- java.awt, java.awt.color, java.awt.datatransfer, java.awt.dnd, java.awt.event, java.awt.font, java.awt.geom, java.awt.im, java.awt.im.spi, java.awt.image, java.awt.image.renderable, java.awt.print AWT (Abstract Windowing Toolkit); the original Java user interface construction framework; augmented and partially superseded by Swing for user interfaces and specialized printing packages.
- java.beans, java.beans.beancontext JavaBeans base classes and support types.
- java.io, java.nio, java.nio.channels, java.nio.channels.spi, java.nio.charset, java.nio.charset.spi Character and bytestream readers and writers, files and channels. Service provider interfaces (SPIs) for channels and character sets.
- java.math Augments the basic library of math functions in class java.util.Math with implementations of big integers and decimals.
- java.net, javax.net, javax.net.ssl Networking support – sockets, HTTP, SSL.
- java.rmi, java.rmi.activation, java.rmi.dgc, java.rmi.registry, java.rmi.server, javax.rmi, javax.rmi.CORBA Remote method invocation (RMI), late-bound construction (activation), distributed garbage collection, distributed service registry, server support, and RMI over CORBA IIOP.
- java.security, java.security.acl, java.security.cert, java.security.interfaces, java.security.spec, javax.security.auth, javax.security.auth.callback, javax.security.auth.kerberos, javax.security.auth.login, javax.security.auth.spi, javax.security.auth.x500, javax.security.cert, org.ietf.jgss Security manager, access control lists (ACLs), certificates, authentication, Kerberos, login, and X500 directory.
- java.sql, javax.sql Support for SQL-based access to relational databases.
- java.text Text manipulation.
- java.util, java.util.jar, java.util.logging, java.util.prefs, java.util.regex, java.util.zip Various utilities, Java archive (JAR) file access, logging, preferences, regular expressions, and ZIP file access.
- javax.accessibility Accessibility.
- javax.crypto, javax.crypto.interfaces, javax.crypto.spec Cryptography – MD5, SHA-1, DES.
- javax.imageio, javax.imageio.event, javax.imageio.metadata, javax.imageio.plugins.jpeg, javax.imageio.spi, javax.imageio.stream Image I/O support.

- javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap, javax.naming.spi Naming and directory access; LDAP.
- javax.print, javax.print.attribute, javax.print.attribute.standard, javax.print.event Printing support.
- javax.sound.midi, javax.sound.midi.spi, javax.sound.sampled, javax.sound.sampled.spi Sound support; MIDI, sampled sound.
- javax.swing, javax.swing.border, javax.swing.colorchooser, javax.swing.event, javax.swing.filechooser, javax.swing.plaf, javax.swing.plaf.basic, javax.swing.plaf.metal, javax.swing.plaf.multi, javax.swing.table, javax.swing.text, javax.swing.text.html, javax.swing.text.html.parser, javax.swing.text.rtf, javax.swing.tree, javax.swing.undo Swing user interface construction framework, including platform-specifc and Java-generic look-and-feel, undo/redo, HTML and RTF support.
- javax.transaction, javax.transaction.xa Transaction support; XA.
- javax.xml.parsers, javax.xml.transform, javax.xml.transform.dom, javax.xml.transform.sax, javax.xml.transform.stream, org.w3c.dom, org.xml.sax, org.xml.sax.ext, org.xml.sax.helpers XML parsing, XSLT, DOM 2, SAX.
- org.omg.CORBA, org.omg.CORBA_2_3, org.omg.CORBA_2_3.portable, org.omg.CORBA.DynAnyPackage, org.omg.CORBA.ORBPackage, org.omg.CORBA.portable, org.omg.CORBA.TypeCodePackage, org.omg.CosNaming, org.omg.CosNaming.NamingContextExtPackage, org.omg.CosNaming.NamingContextPackage, org.omg.Dynamic, org.omg.DynamicAny, org.omg.DynamicAny.DynAnyFactoryPackage, org.omg.DynamicAny.DynAnyPackage, org.omg.IOP, org.omg.IOP.CodecFactoryPackage, org.omg.IOP.CodecPackage, org.omg.Messaging, org.omg.PortableInterceptor, org.omg.PortableInterceptor.ORBInitInfoPackage, org.omg.PortableServer, org.omg.PortableServer.CurrentPackage, org.omg.PortableServer.POAManagerPackage, org.omg.PortableServer.POAPackage, org.omg.PortableServer.portable, org.omg.PortableServer.ServantLocatorPackage, org.omg.SendingContext, org.omg.stub.java.rmi Standard CORBA interfaces and ORB binding.

An interesting category of packages are so-called service provider interfaces (SPIs). These are of interest to implementers of a particular service, rather than to clients. The idea is that a Java standard framework sits between such service providers and clients and mediates. Examples are the SPIs for I/O channels, character sets, authentication, and naming.

Java 2 Enterprise Edition (J2EE)

J2EE includes all of J2SE, including optional packages. Since, in early 2002, J2EE 1.4 had not yet been released, the following numbers cover J2EE 1.3 and, thus, J2SE 1.3. In version 1.3, J2SE comprises 2606 classes and 789 interfaces. To that J2EE 1.3 adds 243 classes and 198 interfaces. J2EE also includes 279 classes and 182 interfaces based on CORBA specifications as well as 673 classes and 92 interfaces contributed by a number of Apache projects. All this adds up to a total of 3801 classes and 1261 interfaces, or a grand total of 5062 types.

Table 14.1 shows the required support of various Java standards in the J2EE versions 1.2.1 and 1.3 (Sun, 2001). Most notably, EJB 2.0 (with its substantial improvements over 1.1) is required and Connector, JAAS, and JAXP have been added to the list of requirements. JNDI and RMI-IIOP are no longer specific to J2EE as they have been reclassified as J2SE optional packages, although they are required in J2EE. As of 1.3, J2SE also includes Java IDL and JDBC Core. The JDBC 2.0 Extensions are optional in J2SE, but required in J2EE. As can be seen in the table, there is a separate evolution of individual Java specifications and the umbrella platform specifications. As long as each individual specification maintains perfect backwards compatibility that is not a major issue.

At the heart of the J2EE architecture is a family of component models. To be precise, there are three groups of component models with a total of nine variations on the component theme, which are, on the client side, application components, JavaBeans, and applets; on the web server tier, servlets and JSPs;

Table 14.1 Matrix of specification versions under two consecutive J2EE platform versions.

J2EE	1.2.1	1.3.1	Interfaces	Classes
J2SE	1.2	1.**3**	789	2606
EJB	1.1	**2.0**	17	11
JAF	1.0	1.0	4	13
JavaMail	1.1	1.**2**	12	85
JDBC	2.0 Ext.	2.0 Ext.	12	2
JMS	1.0	1.0	43	15
JNDI	1.2	(1.2)		
JSP	1.1	1.**2**	6	19
JTA	1.0	1.0	7	10
RMI-IIOP	1.0	(1.0)		
Servlets	2.2	2.**3**	20	16
Connector	–	1.0	24	13
JAAS	–	1.0	5	23
JAXP	–	1.1	39	36
			189	**243**

on the application server tier, EJB in four variations (stateless session, stateful session, entity, and message-driven beans). Using different kinds of components (rather than just different components) is a mixed blessing. On the one hand, it helps to have component models that fit well with a particular architectural area. On the other hand, it makes it harder to refactor an overall system as boundaries of component models may need to be crossed. It is also somewhat unclear how such a fine and diverse factoring of component models caters to the evolution of overall systems architecture. Will the number of J2EE component models continue to grow?

The J2EE architectural overview in Figure 14.3 separates the areas that J2EE supports using specialized component models. Figure 14.3 does not mention JavaBeans. The reason is that JavaBeans components, although preequipped with options to perform in the user interface space, aren't really confined to any particular tier. Instead, JavaBeans and its core technologies could be used in almost any of the spaces shown in the figure. Furthermore, note that the arrows in the figure represent characteristic cases of control flow. They are not meant to be complete. Data flow typically follows the same lines, but in both directions. A binding substrate underpinning all parts of a J2EE system is the naming and directory infrastructure accessible via JNDI (Java naming and directory interface). A second integration plane is the messaging infrastructure accessible via JMS (Java message service). Both JNDI and JMS

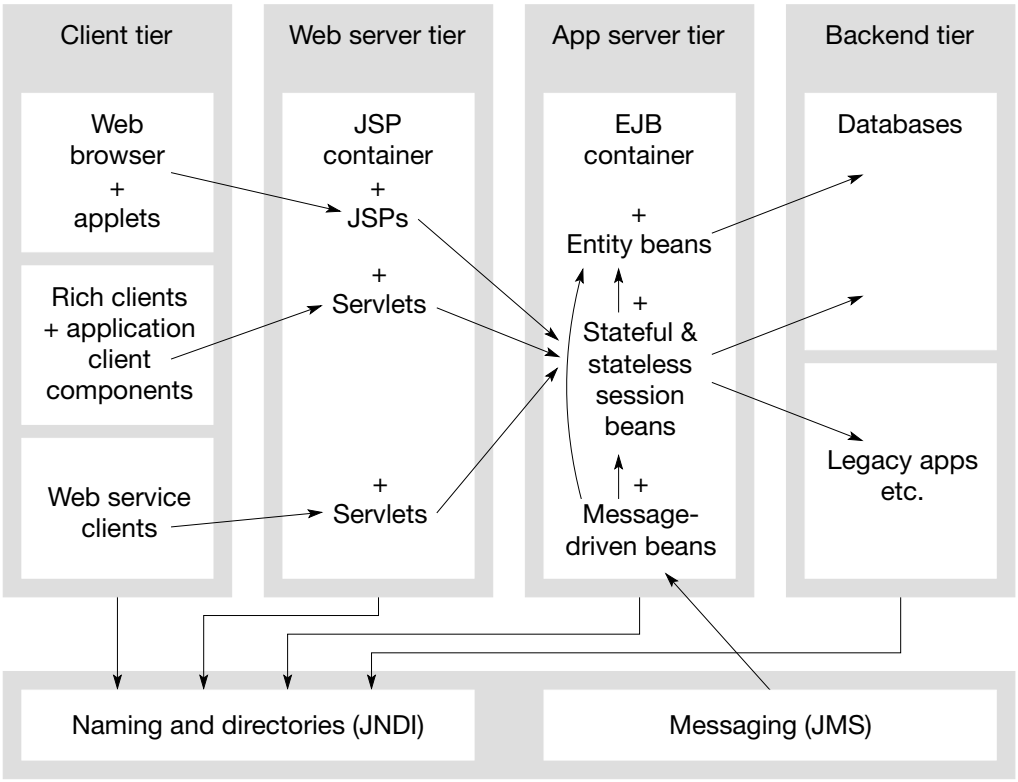

Figure 14.3 Architectural overview of J2EE.

are discussed in section 14.6.3. With the help of message-driven beans in an EJB container, messages can trigger processing on arrival. Messaging and naming/directories are two important integration layer services, but there are also others, such as transactional coordination and security services, that are not covered in Figure 14.3 for the sake of clarity. They are, however, discussed under the heading of enterprise services later.

A detailed discussion of JavaBeans follows in section 14.3. Basic services and advanced services for JavaBeans are discussed in section 14.4. Switching to the enterprise scale, section 14.5 comprises a discussion of the EJB component models. Some details of the most important enterprise services follow in section 14.6.

14.2 Java, the language

Java is an almost pure object-oriented language. It makes some admissions – not everything is an object, for example. All Java code resides in methods of classes. All state resides in attributes of classes. All classes except Object inherit interface and implementation from exactly one other class. Non-object types are the primitive types (boolean, byte, char, short, int, long, float, double) and interface types. Objects are either instances of classes or arrays. Objects can be created but not explicitly deallocated (Java uses an automatic garbage collector for safety). A class can implement any number of interfaces and interfaces can be in a multiple interface inheritance relationship.

All Java classes and interfaces belong to packages. (As of Java 1.1, a class can also be nested inside another class or even inside a method.) Packages introduce a level of encapsulation on top of that introduced by classes. The default mode for access protection allows arbitrary access across classes in the same package. Packages form a hierarchy, but only the package immediately enclosing a class or an interface affects encapsulation and protection. Outer packages merely serve to manage namespaces. All naming in Java is relative to the package name paths formed by appending the names of inner packages to those of outer packages. It is recommended that a globally unique name be used for a top-level package, a typical candidate being a registered company name.

A fully qualified name of a feature of a class takes the form package.Type.feature. Here, package is the path name of the package, Type is the name of the class or interface, and feature is the name of the method or attribute referred to. If internet domain names are used, a package name can take a form such as org.omg.CORBA. Although originally recommended, it is now commonplace to use company-name.productname prefixes for packages, relying on the stability and uniqueness of company instead of domain names. To use another package, the using package can always use fully qualified names to refer to the other package. However, such names can become quite lengthy. By explicitly importing some or all the types of the other package, these type names become directly available in the importing package – Type.feature then

suffices. The unfortunate use of the period to separate all package name particles, top-level type names, nested type names, and features makes the Java notation a.b.c subtle to parse and possibly difficult to understand to the human reader. (C# shares this unfortunate design.)

Figure 14.4 shows the graphical notation used here for Java packages, interfaces, classes, attributes, and methods. The notation follows UML static structure diagrams with a non-standard extension for final classes, which are those that cannot be further extended – that is, they cannot be inherited from. In addition, and not shown in the notational overview, Java methods can be final (not overridable) or static (class instead of instance methods). Attributes can also be final (immutable after initialization) or static (class instead of instance attributes). Interfaces extend any number of other interfaces. Classes extend exactly one other class, except for class java.lang.Object, which has no base class. Classes can also implement any number of interfaces.

Figure 14.5 uses the concrete example of (part of) the java.awt.image package and its relation to the java.lang package. Note that java.awt.image is a subpackage of java.awt. As stated before, there is no special relationship between a package and a subpackage that goes beyond the structure added to the namespace. In particular, both java.awt and java.awt.image contain interfaces and classes.

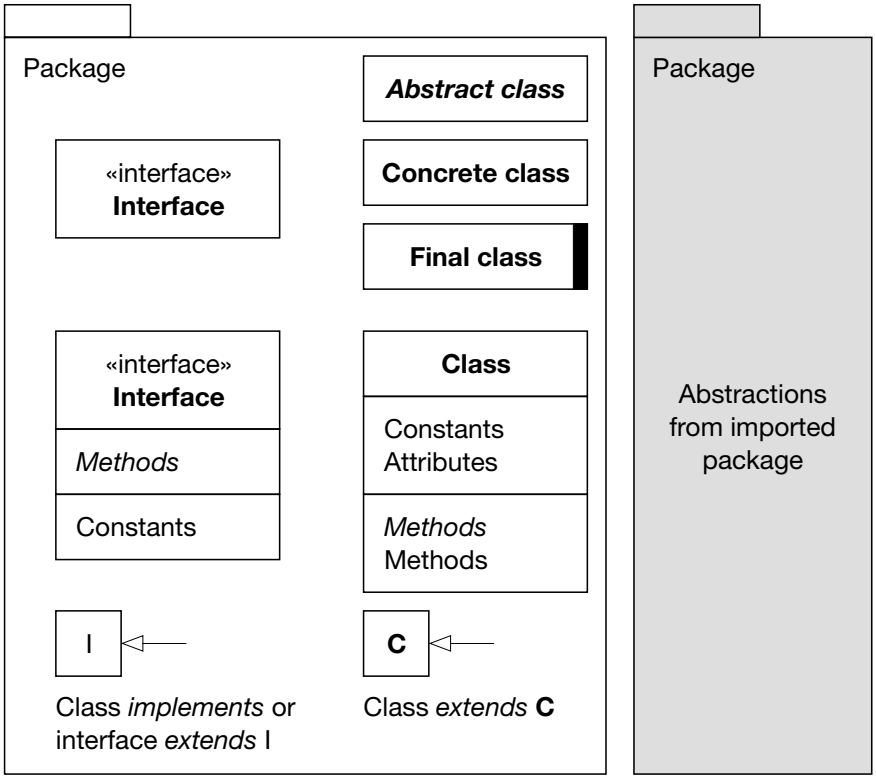

Figure 14.4 Java structuring constructs – names of abstract classes and abstract methods are set in italics.

