Table of Contents for Chapter 9

<<<…>>>

Part 2. Architect's Toolbox - 'bird's view of the terrain'

<<<…>>>

Chapter 9. Java - 'coffee anyone?'

Java burst into IT scene like a meteor – seemingly unexpected, lightning fast and explosive.

'Seemingly' is an operating word here. In early 1990s, industry was longing for widely accepted, open interoperable middleware framework that would give Microsoft run for its money, at the very least.

Java does not replace Corba, but rather nicely complements it. Being a rich computing platform in itself, Java also provides glue for the integration of business components on the server, intranet and Internet across multitude of underlying platforms and technologies.

Java History and Origins

Let us be clear - if not for Java, some other similar proposition would capture mindshare of developer's community, because the time was ripe.
Java has become an almost accidental hero. History of Java presents a very good lesson, helps to understand background and place of Java in the scheme of things.

Without taking away from Java any well-deserved accolades, raise and raise of Java in the beginning followed more the *pull* scenario rather than *push*. In response, Java, like chameleon, changed its colours and adjusted to the demands of the market, re-positioning itself from initial primary emphasis on consumer devices.

Java (under its original name Oak, in honour of the tree outside the James Gosling's office) was developed as part of the Green project at Sun.
Project started in December 1990 by Patrick Naughton, Mike Sheridan and James Gosling, with the aim to build a handheld wireless PDA (called Star7 or *7) and TV set-top box.
Green team cuts regular communications with Sun and embarks on around the clock development for 18 months. Green team demonstrated the working prototype of *7 device in 1992. This is when Java mascot, Duke, was introduced to the world.

TV industry was not ready for such an innovation. Green team, now FirstPerson, realised that brighter future and broader market lies in Internet. FirstPerson returned to Sun.
Java technology was born from the concept of applet on the Web Browser.

On May 23, 1995 John Cage, director of the Science Office for Sun Microsystems, and Marc Andreessen, co-founder of Netscape, announced to the audience of SunWorld conference the launch of Java technology, and commitment to incorporate Java into the Netscape Navigator browser.

Popularity and acceptance of Java technology exceeded all expectations.
In March 1997, there were more than 220,000 downloads of JDK 1.1 software in just three weeks. In April 1997, JavaOne conference attracted 10,000 attendees, becoming world's largest developer conference.

Due to significant qualitative changes in JDK v1.2, Sun announces Java 2.
In December 1998, Java 2 platform ships and Java Community Process (JCP) is formalized. In June 1999, Sun announces three editions of Java platform – J2SE, J2EE, and J2ME.

From applet on set-top box and Internet browser, Java technology re-focused on server and middleware, and made a full circle by enriching client-side capabilities in consumer devices, as well as applets on the browser.

Java Framework

Figure 9.1 depicts all main components of Java in a whole of the *Java Framework* high-level view. Figure defines standard or out-of-the-box Java components that users expect to receive from the vendors of Java Framework who claim Java-certification and Java-compliance.

'Standard' Java Framework is a foundation that will be complemented in the Enterprise Architecture by optional Java packages and classes – vendor-specific or developed in-house. Optional packages deliver required business functionality and integration capabilities.
With evolution of Java Framework in *Java Community Process*, we expect more of the common-purpose optional packages will gain broad acceptance and will be folded into the 'standard' Java baseline.
Meanwhile, Enterprise Architects should watch the evolution of Java standards, and where the current demarcation line between the 'standard' and 'optional' Java resides – this is critical if interoperability between products from different Java vendors is a concern.

