The Java Programming Language/Environment and
Risks/Benefits of Software Engineering with Java

Drs. S. A. Demurjian, Sr. and D.-G. Shin
Computer Science & Engrg. Dept.
The University of Connecticut
Storrs, Connecticut 06269-3155
{steve,shin}@eng2.uconn.edu

Overview of White Paper

Over the past 10 years, there has been an explosive growth of object-oriented software design and development. In the late 1980s and into the early 1990s, C++ dominated the landscape, providing the baseline object-oriented programming features (encapsulation, inheritance, polymorphism, generics) on top of a system-oriented, terse language (C). In the mid 1990s, both Ada95 and Java have emerged to offer new opportunities that are targeted for diverse and significant market segments. Ada95 and its strong ties to DoD and government software, and Java with an increasing impact on commercial Internet-based and general-purpose software, both expand the base of software professionals working on object-oriented platforms.

Java was originally envisioned to augment the capabilities of world-wide web (WWW) browsers through the ability to write applets that can be downloaded from the Internet and manipulated safely within Netscape or Microsoft Explorer. In fact, one of the first commercially available books on Java [49] focused on this feature of the language. Despite this original focus, in the past year, interest in Java has exploded at education institutions, commercial enterprises, and government entities. This has resulted in the following since the first commercial version of Java became available in early 1996:

- Java is taught in educational institutions from freshman undergraduates through graduate student offerings, with textbooks widely available from all major publishers.

- Java is being utilized for distributed, Internet-based applications of all types, including graphical user interfaces (GUls), programming environments, mixed-programming language applications, upgrading and interfacing to legacy systems, etc.

- Java is attractive for general purpose, single-CPU development, since it has the potential to easily evolve software to multiple and varied hardware/OS platforms.

WWW, with its dissemination capabilities, offers a wide range of sites for gathering information on Java: Sun Microsystems [80], the Gamelan site maintained by EarthWeb, Inc. [58], and the Java Report Online maintained by SIGS publications. [79].

The goal of this white paper is to assess the status and future potential of the Java programming language and its associated environment and tool set. To meet this goal, the remainder
of this white paper is organized into six sections. In Section 1, an overview of Java is given, focusing on three areas: the Java environment which includes its platforms and capabilities; the Java language which includes a review of its main features; and the role of Java and its interactions with browsers like Netscape and Microsoft Explorer. While Section 1 briefly highlights the object-oriented features of the Java language, in Section 2, these features are examined in greater detail, providing a view of the language from the software engineering concepts of abstraction, encapsulation, inheritance, polymorphism, and generics. Comparisons to other object-oriented languages are included throughout the discussion of Section 2.

In Section 3, the usage of Java to develop large scale, multi-processor, distributed applications is examined. Successful distributed object computing (DOC) can be addressed from three perspectives. First, when developing new applications, it is often the case that multiple programming languages and varied paradigms must work together, e.g., a Java GUI, a C I/O package, and an SQL database system. This motivates the second perspective, involving the integration of Java with commercial-off-the-shelf (COTS) systems. Of course, when integration occurs, it may be necessary to be innovative and creative to allow interactions with legacy applications, which is the third perspective. Overall, DOC principles as supported in approaches such as DCE [39], DCOM/OLE [30], CORBA [38], must be critiqued for their utility and interaction potential with Java.

In Section 4, the discussion of Section 3 is concentrated on database support, which is often a critical and mandatory component for application development of all types. There are different levels of object persistence that are supported or planned for Java. First, there is the ability to store Java instances persistently in files so that they can be reloaded. Second, there are portions of the Java language that allow integration with SQL-based database systems, thereby providing both persistence and database management capabilities (concurrency, recovery, etc.). Third, there is the evolving relationship of Java to object-oriented database management systems, with most vendors providing/planning integration with Java.

Section 5 focuses on designing and developing Java applications by focusing on the major tools and building blocks that are provided by the development and runtime environment. Java and its role in graphical user interfaces (GUIs), networking, multi-processing via threads, and security and access control techniques, are all explored in detail. In addition, emerging Java development environments that support component-based and visual design of applications are investigated.

Section 6 concludes this white paper with a summary and discussion of the pros and cons of design and development using Java, a review of the future and emerging trends of Java, and recommendations regarding the future use and likely acceptance of the language. A detailed comparison of Java to C, C++, Ada95, Smalltalk-80, and Visual Basic is also provided, to allow the reader to place Java in context with other existing languages.
# Contents

Overview of White Paper 1

1 An Overview of Java 5
   1.1 The Java Environment 5
       1.1.1 The Java Virtual Machine 6
       1.1.2 Just-in-Time Compilers 6
   1.2 The Java Language 7
       1.2.1 The Core Non-Object-Oriented Capabilities of Java 7
       1.2.2 A Brief Tour of Object-Oriented Capabilities of Java 8
       1.2.3 The Java API Packages 9
       1.2.4 Advanced Features: Exceptions, Threads, and Streams 10
       1.2.5 The Java Formal Language Specification 13
   1.3 Java and Browsers 13

2 Object-Oriented Capabilities of Java 14
   2.1 Abstraction and Encapsulation 14
   2.2 Inheritance 16
   2.3 Polymorphism via Dispatching 19
   2.4 Generics 19
   2.5 Java vs. Smalltalk 20
   2.6 Summary of Java's Object-Oriented Capabilities 23

3 Integration and Distributed Object Computing 23
   3.1 A Brief Overview of DOC Models 24
   3.2 The Role of DOC in Java Design and Development 27
   3.3 Integrating Java with Other Programming Languages 28
   3.4 Interfacing Java Applications with Legacy/COTS Systems 30
       3.4.1 The Data-Transformation Approach 30
       3.4.2 ORBs and CORBA 31
       3.4.3 Wrapper via RPC and JNI 32
       3.4.4 Integrating Multiple Legacy/COTS Applications 33
       3.4.5 Java and Future COTS Applications 34
       3.4.6 Java APIs for Legacy/COTS Applications 34

4 Java and Database Support 35
   4.1 Relational Databases: Java Database Connectivity (JDBC) 35
   4.2 JDBC API and JDBC Driver API 36
   4.3 Object-Oriented DBMS 39
   4.4 Available Capabilities for Persistence 40
       4.4.1 Object Serialization 41
       4.4.2 JDBC 41
       4.4.3 JSQQL 42
       4.4.4 OODBMSs 43

5 Designing/Developing Java Applications 44
   5.1 Java and GUs 44
       5.1.1 The AWT API 44
       5.1.2 Java Foundation Classes 46
   5.2 Java and Threads 47
       5.2.1 Creating Threads 47
       5.2.2 Thread Scheduling 48
       5.2.3 Synchronization 48
1 An Overview of Java

Java is a third generation, general-purpose, platform-independent, concurrent, class-based, object-oriented language and associated environment, designed to be simple enough so that software engineers can quickly achieve fluency in the language and effectively utilize the language and environment for developing applets and applications. Java can be utilized in two distinct ways. In its original conceptualization, Java can be used to write special programs called applets that can be downloaded from the Internet and displayed/manipulated safely within a WWW browser. In addition, Java can also be used to design and develop standalone applications, with a wide range of capabilities and functionality. Platform independence is achieved in Java by using bytecode, which is similar in concept to the p-code used in early Pascal compilers that revolutionized compiler design and development. Thus, the Java compiler does not produce native executable code for a particular machine. Instead, the Java compiler produces a special format called Java bytecode that is finally interpreted to the native language of the host machine on the fly.

The remainder of this section provides an overview of the key components and features of Java. In Section 1.1, the Java environment is detailed, focusing on the Java Development Kit and Runtime Environment, the platforms available for Java, and other general issues. Section 1.2 explores the Java language through a presentation of its core non-object-oriented capabilities, its object-oriented features, its rich set of API packages, its support for exceptions, threading, and streams, and its formal language foundations. Finally, Section 1.3 reviews the strong role of Java and WWW via the usage of program applets that are accessible and executable by WWW browsers like Netscape and Microsoft Explorer.

1.1 The Java Environment

Java has two main components, the Java Development Kit (JDK) and the Java Runtime Environment (JRE). The JDK component is a package of programs and support files which is needed to develop Java programs. Included in the support files are class source code and documentation for the complete Java class hierarchy. The JDK contains the command-line driven javac Java compiler. The Java Debugger (JDB) is included with the JDK. It is also command-line driven, and has syntax similar to that of the UNIX dbx and gdb debuggers. The JRE component is needed in order to execute Java applications. It consists of the bytecode interpreter and other files such as the code verifier, as we will discuss in Section 1.1.1. A discussion of browsers and Java applets which can run without the JDK and JRE, is deferred until Section 1.3. Version 1.1.3 of both the JDK [81] and the JRE [93] are available from Sun for Microsoft Windows 95/NT 4.0, SPARC Solaris 2.4 and 2.5, and Solaris 2.5 x86 (non-final, early access version 1.1.3). In addition, there are a wide variety of platforms available as third-party ports [82], including: AIX, Amiga, BeOS, Digital Unix, FreeBSD, HP 3000, HP-UX, IRIX, Linux, MacOS, Netware 4.1/IntranetWare, OpenServer 5, OS/2 Warp, OS/390, OS/400, Psion Series 3/EPOC 16, RiscOS, RiscBSD, and UnixWare. An on-line discussion of general issues related to the Java platform is also available [92].