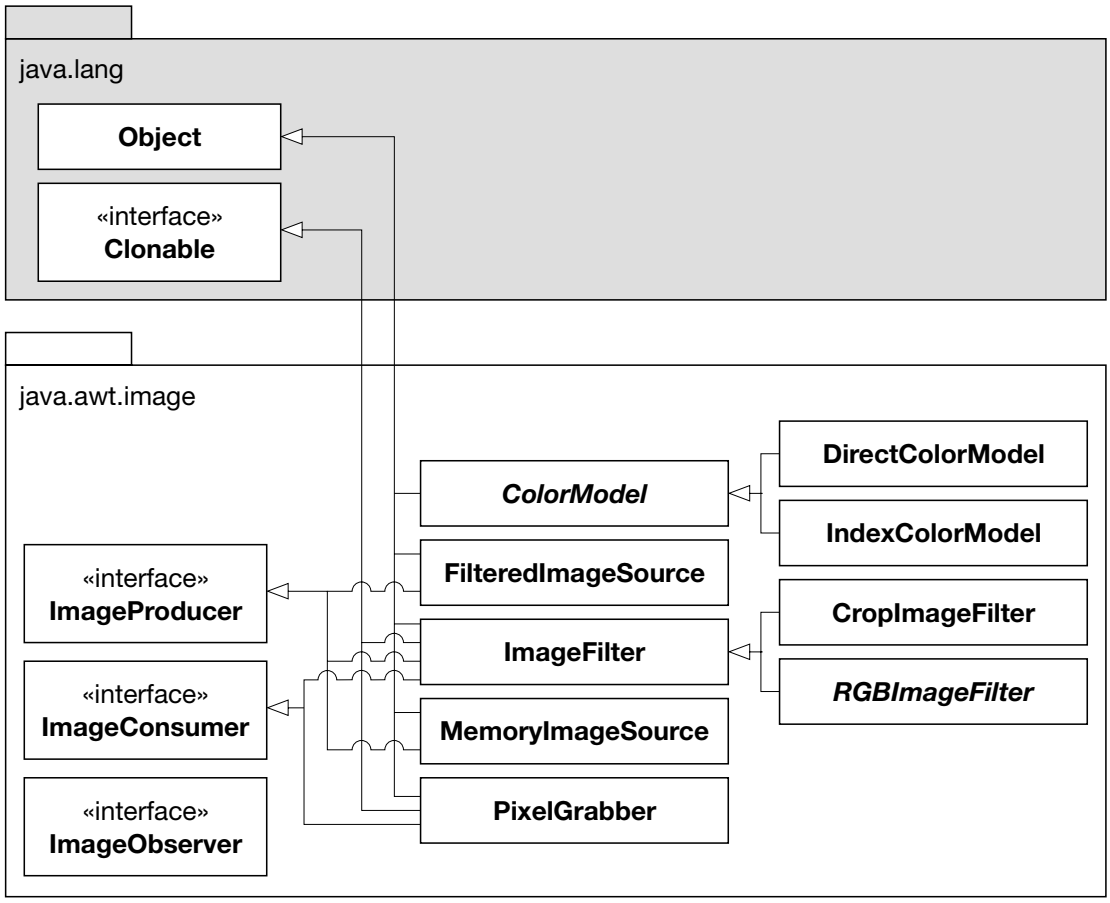

Figure 14.5 Part of the java.awt.image package.

The line between the Java language and some of the fundamental class libraries is not easy to draw. In particular, there are three packages required by the Java language specification (Gosling *et al.*, 1996) – java.lang, java.util, and java.io. The package java.lang is even deeply intertwined with the language specification itself. Many classes and interfaces in this package receive special treatment by the language rules. It would be fair to say that the otherwise modestly sized language Java really includes the definition of java.lang. Unfortunately, in JDK 1.4, java.lang alone defines 82 classes and interfaces, all of which are available in any Java package automatically.

The interweaving of language and standard package constructs is worse than it seems. Not only are all interfaces and classes of package java.lang automatically injected into any other package, but also the language itself builds on some java.lang classes. In particular, these are the classes Object, String, and Throwable, which are entangled with the null reference, the array type constructors, the string literals, and the throw statement respectively. Under the transitive closure over the types used in the signatures of Object, String, and Throwable, the list expands to include classes Class, ClassLoader, and several subclasses of Throwable.

Because Java supports arbitrary and possibly circular dependencies across packages, java.lang can and does have dependencies on other packages. This has been the case since the first release and has worsened in all following releases. The class Throwable relies on types java.io.PrintStream and java.io.PrintWriter, in a debugging method called printStackTrace. This method indirectly causes the classes PrintStream, PrintWriter, FilteredOutputStream, Writer, OutputStream, and IOException from package java.io to be pulled into the language itself. More such dependencies were woven into later versions. So, for example, in J2SE 1.4, java.lang.ClassLoader depends on java.security.ProtectionDomain, java.lang.Process depends on java.io.InputStream, and java.lang.SecurityManager depends on java.net.InetAddress, java.io.FileDescriptor. These pull in many other definitions. In JDK 1.4, the transitive closure of all classes and types directly reachable from the Java language specification contains 146 classes and interfaces!

Java defines two access levels for top-level interfaces and classes – default (package-wide) and public. In the former case, the interface or class is accessible only within its defining package, not even within sub- or superpackages, but in the latter case it is accessible everywhere. The default limitation to intra-package accessibility is not enforceable by the language implementation – anyone can drop a new definition into an existing package. The definition and enforcement of access rights for entire packages is left to the environment. Access rights can be further controlled on the level of class features – that is, nested types, methods and fields. Java defines four levels of static access protection for features of a class, which are private, default (package-wide), protected, and public. Private features are accessible only within their defining class. Default accessibility includes all classes within the same package. Protected features can, in addition, be accessed within all subclasses of the defining class. Finally, public features are globally accessible, provided the defining class is. (Environment-enforced access constraints may exclude access even if the respective definition is declared public.)

Fields can be final, effectively making them constants once initialized. Final methods cannot be overridden. Static features belong to the class rather than to instances of the class. Interface fields are implicitly public, static, and final, making them global constants. Interface methods are implicitly public and abstract. With these restrictions enforced, Java interfaces introduce neither state nor behavior – they are pure types.

14.2.1 Interfaces versus classes

Probably the most innovative aspect of Java is its separation of interfaces and classes in a way that permits single implementation inheritance combined with multiple interface inheritance. This separation eliminates the diamond inheritance problem (p. 111) while preserving the important possibility of a class being compatible with multiple independent interfaces. The diamond inheritance problem is

eliminated because interfaces introduce neither state nor behavior. Thus, there can be no conflicts beyond name clashes in a class that implements multiple interfaces.

Java offers no complete solution to the name conflict problem. If two interfaces introduce methods of the same name and signature but of different return type, then no class can simultaneously implement both interfaces. Clashes of method names among independently defined interfaces are difficult to avoid entirely, but don't seem to pose a significant problem in practice. One reason is that many method names contain the interface name – generally an unfortunate choice. A different scenario, however, literally forces such name clashes. If an interface is versioned, it is likely that the old and new version will share some method names. As pointed out in section 5.1.2, it is essential that components support sliding windows of versions – that is, that they support the interfaces of multiple version generations. In Java, doing so requires us to change the name of an interface for a new version *and* change the name of all methods logically retained in the new version. This requirement is cumbersome enough that it is not part of best practice. Java classes are thus not normally able to simultaneously support multiple versions of their contracts.

Leaving this quite fundamental problem of Java aside, there are other potential pitfalls for designers using a mix of single class and multiple interface inheritance. These are not limited to Java and equally apply to other languages following a similar design paradigm, such as C#. Consider the following example involving both classes and interfaces. The standard package java.util defines the class Observable and the interface Observer, as shown below.

```
package java.util;
public class Observable extends Object {
  public void addObserver (Observer o);
  public void deleteObserver (Observer o);
  public void deleteObservers;
  public int countObservers ();
  public void notifyObservers ();
  public void notifyObservers (Object arg);
  protected void setChanged ();
  protected void clearChanged ();
  public boolean hasChanged ();
}

public interface Observer {
  void update (Observable o, Object arg);
}
```

The idea is that observable objects are instances of classes that extend (inherit from) Observable. Class Observable maintains a list of observers and a flag indicating whether or not the observable object has changed. Observers have

to implement interface Observer. All registered observers will be called when the observable object calls notifyObservers. Figure 14.6 shows a simple class diagram using Observable and Observer to implement a straightforward model view scheme. (The asterisk at the end of the line, from Observable to Observer, is the UML notation showing that one Observable instance can be associated with many observers.)

Observable and Observer aim to support the observer design pattern (Gamma *et al.*, 1995). This pattern is most prominently used in conjunction with model view separations in user interfaces (see Chapter 9). The Observer interface certainly demonstrates a good use of interfaces in Java. However, the Observable class has a design flaw. (Observer and Observable are not used in any of the standard Java packages, although the definitions have never been deprecated. They were superseded by the event and event-listener concepts introduced with the JavaBeans specification described in section 14.3; Sun, 1996.)

The problem is that Observable is a class and not an interface. This forces restructuring of a class hierarchy in cases where a class already depends on a superclass that is different from Object and where instances of this class now should become observable. Why was this done? Observable as it stands does too much. It contains a concrete implementation of the list of observers and the notification broadcast. It also implements an "object has changed" flag. (This flag itself has a few design problems. For instance, at least up to v1.4, the specification states that it is cleared after notifying observers, so re-entrant changes are not notified and no exception is thrown either. In reality, since v1.2, the implementation clears this flag before notifying observers. Another problem is that Observable isn't synchronizing. Concurrent threads may enter race conditions that can lead to inconsistent handling of notifications.)

A simple solution is not to use the Observer/Observable approach. However, to study some further aspects of Java, assume that the interface Observable should be used, but that the observable object already extends some other class. Class Observable cannot therefore be used directly. The astute reader will notice that, again, using object composition can save the situation. To do so, a trivial subclass of Observable needs to be implemented:

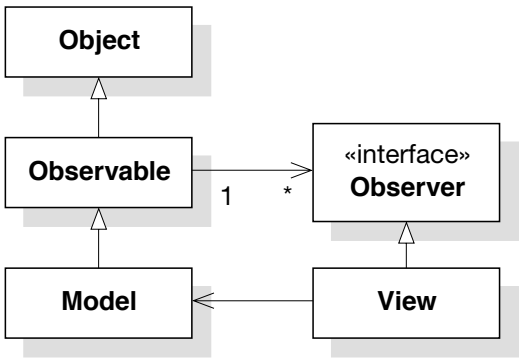

Figure 14.6 Model view construction using Observable and Observer.

```
package fix.it;
public class ObserverRegistry extends java.util.Observable {
  public readonly Object observable;  // reference to actual observable object
  public ObserverRegistry (Object observable) {   // constructor
    this.observable = observable;
  }
  public void setChanged() {
    super.setChanged()
  }
  public void clearChanged() {
    super.clearChanged()
  }
}
```

This trivial subclass removes the protected access mode from methods setChanged and clearChanged. Instances of ObserverRegistry can now be used as part objects of observable objects. Because observers get a reference to an Observable, ObserverRegistry's read-only field observable provides a reference to the object that has actually changed. The registry's constructor sets this reference. For example:

```
package fixed.it;
import fix.it.*;
public class StringBufferModel extends StringBuffer {
  private ObserverRegistry obReg = new ObserverRegistry(this);
  public ObserverRegistry getObserverRegistry () {
    return obReg;
  }
  public void setLength (int newLength)
      throws IndexOutOfBoundsException
  {
    super.setLength(newLength);
    obReg.setChanged();
    obReg.notifyObservers();  // could tell them what has changed ...
  }
  // other StringBuffer methods
}
```

Class StringBufferModel extends the standard class java.lang.StringBuffer, a helper class to support composition of strings. As a string buffer is a mutable object, it is a good candidate for a model-view scenario. For example, a text entry field could be implemented as a view on a string buffer. Unfortunately, StringBufferModel cannot extend Observable directly, as it already extends StringBuffer. StringBuffer was not designed to serve as the superclass of something observable and so extends Object instead of Observable. StringBufferModel solves this by using an ObserverRegistry object instead. Below is the sketch of an observer object displaying a StringBufferModel object's contents.

```
package fixed.it;
import java.util.*;
public class StringBufferView implements Observer {
  private StringBufferModel buf;
  public StringBufferView (StringBufferModel b) {
    buf = b;
    b.getObserverRegistry().addObserver(this);
  }
  public void update (Observable o, Object arg) {
    ObserverRegistry reg = (ObserverRegistry)o;   // checked cast
    Object observable = reg.getTrueObservable();
    if (observable != buf) throw new IllegalArgumentException();
      ... // get changes from buf and update display
  }
}
```

This concludes the example on how to use interfaces and object composition in Java. The example showed only classes implementing a single interface, but the extension to multiple interfaces is straightforward. Where unresolvable naming conflicts occur, a Java compiler rejects a class trying to implement conflicting interfaces. In all other cases there is no interference whatsoever between multiple implemented interfaces. One question remains to be answered, though. How can a client having a reference of a certain interface or class type find out what other interfaces or classes the referenced object supports? The answer is two-fold. Java provides a type-test operator – instanceof – that can be used to query at runtime. Java also has checked type casts that can be used to cast the reference to another class or interface. Here is an example:

```
interface Blue { ... }
interface Green { ... }
class FunThing implements Blue, Green { ... }
```

A client holding a Blue reference can test whether or not the object also supports a Green interface, although Blue and Green themselves are totally independent types:

```
Blue thing = new FunThing ();
Green part = null;
if (thing instanceof Green)   // type test
  part = (Green) thing;   // checked type cast, from Blue to Green
```

It is not possible to statically state that a variable refers to an object implementing both Blue and Green without introducing a common subtype of these two interfaces. The late introduction of such a common subtype for this purpose (say, Cyan extends Blue, Green) does not solve the problem. For instance, class FunThing above should qualify (it implements both Blue and Green), but doesn't as it does not implement the artificially introduced common subtype

Cyan. What is needed is a notion of requiring structural compatibility with a set of interfaces. Büchi and Weck (1998) call such interface sets compound types and identify non-support of compound types as a common problem in all contemporary mainstream languages.

14.2.2 Exceptions and exception handling

The Observer/Observable example above introduced another feature of Java – exception handling. Exceptions and runtime errors are reflected in Java as exception or error objects. These are either explicitly thrown, such as the throw statement in StringBufferView, or thrown by the runtime system, such as on out-of-bounds indexing into an array. All exception and error objects are instances of classes that are derived from class Throwable. A catch branch of a try statement can catch such throwable objects. For example, the notifyObservers method in class Observable could be protected against exceptions raised by notified observers:

```
public void notifyObservers () {
  for (Observer ob = first(), ob != null, ob = next()) {
    try {
      ob.update(this, null);   // observer may throw an exception
    }
    catch (Exception ex) {
      // ignore exception and continue with next observer
    }
  }
}
```

In the example, it is not apparent just by looking at the declaration of method update that it might throw an exception. Exception types for which this is legal are called unchecked exceptions. Java also has exception types that can only be thrown by a method's implementation, if the method declaration announced this possibility. Such declared exceptions are called checked exceptions. For example:

```
class AttemptFailed extends Exception { }
class Probe {
  public void goForIt (int x) throws AttemptFailed {
    if (x != 42) throw new AttemptFailed();
  }
}
class Agent {
  public void trustMe (Probe p) {
    p.goForIt(6 * 7);   // compile-time error!
  }
}
```

Class Agent cannot be compiled. Method trustMe neither declares that it may throw the user-defined exception AttemptFailed nor catches this exception should it be thrown. Note that this check based on static annotations adds a degree of robustness to software. In the example, Agent happens to "know" what Probe needs to avoid failure. Hence, Probe will actually never throw an exception when called from within trustMe. However, a future version of Probe may do so. After all, this possibility has even been declared and is thus part of the contract between Probe and Agent.

14.2.3 Threads and synchronization

Java defines a model of concurrency and synchronization and is thus a concurrent object-oriented language. The unit of concurrency is a thread. Threads are orthogonal to objects, which are therefore passive. Threads can only communicate through side-effects or synchronization. A thread executes statements and moves from object to object as it executes method invocations. Threads can be dynamically created, pre-empted, suspended, resumed, and terminated. Each thread belongs to a thread group specified at thread creation time. Threads belonging to separate thread groups can be mutually protected against thread state modifications. Threads can be assigned one of ten priority levels and can be in either user or demon mode. A thread group may impose limits on the maximum priority of its threads. However, priorities are hints only – some virtual machines ignore them while others differ in their interpretation. Thread termination is not automatically propagated to child threads. A Java application terminates when the last user thread has terminated, irrespective of the state of demon threads. Figure 14.7 shows the states in which a thread can exist and the possible state transitions. The states and transitions are explained below.

Java's threads are lightweight as, on any given virtual machine, they all execute in the same address space. A JVM is free to schedule processing resources by pre-empting executing threads at any time, but JVMs don't have to implement time-slicing. (If progress depends on scheduling, a thread should call Thread.yield – see below.) As threads may be pre-empted, this raises the question of how concurrent access is regulated. The language specification requires atomic ordering of access operations on values of primitive types and reference types. (As a concession to efficient implementation on contemporary 32-bit architectures, there are no such requirements for Java's 64-bit types long and double. However, long and double variables can be declared volatile to prevent interleaved access.) For consistent access to larger units, such as multiple fields of an object or multiple objects, explicit synchronization is required.