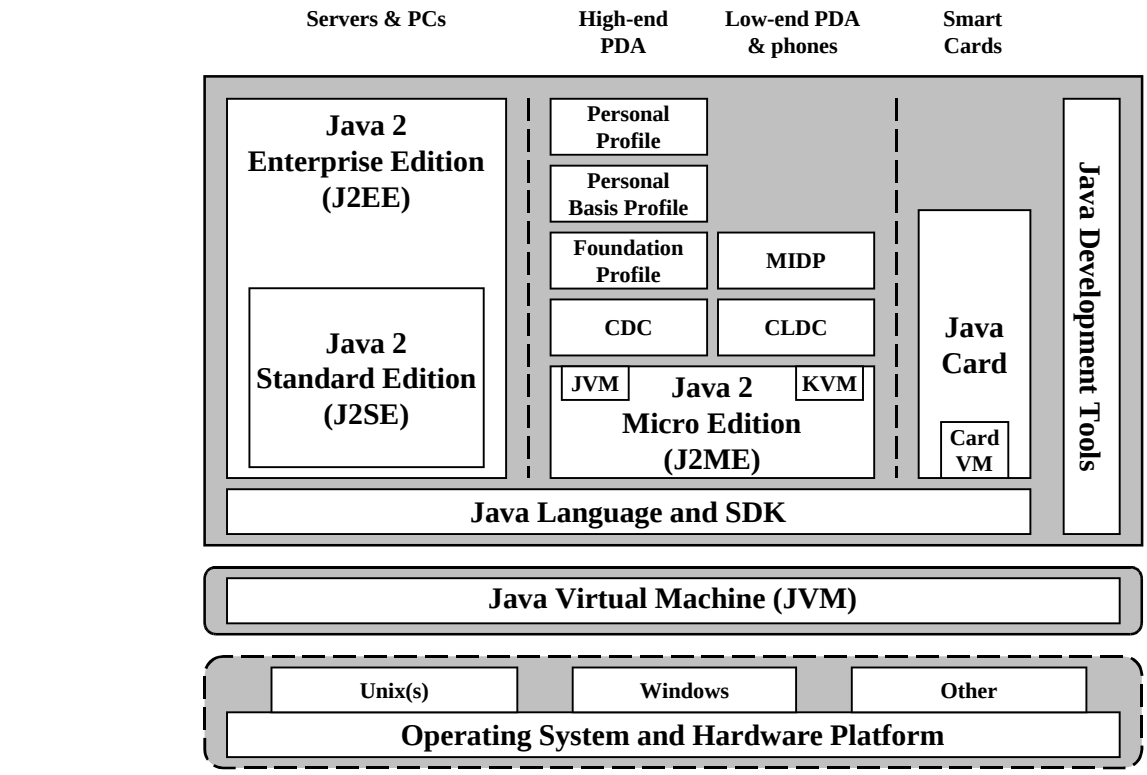

Figure 9.1. Java Framework Components

Java Framework consists of three separate platforms:

- Java 2 Standard Edition (J2SE) – Java language and core APIs
- Java 2 Enterprise Edition (J2EE) – enterprise-specific APIs required to build multi-tier distributed applications. Implementation of J2EE interfaces requires J2SE

- Java 2 Micro Edition (J2ME) – Java for small consumer devices

In addition, Java with smallest footprint targets micro-devices like Smart Cards.

At the moment of this writing, Java community widely uses *Java 2* term in describing components of the Java Framework.
We perceive the reference to version in discussions of Java Framework as a temporary notion that may be omitted in this book without any harm or confusion.
If (or, rather, when) Java community drops or changes version number from the reference to the Java Framework, most of the presentation in this book will remain current.

Java Standard Edition

Java Standard Edition is a foundation of the whole of Java Framework. Standard Edition is required for development and execution of Java applications on desktop and server.
Standard Edition defines Java language, standard *packages* (collection of related Java *classes* implementing some interface or function, or standard library of old), and program life cycle and environment in development and execution.

As a programming language, Java is a clear and simple true object-oriented language with emphasis on programmer's productivity and resilience of resultant program.
Java recognises programmer as a weakest link in common application development and lets system alleviate many error-prone situations in programming. Java strikes the balance between sufficient expressive power of the language for most of the programming challenges in the application development, and too much power that is often abused. As with any craft, powerful tool is good in hands of skilful master, but can be dangerous in hands of novice.