Although only the JDK, the JRE, and a text editor are required to create and execute Java programs, many integrated development environments (IDEs) have quickly emerged to facilitate Java development. Most notable of these is Microsoft Corporation’s Visual J+++. Others include Cafe (by Symantec), Kawa, JPad, and Javelin. Most are available in shareware versions. All of the various IDEs have similar features such as browsing the class inheritance tree in one window while editing source files in another window, file compile and project build options, and automatic source file error line tracking. It is important to note that even with an IDE, the JDK and JRE
are still needed. The IDEs only provide a friendly user interface to the command-line driven JDK and JRE. In the remainder of this section, we examine in detail the compilation and runtime environment and process for Java.

1.1.1 The Java Virtual Machine

In a normal compiled executable program, the object file contains the processor instructions to be executed, and the processor executes the instructions. In order to support platform independence, Java must provide an execution environment that can oversee the execution of applets and applications. The Java Virtual Machine (JVM) is utilized for this purpose. JVM is a program which runs on a particular hardware/OS platform (or ‘real’ machine) which interprets and executes a Java applet/application that is contained in a .class file. The .class file contains both executable JVM instructions (called bytecodes), and additional information such as the class structure, method and data member visibility, and superclass information. Since each JVM interprets the same set of bytecodes, true program portability is achieved by implementing JVMs for a wide variety of platforms. The JVM is also intended to be a small and efficient program, so that Java can be used on computers with fewer resources or, as it is planned, “... in consumer electronic devices such as televisions and cellular phones.” [36, page 976]

Hereafter the JVM will be discussed as though it were a real processor. One must remember that it is only a program which simulates a real processor. The JVM is a stack-based machine, and as such, contains a set of registers that are used to track the ongoing execution/computation. Registers are maintained for the program counter, the local variables needed by a method, the stack of current operations, and a frame pointer for tracking the status of the execution environment [36, page 735].

The JVM implements a verifier which is used to check classes when they are loaded, to “... make sure that all external object references are correct and allowed.” [36, page 738] Since Java is a dynamically bound language, and more precisely, a late bound language (binding occurs at the last possible moment), there is the possibility that a .class file may have been changed either purposefully or inadvertently. After verification of the .class file, it is either allowed to run or prevented from running. The bytecodes executed by the JVM closely resemble assembly instructions for a RISC (reduced instruction set) processor. There is a large proportion of instructions dedicated to moving data to and from the stack, and control transfer instructions. There is also a small number of instructions specifically to handle exceptions and critical regions. The Java standard does not allow programming directly in the bytecode language.

1.1.2 Just-in-Time Compilers

There have recently been dramatic advances in Java compilers. Since the bytecodes are so well defined and so low-level, it is possible to compile sequences of bytecodes directly into native machine instructions. These just-in-time (JIT) compilers can supply dramatic performance improvements at runtime. To illustrate this process, and serve as a basis for discussion, a diagram reproduced from [36, page 978] is shown in Figure 1. Normal JVMs, when executing a class (via a .class file), create a table (known as a virtual table, or V-table) of method names pointing to method bytecode lists. When a method is executed, the bytecode list is looked up in the V-table and executed. With a JIT compiler, the V-table is still created, but each method name points to the JIT compiler itself, with the method’s bytecode list in memory in a different location. When a method is executed, the JIT compiler compiles the required bytecode list to native processor
1.2 The Java Language

This section provides an introduction to the Java language. The discussion is organized into five parts. First, the non-object-oriented capabilities of Java are reviewed, focusing on basic language capabilities, types of variables, and supported language statements. Second, the major object-oriented concepts are examined, including the abstraction/encapsulation support of Java via classes and packages, and inheritance. Next, the application programming interface (API) package concept is detailed, by considering its relationship to other modular based languages and by synopsizing the APIs supported in Version 1.1.3 of JDK with representative examples given. Then, advanced features of the Java language are explained, concentrating on exception handling, threading, and streams. Finally, we briefly report on the formal language specification for Java.

1.2.1 The Core Non-Object-Oriented Capabilities of Java

The core non-object-oriented capabilities of Java are based on C and C++, but organized differently, with redundant concepts omitted and some features from other languages included. For example, there are no preprocessor instructions, no typedef's, and no structure nor group types, all of which are cornerstones of C program development. Header or include files are not supported in Java, having been replaced by the package and import constructs to promote modular-oriented development.

The lexical structure of Java is based on C and C++ but it uses the Unicode character set, which allows for up to 34,168 different characters. Thus, Java is intended to support programming in English and other language alphabets. Java supports the writing of Unicode characters on systems that support only ASCII. Unlike C and C++, Java is a strongly typed language. Thus, many unsafe constructs have been excluded from Java. For example, automatic coercions are not supported; rather, all type changes due to assignment to a different type variable must be done with an explicit cast.
Variables in Java are handled in a much different fashion than C or C++. Basically, there are two types of variables in Java:

- **Primitive types** are variables that are the same regardless of the hardware/OS platform, and include a series of two's complement integers, floating point numbers, boolean, and Unicode characters.

- **Reference types** are used for arrays, classes, and interfaces, to allow variables to be created dynamically as instances.

Variables of primitive types hold values of their type, while variables of reference types hold "pointers" to an instance of that type. The term pointer is used in a logical sense, since there are no physical pointers in Java, e.g., a software engineer cannot develop a linked list in Java which would be possible in a language like C. Since the Java language does not support C structures or unions, and arrays and strings are objects, there is no need for physical pointers. The Java runtime system checks all array indexing to insure indices are within the bounds of the array.

The language statements of Java strongly exploit C syntax and semantics. Assignments and expressions are taken directly from C, with all of the typical operators for mathematical calculations, prefix and postfix increment and decrement, relational and conditional, bitwise calculations, and short-cut assignments. Precedence of these operators, while not identical, is very similar to C++. Control flow statements for looping (for, while, do-while), conditionals (if-then and switch-case), and others (break, continue, label; return) are identical to their C and C++ counterparts. Exceptions are more complex in Java, and will be considered separately in Section 1.2.4.

### 1.2.2 A Brief Tour of Object-Oriented Capabilities of Java

This section contains a brief synopsis of the key object-oriented features and capabilities of the Java language. These concepts are revisited in greater detail in Section 2. The main modeling capability is the Java class, which is similar to a C++ class. Within a Java class, a member (method or variable) can be tagged as private, public, protected, or package (default). A partial class for grocery items at supermarkets is given below:

```java
class Item {
    private String UPC, Name;
    private int Quantity;
    private double RetailCost;
    protected double WholeCost;

    public Item() { ... };
    public void finalize() { ... };
    public boolean Modify_Inventory(int Delta) { ...};
    public int Get_InStock_Amt() { return Quantity; };
}
```

Classes that are related to one another can be grouped together into the package abstraction. For example, a GroceryStore package:

```java
package GroceryStore;
```
could be declared with classes for grocery items, a GUI for store managers, and the UPC scanner
used by cashiers, to name a few. When a package needs to utilize another package or classes that
are external, one or more import statements must be employed, which will be further examined
in Section 1.2.3.

The other key object-oriented feature of Java is its support for inheritance. Inheritance has two
roles in programming. First, it allows controlled sharing between classes, allowing commonalities
to be generalized to some classes, with differences specialized to other classes. Second, inheritance
is incorporated to allow instances of different classes to be treated in a uniform fashion, when it
is semantically meaningful to do so. Inheritance is supported in Java by using the extends
keyword when declaring a class. For example, a number of descendants of Item can be defined
that form an inheritance hierarchy:

class DeliItem extends Item { ... }
   Item
   \  /
   class SaladItem extends DeliItem { ... }; DeliItem ProduceItem
   |  |
   class ProduceItem extends Item { ... }; SaladItem

Note that a discussion on the different ways to define classes and packages, and their visibility
within a Java application is deferred until Section 2.

1.2.3 The Java API Packages

In the early-to-mid 1980s, module based languages like Modula-2 [48] and Ada83 [2, 5, 29] were
proposed. Both of these languages supported abstract data type (ADT) concepts [28, 42]. As a
result, from a public interface perspective, both Modula-2 and Ada83 require software engineers
to clearly identify those portions of a module/package that are exported to the outside world.
To complement this, whenever a software engineer designs and develops a new module/package,
s/he would be required to exactly import those functions and procedures needed from other
modules/packages. Thus, both Modula-2 and Ada83 provided a rich set of module/package
libraries for use in developing applications. Java, through its public interface capabilities and
package concepts (see Sections 1.2.2, 2.1, and 2.2) also requires a clear definition of the exported
portion of all classes/packages, which requires software engineers to specifically enumerate which
packages, classes, and/or methods are imported. Thus, like Modula-2 and Ada83 (and of course,
Ada95), Java provides a set of application programming interface (API) packages.