Java supports thread synchronization either on entrance to a synchronized method or at a synchronized statement. Synchronization forces a thread to acquire a lock on an object before proceeding. There is exactly one lock associated with each object. If the thread in question already holds a lock on that object, it can continue. (This is an important rule avoiding deadlocks in a

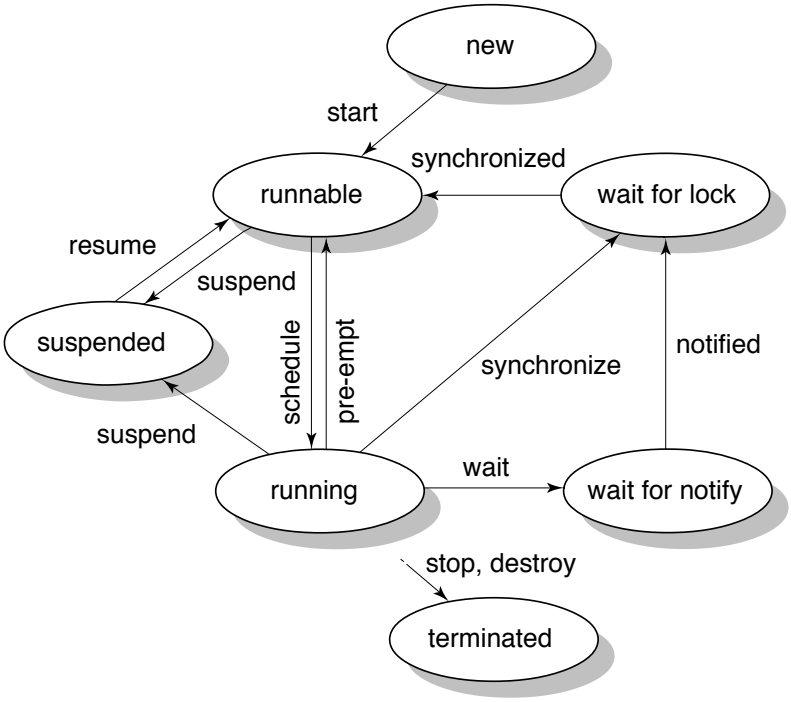

Figure 14.7 States and state transitions of Java threads.

common situation – a thread re-entering an object. Cardelli introduced this approach to avoid what he called self-inflicted deadlocks, 1994.) Synchronized methods use the lock of the object to which they belong. Below is an example of a wrapper for the unsynchronized stacks provided by the standard library class java.util.Stack:

```
class SynchStack {
  private java.util.Stack stack = new java.util.Stack();
  public synchronized void push (Object item) {
    stack.push(item);
  }
  public synchronized Object pop () {
    return(stack.pop());
  }
  public synchronized boolean empty () {
    return(stack.empty());
  }
}
```

The synchronized statement takes the object to be locked as an argument:

```
... // thread may or may not already hold a lock on obj
synchronized (obj) {
  ... // thread now holds lock on obj
}   // lock released, unless it was already held by current thread
```

Without any further measures, a thread waiting for an event would have to poll variables. To avoid inefficiencies, Java allows threads to wait on any object, either indefinitely or up to some specified timeout. For this reason, class Object defines methods wait, notify, and notifyAll. These methods can only be called by a thread that holds a lock on that object. While waiting, the lock is released. On notification, a thread first reacquires the lock before proceeding – by doing so it has to compete with all other threads trying to acquire the lock. Normally, the lock is still held by the notifying thread. Once the lock becomes available, any one of the threads trying to acquire the lock will be chosen. If multiple threads have been notified (or even all that have been waiting by using notifyAll) or if other threads attempt to acquire the same lock, then they will compete for the lock and acquire it, in any order, as soon as the previous owner of the lock releases it.

There is, however, a significant and dangerous subtlety attached to the use of objects as synchronization signals. If an object used for synchronization is accessible to threads from multiple subsystems, then it is unlikely that these threads agree on the semantics of wait/notify on that object. As a result, the same object may end up being used accidentally as a synchronization object by two subsystems. This is especially likely when using a global object. For instance, a widespread pattern uses the meta-object attached to some type for purposes of locking and synchronization. Another danger is that an object is used for conflicting locking and synchronization purposes at base class and derived class levels. To avoid these problems, it is recommended to predominantly use objects that aren't widely accessible for synchronization.

A thread is associated with an owning object at thread creation time. It is through this object that a thread can be suspended, resumed, or terminated. The owning object can also be used to move a thread between user and demon status before the thread is first started. Objects that own a thread are instances (of a subclass) of class java.lang.Thread. Below is an excerpt:

```
public class Thread implements Runnable {
  public static Thread currentThread ();
  public static void yield ();
  public static void sleep(long millis) throws InterruptedException;
  public Thread ();                          // default constructor; use method run
  public Thread (Runnable runObject);        // use method run of runObject
  public void run ();
  public void start ();
  public void stop ();                       // deprecated
  public void suspend ();                    // deprecated
  public void resume ();                     // deprecated
  public void interrupt ();
  public static boolean interrupted ();
```

```
    public boolean isInterrupted ();
    public void setPriority (int newPriority);
    public void setDaemon (boolean on);
    public final boolean isDaemon ();
    public void destroy ();                    // unsafe, not implemented
    public final void checkAccess ();          // check caller permissions
    // other methods deleted ...
}
```

A thread object can itself define the outermost method to be executed by the new thread – method run in class Thread. Alternatively, the thread object is constructed with a reference to an object that implements interface Runnable. In the latter case, thread execution starts by calling the run method of that other object. Either way, the thread terminates if the outermost method returns. A thread also terminates when its outermost method throws an exception.

The definition of class Thread also contains means to control a thread. Static methods are provided for a thread to affect itself by yielding the processor or by sleeping for some set time. Threads that are waiting for a lock or sleeping can be interrupted by another thread, which causes an InterruptedException to be thrown. If a thread is interrupted while not waiting or sleeping, its interrupted flag is set, which can be polled by calling isInterrupted or polled-and-cleared by calling interrupted.

A thread can be terminated by calling its owning object's stop or destroy methods. (Method stop sends a ThreadDeath exception to the thread, whereas destroy terminates the thread immediately and with no clean-up. Objects locked by a destroyed thread remain locked. This is the last resort in case a thread catches ThreadDeath and thereby refuses to terminate.) However, all these methods have been deprecated with v1.2, and one of them has actually never been implemented (destroy). The reason for this deprecation is that these methods are unsafe. Suspending a thread can lead to unavoidable deadlocks if the suspended thread holds a lock that the suspending thread needs to acquire. Stopping a thread causes that thread to release all its locks, potentially leaving the previously locked objects in an inconsistent state. Destroying a thread would leave arbitrary locked objects locked for good.

As of 1.2, the recommendation is to use polling of variables to signal suspension or termination requests. Long-running threads should poll such a variable periodically and, if set, gracefully return to a consistent state and release locks. This is a form of cooperative multitasking that requires the programmer to have substantial insight at a global level.

14.2.4 Garbage collection

Java provides a number of mechanisms to allow programs to interact with the JVM's garbage collector. The first such mechanism is object finalization. It is used by overriding protected method Object.finalize. This method is called by

the collector at some time after an object has been found to be unreachable, but before its memory is actually reclaimed. Unless it is known with certainty that the superclass doesn't have a non-empty finalizer (versioning fragility!), a finalize method should always call the super-finalize method. Unlike object constructors, this chaining is not enforced by the Java language.

The implementation of a finalize method can reach objects that are otherwise unreachable (including its own object!), but it may also reach objects that are still reachable. By assigning a reference from one of the previously unreachable objects to a field of one of the still reachable objects, a previously unreachable object can be "resurrected." That is, after the call to the finalize method, some previously unreachable objects may again be reachable. To prevent endless finalization efforts, the Java garbage collector will call the finalize method of an object exactly once before reclaiming its storage. (The "exactly once" semantics is only guaranteed if an object does become unreachable and if the JVM doesn't fail catastrophically.)

The best finalization strategy is thus to avoid resurrection and to only use finalizers to release external resources. For instance, if an object holds some platform handle to, say, an open database connection, then a finalizer can be used to ensure that the connection is closed eventually. However, note that this guarantee is still fairly weak as there is no guarantee how long it will take before the JVM detects unreachability and before it calls a finalizer. A garbage-collection pass can be "encouraged" by calling System.gc. A best-effort finalization pass can be requested by calling System.runFinalization(). If used with care at strategic points, this can help ensure swift release of critical resources. However, forcing finalization effectively forces partial garbage collection (to detect unreachability) and can thus be expensive. The negative impact on overall performance can be dramatic.

The finalize methods are called on an unspecified thread that is merely guaranteed to not hold any client-visible locks. The specification even allows for multiple concurrent threads to cause concurrent finalization. (In early JVM implementations this led to a subtle security hole, which was that if the thread used was a system thread with high privileges, then the finalize method ran with these privileges and could thus undermine security policies.)

For objects holding internal resources – that is, resources modeled using Java objects – there is no need to implement a finalizer as such internal resources are themselves subject to garbage collection, and finalization if required. The vast majority of Java classes do not require a finalizer.

A second mechanism that helps to steer the garbage collector are special references introduced in SDK 1.2. There are three kinds – soft, weak, and phantom references. Note that standard Java language-level references are now also called strong references. An object is strongly reachable if a sequence of strong references from a root variable leads to that object. It is softly reachable if a sequence of strong and soft references leads to that object. A need to go through at least one weak reference to reach an object makes it weakly reachable.

Finally, if at least one phantom reference is on every path to an object, then that object is phantom reachable.

A soft reference is an instance of class java.lang.ref.SoftReference or its subclasses. A soft reference holds an encapsulated reference to some other object, its referent. If the JVM is under memory pressure, it is free to clear soft references, provided their referents are no longer strongly reachable (reachable via normal Java references only). Although not required, the specification encourages JVM implementations to choose least recently used soft references first. The specification requires a JVM to clear all soft references before throwing an out-of-memory exception. Once cleared, a soft reference will return null when asked for its referent. It is also possible to register reference objects with reference queues. The garbage collector enqueues registered reference objects when clearing their referent. It is usually more efficient to poll or block on a reference queue than to periodically poll all soft references to determine which ones have been cleared. Soft references are useful to implement object caches.

If an object is only reachable via soft or weak references and its soft references have been cleared, then it can be finalized. A weak reference does not ask for holding a referent unless there is a memory shortage – if a weakly referenced object can be collected, then its weak references are cleared and it is readied for finalization. Reference queues work for weak references as well. As a last straw, phantom references can be used to keep the last trace of a finalized but not yet reclaimed object. Phantom references never return their referent (to prevent resurrection), but can be used as unique "names" of their referents in canonicalization algorithms. Reference queues are the only means to extract useful information from phantom references. They are primarily useful to perform a post-finalization cleanup of name tables used for canonicalization. For instance, if a particular object should only be created if no other object has already been created in a certain place, then a phantom reference is a cheap way to account for this condition without keeping any objects unnecessarily alive. However, a memory leak can occur if phantom references, once enqueued, aren't cleared.

It should be clear by now that the garbage-collection steering mechanisms discussed in this section are highly problematical. Slight misuse can lead to significant performance degradation or program misbehavior, although memory safety is always preserved. Luckily, the majority of classes can be written without ever touching on any of these mechanisms.

14.3 JavaBeans

Initially, Java targeted two kinds of products – applets and applications. The composition models are rudimentary, however. Applets can be composed by placing them on the same web page and then, within one applet instance, using class AppletContext to find another applet instance by name. While different applets can be used to fill such named positions, there is no way to determine statically what

type such applets must have. There is also no standardized way to "rewire" applets on a page. In the case of plain Java applications, Java provided even less: Composition there depended entirely on facilities of the operating system. Beyond simplistic cases, applets are not suitable as separately sold components.

JavaBeans fills some of this gap and makes a new kind of product possible – Java components, called "beans" (Sun, 1996). It is unfortunate that the clear distinction between class and object in Java is not carried through in JavaBeans. Although a bean really is a component (a set of classes and resources), its customized and connected instances are also called beans. This is confusing. Thus, in the following, bean refers to the component and bean instance to the component object. "Bean object" would be too confusing, because a bean usually consists of many Java objects.

A bean can be used to implement controls, similar to OLE or ActiveX controls. JavaBeans provides a protocol to handle containment (see section 14.4.5). In addition, JavaBeans has been designed to also enable the integration of a bean into container environments defined outside Java. For example, there is an ActiveX bridge available from Sun that allows a bean instance to function as a control in an ActiveX container.

Beans have been designed with a dual usage model in mind. Bean instances are first assembled by an assembly tool, such as an application builder, at "design-time," and are then used at "runtime." A bean instance can customize appearance and functionality dynamically by checking whether or not it is design-time or runtime. One of the ideas is that it should be possible to strip the design-time code from a bean when shipping an assembled application. A bean instance can also enquire, at "runtime," whether or not it is used interactively, that is, whether or not a graphical user interface is available. Thus, a bean instance can behave differently when being run on a server or as part of a batch job, rather than interactively.

The main aspects of the bean model are the following.

- *Events* Beans can announce that their instances are potential sources or listeners of specific types of events. An assembly tool can then connect listeners to sources.
- *Properties* Beans expose a set of instance properties by means of pairs of getter and setter methods. Properties can be used for customization or programmatically. Property changes can trigger events and properties can be constrained. A constrained property can only be changed if the change is not vetoed.
- *Introspection* An assembly tool can inspect a bean to find out about the properties, events, and methods that a particular bean supports.
- *Customization* Using the assembly tool, a bean instance can be customized by setting its properties.
- *Persistence* Customized and connected bean instances need to be saved for reloading at the time of application use.

14.3.1 Events and connections

Events in the JDK 1.1 terminology – and as used in JavaBeans – are objects created by an event source and propagated to all currently registered event listeners. Thus, event-based communication generally has multicast semantics. However, it is possible to flag an event source as requiring unicast semantics – such a source accepts at most one listener at any one time. Event-based communication is similar to the COM Connectable Objects approach and also can be seen as a generalization of the Observer pattern (Chapter 9).

An event object should not have public fields and should usually be considered immutable. Mutation of event objects in the presence of multicast semantics is subtle. However, where mutation of an event object during propagation is required, these changes should be encapsulated by methods of the event object. The JavaBeans specification gives coordinate transformations as an example, so a mouse-click event may need to be transformed into local coordinates to make sense to a listener.

Listeners need to implement an interface that extends the empty "marker" interface java.beans.EventListener and that has a receiving method for each event the listener listens to. (Marker interfaces are also known as tagging interfaces. They are themselves empty and serve as a mark [or tag] on the type that extends or implements them. This is a form of simple binary meta-attribution. In the case of event listeners, the marker is used to find listeners via introspection.) For example:

```
interface UserSleepsListener extends java.util.EventListener {
  void userSleeps (UserSleepsEvent e);
}
```

A listener can implement a given event listener interface only once. If it is registered with multiple event sources that can all fire the same event, the listener has to determine where an event came from before handling it. This can be simplified by interposing an event adapter. An event adapter implements a listener interface and holds references to listeners. Thus, an event adapter is both an event listener and an event source. A separate adapter can be registered with each of the event sources. Each adapter then calls a different method of the listener. Thus, event adapters can be used to demultiplex events. In principle, event adapters can perform arbitrary event-filtering functions. Note that, in the absence of method types and method variables, event adapters lead either to an explosion of classes or to slow and unnatural generic solutions based on the reflection service. Such generic demultiplexing adapters were indeed proposed in the JavaBeans specification (Sun, 1996, pp. 35–37). Since Java 1.1, this problem has been reduced by supporting in-line construction of lightweight adapter classes – so-called anonymous inner classes, somewhat similar to Smalltalk blocks. A more modern approach uses nested classes to implement the required functionality directly rather than forwarding calls, which minimizes code redundancy.

An event source needs to provide pairs of listener register and unregister methods:

```
public void addUserSleepsListener (UserSleepsListener l);
public void removeUserSleepsListener (UserSleepsListener l);
```

If a source requires unicast semantics, it can throw a TooManyListenersException:

```
public void addUserSleepsListener (UserSleepsListener l)
    throws java.util.TooManyListenersException;
```

Event propagation in a multicast environment introduces a number of subtle problems (see the discussion of callbacks and re-entrance in Chapter 5). The set of listeners can change while an event is propagated. JavaBeans does not specify how this is addressed, but a typical way is to copy the collection of registered listeners before starting a multicast. Also, exceptions may occur while the multicast is in progress. Again, JavaBeans does not specify whether or not the multicast should still continue to address the remaining listeners. Finally, a listener may itself issue an event broadcast on reception of an event. This raises the difficult problem of relative event ordering; JavaBeans does not specify any constraints on the ordering of event delivery (see section 10.6 for a detailed discussion of the ordering issues of event-based programming).

Event ordering issues are aggravated in multithreaded environments such as Java. For example, an event source may hold locks that are required by event listeners to process events. The result would be a deadlock if the event listener uses a separate thread to acquire such locks. Indeed, the authors of the JavaBeans specification (Sun, 1996, p. 31):

> "strongly recommend that event sources should avoid holding their own internal locks when they call event listener methods."

This is clearly subtle. For one, problems only arise if the event listener uses additional threads besides the one carrying the event notification, because the notifying thread continues to hold all the locks held by the event source, which would prevent deadlocks. Also, it is not even clear if this recipe can be followed generally without breaking encapsulation. It may not be at all easy to determine whether and which locks the event source already holds. More precise advice continues:

> "Specifically, [...] they should avoid using a synchronized method to fire an event and should instead merely use a synchronized block to locate the target listeners and then call the event listeners from unsynchronized code."