Java is a strongly typed, well-structured programming language. Java insulates programmer from the physical implementation or low-level data structures, thus greatly promoting the portability of Java programs across various platforms.
Many of common programming errors are to do with memory management, handling of address pointers, allocation and de-allocation of memory for program variables. These memory problems are very hard to detect, that is until 'memory leaks' take up all available space and bring application, or even server, down on their knees. Not so with Java (well, in theory at least). Java does not expose address pointer arithmetic to the programmer, and takes over responsibility for *garbage collection* in the memory – memory cleanup from no longer used program variables.
Java safeguards developers from the *possibility* of creating common memory-related problems.

Java stated goal – 'write once run anywhere' (WORA principle, for short), i.e. anywhere where Java Virtual Machine is available to interpret and execute the Java code.

As an object-oriented language, Java exhibits all uncompromising power of *inheritance*, *polymorphism* and *encapsulation*.
Inventors of Java had to make a judgement call in determining the most important object-oriented features that provided enough power without too much complexity, and did not constrain the developer in most of his or her application development tasks.
For instance, it was decided that classic multiple inheritance is too confusing and error-prone, and the effect can be more easily achieved by other means. Java leverages notion of abstract class that defines the *interface* only, without implementation. Any class *inherits* from one, and only one, class (denoted by keyword *extends* – Java slang for *inherits*). However, Java class can *implement* many additional *interfaces*, thus creating the effect of multiple inheritance.

Java may be a simple object-oriented programming language, but do not get fooled by this. Java Framework as a whole is a comprehensive and complex computing platform that requires great skill and commitment to master – a very rewarding and satisfying professional achievement.

Java Enterprise Edition

Java Enterprise Edition is a specification, not an implementation.

J2EE defines set of abstract interfaces that must be implemented by the vendor of Java Enterprise Edition in order for the Java Application Server to be certified, or be compliant to J2EE.

Important Enterprise interfaces inherited from the Standard Edition:

- ***Remote Method Invocation (RMI)*** – lets you invoke methods on remote objects through local stub (proxy) object. Stub/skeleton mechanism resembles one in Corba, but constrained to the communicating client and server both being in Java. Resemblance to Corba does not stop here - main standard transport for RMI is IIOP, Corba networking protocol. Unlike Corba, RMI handles objects *by value* and not *by reference*, i.e. ships the object over the network
- ***Java Database Connectivity (JDBC)*** – Java interface for connecting to relational databases and handling relational queries. All major relational databases have drivers for JDBC, or JDBC adaptors via ODBC. One can say, JDBC is ODBC done right, for Java

Java Enterprise interfaces in the Enterprise Edition:

- ***Java Server Pages (JSP)*** – Java tag-based scripting language that is mixed with HTML statements and, possibly, with snippets of Java and Javascript code. JSP compiles into Servlets
- ***Servlets*** – Java programs sitting on the receiving end of your URL and handling the generation of HTML page, XML script, or other required HTTP response. Servlets are like CGI programs in the Java world, only running in the Java Virtual Machine of J2EE Application Server's Web Container. *Servlets* were named so in contrast to *applets* – client-side Java programs on the browser. However, servlet's client does not have to be an applet – as the matter of fact, it does not have to be Java client at all, but any client that issues HTTP request
- ***Enterprise Java Beans (EJB)*** – transactional and distributed server-side Java Components. Run in their own EJB Container of the J2EE Application Server, and invoked over RMI-IIOP
- ***Java Naming and Directory Interface (JNDI)*** – Java interface for access to directories like LDAP, and for locating EJBs
- ***Java Messaging Service (JMS)*** – Java generic interface to various Message Queues
- ***Java Transaction API (JTA)*** – transactional Java interface modelled after OMG Object Transaction Service. Supports flat transactions. Can be used explicitly by the programmer, or implicitly in EJBs by defining properties of transactions in deployment descriptors
- *Java Mail*, and required for email attachments ***JavaBeans Activation Framework (JAF)***

Figure 9.2 presents J2EE components and interfaces.