The Java Platform 1.1.3 Core API, available online [64], is shown below:

```
package java.applet
package java.awt
package java.awt.datatransfer
package java.awt.event
package java.awt.image
package java.rmi
package java.rmi.dgc
package java.rmi.registry
package java.rmi.server
package java.security
```
<table>
<thead>
<tr>
<th>package java.beans</th>
<th>package java.security.acl</th>
</tr>
</thead>
<tbody>
<tr>
<td>package java.io</td>
<td>package java.security.interfaces</td>
</tr>
<tr>
<td>package java.lang</td>
<td>package java.sql</td>
</tr>
<tr>
<td>package java.lang.reflect</td>
<td>package java.text</td>
</tr>
<tr>
<td>package java.math</td>
<td>package java.util</td>
</tr>
<tr>
<td>package java.net</td>
<td>package java.util.zip</td>
</tr>
</tbody>
</table>

Many of these APIs will be discussed in later sections of this white paper, including the java.awt (Abstract Windows Toolkit) for GUI development, java.beans for reusable software components that can be visually manipulated in builder tools, and java.security for private and public key encryption. Each API contains a complete description of the package, which includes the classes and public methods that can be imported and utilized when developing Java applications. For example, the package java.security [75] contains five classes: Certificate, Key, Principal, PrivateKey, and PublicKey. Each class is then described according to the methods in the public interface that are importable by other classes/packages. For instance, the Certificate class contains methods such as decode(), encode(), getFormat(), getPublicKey(), and so on. In addition to these APIs, there is also a on-line list [65] that tracks all Java APIs, including ones in the planning stages with their probable release dates.

### 1.2.4 Advanced Features: Exceptions, Threads, and Streams

Exceptions are traditionally intended to handle situations where an error in an executing program can be anticipated, caught, and processed (e.g., divide by zero). Exception handling in a programming language involves a number of concepts and constructs. First, there must be a language construct to encapsulate a block of code that has the potential to raise an exception. As the code within this block is executing, various conditions can be checked, and when the correct situation occurs, an exception can be raised or thrown. This thrown exception is then processed by another block (requiring another language construct) that typically follows the original block.

Exception handling is provided in programming languages so that software engineers can differentiate error handling actions from the "regular" code, promoting separation of concerns during design and implementation.

Since Java is object-oriented, exceptions are handled with an object-oriented emphasis. When an exception is raised within a Java method, an exception object is created and handed to the runtime system, throwing an exception. This exception object contains information about the context in which the exception occurred. To process the exception object, the runtime system tries to find a handler for the exception by performing a backward search through the call stack. If there is no handler to catch the exception, the Java program terminates. Java exception handling is contained in the java.lang API in the Throwable class. All Java exceptions must be instances of the Throwable class or one of its descendents. The definition of the exceptions that a method is able to throw must form part of the public interface of the method. This is different from C++ where this definition is optional. There are some checked exceptions that are required to be handled (like IOException), while runtime exceptions may or may not be handled at the discretion of the software engineer.

To support exceptions in Java, a number of language constructs are available, including the try, catch, throws, and finally statements. The try, catch, and throws statements in Java operate in a similar fashion to their C++ counterparts. However, unlike C++, Java exception handling includes the finally statement, which is used to clean up the state of the computation in the event of an exception. The finally statement is always executed at the end, regardless of
the way the try statement ended, and is intended to leave the system in a consistent state. The finally statement is useful to avoid repeating code in the catch statements for those things that have to be executed in any case. The finally statement is especially important when unexpected runtime exceptions occur and they are not handled within the same method. This is since the compiler does not require the definition of a handler for a runtime exception but at runtime these exceptions may occur and the runtime system may try to handle them in a higher level method.

To illustrate the exception handling capabilities of Java, consider the writeList method in Figure 2, reproduced from [6, page 273], within which try, catch, and finally blocks have been included. The try block of code contains the actions of the writeList method that will cause various kinds of exceptions to be raised, each of which is caught and processed accordingly. Exceptions will be raised in two situations, when I/O errors occur involving the calls to FileOutputStream and when an array access goes out of bounds as a result of the call to vector.elementAt(i). Both of these two Java methods contain, in their implementations, the throws statements for the two exceptions that are caught in writeList. The two catch state-

```java
public boolean writeList() {
    PrintStream pStr = null;

    try {
        System.out.println("Entering try statement");
        pStr = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("OutFile.txt")));

        for (i = 0; i <= size; i++)
            pStr.println(vector.elementAt(i));
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught out of bounds exception" + e.getMessage());
    }
    catch (IOException e) {
        System.err.println("Caught I/O exception" + e.getMessage());
    }
    finally {
        if (pStr != null) {
            System.out.println("Closing PrintStream");
            pStr.close();
        } else {
            System.out.println("PrintStream not open");
        }
    }
}
```

Figure 2: An Exception Handling Method in Java.

ments in writeList handle exceptions for the attempt to access an array index that exceed the bounds of the array and an I/O exception related to the attempt to open OutFile.txt. Regardless of the situation, the finally block does the clean up that is required whenever either of the exceptions is thrown.
To support multi-process and multi-tasked application development, Java provides the ability to spawn and manage multiple threads of control. Recall that a thread is a single sequential flow of control within a program. In languages like C, it is possible to spawn multiple processes from the same program as separate threads of control. Java takes this concept a dramatic step forward, to allow the use of multiple threads in a single program which may all run at the same time, synchronized or not, while performing different tasks. This leads to a more elegant and straightforward support for concurrent programming.

A thread is called a lightweight process, since it runs within the context of a program and takes advantage of the resources allocated for that program’s environment. Java threads are implemented by the Thread class, which is part of the java.lang API. Thread implements a system-independent definition of Java threads, with the actual implementation of concurrency provided by a system specific implementation. This is since threads in Java must be OS dependent. On certain platforms Java threads must run preemptively. On others, they do not, and programs must call the sleep() or yield() function of the Thread class in order to allow other threads to obtain CPU cycles. The Thread class also provides synchronized methods, which act identically to critical sections, and methods like wait() and notify() to schedule threads. The Thread class is versatile to allow for different computing models of threads to be supported based on the target hardware/OS platform.

To use threading in a Java applet or application, a thread can be declared in two distinct ways:

1. by defining a subclass of the Thread class and overriding the run() method, or
2. by creating a thread that implements a Runnable interface.

The option that is chosen depends on the software engineer’s criteria. For instance, implementing a Runnable interface is chosen whenever the thread is already a subclass, since there is no multiple inheritance. Threading is similar in concept to tasking in Ada. In Ada, tasking allows a software engineer to establish multiple tasks that communicate and synchronize with one another. Tasking in Ada is included at the language construct level, with specific keywords and language constructs provided to support multi-process behavior. This is stark contrast to Java, where the threading is part of an API package and not a part of the language syntax nor semantics.

The other critical aspect for writing Java applications or applets is the utilization of streams, which are essentially a flowing sequence of characters. Java has the java.io package [88] that contains a series of input and output streams that programs can use to read and write data. The two main classes of java.io are InputStream and OutputStream, which are abstract superclasses that provide a minimal programming interface and a partial implementation of input and output streams, respectively. InputStream [89] defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. OutputStream [90] defines methods for writing bytes or arrays of bytes to the stream.

InputStream and OutputStream are roots of inheritance sub-hierarchies in the java.io package, that are specialized to implement specific input and output functions. For example, PipedInputStream and PipedOutputStream are used to implement pipes, which support the ability to channel the output from one program (or thread) into the input of another, i.e., analogous to Unix pipes. FileInputStream and FileOutputStream are used to read and write files in the native file system. This provides the abstraction for platform independence I/O; the implementations of these two classes in the JRE maps down to the specific hardware/OS configuration.
The classes ByteArrayInputStream and ByteArrayOutputStream are used to read and write byte arrays. Finally, StringBufferInputStream is used to read from a StringBuffer, but it does not have an output partner, e.g., similar to sscanf in C which allows the reading of input from a string.

In addition to these basic capabilities, advanced I/O features are supported. For example, it is possible to process filtered streams. A filtered stream is attached to another stream to filter data as it is written to or read from the original stream. The java.io package contains the following filtered streams: DataInputStream, DataOutputStream, BufferInputStream, BufferOutputStream, LineNumberInputStream, PushbackInputStream, and PrintStream. Software engineers can customize filtered streams by defining a subclass of FilterInputStream or FilterOutputStream and overriding at least the read and write methods. Finally, Java also provides support for random access files to permit non-sequential, or random, access to the contents of a file. RandomAccessFile [91] is a class of the java.io package, used to both read and write files, depending on the parameters given to the constructor. This class does not inherit from InputStream or OutputStream, and so the same filters cannot be applied to random access files. However, this class does implement the DataInputStream and DataOutputStream interfaces.