As indicated above, "to locate the target listeners" is best interpreted as performing a synchronized copy of the collection holding references to the target listeners. Otherwise, if listeners were added or removed during an ongoing notification, then standard Java collections would throw a ConcurrentModificationException,

following the "fail fast" pattern. Although this specific advice clearly needs to be followed, it is not sufficient to avoid subtle deadlocks that result from a combination of multithreading and event-based communication. More effective advice is to not use event-based models in combination with multiple threads. (For a discussion of Java threading in general, see section 14.2.3.)

14.3.2 Properties

A bean can define a number of properties of arbitrary types. A property is a discrete named attribute that can affect a bean instance's appearance or behavior. Properties may be used by scripting environments, can be accessed programmatically by calling getter and setter methods, or can be accessed using property sheets at assembly time or runtime. Typical properties are persistent attributes of a bean instance. Property changes at assembly time are used to customize a bean instance. A property change at runtime may be part of the interaction with the user, another bean instance, or the environment.

Access to properties is via a pair of methods. For example, the setter and getter methods for a property "background color" might take the following form:

```
public java.awt.Color getBackground ();
public void setBackground (java.awt.Color color);
```

To optimize the common case where an array of property values needs to be maintained, indexed properties are also supported. The corresponding setter and getter methods simply take an index or an entire array of values:

```
public java.awt.Color getSpectrum (int index);
public java.awt.Color[] getSpectrum ();
public void setSpectrum (int index, java.awt.Color color);
public void setSpectrum (java.awt.Color[] colors);
```

A property can be bound. Changes to a bound property trigger the firing of a property change event. Registered listeners will receive an event object of type java.beans.PropertyChangeEvent that encapsulates the locale-independent name of the property and its old and new value:

```
public class PropertyChangeEvent extends java.util.EventObject {
  public Object getNewValue ();
  public Object getOldValue ();
  public String getPropertyName ();
}
```

The methods required to register and unregister property change listeners are:

```
public void addPropertyChangeListener (PropertyChangeListener x);
public void removePropertyChangeListener (PropertyChangeListener x);
```

The interface java.beans.PropertyChangeListener introduces the method to be called on property changes:

```
void propertyChange (PropertyChangeEvent evt);
```

A property can also be constrained. For a constrained property, the property setter method is declared to throw PropertyVetoExceptions. Whenever a change of a constrained property is attempted, a VetoableChangeEvent is passed to all registered vetoable-change listeners. Each of these listeners may throw a java.beans.PropertyVetoException. If at least one does, the property change is vetoed and will not take place. If no veto exception is thrown, the property is changed in the usual way and a PropertyChangeEvent is passed to all change listeners.

The methods required to register and unregister vetoable property change listeners are:

```
public void addVetoableChangeListener (VetoableChangeListener x);
public void removeVetoableChangeListener (VetoableChangeListener x);
```

The interface java.beans.VetoableChangeListener introduces the method to be called on property changes:

```
public void vetoableChange (VetoableChangeEvent evt)
    throws PropertyVetoException;
```

Normally, properties are edited by property editors as determined by a java.beans.PropertyEditorManager object. This object maintains a registry to map between Java types and appropriate property editor classes. However, a bean can override this selection by specifying a property editor class to be used when editing a particular property of that bean. Furthermore, where customization of a bean is complex or involves many properties, a bean can also nominate a customizer class that implements a specific customization interface. A customizer will normally take a separate dialog window, whereas typical property editors are controls. Thus, property editors can be used in the dialog implemented by a customizer.

14.3.3 Introspection

Events and properties are supported by a combination of new standard interfaces and classes. Examples are EventListener, EventObject, EventSetDescriptor, and PropertyDescriptor (for more information on the property and event models, see the following sections). The use of interfaces such as EventListener allows an assembly tool using the reflection services (see below) to discover support of certain features by a given bean. In addition, JavaBeans introduces the new notion of method patterns. (In the JavaBeans specification, these are called "design patterns"; Sun, 1996. This is confusing. The term "method pattern" used here is

much closer to what is meant.) A method pattern is a combination of rules for the formation of a method's signature, its return type, and even its name. Method patterns allow for the lightweight classification of individual methods of an interface or class. For example, here is the method pattern used to indicate a pair of getter and setter methods for a specific property:

```
public <PropertyType> get<PropertyName> ();
public void set<PropertyName> (<PropertyType> a);
```

For a property "background" of type java.awt.Color, this pattern yields the following two methods:

```
public java.awt.Color getBackground ();
public void setBackground (java.awt.Color color);
```

The use of conventional names in method patterns can be avoided – a bean can implement interface java.beans.BeanInfo. If this interface is implemented, an assembly tool will use it to query a bean instance explicitly for the names of property getters, property setters, and event source registration methods. The signatures prescribed by the patterns still need to be followed, of course, as otherwise the assembly could not take place. The following excerpts from class BeanInfo and related classes (JavaBeans as in JDK 1.4) hint at how BeanInfo information can be used to find out about the supported events, properties, and exposed methods:

```
package java.beans;
public interface BeanInfo {
  // some constants elided

  public BeanInfo[] getAdditionalBeanInfo ();
    // current info takes precedence over additional bean info
  public BeanDescriptor getBeanDescriptor ();
  public int getDefaultEventIndex ();
  public int getDefaultPropertyIndex ();
  public EventSetDescriptor[] getEventSetDescriptors ();
  public Image getIcon (int iconKind);
  public MethodDescriptor[] getMethodDescriptors ();
  public PropertyDescriptor[] getPropertyDescriptors ();
}

public class FeatureDescriptor {
  public Enumeration attributeNames ();
  public String getDisplayName ();
  public String getName ();
  public String getShortDescription ();
  public boolean isExpert ();   // feature for expert users only
  public boolean isHidden ();   // feature for tool-use only
  public boolean isPreferred ();   // important for human inspection
```

```
  public Object getValue (String attributeName);
  public void setValue (String attributeName, Object value);
    // setValue and getValue allow to associate arbitrary named attributes
    // with a feature
  // other set-methods elided
}

public class BeanDescriptor extends FeatureDescriptor {
  // constructors elided
  public Class getBeanClass ();   // the class representing the entire bean
  public Class getCustomizerClass ();   // null, if the bean has no customizer
}

public class EventSetDescriptor extends FeatureDescriptor {
  // constructors elided
  public java.lang.reflect.Method getAddListenerMethod ();
  public java.lang.reflect.Method getRemoveListenerMethod ();
  public java.lang.reflect.Method getGetListenerMethod ();
  public java.lang.reflect.MethodDescriptor[] getListenerMethodDescriptors ();
  public java.lang.reflect.Method[] getListenerMethods ();
  public Class getListenerType();
  public boolean isInDefaultEventSet ();
  public void setInDefaultEventSet (boolean inDefaultEventSet);
  public boolean isUnicast ();
    // this is a unicast event source: at most one listener
  public void setUnicast (boolean unicast);
}

public class MethodDescriptor extends FeatureDescriptor {
  // constructors elided
  public java.lang.reflect.Method getMethod ();
  ParameterDescriptor[] getParameterDescriptors ();
}

public class ParameterDescriptor extends FeatureDescriptor {}

public class PropertyDescriptor extends FeatureDescriptor {
  // constructors elided
  public Class getPropertyEditorClass ();
  public Class getPropertyType ();
  public java.lang.reflect.Method getReadMethod ();
  public java.lang.reflect.Method getWriteMethod ();
  public boolean isBound ();
    // change of bound property fires PropertyChange event
  public boolean isConstrained ();   // attempted change may be vetoed
  // set-methods elided
}
```

Method patterns are an interesting deviation from the established Java design principles. Normally, a Java class that has a certain property, or implements certain functionality, signals this by implementing a corresponding interface. Consider the following hypothetical substitution of the method pattern for property getters and setters. A standard empty interface Property is provided. For each property, a subinterface is defined. A bean class that has some of these properties then has to implement the corresponding property interfaces:

```
interface BackgroundProp extends Property {    // hypothetical!
  public Color getBackground ();
  public void setBackground (Color color);
}
```

The names of the set and get method should still contain the property name to avoid conflicts with set and get methods from other property interfaces. Java reflection could now be used directly to look for property interfaces in a class. The triggering key would be extension of interface Property rather than detection of methods of a certain pattern. It is not clear why this straightforward use of interfaces and reflection, instead of method patterns, was not used for the specification of JavaBeans. This is particularly surprising as a similar approach is used for event listener interfaces.

The feature introspection mechanism of JavaBeans allows attachment of arbitrary named custom attributes (name-value pairs) to features of a bean. As of SDK 1.4, this mechanism became fragile – class Introspector now uses soft references to hold on to BeanInfo instances. As a result, any bean-info object can be garbage collected as soon as the last client reference is gone. Sun recommends checking attribute attachments every time a bean-info object is acquired – and re-attaching attributes in case they aren't there any more (because the previous info object has been collected). Doing so can become quite tedious and requires keeping all attachment code central. The option to use attached attributes to communicate between separate parts of a system is essentially gone.

14.3.4 JAR files – packaging of Java components

Java class files were the only pre-beans means of packaging Java components in pre-1.1 JDKs. All that a Java class file can contain is a single compiled class or interface. A class file also contains all meta-information about the compiled context. Resources cannot be included in a class file. To ship a component, many separate files would need to be distributed.

The problem is solved by using Java Archive (JAR) files to package a JavaBean. JAR files were originally introduced to support JavaBeans, but have since been used to package all other Java components as well. Technically, a JAR file is a ZIP-format archive file that includes a manifest file. Manifest files

are used to provide information on the contents of an archive file (see below). The archive may include:

- a set of class files;
- a set of serialized objects that is often used for bean prototype instances;
- optional help files in HTML;
- optional localization information used by the bean to localize itself;
- optional icons held in .icon files in GIF format;
- other resource files needed by the bean.

The serialized prototype contained in the JAR file allows a bean to be shipped in an initialized default form. Serialization is performed using the object serialization service (see section 14.4.2). New instances of such a bean are created by deserializing the prototype, effectively producing a copy.

There can be multiple beans in a single JAR file – potentially, each of the contained classes can be a bean and each of the serialized objects can be a bean instance. The manifest file in the JAR file can be used to name the beans in the JAR file.

14.4 Basic Java services

Over the years, there have been many additions to the services standardized for Java. This section covers reflection, object serialization, and the Java native interface (JNI).

14.4.1 Reflection

The Java core reflection service is a combination of original Java language features, a set of support classes (introduced with JDK 1.1), and a language feature to support class literals (expressions of type Class, such as MyFoo.class, introduced with JDK 1.2). The reflection service, curbed by the active security policy, allows:

- inspection of classes and interfaces for their fields and methods;
- construction of new class instances and new arrays;
- access to and modification of fields of objects and classes;
- access to and modification of elements of arrays;
- invocation of methods on objects and classes.

The reflection service thus now covers all the Java language's features. The Java language-level access control mechanisms, such as privacy of a field, are enforced. (Unrestricted access can be useful to implement trusted low-level services, such as portable debuggers. A special interface for such unrestricted access is part of the Java platform debugger architecture – JPDA.) To enable reflective operations, the reflection service introduces a package java.lang.reflect. (The default import of java.lang into all packages makes

java.lang itself effectively inextensible as otherwise there would be a risk of name clashes with existing packages imported using the package.* syntax. The introduction of java.lang.reflect, instead of introducing the new classes into java.lang itself, avoids the conflict. This is somewhat unfortunate as the original JDK 1.0 reflection classes, such as Class, are in java.lang. Also, once compartmentalized, further additions that aren't about reflection didn't logically fit into java.lang.reflect – as of J2SE 1.4 there's also java.lang.ref to support special references; see section 14.2.4.)

Classes Field, Method, and Constructor provide reflective information about the field, method, or constructor that they describe and allow for type-safe use of this field, method, or constructor. All three are final and without public constructors. All three implement interface Member, which makes it possible to find out how the member is called and determine the member's modifiers and to which class or interface it belongs. Below are excerpts of some of these interfaces and classes:

```
package java.lang.reflect;
public interface Member {
  public abstract Class getDeclaringClass ();
  public abstract String getName ();
  public abstract int getModifiers ();
    // decodable using class java.lang.reflect.Modifier
}

public final class Field implements Member {
  public Class getType ();
  public Object get (Object obj)   // if static field, obj is ignored
        throws NullPointerException, IllegalArgumentException,
          IllegalAccessException;
  public boolean getBoolean (Object obj)
        throws NullPointerException, IllegalArgumentException,
          IllegalAccessException;
  // similar for all other primitive types; avoids wrapping in get
  public void set (Object obj, Object value)
        throws NullPointerException, IllegalArgumentException,
          IllegalAccessException;
  public void setBoolean (Object obj, Boolean z)
        throws NullPointerException, IllegalArgumentException,
          IllegalAccessException;
  // similar for all other primitive types; avoids wrapping in set
}

public final class Method implements Member {
  public Class getReturnType ();
  public Class[] getParameterTypes ();
  public Class[] getExceptionTypes ();
  public Object invoke (Object obj, Object[] args)
```

```
        // returns null if return type is void
        throws NullPointerException, IllegalArgumentException,
          IllegalAccessException,
          InvocationTargetException;      // wrapper for exception thrown
                // by invoked method
}
```

Class Class (still in java.lang, not java.lang.reflect) has methods to return instances of these classes when querying for the features of a particular class. The important methods of class Class are:

```
public final class Class {
  public static Class forName (String className)
        throws ClassNotFoundException;
  public Object newInstance ()
        throws InstantiationException, IllegalAccessException;
  public boolean isInstance (Object obj);
  public boolean isInterface ();
  public boolean isArray ();
  public boolean isPrimitive ();
  public String getName ();
  public int getModifiers ();   // decodable using java.lang.reflect.Modifier
  public ClassLoader getClassLoader ();
  public Class getSuperclass ();
        // null if primitive, interface, or class Object
  public Class[] getInterfaces ();
  public Class getComponentType ();
        // type of array components; null if not array
  public Class getDeclaringClass ();
        // next outer class, if this is an inner class or interface
  public Class[] getClasses ();
        // public inner classes or interfaces, incl. inherited ones
  public Field[] getFields () throws SecurityException;
        // public accessible fields, incl. inherited ones
  public Method[] getMethods () throws SecurityException;
        //public methods, incl. inherited ones
  public Constructor[] getConstructors () throws SecurityException;
        // public constructors
  public Class[] getDeclaredClasses () throws SecurityException;
        // all inner classes or interfaces, excl. inherited ones
  public Field[] getDeclaredFields () throws SecurityException;
        // all fields, excl. inherited ones
  public Method[] getDeclaredMethods () throws SecurityException;
        // all methods, excl. inherited ones
```

```
    public Constructor[] getDeclaredConstructors () throws
  SecurityException;
        // all constructors
    // further methods to get resources and
    // (declared) field, method, constructor, or class by name
  }
```

There are several constant objects (public static final) for the languages primitive types. For example, there is an object java.lang.Boolean.TYPE that is the Class object for primitive type boolean. While still supported, the preferred and universal way to get to a Class object since JDK 1.2 is the form boolean.class. Class Array supports dynamic construction and use of arrays. Class Modifier simplifies the inspection of modifier information on classes, fields, and methods.

```
package java.lang.reflect;
public final class Modifier {
  public static boolean isPublic (int modifiers);
    // true if modifiers incl. public
  // similar for private, protected, static, final, synchronized, volatile,
  // transient, strictfp, native, interface, abstract
  // note that "interface" is viewed as a class modifier!
}
```

14.4.2 Object serialization

Up to JDK 1.0.2, Java did not support serialization of objects into bytestreams – only primitive types were supported. If an application wanted to write an entire web of objects to an output stream, it needed to traverse and serialize the objects itself, using some ad hoc encoding scheme. The Java object serialization service overcomes this by defining a standard serial encoding scheme and by providing the mechanisms to code and decode ("serialize" and "deserialize") webs of objects.

To be serializable, an object has to implement interface java.io.Serializable. In addition, all fields that should not be serialized need to be marked with the modifier transient. This is important, because fields may refer to huge computed structures, such as caches, or values that are inherently bound to the current JVM incarnation, such as descriptors of open files. For objects implementing Serializable, sufficient information is written to a stream such that deserialization continues to work, even if different (but compatible) versions of classes are used. Methods readObject and writeObject can be implemented to control further what information is written or to append further information to the stream. If these methods are not implemented, all non-transient fields referring to serializable objects are automatically serialized. Shared references to objects are preserved.

To make serialization safe and configurable, methods readObject and writeObject are private! Therefore, there can be one such method per subclass level. Reflection is used to find these methods for each extension level. If these methods exist, they should call a method defaultReadObject (or defaultWriteObject) before handling additional private data. The reason is that readObject and writeObject on a given class extension level handle only the fields introduced in this level, not those in subclasses or superclasses.