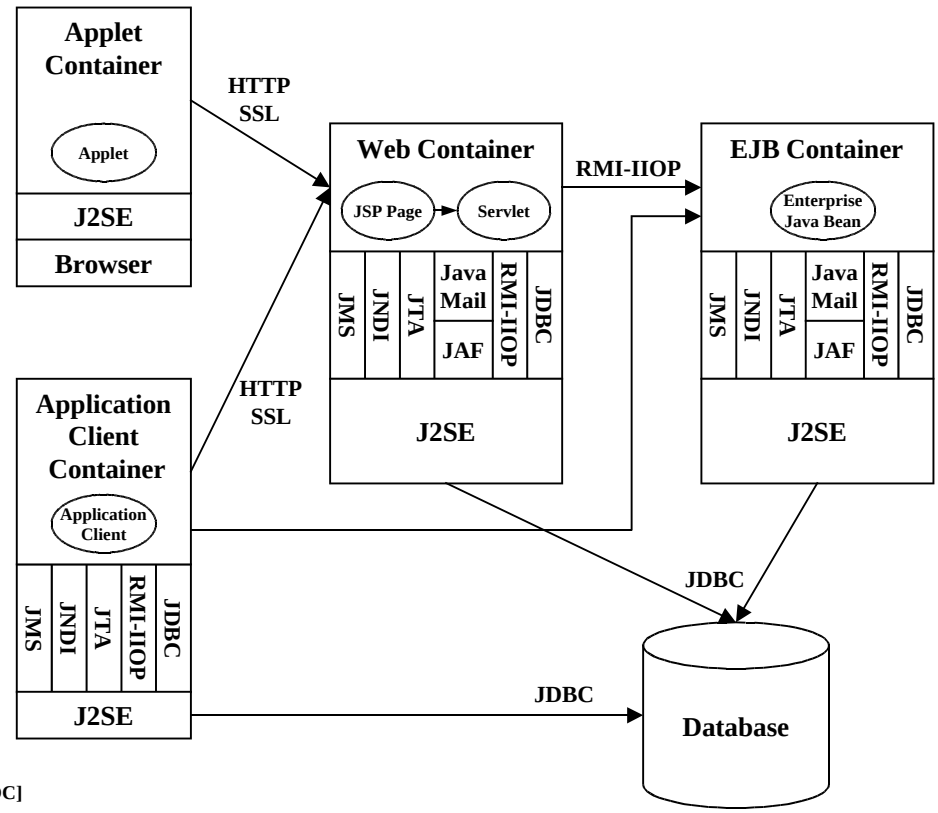

Figure 9.2. J2EE Components

Note that J2EE Web Container and EJB Container do not have to be in the same JVM, or on the same box for that matter. And, you can deploy and use one without another, as architecture of your application requires.

Enterprise Java Beans (EJB)

One source of confusion in Java Framework is distinction between JavaBeans and Enterprise Java Beans (EJB). JavaBeans and EJB are completely different beasts.
JavaBean is a Java class that complies with some naming conventions for methods and attributes (sometimes ambitiously called 'patterns') so that self-discovery of the JavaBean class through Java *reflection* and *introspection* is greatly simplified. JavaBeans are very popular in visual builder tools, but may be used in application development just as well.
JavaBean may present a viable alternative to EJB where distributed and transactional capabilities of EJB are not required. JSP tags provide easy interface to JavaBeans. If you can do without EJB overheads, do it.
In the enterprise computing, by 'beans' we usually mean EJB, not JavaBeans.

EJB technology has matured sufficiently to become a mainstream for scaleable, transactional, highly available, secure Java implementation of distributed business components in the Enterprise.
EJB insulates and implements business logic or integration adaptor that can be used by many client applications (as long as they talk Java). Applications can mix and match EJBs in various patterns. EJBs can run on different J2EE-compliant Application Servers.