### 1.2.5 The Java Formal Language Specification

The formal language specification for Java has been documented extensively in written [22] and electronic [63] forms. Java is an LALR(1) grammar, which uses left-to-right input scanning, constructs a rightmost derivation in reverse for an input source program, and uses one lookahead symbol. LALR(1) grammars are supported by the compiler writing tools lex and yacc. As a grammar, Java is simpler then Ada95 and C/C++, since many of its capabilities are embodied in the APIs rather than within language constructs. For example, threading in Java is an API, unlike Ada which has a separate language construct. As indicated in earlier sections, Java has eliminated many of the more dangerous C/C++ language features, and as such, the grammar is more complete in its capturing of all legal language statements. This has lead to Java being a strongly-typed language like Ada, unlike C/C++, which often allow software engineers to circumvent language requirements when writing programs.

In fact, as documented in Chapter 19 of [22], there are only six major difficulties with the grammar which are generally traced to the syntactic oddities of C and C++ features that are used in Java. Let us briefly consider one such grammar difficulty. Many Java statements involve a sequence of identifiers separated by periods. Referring to a class within a package (GroceryStore.Item) or calling a method (victor.elementAt(i)) are two such examples. But, it is not possible with only a single lookahead to differentiate between these two cases. Thus, the grammar for Java was rewritten to simply recognize names, which are then reconciled later based on context when more input has been parsed. The key issue is that despite the fact that Java is a substantial object-oriented programming language, with a sizable grammar, there were only six non-LALR(1) language difficulties that had to be resolved. This indicates that unlike C and C++, Java has stronger theoretical programming language foundations.

### 1.3 Java and Browsers

Applets are small, independent programs that are written for the explicit purpose of being embedded into WWW pages, and once embedded, are then executable via a Java-enabled browser. This is in contrast to stand-alone Java applications which run independently of a browser. The
three most popular and best supported Java-enabled browsers are Netscape from Netscape Communications Corporation, Internet Explorer from Microsoft Corporation, and HotJava from Sun Microsystems. There are numerous other WWW browsers available, but many of them do not support Java.

WWW browsers provide a feature-rich environment in which a Java applet can run. An applet can run on any system with a Java-enabled browser, even if the system has no other Java software installed. Applets running in browsers are subject to severe security restrictions, including:

"1. applets can never run any local executable program;
2. applets cannot communicate with any host other than the server from which they were downloaded ... 
3. applets cannot read or write to the local computer's file system ...
4. applets cannot find any information about the local computer, except for the Java version used; the name and version of the operating system; and the characters used to separate files ..., paths ..., and lines ..."[8, page 320]

Netscape is probably the most popular Java-enabled browser, with versions available for X-Windows, Solaris, Windows 3.1, Windows 95, and Windows NT. Internet Explorer is available only for Windows 3.1, Windows 95, and Windows NT platforms. HotJava 1.0 is available for SPARC Solaris (version 2.5, CDE only), Windows NT for the Intel processor, and Windows 95 systems (256 colors or better). HotJava will not run on Solaris 1.x (SunOS 4.x), Windows 3.1, or Windows 3.11. HotJava was written in Java, and is programmable and extensible using Java. As a result, HotJava’s performance is slightly worse than Netscape and Explorer. HotJava allows varying levels of security with signed applets. A digital signature can indicate that the applet is trusted and can be given access to more system resources.

2 Object-Oriented Capabilities of Java

This section reviews the ability of the Java programming language to support the key software engineering and object-oriented capabilities, namely: abstraction and encapsulation (Section 2.1); inheritance (Section 2.2); polymorphism via dispatching (Section 2.3); and generics (Section 2.4). These topics are examined in turn by discussing the concepts, detailing their realization in Java, and contrasting Java’s capabilities with C++ and/or Ada95. Note that hiding is discussed when reviewing abstraction/encapsulation and inheritance, since both capabilities involve visibility. This section also compares Java to Smalltalk (Section 2.5), to carefully critique Java against what many consider to be a pure object-oriented programming language. This section concludes with summary remarks (Section 2.6).

2.1 Abstraction and Encapsulation

Any domain or application can be divided and decomposed into major building blocks and components, achieving a separation of concerns. This decomposition allows the application requirements to be further defined and refined, while partitioning these requirements into a set of interacting components, thereby achieving a modularity of behavior. In object-oriented programming, abstraction and encapsulation are utilized as the major building blocks. Through abstraction, the details of an application’s components can be hidden, providing varying levels of detail on the composition of a design. Abstractions can range from base data types (integer, float, string,
etc.) to user-defined data types (ADTs or classes) to constructs that allow groups of ADTs to be bound into a single conceptual unit (packages or modules). Regardless of the approach, the key is to encapsulate the internal structure and function of each abstraction, thereby hiding the implementation from those other components that are not allowed access. This also allows changes to be made to the internal structure and function of each component, achieving representation independence, since the external view of the abstraction is not impacted.

Java supports three levels of abstraction/encapsulation: class, package, and world. As we will see in Section 2.2, the class abstraction is extended when inheritance allows superclasses and subclasses to be defined. The class abstraction/encapsulation in Java is similar to an ADT at the design level, or a C++ class/Ada95 package at the programming level. Within a Java class, a member (method or variable) can be tagged as private, public, protected, or package (default). Public, protected, and private are similar to C++, and allow the software engineer to define access with respect to the users of a class. Consider the class defined below for grocery items at supermarkets:

```java
class Item {
    private String UPC, Name;
    private int Quantity;
    private double RetailCost;
    protected double WholeCost;

    public Item() { ... };
    public Void finalize() { ... };
    public boolean Modify_Inventory(int Delta)
    { int i = Get_InStock_Amt();
        if (Delta >= 0) {
            Quantity += Delta;
            return true;
        } else { return false; }
    }
    public int Get_InStock_Amt() { return Quantity; };
    public double Compute_Item_Profit()
    { return (RetailCost - WholeCost); }
    protected boolean Modify_WholeSale(double NewPrice); { ... };
}
```

Public, protected, and private members are all visible to the class itself. Public members are visible to the outside world, which includes all other classes and packages. When a member (variable or method) declaration is preceded by the modifier static, it becomes a class member, available to all instances of a class. In this case, the encapsulation of a member expands from within a single instance of a single class to all instances of a single class. In Item, private members (variables or methods) are visible within the implementations of the methods, as shown in the above Item class declaration. Classes which contain declarations to variables of their own types can access the private data. For example, if the Item class contained a method with a parameter of type Item, say I1, both this.Quantity and I1.Quantity are accessible within the Item class declaration.

The highest level of abstraction/encapsulation in Java is the package, which allows collections of one or more classes to be bound into a single named unit. For example, consider the two declarations of a GroceryStore package:

```java
package GroceryStore;   package GroceryStore;
```
class Item { ... };

public class Item { ... };

class ManagerGUI { ... };

public class ManagerGUI { ... };

class Scanner { ... };

public class Scanner { ... };

...               ...

with classes defined to represent grocery items, a GUI for store managers, and the UPC scanner used by cashiers, to name a few. In the version on the left, the classes are only visible within the package in which they are defined. In the version on the right, any class preceded by the public qualifier is visible both within the package and externally. A package without at least one public class isn’t reasonable, since there must be a way for entry to the package to be provided.

Assuming a declaration of GroceryStore similar to the package on the right, with at least some public classes, we can now focus in on the protected access specifier and its usage within a package. The protected members can be utilized by other classes in the GroceryStore package, specifically, by the ManagerGUI class to allow changes to wholesale prices via the method Modify_WholeSale(…). It is also possible to change protected variables, allowing ManagerGUI to have direct access to WholeCost. (Note: This may not be desirable from a software engineering practices perspective). The public members can also be utilized by the other classes in a package. When members (variables or methods) have the access specifier (public, protected, or private) omitted, then they become package members, and are only accessible by other classes within the same package. This type of control clearly limits the visibility of members, supporting hiding.

The abstraction/encapsulation of Java clearly exceeds the capabilities of C++. While public, protected, and private in Java are similar to C++, the package specifier and the package abstraction offer encapsulation opportunities that aren’t present in C++. While a C++ file can contain multiple classes, there is no binding of those classes into a logical unit, which the package definition does in Java. The package access specifier also supports the ability of sharing across classes within a single package, while simultaneously hiding this information from other classes. Such a capability is only partially achieved in a tedious fashion using the friend specifier in C++.

2.2 Inheritance

Inheritance and hiding are strongly related in object-oriented design and programming. Inheritance is the controlled sharing of information and methods between related classes of an application. At the type level, this involves an understanding of which attributes and methods of the supertype are available for use by the subtype. Different programming languages have different interpretations of inheritance: C++ allows the public and protected information to be acquired by the subtype from the supertype, while making the private parts unavailable; Eiffel only allows public methods to be inherited. From this perspective, information hiding is a key issue, since through inheritance we can control access to the information and methods of an class.