As an alternative to implementing interface java.io.Serializable, a class can implement interface Externalizable. Then none of the object's fields is automatically handled and it is up to the object to save and store its contents. Externalizable has methods writeExternal and readExternal for this purpose. These methods are public, and objects implementing Externalizable open themselves for access to their state that bypasses their regular public interface. This requires some care to avoid safety problems.

A simple versioning scheme is supported – a serializable class can claim to be a different version of a certain class by declaring a unique serial version unique ID (SUID). A SUID is a 64-bit hash code ("fingerprint") computed over the name of a class and all implemented interfaces' features of that class, including the features' types or signatures. It does not cover superclasses, because each extension level has its own serial version ID and is responsible for its own evolution. The serial version ID is computed automatically when an instance of a class is serialized and that class does not declare a serial version ID. However, if it does, then the class declares itself to be compatible with the class that originally had this ID. The readObject method can be used to read serialized state from other versions and thus preserve compatibility with serialized versions.

Object serialization must be used with care to avoid security holes. For example, if a serialized stream can be tampered with, then it can be arranged to deserialize a critical object in a way that its internal subobjects are shared by some rogue object. These alias references can then be used to bypass security checks of the critical object. Before J2SE 1.4, it was recommended to eagerly copy such private subobjects after deserialization in order to break unwanted aliases. Such a deep copy is expensive and non-trivial to implement. In version 1.4, new methods readUnshared and writeUnshared are provided to solve this problem.

Object serialization creates a stream of bytes in a single-pass process – that is, with no back-patching. Hence, while still serializing a web of objects, the part of the stream that has been produced already can be forwarded to filters or the destination. The stream is fully self-describing down to the level of Java primitive types – a tag that describes the type of the following item precedes every value in the stream. Compared with compact native formats, this can be quite costly. However, the added robustness of the serial format probably outweighs the higher cost.

A major drawback of the current serialization service is the missing support for graceful degradation in the case of missing or incompatible classes at the receiving end. For example, if the serialized object is a document, then it

should be possible to deserialize and use that document even if some of its embedded objects cannot (currently) be supported on the receiving platform. The current service simply throws a ClassNotFound exception and does not offer means to resynchronize and continue deserialization. In the document example, it would also be desirable if an unsupported object could be kept in serialized form to include it when serializing the document again. This is also not supported by the current serialization service.

14.4.3 Java native interface

The Java native interface (JNI) specifies, for each platform, the native calling conventions when interfacing to native code outside the Java virtual machine. JNI also specifies how such external code can access Java objects for which references were passed. This includes the possibility of invoking Java methods. JNI does not specify a binary object model for Java – that is, it does not specify how fields are accessed or methods are invoked within a particular Java virtual machine. Interoperation between Java virtual machines on the same platform remains an unresolved issue, as does interfacing with services such as just-in-time compilers. JNI allows native methods to:

- create, inspect, and update Java objects;
- call Java methods;
- catch and throw exceptions;
- load classes and obtain class information;
- perform runtime type checking.

The actual layout of objects is not exposed to native code. Instead, all access is through so-called JNI interface pointers (Figure 14.8) that use a runtime structure identical to that of COM (p. 331).

Despite the superficial closeness to COM, JNI is different and not automatically compatible with COM. A JNI interface pointer is used only to refer to a thread-specific context and does not correspond to an individual Java object. The JNI interface does not include standard COM functions QueryInterface, AddRef, or Release. If these are added, the entire JVM could function as one

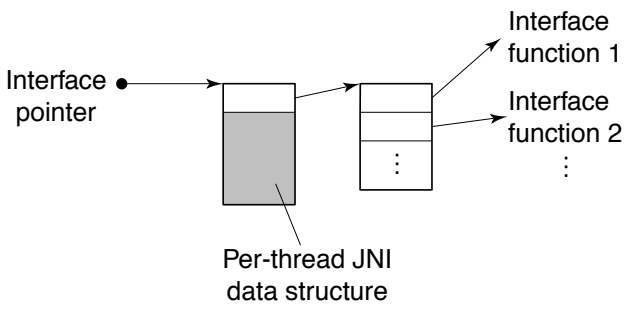

Figure 14.8 JNI interface pointer.

COM component object. A native method is called with a JNI interface pointer as its first argument. Other arguments, including Java object references, are passed directly, but all access to Java objects is via functions provided by the JNI interface pointer.

For the Java garbage collector to work, the JVM keeps track of all references handed out to native methods. References are handed out as local references, and these are released by the JVM on return of the native method. Native code can turn a local into a global reference. It is the responsibility of the native code to inform the JVM when a global reference is no longer needed.

All access to fields or invocation of methods of Java objects from native methods is performed using one of several accessor functions available via the JNI interface pointer. Below are two sample JNI functions that show how to invoke a Java object's method, using C++ notation:

```
JNIEnv *env;   // the JNI interface pointer
jmethodID mid = env->GetMethodID (classPtr, methodName,
                    methodSignature);
  // methodSignature is a string representing the mangled signature
jdouble result = env->CallDoubleMethod(obj, mid, args);
  // factoring of GetMethodID useful to avoid repeated method lookup
```

JNI specifies how to "mangle" signatures of methods. In the above example, this is used to resolve overloaded methods when performing a method lookup. As JNI does not reveal how a JVM implements objects or dispatches methods, access is relatively expensive.

14.4.4 Java AWT and JFC/Swing

The Java abstract windowing toolkit (AWT) and Java foundation classes (JFC) are central to any Java development providing a graphical user interface. Here is a brief summary.

- *Delegation-based event model* Perhaps the most dramatic change in JDK 1.1. The previous event model was based on inheriting from component classes and overriding event handler methods. The subtle interaction of super-calls, open lists of possible events, and local handling decision procedures was found to be too complex and error-prone. (The 1.1 model has been described in section 14.3.1.) "Delegation-based" is a misnomer, following the unfortunate example of the use of the term delegation in COM. The 1.1 JDK really provides a forwarding-based event model. Object connection and composition are used in favor of implementation inheritance.
- *Data transfer and clipboard support* Like the COM universal data transfer service, AWT defines the notions of transferable data items.

Internet MIME (multipurpose internet mail extensions) types are used to interact with non-Java applications. Data transfer between Java applications can also directly use Java classes.

- *Drag and drop* Support for drag and drop among Java and non-Java applications (by hooking into the underlying system's drag and drop protocols, such as the OLE one on Windows).
- *Java 2D* Classes for advanced 2D graphics and imaging. Java 2D covers line art, text, and images. Supported features include image compositing, alpha blending (transparency), accurate color space definition and conversion, and display-oriented imaging operators.
- *Printing* The printing model is straightforward. Graphical components that do not explicitly handle printing will be printed using their screen-rendering methods. Thus, for simple components, printing is free. However, the printing model does not address the subtleties resulting from printing embedded contents that need to spread over multiple pages. (This is addressed, for example, by the ActiveX printing model, which allows embedded controls to print across several pages in cooperation with their container. This is a complex model, though, and few if any ActiveX containers actually implement this advanced printing protocol.)
- *Accessibility* Interface that allows so-called assistive technologies to interact with JFC and AWT components. Assistive technologies include screen readers, screen magnifiers, and speech recognition.
- *Internationalization* Based on the Unicode 2.1 character encoding, support is provided to adapt text, numbers, dates, currency, and user-defined objects to the conventions of a locale, as identified by a pair of language and region identifiers.
- *Swing components and pluggable look and feel* In JDK 1.0, most user interface components (buttons, list boxes, and so on) relied on native implementation in so-called "peer classes." This has the advantage that a Java application truly looks and feels like a native one. Introduced later, so-called Swing components (which are JavaBeans) are independent of native peers and support pluggable look-and-feel. Whatever look-and-feel is selected, it will be consistent across all platforms. Swing components are written entirely in Java, without relying on native window-system-specific code.

14.4.5 Advanced JavaBeans specifications

The following four specifications generalize the JavaBeans model – the containment and services protocol, the Java activation framework (JAF), the long-term persistence model, and the InfoBus specification. Each is described briefly below.

Containment and services protocol

The containment and services protocol supports the concept of logically nesting JavaBeans bean instances. All beans can assume the services of the JVM and core Java APIs. With the containment and services protocol, a nested bean can acquire additional services at runtime from its container and a container can extend services to its nested beans.

Java activation framework (JAF)

JAF is used to determine the type of arbitrary data (by following the MIME standard), the operations ("commands") available over that type of data, and to locate, load, and activate components that provide a particular operation over a particular type of data. The central piece of a JAF implementation is a command map – a registry for components that is indexed by pairs of MIME types and command names.

Long-term persistence for JavaBeans

Long-term persistence – also called archiving – is an alternative to object serialization (see section 14.4.2). The file format created by this archival process is XML with a DTD or one of two proprietary Java file formats. While object serialization aims to fully persist the entire state held in a graph of objects, archiving only captures the part of the state that can be set through public properties. As such public interfaces are far more stable over time than the object's implementations, it is much less likely that a future version of a component can no longer read the state written by an older version (or, indeed, the other way around.)

The process of creating and consuming such archived state is interesting. It uses the same basic infrastructure as object serialization, but limits itself to publicly settable properties. To prevent the writing of default values, the proposed persisting form is immediately read again to create a copy of the original object graph. Any attempt to set a property to a value that it already has (because of defaults) is detected and such values are eliminated from the persisted form. This approach compresses persisted forms nicely, but it also introduces a new form of implementation dependency. If a particular default value wasn't picked according to a bean's specification, but merely for implementation convenience, then it may be different in a future bean implementation. Error handling is also done in an original way – exceptions during reconstruction from persisted form are effectively ignored – in a hope that a partially reconstituted object graph is better than none. This is problematical and a protocol should be provided to allow the detection and handling of inappropriate reconstruction failures.

InfoBus

The InfoBus specification creates a generic framework for a particular style of composition. The idea is to design beans to be InfoBus-aware and categorize beans into data producers, data consumers, and data controllers, all of which can be coupled by an information bus determining data flow.

The InfoBus protocol distinguishes six steps. First, any Java component can connect by implementing InfoBusMember, obtaining an InfoBus instance, and having the member join it. Second, members receive notifications by implementing an interface and registering it with the InfoBus. Two event listener interfaces are defined to support data consumers receiving announcements about data availability and data producers receiving requests for data. Third, based on the name of some data item, data producers and data consumers rendezvous to exchange data. Fourth, to allow producers and consumers to operate in terms of their internal data representations, the InfoBus provides a set of interfaces for various standard protocols, which are used to create data items with common access. Fifth, data values are typically of primitive type or collections to minimize dependencies between consumers and producers. Sixth, a consumer can attempt to change data items, but the producer can enforce policies as to what changes are acceptable.

Data controllers implement the InfoBusDataController interface and participate in the distribution of InfoBusEvents to consumers and producers on an InfoBus. Multiple controllers can be added to the bus in order to optimize communication, not interfere with actual data item exchange between producers and consumers. A data controller can intercept requests and, for instance, implement a cache for data items from an expensive producer. Requests from consumers and producers are offered to controllers in sequential order – the first controller to declare the request handled stopping further propagation. A default controller at the end of this chain handles all remaining requests.

14.5 Component variety – applets, servlets, beans, and Enterprise beans

The Java universe defines five different component models, and more may arrive in the future. Besides the applet and JavaBeans models (both part of J2SE), there are Enterprise JavaBeans, servlets, and application client components (all part of J2EE). This section presents a brief overview of these different component models.

The two critical contributions to the J2EE server-side models are servlets/JSPs and EJBs. These are covered in more detail in following subsections. All components in the J2EE space are packaged into JAR files, which can be included in a J2EE application. An important aspect of all J2EE components is the support of deployment descriptors. These are XML files co-packaged with a component that describe how a particular component

should be deployed. Deployment is the act of readying a component for an actual deployment context – a step that can be, and often is, separate from the notion of installation. (It is possible to deploy a component once and then install it many times. See sections 8.6 and 8.11.) The detailed nature of deployment descriptors depends on the particular component model. For instance, the descriptor of an EJB entity bean (see section 14.5.2) may request container-managed persistence and detail how properties of the entity bean should be mapped to tables in a database.

Applets were the first Java component model, aiming at downloadable lightweight components that would augment websites displayed in a browser. The initial applet security model was so tight that applets could not really deliver much more than "eye candy" – an area that was rapidly captured by other technologies, such as GIF89a animated images, Macromedia's Shockwave and Flash technology, JScript, and refinements to the HTML world itself, including the introduction of DHTML (dynamic HTML). Taking advantage of such browser-level technologies, most J2EE-based applications use servlets and JSPs to generate scripted HTML pages instead of sending applets.

The second Java component model, JavaBeans, focuses on supporting connection-oriented programming and is, as such, useful on both clients and servers. Historically, JavaBeans are more popular in the scope of client-side rich applications and sometimes perceived as being replaced by EJB on the server. This view is, technically, wrong. EJB, beyond its name, shares little with JavaBeans. JavaBeans remain useful when building client-side applications as they explicitly support a visual application design mode. (As of J2SE 1.3, a bean can also be an applet. However, so far the support is limited – bean applets will always receive an empty applet context and stub.)

EJB, Java's third component model, focuses on container-integrated services supporting EJB beans (components) that request services using declarative attributes and deployment descriptors. In a later revision, a container model was added to JavaBeans as well, but JavaBeans containers are very different from EJB containers. The former are a mere mechanism for containment, while the latter are partially driven by declarative constructs. As JavaBeans does not require the presence of an interactive user outside of design-time, it is conceivable to use JavaBeans to construct more complex EJBs. (JavaBeans and EJBs correspond roughly to component classes and serviced component classes in the .NET Framework.)

The fourth Java component model is servlets. These pick up the spirit of applets, but live on a server and are (usually) lightweight components instantiated by a web server processing, typically, web pages. A matching technology, Java ServerPages (JSP), can be used to declaratively define web pages to be generated. JSPs are then compiled to servlets. (Servlets and JSP correspond roughly to page classes and pages in ASP.NET; see section 15.13.3.)

J2EE introduces a fifth Java component model – application client components. These are essentially unconstrained Java applications that reside on

clients. A client component uses the JNDI enterprise naming context (see section 14.6.3) to access environment properties, EJBs, and resources on J2EE servers. Such resources can include access to e-mail (via JavaMail) or databases (via JDBC).

At a distribution and deployment format level, J2EE enterprise applications are packaged in .ear (enterprise archive) files that contain .war and .jar files. Servlets and JSPs are packaged in .war (web archive) files; applets, JavaBeans, and EJBs are packaged in .jar files. All these files follow the ZIP archive file format (www.pkware.com).

J2EE deployment descriptors serve two purposes. First, they enable the component developer to communicate requirements on the side of the component. Requesting container-managed persistence is an example. Second, they enable the component deployer to fill in the blanks. For example, for a component that requests container-managed persistence, a deployer would specify exactly how the state of that component's instances should be mapped to particular relational tables. The role of the deployer is distinct and often served by a person different than the component developer, potentially in a separate organization. (For comparison, in COM+ and CLR, the equivalent of deployment descriptors shows in two different places, aligned with these two different roles. Deployment information provided by the developer is capture in attributes that are closely aligned with the code. In the CLR case, these are conveniently attached using language-supported custom attributes. Deployment information that is modified or provided by the deployer is kept in separate XML-based configuration files.)

The variety of supported Java component models is a reflection of different needs. However, to actually establish component markets in these various areas, deeper standardization of domain-specific concepts has to happen. Today, only a few EJB components, to name just one case, are used beyond even the one enterprise application for which they were developed.

14.5.1 Java server pages (JSP) and servlets

Serving web pages and other formatted contents can be viewed as a combination of three core functions. First, incoming requests for such contents need to be accepted, checked for proper authorization, and routed to the appropriate components ready to handle a particular request. Second, such components need to draw on information sources and processes to retrieve or synthesize the requested contents. Third, the retrieved or generated contents need to be sent to the requesting party.

The prototypical model handling these three steps is that of a web server. Incoming HTTP requests are received by a web server that interprets the request and the target URL, including possible parameters. A simple web server will either find a static HTML page, typically in the local file system, or use the URL to activate a component via a simple interface called CGI

(common gateway interface). The component receives the URL (and, in particular, the parameters that it may include) and generates an HTML page. In both cases, the server will then send the HTML page back to the client, typically a web browser.

The same model can also be used to serve up content that is not in HTML format. For instance, a web server can be used to provide XML web services. In that case, incoming SOAP requests are routed to the right component, processed, and SOAP replies are sent back.

A common property of the above scenarios is that the web server infrastructure is common up to the point where specific components need to take over to process requests and retrieve or synthesize replies in a customizable way. For instance, the Windows internet information service (IIS) handles incoming HTTP requests, determines whether or not to send back information found in a file directly, or activate an appropriate component and send back the results produced by that component.

Writing the customizing components requires not much more than a simple interface to the server infrastructure – at least when aiming for simple cases only. An example of such an interface in the context of IIS is ISAPI (internet server API), an efficient low-level interface that enables high performance, but that is quite demanding on the developer of a customizing component. To simplify this situation for the bulk of website requirements, Microsoft introduced ASP (active server pages), a model that allows raw web pages to include server-side script code. The ASP server executes such scripts, which will typically synthesize HTML fragments as their result. ASP then places computed HTML fragments where the script used to be and returns the resulting HTML page to the client.