Types of EJB:

- ***Session EJB*** – represents a single client session. Session EJB can be *stateful* or *stateless*. Stateful Session EJBs maintain the conversational state for the duration of the user session, possibly by persisting the session state on disk. Stateless Session beans may offer better performance, and more options for load balancing

- ***Entity EJB*** – represents a business obect in a persistent storage, and has a unique object identifier – *Primary Key*. Persistence in Entity EJB can be *Bean-Managed* (BMP) or *Container-Managed* (CMP). BMP provides do-it-yourself database access option for developers of Entity EJB with BMP, when all SQL requests to the database are written 'by hand'. In conrast, CMP allows for declarative handling of entity bean persistence, with EJB container providing database access and OO-Relational (object-oriented to relational) mapping for you – just sit back and watch your Entity EJB being persisted by EJB Container. Properties of EJB defined in EJB deployment descriptors in XML
- ***Message****-Driven EJB* – provides asynchronous interface to Message Queues via JMS. Message-Driven Bean (MDB) acts as a JMS message listener. Future implementations of MDB may be able to process other than JMS messages

EJBs with their Deployment Descriptors can be packages and shipped for deployment on other J2EE-compliant Application Server.

Java Micro Edition

Java Micro Edition (or, J2ME – Java 2 Platform Micro Edition) is the Java platform for consumer and embedded devices that address specific needs for portability and mobility in Java.
Examples of portable devices are mobile phones, PDA, TV set-top boxes, and great variety of embedded devices (your car or fridge may have one).

Portable consumer devices cater for stringent useability requirements with highly personalised user experience, and usually posses comparatively limited grunt, memory, and real estate for display and controls.

In short, Micro Edition is a scaled-down specialized subset of Java with smaller footprint.

Two main *Configurations* of Java Micro Edition are *Connected Device Configuration* (CDC) and *Connected Limited Device Configuration* (CLDC).
CDC is designed for high-end consumer devices. CDC includes full-featured JVM and large subset of the Java Standard Edition platform. Most of the CDC-targeted devices have 32-bit CPU and minimum of 2MB of memory.
CLDC is a smaller configuration for devices like mobile phones, pagers, low-end PDA. These devices typically have either 16- or 32-bit CPU, and a minimum of 128KB to 0.5MB of memory. CLDC runs on scaled-down implementation of virtual machine as well – KVM instead of JVM.

To cater for great variety of consumer devices, Java Micro Edition clearly separates device *Profiles* from platform *Configurations*.

Foundation Profile (FP) is the lowest level *Profile* for CDC, and can be used for deeply embedded implementations without the user interface.
Personal Basis Profile (PBP) and *Personal Profile* (PP) required for CDC-devices with Graphical User Interface. Personal Profile provides full GUI or Internet applet support on high-end consumer devices, such as high-end PDAs and game consoles.

Mobile Information Device Profile (MIDP) is designed for mobile phones and entry-level PDAs.

Java Community Process

Java Community Process (JCP) is a formal process for developing and revising Java technology specifications by open international organisation. Also, JCP builds the reference implementations and test suites.

JCP itself is constantly reviewed and refined to facilitate openness, broad acceptance of Java technologies and industry consensus on proposed Java specifications amongst various vendors and stakeholders.

JCP participation is open to everyone. However, orderly process is maintained by the Executive Committee, and by the Expert Group assigned to lead the JSR.

Sun played prominent role in setting up and leading the JCP, and remains the influential Executive Committee member. However, Sun does not control or dominate groups developing and maintaining Java specifications.

Java standardisation process is initiated by submission of proposal in a form of Java Specification Request (JSR).
JCP defines procedure, milestones and deliverables for the life cycle of the JSR from initiation phase, through to Public Review, to Final Approval Ballot at following Maintenance Reviews.

Upon initiation, JSR is allocated number for future reference. For instance, JSR 168 (Portlet Specification) aims to enable interoperability between Portlets and Portals by defining “APIs for Portal computing addressing the areas of aggregation, personalisation, presentation and security” [WWW JCP].

Many JSRs are in progress in different phases at any particular time.

<<<…>>>