The inheritance concept has also played different roles in programming languages and database systems [9]. In an object-oriented programming language, inheritance is incorporated to allow instances of different classes to be treated in a uniform fashion, when it is semantically meaningful to do so, i.e., substitutability [52], which is related to polymorphism. This contrasts with the database interpretation when work on generalization was first conducted, where the underlying motivation for inheritance was to allow "... relevant details to be introduced in a controlled manner ..." [43], which emphasizes abstraction. Regardless of the programming or database
view of inheritance, all inheritance relationships between classes that share a common parent (or grandparent) form a hierarchy with an identifiable root (ancestor). Inheritance is a critical capability in object-oriented design and programming, since it is strongly tied to incrementality, evolution, and reuse.

Inheritance is supported in Java by using the extends keyword when declaring a class. For example, continuing with the supermarket domain, it is possible to declare a number of descendants of Item:

```java
class DeliItem extends Item { ... };
class SaladItem extends DeliItem { ... };
class ProduceItem extends Item { ... };
```

The members (variables and methods) of a superclass available in a subclass are as follows:

- A subclass inherits all public/protected members of a superclass.
- A subclass inherits all package members of classes in the same package as the subclass.
- A subclass does not inherit private members of a superclass.
- A subclass does not inherit members of a superclass that have the same name as members in the subclass.

Using these rules, a software engineer can develop inheritance hierarchies that contain classes which are abstract, public, and final.

An abstract class in Java can contain variables, and both regular and abstract methods, and is neither intended nor allowed to be instantiated. The Item class given in Section 2.1 can be reworked to the following:

```java
abstract class Item {
    protected String UPC, Name;
    protected int Quantity;
    protected double RetailCost;
    protected double WholeCost;

class Item() { ... };
abstract public void finalize();
abstract public boolean Modify_Inventory(int Delta);
public int Get_InStock_Amt() { ... };
public double Compute_Item_Profit() { ... };
protected boolean Modify_WholeSale(double NewPrice); { ... };
};
```

Notice that in the Item class, the finalize and Modify_Inventory methods have been made abstract, to signify that their implementations will be provided in the descendants of Item. Note that with inheritance, all public and protected members of Item are passed to its subclasses, and in turn to their subclasses, and so on. DeliItem, SaladItem, and ProduceItem would all acquire the protected and public members.

The passing of protected information raises a security issue. While protected members are not visible to the outside world, if the class is visible, an unscrupulous user could declare a subclass
and get access to protected members. Such a situation is possible in C++. To counter this, Java has included the final designation for classes. When a class is final, it cannot be subclassed. If SaladItem and ProduceItem are both leaf nodes in an inheritance hierarchy, then the keyword final should precede the keyword class. The ability to specify final classes in Java is a level of security not found in languages like C++ and Ada95. Java also allows the definition of final methods, which can't be overridden in a subclass/descendant.

Some languages like C++ and Ada95 support multiple inheritance, where a given subclass inherits behavior from two or more superclasses. One problem with multiple inheritance is when two or more superclasses have the same named method implemented in different ways. It is unclear in this situation which method should be inherited by the subclass. This type of ambiguity causes problems in Java particularly, where the bytecode must be generated and runnable on multiple platforms. However, while Java does not support multiple inheritance, it does have a capability, called interfaces, that can be used to model multiple inheritance at a design level. An interface in Java is a set of methods (no implementations) coupled with a set (possibly empty) of constant values. A Java class can be defined that simultaneously inherits from a superclass and implements one or more interfaces. For example, consider that in a University application that there are the following classes grouped by inheritance:

```
 Student               Employee
   / \                          |
 UnderGrad  Graduate  Faculty
```

One responsibility of both graduate students who are teaching assistants and faculty involves the teaching activity. In Java, a software engineer could define a Teaching interface, which is implemented by both Graduate and Faculty:

```java
interface Teaching {
    int NUMSTUDENTS = 25;
    void recordgrade(Undergrad u, int score);
    void advise(Undergrad u);
    ... etc ...
}

class Graduate extends Student implements Teaching {
    ...
}

class Faculty extends Employee implements Teaching {
    ...
}
```

The methods of Teaching, which are not implemented, represent those teaching activities that are common to both graduate students and faculty. However, their implementations will be different since their responsibilities are different. For example, the recordgrade method for a Graduate might only allow project grades to be set, while recordgrade for faculty would allow project and exam grades to be set, which requires a different implementation. The multiple inheritance is considered at a design level since the declaration of the Teaching interface does not contain any implementation. Note that when a class implements an interface, it has to provide an implementation for all of the methods that are declared in the interface; if the class is defined as “abstract” some implementations can be pending to non-abstract descendants.
2.3 Polymorphism via Dispatching

Polymorphism is the ability of a variable to hold more than one type, and is typically utilized in the context of inheritance hierarchies within class libraries. When an object-oriented programming language supports polymorphic behavior, then for a given inheritance hierarchy it is possible that a variable defined to be a supertype within the hierarchy can hold an instance of one of its subtypes. To realize polymorphism at runtime, dispatching is employed. Dispatching allows a run-time or dynamic choice of the method that is to be called based on the class type of the invoking instance. As a concept, the effective use of dispatching is tightly bound with inheritance, and offers many benefits:

a. versatility in the design and use of inheritance hierarchies;
b. promotion of reuse and evolution of code, allowing hierarchies to be defined and evolved over time as needs and requirements change; and,
c. development of code that is highly general and easier to debug (and hence reuse/evolve).

But, as with all useful techniques, there is a caveat, namely, dispatching incurs a cost or overhead at both compile and runtime.

Java supports polymorphism and dispatching within inheritance hierarchies. Unlike C++, the software engineer is not required to specify the virtual methods. Rather, in Java, methods of the same name (and signature) within a subclass override the method of the superclass. In our example of Sections 2.1 and 2.2, if a collection of Items is maintained, with instances of the collection being of types DeliItem, SaladItem, ProduceItem, etc., then methods that are redefined in these types will override Item methods. For example, the Modify_WholeSale is one such method that, as an abstract method, will be declared/implemented in descendants of Item. At runtime, the type of the instance within the collection of Items will determine which Modify_WholeSale method that is called. The runtime assurance is achieved via dynamic binding at the programming language level. At compile time, the software engineer is given a guarantee that a Modify_WholeSale will be found at runtime.

2.4 Generics

A generic is a type-parameterizable class that greatly facilitates software reuse by allowing a single software component to be utilized for many needs. For example, instead of having a stack that is bound to a specific data type (say, integer), the stack can require that the data type be provided as part of its initialization. Thus, the creation of a stack (e.g., Stack(Real), Stack(Char), etc.) binds the stack’s methods to the appropriate types. Non-object-oriented languages such as Standard ml (sml [32]) and Ada83 support generics.

However, while C++, Ada95, and Eiffel all support generics, Java does not. Interfaces in Java, as discussed in Section 2.2, can be utilized to mimic generic-like behavior. In [6], the following example of an interface is given:

```
interface Collection {
  int MAXIMUM = 500;
  void add(Object obj);
  void delete(Object obj);
  Object find(Object obj);
  int currentCount();
}
```
The Collection interface can be then implemented by one or more classes to capture behavior, that while similar from an external perspective (the Collection interface), is different from an internal perspective (the actual semantics captured in the implementation). For example, there could be the following class implementations of Collection:

```java
class Set implements Collection {
    ...
    void add(Object obj){
        ...
    }
    void delete(Object obj){
        ...
    }
    Object find(Object obj){
        ...
    }
    int currentCount(){
        ...
    }
}
```

```java
class Queue implements Collection {
    ...
    void add(Object obj){
        ...
    }
    void delete(Object obj){
        ...
    }
    Object find(Object obj){
        ...
    }
    int currentCount(){
        ...
    }
}
```

The Set class uses Collection to implement set-like behavior, which would require the implementation of the add method to prohibit duplicates. The Queue class would have a different implementation of the methods to capture the behavior desired by the software engineer.

### 2.5 Java vs. Smalltalk

The Smalltalk language and environment [20, 21] has been in use since the early 1980s, and is often utilized when practitioners and researchers benchmark the object-orientedness of a language. While the discussion in the earlier portions of Section 2 have focused on concepts, this subsection concentrates on a concrete and detailed comparison of the two languages, and their capabilities, environments, and platforms. Specifically, this section compares the two languages (Java and Smalltalk) in terms of abstraction/encapsulation, object-oriented programming, language and environment characteristics, stability and reliability, and platforms and cost.

**Abstraction/Encapsulation:** As discussed in Section 2.1, these features of a language are critical for understanding its object-oriented modeling potential. In a Smalltalk application, the largest abstraction unit is referred to as an image. All Smalltalk objects live in the same image and share the same name space. It is very common for different software engineers to develop their portions of an application separately and use some kind of naming convention for those classes that are in common across the entire application. Unfortunately, this often produces hard to read class names, or may even result in two (or more) software engineers viewing the same named class in two (or more) distinct and conflicting ways. In contrast, the largest abstraction unit in Java is a package, which contains a collection of classes and interfaces. The package construct requires additional design/implementation rigor by a software engineer. Specifically, inside each Java source file, a software engineer must specify the package(s) that are imported. Thus, in Java, two classes can have the same name if they belong to different packages. This is cleaner than Smalltalk.