Java server pages (JSP) improved on the ASP model by compiling JSP pages (pages containing HTML with inserted Java fragments) into servlets. A JSP server activates servlets as needed to handle requests. In addition, servlets are also supported by explicit API definitions. It is possible and common to implement servlets instead of writing JSP pages. The result is an inversion of the page programming model. At the level of ASP and JSP pages, code fragments reside in line in HTML (or other markup, such as XML). At the level of servlets, HTML (or other markup) resides in line in Java source code. (This is a simplified description as, in both cases, the in line fragments can be included by reference to separate files.) The following simple example shows how a doGet method on a servlet produces a simple HTML result:

```
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
  protected void doGet (HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, java.io.IOException
```

```
    {
      java.util.Calendar calendar = new java.util.GregorianCalendar();
      int hour = calendar.get(currTime.HOUR);
      int minute = calendar.get(currTime.MINUTE);
      ServletOutputStream out = response.getWriter();
      out.print("<HTML><BODY>\r\n");
      out.print("The time is: " + hour + ":" + minute +
                " – or it was when I looked.\r\n");
      out.print("</BODY></HTML>\r\n");
    }
  }
```

As can be seen, the servlet programming model is quite natural if relatively small static HTML (or other markup) fragments need to be combined with computed results. The servlet container supports servlets by providing simple access to HTTP Request parameters and by managing sessions. However, if large amounts of static markup contents need to be produced with print calls, readability quickly suffers. In addition, if the markup fragments need to be open for localization (such as translation into another language), then embedding these fragments as string literals into source code is a bad choice. The fragments can be factored into separate files to enable reasonable localization, but that leads to further reduction of code readability.

The dual model to embedding markup in source code is to embed source code in markup. This is exactly what a JSP page does, where the content of <% ... %> tags is interpreted as Java source code. The following example shows a JSP page that leads to the same result as the servlet above. Note the use of tag <%= ... %> to request evaluation of a Java expression and output of the result by printing to the page stream.

```
<HTML><BODY>
<%
  java.util.Calendar calendar = new java.util.GregorianCalendar();
  int hour = calendar.get(currTime.HOUR);
  int minute = calendar.get(currTime.MINUTE);
%>
The time is: <%= hour %>:<%= minute %> – or it was when I looked.
</BODY></HTML>
```

When processed, this JSP page causes the generation of, essentially, the servlet above. It is thus appropriate to always think of web page and other contents handling as being performed by servlets, even if the source is kept in the form of JSP pages. To keep JSP pages largely contents-oriented and maintain both readability and localizability, it is useful to minimize code in JSP pages. Unlike contents, Java code is supported by natural abstraction mechanisms – packages, classes and methods. Thus, instead of embedding large pieces of Java

code in JSP pages, it is preferable to keep most code in separate Java packages and reduce code in JSP pages to the necessary glue – invocation of methods on appropriately created objects and insertion of results into page streams. This helps avoid another source of possible confusion – server-side Java and client-side JavaScript may coexist in the same JSP page, which clearly does not help readability.

By means of special and custom tags, JSP can go further towards eliminating server-side code in JSP pages. For instance, there is a set of tags to create JavaBeans instances and access their properties from JSP pages. Custom tag extensions can be added easily, for instance by implementing just two methods (doStartTag and doEndTag) on a class derived from class javax.servlet.jsp.tagext.TagSupport. In the following example, two custom tags are used to retrieve the current hour and minute, respectively, and insert the value into the page output stream.

```
<% taglib uri="/hour" prefix="calendar" %>
<% taglib uri="/minute" prefix="calendar" %>
<HTML><BODY>
The time is: <calendar:hour>:<calendar:minute> – or it was when I looked.
</BODY></HTML>
```

The JSP standard tag library (JSTL) includes control flow tags such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. JSTL also introduces an expression language to simplify page development and a framework for integrating existing custom tags with JSTL tags.

When combining the servlet concepts with libraries that support the processing of XML, JSP servlets can be used to implement simple web services. To do so, the JSP server itself needs to be augmented with support for the underlying web service protocols (essentially SOAP and possibly web service standards for security and other aspects). This is essentially the path that ASP.NET follows to support web services (see section 15.13.3).

Servlets do not have to directly produce the contents stream eventually output by the server to the client. Instead, a request can be handled by one servlet that contains relevant business logic. Once such logic has been processed, that servlet can hand off output generation to another servlet class. In other words, it is possible to factor servlets, yielding more specialized and easier to maintain components.

If web requests are only one point of entry into larger enterprise applications or if complex integration across applications is required, it becomes useful to further factor the approach. To do so, servlets can build on the services of EJBs (see the next subsection). Where such a separation is performed, it is important to recognize the overheads naturally introduced by building on two separate component models and their (partially) separate infrastructure. It is possible to colocate a JSP server and an EJB container on the same machine

and even in the same process, sharing the same JVM instance. However, it is equally possible to distribute JSP and EJB processing across separate machines, potentially enabling better scalability – but at an increased base overhead.

14.5.2 Contextual composition – Enterprise JavaBeans (EJB)

The naming of Enterprise JavaBeans suggests a close relationship to JavaBeans (see section 14.3). This is, in fact, not the case. The JavaBeans approach to composition is connection-oriented programming or, as it is sometimes called, wiring. Beans can define both event sources and event listeners. By connecting one bean instance's listener to another bean instance's event source, events flow. Later improvements of the JavaBeans model introduce support for hierarchical container structures (see section 14.4.5), so, instead of placing all bean instances in a flat space and relying on peer-to-peer connections only, containers allow the hierarchical creation of subsystems. Another addition to JavaBeans, the InfoBus, allows for a flexible decoupling of event sources and event listeners by routing some or all communication through a bus structure that allows for the interception of messages and application of policies without requiring cooperation from the event source or listener and without a need to re-wire. With JavaBeans containment and services infrastructure in place and the availability of the InfoBus, JavaBeans could move beyond connection-oriented composition to forms of contextual composition and data-driven composition. In practice, this potential is rarely exploited as, while the infrastructural design is in place, the critical mass of contextual services or messaging infrastructure is not.

EJB follows an entirely different path. There are no provisions for connection-oriented programming at all. (Adding these is one of the main improvements of CCM over EJB; see section 13.3.) Instead, EJB components (e-beans) follow a relatively conventional model of object-oriented composition. For example, if an e-bean instance needs a fresh instance of another e-bean, it simply creates one. If it needs to communicate with another e-bean instance, it simply calls methods. That is, EJB is not about systematically improving compositionality of e-beans via wiring of connections. E-beans are just as composable as their specific design made them and the generic EJB architecture does little to improve or hinder. For the remainder of this subsection, e-beans will be called just beans for brevity and to align with the common EJB lingo. It is important, however, to remember that EJB beans and JavaBeans beans are entirely different.

If EJB is weak at the connection-oriented composition level, it is strong at the level of contextual composition. (Since EJB 2.0, data-driven composition is also well covered; see the following subsection.) Contextual composition is about the automatic composition of component instances with appropriate services and resources (see section 21.2 for a theoretical discussion). Two important examples of services in the EJB space are the handling of transac-

tions and security policies. An EJB container configures services to match the needs of contained beans. These needs are expressed declaratively in a bean's deployment descriptor (see section 14.5). The skeleton of an EJB deployment descriptor has the following form:

```
<!DOCTYPE ejb-jar PUBLIC
  "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
  "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <session> ... </session>
    <entity> ... </entity>
    <message-driven> ... </message-driven>
  </enterprise-beans>
  <relationships>
    <ejb-relation> ... </ejb-relation>
  </relationships>
  <assembly-descriptor>
    <security-role>
      <description> ... </description>
      <role-name> ... </role-name>
    </security-role>
    <method-permission>
      <role-name> ...as defined in a security role entry... </role-name>
      <method>
        <ejb-name> ...bean name... </ejb-name>
        <method-name> ...method name or * for all methods...
                                                  </method-name>
      </method>
    </method-permission>
    <container-transaction>
      <method> ...as above... </method>
      <trans-attribute> ... </trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

In this example skeleton, three different beans (a session, entity, and message-driven bean) are described (with all details elided). The following relationships map declares relationships among entity beans, which are used by the container to manage relationships and persistence of entity beans. For example, the container can use declared relationships to automatically find the entity bean instance at the other end of a relationship of an entity bean instance. The application descriptor section covers security and transaction requirements. For instance, it might grant the right to call a particular set of methods to anyone

acting in a particular named role. (Mappings from role names to specific roles in the deployment environment are declared separately using a deployment tool.) Likewise, the transaction attribute might be set to Required, which instructs the container to create a new transaction if the bean's caller isn't already enlisted in one. As deployment descriptors can get long and detailed, good tool support at development and deployment-time is essential. (One issue here is that a lot of information appears redundantly in multiple places. For example, the name of a method on an EJB object interface appears in the deployment descriptor, on the object interface – twice if local and remote are supported – and on the bean class. All four occurrences cannot be checked by the compiler for mutual consistency.)

Deployed beans are contextually composed with services and resources by an EJB container. Contextual composition works by placing a hull around instances and intercepting communication from and to that instance. The hull itself can be thought of as a wall of proxies placed on all references to and from the instance inside the hull. To enable interaction between services and an instance, contextual access to services is provided to the instance – thus contextual composition. That is, the instance receives some form of reference to its context. The combination of the service implementations, intercepting hull, and context is referred to as a container in EJB. EJB containers are provided by servers. In the case of J2EE, the standard architectural enclosure for EJB, this server is a J2EE application server that will also provide containers for servlets (see previous subsection).

EJB realizes the hull around beans by not allowing any direct access to a bean's fields, methods, or constructors from any other bean, including beans colocated on the same server in the same container. Instead, all access to beans is required to go through one of two interfaces – the EJB home interface for life-cycle operations and the equivalent of static methods and the EJB object interface for all methods on the bean instance. Non-local clients will see these interfaces implemented on stub objects that use RMI or RMI-over-IIOP to communicate with the corresponding EJB container, which then relays calls to bean instances it contains. As of EJB 2.0, local clients can request local versions of both home and object interfaces. Local clients are still not allowed to access other beans directly and, as home and object interfaces never return anything but home and object interfaces of other beans, there is actually no way to do so. (Technically, a local bean could use reflection to subvert containment. However, a server implementation could use deployment-time code inspection and run-time checking techniques to discover and prevent such attempts.) Figure 14.9 shows how client, server, container, bean, home, and object relate in EJB.

The EJB specification does not detail how a particular container wraps bean instances. As part of the deployment process, tools provided with a container implementation take a bean in a JAR file, including the compiled interfaces and classes defining the bean and its deployment descriptor. These tools generate the bean's EJB object and EJB home. They are free to either generate classes implementing the local and remote EJB object and home interfaces

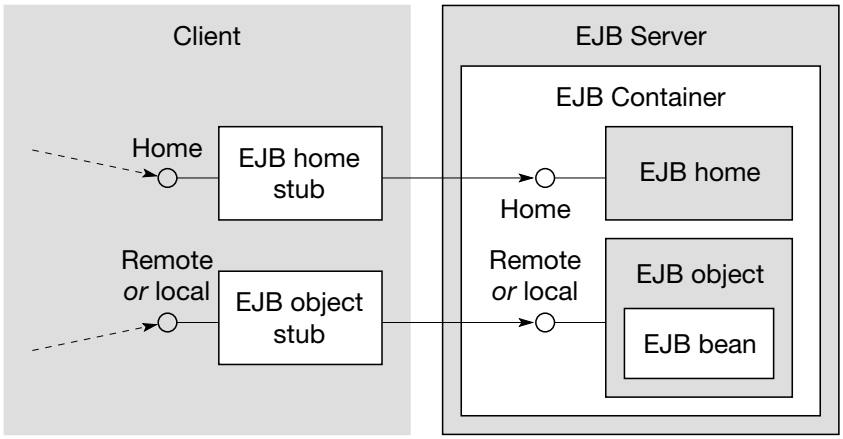

Figure 14.9 Isolation of EJB beans in containers via EJB home and EJB object interfaces. After Fig. 2-2 in Monson-Haefel, R. (2001) *Enterprise JavaBeans, 3rd Edition*, reprinted by permission of O'Reilly and Associates, Inc.

that wrap the bean's class, to synthesize subclasses that combine container-supplied object and home code with the code found in the bean's class, or even to synthesize classes that directly combine the implementation of the bean's class and the container-specific code. (In the latter cases, reflection would reveal additional members on the bean's class and, again, the EJB server could take deployment and runtime measures to prevent that.)

Beans of many flavors

There are four kinds of EJB beans – stateless session, stateful session, entity, and message-driven beans. Message-driven beans are new in EJB 2.0 and somewhat different from session and entity beans (they are described in the following subsection). The three flavors of session and entity beans are covered in more detail further below. They are all united by a common top-level contract between beans and containers and their use of deployment descriptors. Session and entity beans in addition share the design of EJB object and EJB home interfaces. (These are not standardized Java interfaces but are synthesized according to a standard pattern for any given bean.)

Every EJB home interface has a standard method create to instantiate a bean. In the case of entity beans it also has a standard method findByPrimaryKey to locate an existing instance by its primary key. A home interface can have additional, bean-specific methods as specified in the deployment descriptor. As such methods are not associated with any specific bean instance, they roughly correspond to static methods on a Java class. The methods on an EJB object interface are all bean-specific as specified in the deployment descriptor.

The EJB container cycles beans through defined lifetime stages. Immediately after creation, setEntityContext or setSessionContext is called to establish a backlink to the container's context for the new bean. Then ejbCreate is called, which is matched by a call to ejbRemove just before a bean instance is finally released. For entity beans, removal also implies deletion from the database as,

unlike session beans, entity beans are thus only removed when their deletion is explicityly requested (via their home interface). As a container may need to keep many bean instances logically around, but parked at some low-cost level, it can call ejbPassivate to request release of any non-essential resource held by a bean instance. Then the container serializes a stateful session bean to some exterhal store or writes an entity bean back to the database, respectively. The container calls ejbActivate to return a bean to a ready state, after deserializing it from external store or loading it from a database.

Although the container context is attached to a bean in such a special way, it hasn't been designed in a particularly durable way. Of its ten methods, three have been deprecated. The remaining seven allow a bean some control over the current transaction, limited access to the security context, and access to the bean's own local and remote home. However, the "mother of all contexts" is not available – the initial context of JNDI (section 14.6.3). Instead, every bean has to find this initial context in a way that, up to EJB 2.0, is not portable.

An EJB container serializes all invocations of beans, so there is no need to synchronize within beans and the use of the synchronized keyword on bean methods and the creation of new threads is even illegal. The container also protects beans from re-entrant calls (with the exception of entity beans that explicitly declare that they can tolerate re-entrancy; see below), throwing an exception whenever a re-entrant call is attempted. Finally, the container isolates faults, so if a bean throws an exception, its EJB object is immediately invalidated and the offending bean instance destroyed, the remove method having not been called.

Session beans

A session bean is created by a client as a session-specific contact point. For instance, a servlet may create a session bean to process web requests that the servlet received. A stateless session bean does not maintain state across multiple invocations – a container can therefore decide to keep instances in a pool instead of creating new ones for every invocation in every session. A stateful session bean, on the other hand, remains associated with the one session for its lifetime and thus can retain state across method invocations.

The deployment descriptor of a session bean, in skeleton form, has the following form:

```
<session>
  <ejb-name> ...name of the session bean... </ejb-name>
  <home> ...name of EJB home interface... </home>
  <remote> ...name of EJB object interface... </remote>
  <local-home> ...name of local EJB home interface... </local-home>
  <local> ...name of local EJB object interface... </local>
  <ejb-class> ...name of class implementing this EJB... </ejb-class>
  <session-type> ...Stateless or Stateful... </session-type>
  <transaction-type> ...Container or Bean... </transaction-type>
```