**Object-Oriented Programming:** One key aspect of object-oriented programming is the degree to which the language treats data as objects. In Smalltalk, all kinds of data, including
primitive types like integer and character, are treated as objects. This is not the case in Java, where primitive types are handled in a similar fashion to imperative programming languages like C, Ada, and Pascal. A second key aspect of object-oriented programming is inheritance, as we outlined in Section 2.2. Both Java and Smalltalk support single inheritance. However, Java has more restrictions on overriding superclass methods in the subclasses. Unlike Smalltalk, in Java, static and class methods that are defined in a superclass cannot be overridden in the subclass. But, Java does support the designation of a final method that cannot be overridden in a subclass, a feature which Smalltalk does not provide. In addition, Java subclass methods cannot override the return type of the same method defined in the superclass.

Another key object-oriented aspect that is present in more recent languages is the concept of an abstract class and abstract methods as introduced in Section 2.1 for Java. In Java, this feature is supported by the language and environment. In Smalltalk, this feature is dependent on a software engineer’s discipline to imitate the capability. Without compiler support that enforces abstract classes/methods, there is the potential for misuse and inconsistency. Within individual classes, Smalltalk supports the definition of both class variables and class instance variables. Java on the other hand is limited to class variables, which are shared by all instances of a class. Both Smalltalk and Java allow instance methods and class methods to be defined. However, Smalltalk instances can access only instance methods while Java instances can access both kinds of methods. Finally, even though garbage collection is not a necessary aspect of object-oriented programming languages, both Smalltalk and Java have this feature.

**Language and Environment:** There are many different comparisons of Smalltalk and Java that can be made based on their various language and environment features. At the language level, Sun currently controls the standard of Java, dictating the features and makeup of the language. Consequently, it has been promoted by Sun that Java source files are compatible on the compilers for different hardware/OS platforms. In reality, there are some incompatibility problems for Java. For example, the lack of case sensitivity on Windows 95 could affect the port of Java source code to a Unix platform, if the software engineer hasn’t been careful in the naming of source files and classes. Even though Smalltalk has a standards committee, in practice, different Smalltalk source files have not been 100% compatible across compilers/platforms. When considering source files and runtime issues, in Java each public class has its own .java and .class files. The .java files are the source files and the .class files are binary Java bytecode files. Each Smalltalk program has a .im (image) and .cha (changes) files. The image file contains Smalltalk bytecode. The changes file contains the source code and all of the editing actions that have occurred which modify the image.

From an extensibility perspective, Smalltalk is a reflexive system. If desired, the software engineer can change anything above the virtual machine since the development codes reside in the same image as the rest of the system. On the other hand, since Java has been designed to be secure over the WWW, the appletviewer doesn’t allow a software engineer to change some of the base classes, thereby limiting extensibility. WWW usage and support is also dramatically different in the two languages. First, there is no standard, popular application framework for writing Java applets that is similar to the Smalltalk-80 framework, i.e., Java’s Abstract Windows Toolkit (AWT) API doesn’t have the concept of Smalltalk’s Controllers. Further, Java and its applet capability were specifically intended to interact with commercial WWW browsers like Netscape and Microsoft Explorer. While it is possible
to write Smalltalk applets that can be run in a Smalltalk WWW browser, such browsers are
not in widespread use. Different Smalltalk dialects come with varying degrees of database
connectivity. For example, the GemStone Data Management System [4] extends Smalltalk
with persistence and database management capabilities. Finally, in Smalltalk, sending the
message #perform: to any object with a selector(method name) as the argument has the
same effect as sending that message to that object. Java doesn’t support such a capability
since it is conflict with some of the security requirements of the language that are dictated
by its WWW usage.

**Stability and Reliability:** Java is a very new language, whose content continues to change
and evolve over time. It is expected that Java will change and improve to stay competitive
in coming years. Smalltalk, since it is a much older language, is changing and improving
at a much slower rate. As a rapidly evolving language, Java has a number of bugs and
problems. For example, Netscape and Sun’s application viewer may treat the same Java
applets differently. In fact, Java applications developed on Windows 95 may have a different
look-and-feel on a Solaris platform. Also, Java APIs in newer releases haven’t always been
totally compatible with earlier releases. However, as Java matures and stabilizes, bugs
and problems should reach a stable (and hopefully minimal) state. On the other hand,
Smalltalk has a much longer history, and as such, it is expected that older classes (the
equivalent to Java APIs) are very reliable. Like any language, newly developed classes,
tools, and frameworks for Smalltalk will have bugs that will be fixed over time.

**Platforms and Cost:** Both languages use virtual machines that are portable to different hard-
ware/OS platforms at the bytecode level. Sun’s Java Development Kit is available in
SPARC/Solaris, Windows NT/95, and Macintosh, and a wide range of third-party ports [82]
as given in Section 1.1. Classifying the availability of Smalltalk is more complex. Usually,
source files in Smalltalk are not compatible with different Smalltalk dialects, and each di-
ialect supports a different number of platforms. Smalltalk/X is available for Solaris, Linux,
Irix, Ultrix, and NextStep. IBM Smalltalk is available for OS/2, Windows, and AIX. Visual-
Works is available for Windows, OS/2, UNIX, and Macintosh. VisualSmalltalk is available
for Windows and OS/2. ENFIN Smalltalk is available for Windows, OS/2, AIX, Solaris,
and HP-UX. QKS Smalltalk Agents is available for Macintosh only. The different dialects
and their varied availability (and oftentimes incompatibility) gives a significant edge to Java
in uniformity and consistency across platforms.

From a cost perspective, it is difficult to undercut Java. The Java Development Kit from
Sun is free of charge for all platforms. The final product doesn’t have a runtime fee.
Commercial development tools for Java have also tended towards low cost, usually less
than $500, e.g., Symantec Cafe only costs about $100. On the other hand, commercial
Smalltalk development tools are more expensive, typically costing $2,000 and up. While
there are non-commercial versions of Smalltalk available (e.g., gnu Smalltalk, Smalltalk-
X, and little Smalltalk), serious development often requires a more stable and supported
platform. At the present time, the commercial development environment of Smalltalk is
much more mature than that of Java.

Overall, Java and Smalltalk can both be classified as easy to learn object-oriented languages.
Clearly, Smalltalk is more mature than Java due to its longer presence and history. However,
Java’s potential for cross-platform, portable, and distributed development may revolutionize the
computing community.
2.6 Summary of Java’s Object-Oriented Capabilities

Overall, Java measures up quite well with other object-oriented programming languages. The lack of multiple inheritance at the implementation level is often cited as a major drawback of Java, but in practice, multiple inheritance is rarely found. Java does support multiple-inheritance-like behavior at a design level via interfaces (Section 2.2), but falls far short of true multiple inheritance at the implementation level. The absence of generics is the major deficiency of Java, and greatly impacts the reuse potential of the language. While class interfaces (see Section 2.4 again) do offer generic-like behavior, there is no way for a software engineer to write a type-parameterizable Collection that can be utilized by many different classes.

However, these two deficiencies of Java are more than offset by the tightening of the language capabilities to support encapsulation (Section 2.1), inheritance (Section 2.2), and polymorphism (Section 2.3). The package capability of Java, along with the means to control visibility both within and between packages is similar to Ada95. Like Ada95 and other modular-based languages, Java also requires software engineers to explicit import information needed from other packages when defining their own classes/packages. From a software-engineering-practices perspective, Java is clearly superior in this regard to C++. Inheritance-wise, Java is similar to both C++ and Ada95, but differs from both with the final class specifier, that effectively turns off classes and prevents subclasses as a means to circumvent hiding and security. While the final class specifier is similar to private derived inheritance in C++, in that both make protected members inaccessible, Java expands this to not even allow the subclass to be extended. Polymorphism in Java is similar to C++ and Ada95, but it seems simpler to use in Java. Finally, Java does match up well against Smalltalk (Section 2.5), and as Java continues to evolve it will be interesting to see if this comparison holds firm.

3 Integration and Distributed Object Computing

The emergence of distributed object computing (DOC) technology and its competing and/or complementary paradigms such as ODP [25], DCE [39], CORBA [38], and DCOM/OLE [30], has opened new horizons to the object-oriented approach by enabling the parallel and/or distributed processing of large and computation-intensive object-oriented applications on heterogeneous hardware and software environments. This is precisely one of the major market segments that Java application development is targeted on. Furthermore, DOC is positioning itself as the middleware that will enable a promising synergism between the object-oriented paradigm and innovative WWW technologies, which are two key facets for which Java has been championed.