```
    <ejb-ref> …import of other beans referenced by this bean… </ejb-ref>
    <security-identity> …run-as role-name or use-caller-identity…
                        </security-identity>
</session>
```

Besides assigning a name to the session bean itself, the descriptor assigns names to the EJB home and object interfaces and, if supported, their local counterparts. (A bean that does not declare local interfaces will not receive any when deployed.) The EJB class is the main class implementing the bean. The session type determines whether or not the container will associate the bean with a particular session and, therefore, whether the container will create a new instance per call or not. Session (and message-driven) beans have an option to explicitly control transactions – so-called bean-managed transactions – or follow the default of container-managed transactions. Explicitly controlling transactions is subtle and error-prone, but it allows for covering scenarios where the container-managed approach does not work. This is usually the case only when interfacing with external systems that are not coordinated with the container. The list of EJB references is like an import clause in that it lists the names of other beans and their local and remote EJB home and object interfaces. During deployment, this list is used to make sure that a deployed bean refers to the right synthesized interfaces and classes. Finally, the security identity field (new in EJB 2.0) allows a bean to either run under the caller's identity or under some specified role.

Entity beans

The idea behind entity beans is to use objects corresponding to database entities and encapsulate access to actual database records. An entity bean can correspond to a single row in a relational table, but more likely it corresponds to a row in the result of a join operation. The mapping of entity beans to and from database rows is called persistence. Entity beans can manage their own persistence by directly using JDBC to operate over databases. Alternatively, they can use container-managed persistence driven by object-to-table mappings defined during the deployment process.

Entities in a database schema are related to each other. Thus, there is a need to find entity bean instances by navigating entity relationships. The corresponding mechanisms can again be either bean or container-managed relationships, as selected by the deployment descriptor. It is interesting to note that container-managed relationships map entity relationship (ER) models straight into object models. The database model is no longer relational, but instead of network-style. Network databases were introduced well before relational ones and they can have performance advantages. However, as they do not support concepts of normalization, network databases are known to be fragile. If the schema changes, client code needs to change as well. By keeping the data in normalized form (in a relational database), ER model changes will only affect the code inside entity beans.

An entity bean is either newly created or located by its primary key. In the former case, a new entity bean is created and associated with a new primary key. In the latter case, the container checks whether or not the corresponding bean instance is already available. If so, it is returned. If not, the container creates a new instance and associates it with the primary key. Then either the container or, in the bean-managed case, the bean class locate the underlying data in a database in order to initialize the new instance.

The deployment descriptor for an entity bean takes the following skeleton form:

```
<entity>
  <ejb-name> ...name of the entity bean... </ejb-name>
  <home> ...name of EJB home interface... </home>
  <remote> ...name of EJB object interface... </remote>
  <local-home> ...name of local EJB home interface... </local-home>
  <local> ...name of EJB object interface... </local>
  <ejb-class> ...name of class implementing this EJB... </ejb-class>
  <persistence-type> ...Container or Bean... </persistence-type>
  <prim-key-class> ... </prim-key-class>
  <reentrant> ...True or False... </reentrant>
  <abstract-schema-name> ... </abstract-schema-name>
  <prim-key-field> ...field name... </prim-key-field>
  <cmp-field> <field-name> ...field name... </field-name> </cmp-field>
  <security-identity> ...run-as role-name or use-caller-identity...
                      </security-identity>
</entity>
```

The initial fields match those of session bean descriptors. The persistence type can be container- or bean-managed, as discussed above. The primary key class defines which Java class to use to hold primary key values. This can range from java.lang.Integer to complex compound key types. The re-entrance field can be used to enable re-entrant calls as, by default, re-entrance of EJB beans is prohibited and causes the container to throw an exception. This cannot be overridden for session beans, but entity bean designers can decide to support re-entrancy as it can be difficult to prevent cases of re-entrance when navigating from entity to entity along the relationship graph. The abstract schema name is one that is unique to the particular bean that is used in EJB's query language (EJB QL; see below). Each field descriptor for container-managed persistence is interpreted as the name of a property (as of EJB 2.0) that has a setter and a getter method to be called by the container.

Entity relationships and database mapping

As mentioned above, combining the descriptors for relationship among entity beans and those for container-managed persistent fields yields an abstraction

for a flexible object-to-relational mapping. This approach is at the heart of the improved portability of entity beans in EJB 2.0 from one J2EE server product to another. (EJB 1.1 introduced a container-managed persistence model that was severely limited and essentially made portable entity beans non-existent. While not described in this book, EJB 1.1 persistence lives on, but to enable backwards compatibility, an EJB 2.0 container has to also implement EJB 1.1 persistence.)

EJB 2.0 supports one-to-one, one-to-many, and many-to-many relationships – all in both unidirectional and bidirectional versions. In addition, EJB 2.0 introduced a query language similar to SQL called EJB QL. This language is used in deployment descriptors to declaratively specify additional find methods for the home interfaces of entity beans. For example, to define a find method that locates an entity by name (rather than by primary key), the following fragment could be added to a deployment descriptor:

```
<entity>
  ...
  <query>
    <query-method>
      <method-name>findByName</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
      </method-params>
      <ejb-ql>
        SELECT OBJECT(x) FROM Employee e WHERE e.name = ?1
      </ejb-ql>
    </query-method>
  </query>
</entity>
```

This descriptor fragment causes the implementation of a findByName method that takes a single string-typed argument. The SELECT statement is similar to SQL; ?1 refers to the first (and, in this case, only) method argument. In EJB 2.0, the specification of EJB QL still has a number of shortcomings (Monson-Haefel, 2001). For example, it is not possible to select rows in sorted order and it is not possible to properly handle dates. The lack of support for sorted order, especially, is a significant drawback for high-performance applications.

Container-managed persistence and relationships, as defined in EJB 2.0, are a powerful abstraction of the data-mapping machinery. However, there is a danger that developers lose touch with actual performance. At the time of writing, IBM's WebSphere implementation of J2EE 1.3.1 was one of the few that integrated the EJB 2.0 container mechanisms with the caching system of the database server. Without such a deep integration, container-managed persistence and relationships can lead to surprising performance problems because entities are brought in either in small increments or at once – both of which can seriously degrade performance.

14.5.3 Data-driven composition – message-driven beans in EJB 2.0

With the introduction of EJB 2.0, support for data-driven composition was added to the EJB model. In essence, this is done by adding an entirely new e-bean type – message-driven beans (md-beans). Like session and entity beans, md-beans reside inside an EJB container. Like session bean instances, they have no container-managed peristent state. Unlike session and entity beans, they also don't have a remote or local interface or home interface. The only way to instantiate and use an md-bean is to register it for a particular message queue or topic as defined by the Java message service (JMS, see section 14.6.3). An md-bean can be registered with exactly one such queue or topic only, requiring external collection of messages meant to be handled by an md-bean into a single queue or topic.

When the container schedules a message in a queue or topic for processing, it creates a new instance of any bound md-bean (or recycles an existing instance) and then calls its ejbCreate method. The container calls setMessageDrivenContext to associate the new md-bean with its context. Then the actual message handling is requested by calling the one central method of any md-bean – onMessage. An md-bean handling a message behaves a bit like a stateless session bean handling an incoming method call – it does not maintain specific state when returning from onMessage. The container is free to recycle an md-bean instance to handle multiple messages. Eventually, the container cleans up by calling the ejbRemove method, allowing the md-bean to release any resources it might have acquired. Unlike any other bean, an md-bean cannot return invocation results. To have any permanent effect whatsoever, an md-bean draws on other beans including entity beans. It can also draw on any resource accessible via its context, which, in EJB 2.0, has to include a JMS provider via which the md-bean can send messages. There are many more options, such as accessing databases or other external resources via JDBC or JCA resource connectors, respectively (see section 14.6.3).

It is possible to write all application logic using md-beans. The resulting message-queue-oriented design is fully asynchronous and thus not suitable for serving interactive sessions. However, the gained flexibility at the message queuing and scheduling level leads to a composition model that is well suited to workflow-oriented automatic distribution of work items.

14.6 Advanced Java services

This section covers Java support for distributed computing at an enterprise scale. There are actually four different models supporting distributed computing in Java – RMI, RMI over IIOP, CORBA, and EJB containers (that themselves build on RMI or RMI over IIOP). EJB was covered in the previous section; this section covers the remaining approaches, followed by the most important services supporting distributed applications.

14.6.1 Distributed object model and RMI

Distributed computing is mainly supported by the object serialization service (as described above) and the remote method invocation (RMI) service, both introduced with JDK 1.1. This subsection describes RMI and RMI over IIOP, which are subtly different.

A distributed object is handled via references of interface type – it is not possible to refer to a remote object's class or any of its superclasses. Interfaces that can be accessed remotely have to be derived from java.rmi.Remote. A remote operation can always fail as a result of network or remote hardware problems. All methods of a remote interface are therefore required to declare the checked exception java.rmi.RemoteException. Parameter passing to remote operations is interesting. If an argument is of a remote interface type, then the reference will be passed. In all other cases, passing is by value – that is, the argument is serialized at the call site and deserialized before invoking the remote interface. Java objects are not necessarily serializable. An attempt to pass a non-serializable object by value raises a runtime exception. If Java RMI conventions were made parts of the language, then the compiler could statically enforce that only serializable objects are passed by value and that all methods declare RemoteException.

The Java distribution model extends garbage collection as well. Fully distributed garbage collection is supported, based on a careful bookkeeping of which objects may have remote references to them. The collector is based on the work for Network Objects (Birrel, 1993). Distributed garbage collection is the most outstanding feature of Java RMI compared with almost all other mainstream approaches today. The only other approach that also builds on Network Objects and the idea of leased references is CLI remoting (see section 15.13.1), which was introduced about four years after the debut of Java RMI.

The Java distributed object model has a number of quirks, however. First, Java RMI interferes with the notion of object identity in Java. Second, Java RMI interferes with the Java locking system's semantics, which normally prevents self-inflicted deadlocks. These two problems are explained below. (Both of these problems are solved by the CLI approach.)

Java RMI affects object identity as a result of its model of implementing remote references. If a remote interface reference is passed around, proxy objects are created on remote sites. A reference to a remote interface, once passed in a remote method invocation, is thus not a reference to the remote object but to the local proxy of that object. Even if such a remote reference is sent back to the object's home server, it will still point to a proxy – one that is in the same server that the object itself is residing in. It is thus not possible to send out a reference, get it back, and compare it against a local object to see whether or not the returned reference matches that of the local object. (In contrast, DCOM and CLR maintain object identity on proxies. That is, they do not create multiple proxies for the same remote object in the same client context.)

Self-inflicted deadlocks are caused by locking systems that do not allow a thread to re-enter an area locked by that same thread. In regular Java, threads can acquire locks any number of times without causing a deadlock. In Java RMI, the situation is different. The notion of a thread identity does not span multiple machines. If a remote invocation performs a nested remote invocation back to the original requester, then a distributed deadlock can occur. Such a deadlock is caused by the original requester holding locks that are also needed by the recursive callback. (Compare this to logical threads, supported by DCOM and CLR, that can span process and machine boundaries.)

The special handling of identities by Java RMI has further effects. Several methods defined in java.lang.Object had been introduced with the Java object identity model in mind. Under Java RMI, the Object method's equivalents, hashCode and toString, need to be implemented differently. Proper handling can be achieved by extending class java.rmi.RemoteObject when creating a class that implements remote interfaces. Obviously, this precludes extending some other class and it may therefore be necessary to override these three methods "manually."

The situation is somewhat worse for the Object methods getClass, notify, notifyAll, and wait. These are declared final in java.lang.Object and cannot be adjusted by RemoteObject or another class. Although none of these methods malfunctions in the context of Java RMI, they all have potentially unexpected semantics. When operating on remote references, all these methods operate on the proxy object. For getClass, this makes the synthesized proxy class visible, instead of returning the real class of the remote object. For the wait and notify operations, there is no synchronization between the local proxy and the remote object. (The latter is usually desirable, as remote locking would be a fairly inefficient operation.)

14.6.2 Java and CORBA

An OMG IDL to Java binding and, an OMG first, a Java to OMG IDL reverse binding were defined in 1998 as part of CORBA 2.2 (see section 13.1.2). An important reason to incorporate CORBA into Java projects is to enable the use of IIOP for communication with non-Java subsystems. For access to CORBAservices it is usually more convenient to go through Java-specific access interfaces that can map to CORBA-compliant and other service implementations. Several of these Java service interface standards are discussed in the following subsections.

CORBA and Java usually coexist in almost all application server products today. It is therefore often reasonable to assume the presence of CORBA mechanisms (for interoperation) when implementing for the application server tier.

RMI is normally implemented using a proprietary protocol (Java Remote Method Protocol – JRMP), which limits the use of RMI to Java-to-Java com-

munication. The RMI-over-IIOP specification was introduced in 1999 and is part of the JDK since J2SE 1.3. It supports a restricted RMI variant to be used over the CORBA IIOP protocol, reaching any CORBA 2.4-compliant ORB. This specification requires a recent addition to CORBA that enables the sending of objects by value – that is, the sending of a copy of the object rather than the sending of a reference to the object staying behind.

RMI-over-IIOP does not support the RMI distributed garbage collection model and thus falls back on CORBA's lifecycle management approach to deal with the lifetime of remote objects explicitly. In addition, RMI-over-IIOP creates proxies that do not allow normal Java instanceof/cast mechanisms to be used to discover interfaces or subclasses. Instead, a service method must be called to determine whether or not some type is supported – and, if so, that method returns a new proxy. The fact that a new proxy instance may be returned does, as such, not add new complications as RMI is already removing the object identity property and several proxy instances may refer to the same remote object.

With J2SE 1.4, introduced in 2002, support was added for POA (portable object adapter), portable interceptors, INS (interoperable naming nervice), GIOP 1.2 (general interoperability protocol), and dynamic any's. J2SE 1.4 also includes a tool to remotely manage a persistent server and an ORB daemon (ORBD) that enables clients to transparently locate and invoke persistent objects on CORBA-supporting servers.

14.6.3 Enterprise service interfaces

Important parts of the J2EE architecture are several suites of interfaces that address enterprise-level services. Such service interfaces could also be established via CORBA (see section 14.6.2). However, Java-CORBA integration necessarily introduces some friction. In contrast, the Java-centric interfaces discussed in this section are designed to minimize any such friction, both from a client's and an implementer's point of view.

Java naming and directory interface (JNDI)

A universal problem in computing systems is the location of services by exact name or attributes. Naming services address the former and directory services the latter problem. Examples of naming services include the internet domain name service (DNS), the RMI registry, and the CORBA naming service. Examples of directory services include LDAP-compliant directories such as Novell's eDirectory, Microsoft's Active Directory, or the open source OpenLDAP (www.openldap.org).

JNDI provides uniform APIs for naming (javax.naming) and directory (javax.naming.directory) services. The most commonly used interface – Context – makes a particular naming context available on which method lookup can be used to locate objects by name. A naming context can also be used to list all

bindings in the context, remove a binding, or create and destroy subcontexts. An important naming context for EJB beans is the environment naming context (ENC) provided by the EJB container. It enables access to environment properties, other beans, and resources.

Interface DirContext extends Context to provide directory functionality of examining and updating attributes associated with an object listed in the directory and searching a directory context by value. As DirContext extends Context, a directory context is also a naming context. Most contexts are themselves found by recursive lookup on another context. The starting point is the initial context that is always accessible by instantiating class InitialContext.

JNDI also defines an event API (javax.naming.event), an LDAP API (javax.naming.ldap) that supports the LDAP v3 features that go beyond the DirContext features, and a service provider interface (javax.naming.spi) that enables providers of naming and directory services to hook into JNDI. The event mechanism is used to register for change notifications. J2SE 1.4 comes with four built-in service providers – CORBA naming, DNS, LDAP, and RMI.

Java message service (JMS)

Asynchronous messaging enables composition models that decouple and overlap operations of the instances communicating by messages. Transacted message queues establish the level of reliabilty that normally requires synchronous call-based models. Flexible message routing, multicasting, and filtering further improve flexibility. JMS is a Java access mechanism to messaging systems – it doesn't implement messaging itself. JMS support message queues for point-to-point delivery of messages. It also supports message topics that allow for multiple targets to subscribe. Messages published to a topic are delivered to all subscribers of that topic.

JMS supports a variety of message types (bytes, streams, name-value maps, serialized objects, and text). Using declarations that are close to the SQL WHERE clause, message filters can be set up.

Java database connectivity (JDBC)

JDBC followed the popular Microsoft ODBC (open database connectivity) standard in establishing a common way of interacting with a database. The JDBC API is split into the core API (found in package java.sql and part of J2SE) and the JDBC optional package (found in javax.sql and optional in J2SE, but mandatory in J2EE). Like ODBC, JDBC depends on drivers to map the JDBC API to the native interfaces of a particular database.

There are four types of JDBC drivers. Type 1 and Type 2 drivers use native (non-Java) code accessed via JNI. Type 1 drivers use native code with a common interface, while Type 2 allows for database-specific interfaces. The most common Type 1 driver is the JDBC-ODBC bridge included in the JDK – it maps JDBC calls to ODBC calls. This is relatively slow as ODBC uses its

own driver model to access specific databases. Type 3 and Type 4 drivers are pure Java.Type 3 accesses a database indirectly via a network protocol and a database gateway, while Type 4 accesses a database directly. Drivers can be selected without affecting client code as the JDBC API itself is unaffected by the choice of driver. In terms of typical performance, Type 4 is usually the best, followed by Type 2, then Type 1, and finally Type 3.

Available JDBC drivers are tracked by the driver manager. Typically, a driver registers with the driver manager using a static initializer. That is, merely loading a driver suffices to cause registration. The following statement is commonly used to load a driver – the JDBC-ODBC bridge driver in this example.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Once a driver has been located, the Driver interface can be used to create a database connection, which is reflected by a returned object that implements interface Connection. The Connection interface is the main JDBC hub – it can be used to retrieve meta-information about the database itself, create database statements (SQL statements), and manage database transactions. Statements come in three variants – plain, prepared, and callable statements. Prepared statements are effectively SQL templates, which are SQL commands with variables. Before executing a prepared statement, these variables are filled in by calling methods that associate a variable with a value. Callable statements invoke SQL stored procedures. Some statements perform a database command that has no results (such as deleting a row of data), but most yield a result. Results are objects that implements interface ResultSet. Unless a result is read-only, it can be updated and pushed back to the database.

Database connections (and open statements per connection) are a relatively expensive resource and most systems impose tight limits. This is an area where reliance on the JVM's garbage collector is not acceptable practice. Instead, developers must take care to explicitly call the corresponding close methods to ensure release of such resources as soon as they are no longer needed. There are further complications in this space. For instance, while prepared statements are good candidates for shared use – they can be created once and then used repeatedly with variable bindings – they are always associated with a single specific connection. However, connection pooling is a common technique to enhance performance, forcing the creation of a new statement every time as the connection instance will vary.

Java transaction API and service (JTA, JTS)

While transaction management is almost always delegated to an EJB's container, there are cases where explicit transaction management is required. The CORBA object transaction service (OTS) or its Java implementation (JTS) can be used for this purpose. However, a much simpler interface was introduced with EJB, namely the Java transaction API. It comprises a low-level XA interface

(the X/Open transaction API standard) used by server/container implementations and a high-level client interface accessible to EJB beans implementations. The most common high-level interface is javax.transaction.UserTransaction with six simple methods (with exception lists elided for brevity):

```
package javax.transaction;
public interface UserTransaction {
  void begin () throws …;
  void commit () throws …;
  int getStatus () throws …;
  void rollback () throws …;
  void setRollbackOnly () throws …;
  void setTransactionTimeout (int seconds) throws …;
}
```

The use of JTS (or OTS) requires explicit and careful enlistment of resources in transactions and, thus, the explicit demarcation of transaction boundaries. With the high-level JTA interfaces, this error-prone function is performed by the EJB container. However, explicit transaction management is still error-prone and can lead to inconsistencies or inefficencies due to resources held by long-running transactions.

J2EE connector architecture (JCA)

Introduced with J2EE 1.3, JCA standardizes connectors between a J2EE application server and enterprise information systems (EIS) such as database management, enterprise resource planning (ERP), enterprise asset management (EAM), and customer relationship management (CRM) systems. Enterprise applications not written in Java or within the J2EE framework are also candidates for JCA connection. The JCA is defined in package javax.resource and its subpackages (cci, spi, and spi.security). (JCA is a case of frontal acronym collision as the same acronym is also used for the Java cryptography API.)

JCA defines resource adapters that plug into a J2EE application server – one such adapter per EIS type. Resource adapters interact with an application server to support connection pooling, security, and transactions. This collaboration is formalized by three corresponding JCA system contracts that specify how application servers and application components establish connections as well as security and transaction contexts. In addition, the JCA common client interface (CCI) defines a client API for application components that need to access an EIS. The CCI is preferably used by enterprise application integration (EAI) frameworks and by tools.

In early 2002, nine J2EE application servers supported JCA, including BEA's WebLogic, IBM's WebSphere, Borland's Enterprise Server and Oracle's

9iAS application server (java.sun.com/j2ee/connector/products.html). At the same time, 16 EIS and EAI products supported JCA, with an additional 16 announced or in beta. Connectable EIS include BEA's Tuxedo, SAP's R/3, IBM's CICS, IMS, and MQ/Series, JD Edward's, Oracle's, Peoplesoft's and SAP's ERP systems, and Siebel's CRM system.

14.6.4 Java and XML

Sun has been one of the early promoters of XML. Yet, initially, XML support for Java was limited to interfaces supporting the processing of XML documents, presenting the XML document (DOM) and XML streaming (SAX) models. More complete support for XML, including XML Schema, and support for web services standards has been added as a pre-release ("early adoption") in early 2002.

The Java Architecture for XML Binding (JAXB) provides an API and tools that automate the mapping between XML documents and Java objects, although the early access version released in early 2002 only supports DTDs, not XML Schema.

Java API for XML messaging (JAXM) is a J2SE optional package that implements the simple object access protocol (SOAP), v1.1 with attachments. Messaging profiles are included to enable pluggability of higher-level protocols.

The Java API for XML-based RPC (JAX-RPC) supports the construction of web services and clients that interact using SOAP and that are described using WSDL.

Java API for XML processing (JAXP) is a collection of DOM, SAX, and XSLT implementations.

Java API for XML registries (JAXR) provides uniform and standard access to different kinds of XML registries, including the ebXML registry and repository standard and the UDDI specification.

14.7 Interfaces versus classes in Java, revisited

Java separates and supports classes and interfaces, where an interface is essentially a fully abstract class. As Java classes can be fully abstract, Java programmers have to choose whether to use an interface or an abstract class. It is interesting to observe the current trend away from the use of classes and toward the increasing use of interfaces. As interfaces rule out implementation inheritance, this trend also favors object composition and message-forwarding techniques. A few examples of this ongoing trend are described below.

JavaBeans still allows classes to surface on bean boundaries. Even implementation inheritance across bean boundaries is commonly used, although usually to inherit from base libraries, not from other beans. However, the JavaBeans specification (as of v1.00-A) recommends not using the Java type test and

guard operators (instanceof and checked narrowing cast). Instead, programmers should use the methods isInstanceOf and getInstanceOf in class java.beans.Beans. The intention probably was to provide a hook for post-1.00 beans that can be represented to the outside by more than one object, similar to the QueryInterface mechanism in COM, although this has yet to happen.

The Java AWT event model has been changed from an inheritance-based solution in JDK 1.0 to a "delegation-based" (really a forwarding-based) solution in JDK 1.1. In addition to the resulting advantages and increased flexibility, it was admitted that the 1.0 approach led to undue complexity and was a source of subtle errors.

Whereas the non-distributed Java object model supports classes and interfaces, the distributed Java object model (based on RMI; see section 14.6.1) restricts remote access to interfaces. In other words, where distribution is used or planned for, direct use of classes is not advisable and all access should be via interfaces.

Enterprise JavaBeans also restricts bean access to interfaces, but it goes even further than RMI as there can be only one remote interface on an EJB bean. If an EJB bean would need to implement multiple interfaces, it suffices to define a single interface extending these interfaces.

It is interesting to see how the tension between interfaces and classes shaped up as the Java specifications evolved. For instance, the ratio of classes to interfaces has markedly shifted from almost exclusive use of classes in the early Java packages to much heavier use of interfaces in later Java specifications. A Sun initiative, code-named "Glasgow," aimed to generalize JavaBeans. Besides other enhancements (see sections 14.4.4 and 14.4.5), this specification was supposed to support object composition based on containment and aggregation. While Glasgow did add many new features to JavaBeans and the larger user interface toolkit, including an interesting container design, it did not deliver aggregation support.

14.8 JXTA and Jini

JXTA and Jini address a similar problem – the federation of systems over loosely coupled distributed systems. Jini focuses on Java everywhere, with a preference for (although not a strong restriction to) Java-specific networking protocols such as RMI (www.sun.com/software/jini/; www.jini.org; Arnold *et al.*, 1999). Jini also moves Java components to where they are needed. Thus, Jini requires the Java runtime environment on at least some of the involved computers and devices. The surrogate project (www.jini.org) created the Jini surrogate architecture, a design that enables non-Java endpoints to be hooked into a Jini federation by having a Jini surrogate act on their behalf. (In standard networking terminology, a surrogate is a protocol bridge.) For instance, a surrogate could be used to bridge between a Jini federation and a Universal Plug-and-Play federation.

JXTA (pronounced "juxta", short for juxtaposition; Gong, 2001) is a Sun initiative that aims at open peer-to-peer computing, preferring XML-based conventions and protocols on the network (wwws.sun.com/software/jxta/; www.jxta.org; Brookshier, 2002). There is no requirement that JXTA participants run Java, although JXTA naturally defines how Java-based endpoints can be formed. The Jini surrogate architecture can be used to bridge between JXTA and Jini.

Jiro is another Java-based approach that aims to use federation to specifically aid systems management. As it stands in early 2002, Jiro focuses even more specifically on the domain of managing storage systems. See section 18.5 for a brief discussion of Jiro.

All three initiatives (Jini, JXTA and Jiro) have some overlap with the space of web services. Jini's and Jiro's reliance on sending Java components across the network limits it to non-XML web services. JXTA builds on XML conventions and protocols, but, as it stands, does not use the standard protocols of XML web services, such as SOAP, WSDL, and UDDI (see section 12.4.5).

14.8.1 Jini – federations of Java services and clients

Jini is a specification that describes how federations of Java services and clients of such services can be formed over loosely coupled systems. Three problems need to be addressed – how clients locate services, how services make themselves available, and how the overall system copes with partial failure and reconfiguration. Jini addresses the problems of locating and publishing services and partially addresses the problems of robustness under various failure modes.

Jini defines special services called lookup services that serve two purposes. First, clients query the lookup services in order to find specific services and, second, services publish themselves by registering with lookup services. This introduces a bootstrap problem – how do clients and services locate lookup services? Jini defines both a broadcast protocol for lookup service location within smaller networks that support such broadcasts and a unicast protocol to hierarchically locate lookup services in larger networks that don't support full broadcasts.

When a Jini service registers with a lookup service, it specifies a desired lease period. This is similar to the RMI distributed garbage collection scheme (see section 14.6.1) as, once the lease expires, the lookup service removes the registration entry of that service. A lookup service is free to constrain the lease period and grant a lease only to an upper bound it can accommodate. In particular, Jini lookup services will never grant an infinite lease period.

A Jini service can register with more than one lookup service and multiple Jini services can register under the same service type. A Jini client can consult multiple lookup services. Hence, it is possible – and in a larger federation even likely – that a client locates multiple services that could be used. It then picks one of them and establishes contact. Once a client has located a service, Jini is

out of the loop. If a service fails while a client is connected, then it is that client's responsibility to either take appropriate measures or else simply fail as well. As Jini services can be – and often are – clients of other Jini services, such failure propagation can cause a ripple effect.

The dynamic proxy mechanism introduced in J2SE 1.3 can be used to isolate clients from certain service failures (Ledru, 2002). The idea is to automatically fail over from a failed service to an equivalent one and do so inside a proxy, such that a client doesn't observe such a change. Clearly, there is a strong assumption to be made – that the failed session didn't build up any session-specific or even longer-term state in the service. If such state existed, it would be lost on fail over. In other words, even with such smart proxies in place, it is still the responsibility of the developer of Jini services and clients to deal with partial failure modes at a state level properly.

14.8.2 JXTA – peer-to-peer computing

JXTA is supposed to span programming languages, platforms (such as Unix versus Windows), and networking infrastructure (such as TCP/IP versus Bluetooth). The JXTA protocols establish a virtual network on top of existing networks, hiding their underlying physical topology. Peer discovery and organization into peer groups is performed over this virtual network. The JXTA protocols also address the advertising and discovery of resources and communication among and monitoring of peers. The following five abstractions form the JXTA virtual network – uniform peer ID addressing, peer groups, advertisements, resolvers, and pipes. JXTA defines a search approach that helps to locate peers and resources efficiently in large networks.

Figure 14.10 shows the main layering of the JXTA architecture (Gong, 2001). JXTA comprises the following core protocols – peer discovery, peer resolver, peer information, peer membership, pipe binding, and endpoint routing. Peers and resources are identified uniformly by UUIDs (128-bit "flat" unique IDs). XML documents are used to advertise services and resources. Peers are required to support a set of peer-level protocols. In addition, groups of peers can form a virtual entity that supports additional group-level protocols. Communication is by means of messages sent via pipes. Pipes are directed, unreliable, unicast or multicast constructs that deliver messages between peer endpoints. Thus, a pipe can be seen as a named unreliable message queue. Higher-level communication semantics, such as reliable delivery, can be built on top of basic pipes. The motivation is to get as close as possible to the physical networks (say, UDP instead of TCP/IP), which usually do not provide reliable communication.

The peer discovery protocol serves a peer to discover its peers, peer groups and advertised resources (by discovering the describing advertisement, an XML document). If no ID is specified, this protocol will return all advertisements of a particular peer or peer group. By starting with a unique world peer

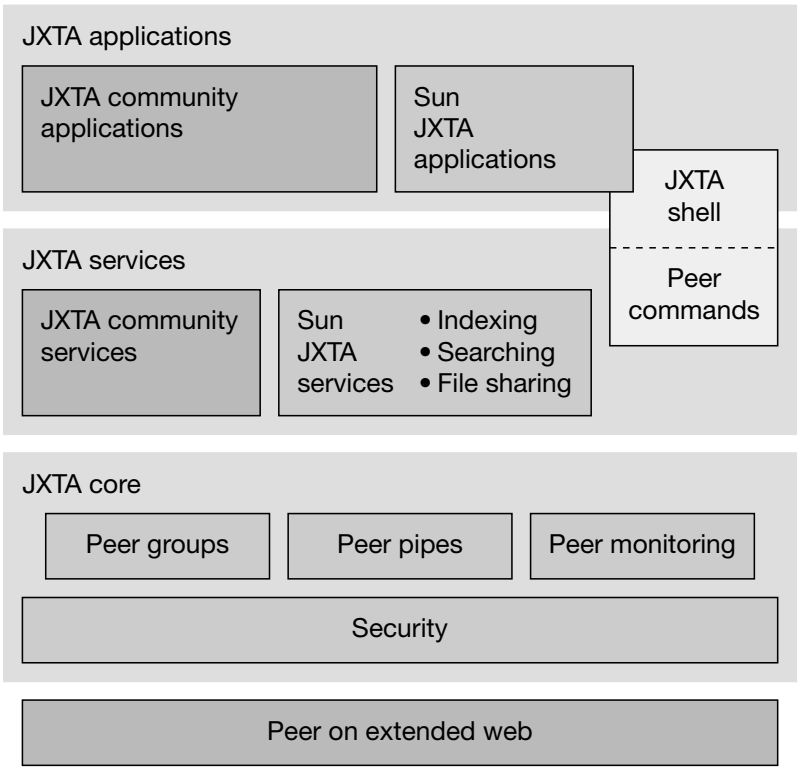

Figure 14.10 JXTA architectural layering. (From Gong, L. (2001) Project JXTA: A Technology Overview. Sun Microsystems, Palo Alto, CA. (**www.jxta.org/project/www/docs/TechOverview.pdf**).)

group, a system can be bootstrapped. Notice that this is conceptually very similar to the internet's domain name service (DNS) supported by a hierarchy of DNS servers, where nodes are identified by their IP address, which as of IPv6 is also a flat 128-bit value.

The peer resolver protocol is used to implement advanced search functionality. It is typically implemented by peers that maintain repositories. The peer information protocol can be used to retrieve status (liveness) and capability information on other peers. The peer membership protocol allows peers to discover credentials required to join a particular group, apply for membership, update membership and credential information, and cancel membership. Protection is provided in the form of authentication and security credentials.

The pipe-binding protocol allows peers to bind a pipe to one of their endpoints. Pipes can be created, opened, closed, or deleted. An open pipe can be used to send and receive messages. Opening a pipe binds it to an endpoint; closing it causes unbinding. Endpoints specify the actual sources and sinks of messages within a peer implementation. The peer endpoint protocol enables peers to become routers. Thus, if a peer needs to determine how to reach another peer, it uses this protocol to discover possible routes. Any peer that implements the protocol indicates that it can route messages over certain networks to certain other peers.

On reflection, JXTA is a bit like a virtual internet over the internet. While creating a clear opportunity for redesign of this space, it is not yet clear whether or not the benefits of a hopefully cleaner design can outweigh the disadvantages of separation from the internet via this abstraction layer. The internet was designed to scale to enormous size and has certainly successfully done so. Its protocols may show signs of their age, but they don't seem to show fundamental flaws that would indicate an approaching end of scalability. In particular, the large computers from the early days of the internet are now matched by the capabilities of even the smallest devices. Useful implementations of SNMP, HTTP, and TCP/IP can now be squeezed into tiny devices (www-ccs.cs.umass.edu/~shri/iPic.html; www.ipsil.com). It might well be that JXTA would be more effective by complementing the existing internet protocol family, including the emerging web services protocols.

14.9 Java and web services – SunONE

SunONE (Sun open network environment) is an extension of J2EE that relies on specialized servlets to handle web service protocols. SunONE also incorporates the J2EE server products formerly marketed by iPlanet (note that the alliance between Netscape and Sun that was called iPlanet expired in early 2002, leaving the iPlanet products with Sun. In early 2002, iPlanet accounted for around 7 percent market share in the J2EE space, following IBM's WebSphere and BEA's WebLogic at 34 percent each and closely followed by Oracle at 6 percent.)

With the release of early adopter versions of the Java web services developer pack (Java WSDP) in early 2002, Sun provides support for SOAP, WSDL, and UDDI. Java WDSP includes the Java APIs for XML messaging (JAXM), XML processing (JAXP), XML registries (JAXR), and XML-based RPC (JAX-RPC). It also includes the JSP standard tag library (JSTL), the Ant build tool, the Java WSDP registry server, the web application deployment tool, and the Apache Tomcat web server container.