Several models have been proposed to embody DOC principles, and they all define an Interface Definition Language (IDL) for the description of standard object interfaces. IDL is a declarative language that is used to describe the services an object provides. IDL supplies a set of basic and constructed types, but no flow control constructs (e.g., if, loop, etc.) since it is not a programming language. Interfaces defined with IDL must be mapped into some actual programming language (e.g., C++ or Java) and compiled to the rest of the application. IDLs are usually not compatible from one DOC model to the other. An API for Java IDL is in the early-release stage, and is intended to provide integration with CORBA [83]. The Java IDL will have a number of components, including: “... a Java IDL Language Mapping Specification, an IDL-to-Java compiler and a portable, Java ORB core that supports IIOP thus allowing developers to build Java applications that are integrated with heterogeneous business information assets.” [65]

The purpose and promise of Java is to provide the tools (via the APIs) that facilitate com-
munication and interaction between programs (processes), for not only homogeneous code (all Java source), but for applications that require multiple programs (processes) written in different (object-oriented) programming languages. This is required for not only new programs (processes), but for existing legacy systems, and for the ability to interface Java to commercial-off-the-shelf (COTS) systems. The remainder of this sections covers four topics related to integration and DOC: an overview of DOC models with an emphasis on CORBA (Section 3.1); an examination of the role of Java in the design and development of DOC applications (Section 3.2); an exploration of interactions between Java and other programming languages (Section 3.3); and, an investigation of techniques that allow Java applications to interface to legacy and COTS systems via DOC and programming-level interactions (Section 3.4).

3.1 A Brief Overview of DOC Models

DOC models rely on the client-server paradigm to regulate the communication exchanges between objects across heterogeneous platforms. The transparent handling of communication activities as well as the proper matching of a client request to the appropriate server is taken care by a special component, called an Object Request Broker (ORB). The ORB (also called the broker) is the core module of a DOC model. It has the responsibility of delivering client requests to remote servers and returning any responses, while hiding the disparities of the underlying environments.

Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object’s interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

DCE (Distributed Computing Environment) [39, 41] was originally a procedural platform (with a C based IDL) from the Open Group (formerly, the Open Software Foundation – OSF). DCE relies on RPC and a directory service to achieve the ORB functionality. DCE is one of the first distributed computing models to gain wide acceptance. It is a reference implementation since there is some actual code that can be acquired from the Open Group for the development of compliant distributed computing platforms. At the time of this writing, significant parts of DCE 1.1 source code can be freely downloaded from the site [56]. The latest release of DCE (i.e., DCE 1.2) has extended support for object-oriented programming. Note also that Hewlett-Packard has implemented its own object-oriented version of DCE, called OODCE [17].

COM/OLE (Component Object Model/Object Linking and Embedding) [30] is a proprietary solution to distributed object computing, from Microsoft. Similarly to other distributed computing frameworks (DCF), COM specifies conventions and provides services for defining and using objects in heterogeneous systems. It also provides an IDL for specifying interfaces to objects. COM/OLE’s IDL is compatible with neither DCE’s nor CORBA’s IDL.

CORBA (Common Object Request Broker Architecture) [38, 50, 51] is an industry standard for distributed object computing from the Object Management Group (OMG). Unlike the two previous DCFs, CORBA is not a reference implementation but a specification that defines an Object Management Architecture (OMA) as shown in Figure 3. OMA contains common facilities for objects at the application level (e.g., mailing and printing), lower-level object services (e.g., naming, concurrency control, transaction services, persistency, object query, object collection,
system managements, etc.), and the ORB as a common object communication bus. To deliver these facilities and services, OMA is composed of an Object Model and a Reference Model [50]. The Object Model defines the ways that objects which are distributed across a heterogeneous environment can be described, while the Reference Model characterizes interactions between those objects. The core component of the object management architecture is the Object Request Broker (ORB) which is a common communication bus for objects [51], as shown in Figure 4. Figure 4 shows a number of components, which interact with one another to provide the required runtime services. Interactions are best described in the context of a client making a request for services. When such a request is made, the components in Figure 4 offer the following functionality:

- The Client IDL Stubs component provides the static interfaces to the object services that a client requires.

- The Dynamic Invocation Interface component is utilized to discover the method to be invoked at runtime to satisfy a client’s request.

- The Interface Repository component allows the user to obtain and modify the descriptions of all of the registered component interfaces, the methods that they support, and the parameters that they require.
- The ORB Interface component consists of a selected set of APIs to local services that may be of interest to an application and the server.

- The Static Skeletons (server IDL stubs) component provides static interfaces to each service exported by the server.

- The Dynamic Skeleton Invocation component provide a runtime binding mechanism for servers that need to handle incoming method calls for components that do not have IDL-based compiled skeletons (or stubs).

- The Object Adapter component accepts requests for service on behalf of the server’s objects. This component provides the runtime environment for instantiating server objects, passing requests to them, and assigning object IDs to them. It also registers the classes it supports and their runtime instances with the Implementation Repository component.

- The Implementation Repository component provides a runtime repository of information on the classes that a server supports, the objects that are instantiated, and their object IDs.

All of the components in Figure 4 provide the runtime support for DOC technology.

CORBA also defines its own IDL language for the description of object interfaces. OMG’s IDL is a purely descriptive language, programming language neutral, and network neutral. Mappings to C, C++, SmallTalk, and Ada95 are fully specified. To keep up with the high demand for Java/CORBA integration, a mapping for the Java language has been slated [50]. Several commercial implementations of the latest version of the specification (CORBA 2.0 [38]) have been released. Table 1 lists the most publicized of them. Remarkably, all of the commercial platforms shown in Table 1 are compatible with one another. This rather unique feature is achieved through their common support of the Internet Inter-ORB Protocol (IIOP) that allows interoperability between vendor specific implementations. A testbed, called CORBAnet [55], has been developed to demonstrate interoperability between ORBs from various vendors relying on the IIOP protocol.

<table>
<thead>
<tr>
<th>Product Name</th>
<th>Vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAIS</td>
<td>ICL</td>
</tr>
<tr>
<td>DSOM</td>
<td>IBM</td>
</tr>
<tr>
<td>HP ORB Plus</td>
<td>Hewlett-Packard</td>
</tr>
<tr>
<td>NEO</td>
<td>Sun</td>
</tr>
<tr>
<td>ObjectBroker</td>
<td>Digital</td>
</tr>
<tr>
<td>ObjectDirector</td>
<td>Fujitsu</td>
</tr>
<tr>
<td>Orbix</td>
<td>IONA</td>
</tr>
<tr>
<td>PowerBroker CORBAplus</td>
<td>Expertsoft</td>
</tr>
<tr>
<td>SmalltalkBroker</td>
<td>DNS Technologies</td>
</tr>
<tr>
<td>SORBET</td>
<td>Siemens Nixdorf</td>
</tr>
<tr>
<td>NonStop DOM</td>
<td>Tandem</td>
</tr>
<tr>
<td>Visibroker</td>
<td>Visigenic</td>
</tr>
</tbody>
</table>

Table 1: CORBA Implementations Compliant with CORBAnet.
3.2 The Role of DOC in Java Design and Development

In general, DOC models are embodied in Distributed Computing Frameworks (DCF). A DCF consists of multiple components which have been integrated to work closely together. Minimally, there is an IDL language, an ORB server, and an interface repository. Optionally, a DCF can define its own security service and distributed file system. DCF is called middleware or enabling technology. It is not intended to exist alone, but instead should be integrated or bundled into a vendor's operating system offering. For example, DCF's security and distributed file system can completely replace their current, non-network, analogs.

The rise of the Internet and the current boom of intranets is forcing software engineers and network managers to find better ways to exploit, and when necessary, to spread out the huge computing intelligence that is locally clustered. Though complex, DOC constitutes a robust and open way to bridge the gap between corporate networks and the applications that run over them [31]. DOC offers an attractive solution for object-oriented components across heterogeneous platforms, to interact and participate to the resolution of a problem. However, the software engineer still needs to decide what objects and/or computing entities must be distributed before he/she can define appropriate IDL interfaces for them, map them to the programming language of his/her choice, and compile them with the rest of the application. In other words, DOC technology does not address the critical issues of determining what are the key object-oriented application components and how they are distributed. It just provides the means to carry out the distributed computations, once the what and how decisions have been made. To the contrary, the emergence of DOC is making the need for a rigorous and consistent approach to address the aforementioned questions at the appropriate level (i.e., design stage) more pressingly than ever before.

The early access specification of the API for the IDL-to-Java Mapping [83] is available for downloading from the WWW. A detailed discussion of the mapping is available on-line [85], along with a preliminary version of the tool idltojava [86] that compiles IDL files to Java source code. In addition, there are many other commercial and shareware products that are available, including:

- VisiBroker for Java by VISIGENIC Software [124]: Supports both client and server Java programs, and provides complete support for connecting to any CORBA object that adheres to the Internet Inter-ORB Protocol (IIOP).

- Joe by SUN [113]: Allows Java client applets/applications to connect to NEO objects (SUN's CORBA product).

- OrbixWeb by IONA Technologies [120]: Allows Java client applets/applications to connect to Orbix objects and to interoperate with any CORBA 2.0 compliant service.

- JIDL by Sandia National Laboratories [84]: A public domain CORBA IDL-to-Java compiler.

- JYLU at Stanford University [114]: A CORBA ORB for Java clients/servers.

- HORB by Electrotechnical Laboratory [59]: A Java ORB that supports communication between Java servers and Java clients. Note that HORB is not CORBA compliant.

The above list is representative of currently available technology for mapping IDL to Java. It is expected that the choices will continue to expand in the future. The availability of IDL-to-Java mapping and associated tools will allow Java to serve as a target for supporting DOC. This will aid in the ability to interface Java with COTS and legacy systems (see Section 3.4).
3.3 Integrating Java with Other Programming Languages

The ability to integrate Java with software written in other programming languages has significant implications in a number of different areas. First, whenever new languages emerge, there is always an interest in migrating existing applications directly to the new language. Rewriting code is one solution, but the more popular choice is automated language-level translation at a source-code level. The migration of existing applications to a new language allows software engineers (and the enterprises they work for) to accrue all of the benefits and advantages of the new language. For example, an existing C++ application that is rewritten in Java still has the object-oriented benefits related to reuse, evolution, etc., but immediately acquires platform independence in the short term and in the longer term has the potential for upgrade to client/server or DOC solutions.

There have been a number of products that address automated language-level translation, both to and from Java. PERC from Synkronix [122, 123] is a COBOL-to-Java cross compiler that supports source code translation from ANSI COBOL-85 to Java, including language changes in the ANSI 1989 and 1993 addenda. Other features include support for SQL/database access via JDBC, JavaBeans compliancy, and multi-threading via COBOL that are then translatable to Java via PERC. Language translators like PERC allow existing COBOL code to be reused in new Java based applications. Moreover, retraining of all software engineers to Java is not necessary if the translator is robust enough to handle all COBOL programs as input. PERC also allows programs that were platform specific to become platform independent, opening a larger potential market for existing products. Other non-commercial products include: JCC, a Java to C translator [112] and C2J++, a C++ to Java translator [111]. C2J++ claims to handle the majority of the translation, unable to process variable declarations on the stack, template definitions, and overloaded operators in the code.

A second area of programming language interaction occurs directly at the language/runtime environment levels. Most programming language compilers are targeted to the native assembly language when processing source code, making compilers platform specific. However, there is no intrinsic reason why a compiler cannot generate Java bytecode. Remember, the JVM executes bytecode, and while bytecode can be created from Java source, it may also be created from source code of other programming languages. The advantages to such an approach are twofold: existing programs can be reused as is by compiling to bytecode, or, new programs can be written that take advantage of both the original programming language and Java. We have found two products of note that offer such a capability, AppletMagic from Intermetrics [54] and Kawa by Cygnus [116].

AppletMagic is a compiler and IDE that generates Java bytecodes from Ada95 programs. AppletMagic works by translating Ada95 source code to Java bytecodes, with the ability to fully translate Ada95 packages/types to Java packages/classes. It also supports the transformation of Ada95's access-to-subprogram types, task entries, and nested subprograms to Java bytecode, features which are not directly supported by the Java language. But the totality of the translation is unclear. Specifically, in CD Rom documentation [53], the statement “Almost any Ada95 feature has a direct mapping in J-code and every capability of J-code is readily represented in an Ada95 construct.” doesn’t really answer the question of translation completeness. AppletMagic works by allowing the Java API packages java.applet, java.awt, java.io, java.lang, java.net, and java.util to be utilized directly in Ada95 code via the with and use statements in Ada95. By using these key APIs, Ada95 programs can easily access the browser, GUI, networking, and other features from Java.

Kawa is an IDE for Scheme, written in Java, that compiles Scheme programs into Java bytecodes. There are a few minor features of Scheme that are not supported, but in general, any
Scheme program can be translated/compiled into Java bytecodes. Kawa provides an API library to support the various Scheme types in Java. For example, Scheme integers are implemented in the class kawa.math.IntNum and Scheme strings are implemented in the class kawa.lang.fstring. Kawa also provides the ability to make direct calls to Java methods from Scheme programs, which is similar to AppletMagic's use of Java APIs from Ada95 code. Products such as AppletMagic and Kawa are likely the forerunners of future IDEs that will provide similar capabilities for existing programming languages, thereby facilitating solutions to integrating Java with COTS and legacy software. However, all such products need to be critiqued from the perspective of completeness of translation, to ensure that software engineering intervention is not needed to clean-up or fix translation shortcomings.

A third area of programming language interaction occurs at a OS level. When two or more programs written in different languages need to communicate with one another, there are a number of possible solutions:

- Each program is a separate process level, and the interactions that occur between them utilize well-understood communication protocols like pipes or sockets. If the programs are being newly developed, then the communication requirements can be integrated as part of a message-passing front-end to each program. Otherwise, it may be necessary to build new interfaces to the existing programs to allow the inter-process communication to occur. Inter-process communication between programs written in the same or different (object-oriented) programming language is a well-understood problem with standard and known solutions.

- DOC as discussed in Sections 3.1 and 3.2 can function as the common communications medium for both newly designed/developed software and for integrating with COTS/legacy software.

- When one program needs to utilize the services of another program, it may be possible to make a direct call to the executable of the desired program.

Note that the first two solution approaches are appropriate for COTS and legacy software interactions with Java, which will be examined in Section 3.4. The third solution is supportable in Java using the Java Native Interface, JNI [87].

JNI is the bridge between Java applications and the native interface. By using JNI, it is possible to have your Java applications directly call a program written in another language (like C or C++), or to alternatively, allow an application written in another language to call a Java application. JNI is essentially a trap-door that allows sophisticated software engineers to circumvent the platform independence of Java. This occurs for a number of reasons. First, the Java APIs may not contain a platform-independent means to utilize a platform-specific feature that is needed in your Java application, i.e., a specialized piece of hardware that operates off the serial or parallel port of a PC. Second, a significant, tested, and important program written in another language needs to interact with a Java application. This “important” program may be a legacy or COTS application that must be utilized as is without a total recording in Java. Third, to address performance, it may be necessary to write a segment of your Java application in assembly language. There are important caveats that accompanies any utilization of the JNI, outlined in [16], and reformulated below:

* By using JNI, you are no longer 100% portable, which may require the development and maintenance of multiple versions of an application for each hardware/OS platform.
* JNI requires significant programming skill for correct and consistent usage, and thus, code written with JNI may be prone to more errors at both compile and runtime.

* JNI effectively opens the security door since native code has been introduced.

The interested reader is also referred to [16] for a detailed discussion of utilizing the JNI to both call a C program from Java and to call a Java method from a C program.

3.4 Interfacing Java Applications with Legacy/COTS Systems

Whenever a new programming language or other computing technology becomes prominent, one of the first questions that is asked involves the potential impact and possible interactions with legacy systems. A legacy system is an existing application that is tied to the core operations of some entity (business or government agency). The issues related to motivating and defining legacy applications and their interactions with today's applications is a popular topic [34, 35, 37, 40, 47], particularly with the advent of Java and CORBA. Most entities must continue to utilize legacy applications, since the data that they generate, store, and/or access are critical to meet current operational needs in a variety of ways. Specifically, legacy data is needed for official reporting, analyses, dissemination, and availability to future systems. While the first two needs are often met by the existing functionality of a legacy application, the last two, dissemination and availability, are what requires entities to consider the utilization of new technologies like WWW and Java. The problem is made more difficult since the majority of legacy applications are unmodifiable; without the ability or knowledge to make changes, there must be creative solutions that integrate legacy applications with modern technologies (e.g., C++, Java, OODBMS, Ada95, etc.), effectively migrating them to a new generation of computing and users.

There are many possible ways to integrate legacy applications with modern technologies, including: the data-transformation approach that relies on commercial database management system technology as a means for data exchange between a Java client and a legacy application; the ORB approach that utilizes DOC and CORBA ideas and introduces the concept of a wrapper to integrate the legacy application to an ORB that is then accessible via Java clients; the Java-client wrapper approach that avoids ORB/CORBA by integrating the Java-client with a wrapper that maps down to the native OS/network level, which is then accessible to the legacy application; and, an approach that illustrates the way that two or more legacy applications can be integrated to multiple-java clients. In all of these approaches, the legacy application is evolving to a DOC environment, functioning as a server to one or more Java clients. Note that while we have introduced this material using legacy application concepts, all of the discussion to this point and in what follows is also applicable to commercial-off-the-shelf (COTS) applications. The remainder of this section presents and critiques each of the four approaches in turn. This is followed by a discussion of integrating Java applications with “future” COTS, which illustrates the true potential and versatility of the language. Finally, a summary of the various, relevant APIs from Java that are needed to facilitate and support the integration with legacy/COTS applications is provided.

3.4.1 The Data-Transformation Approach

The data-transformation approach, outlined in [40], is a data-centered solution that is appropriate for legacy applications that generate data. The data-transformation approach is illustrated in Figure 5 for a Java client. In Figure 5, the process is as follows:
• Data is extracted from the legacy application and stored into a relational database system (RDBS), where it is now accessible to a Java client.

• If the legacy application is dynamic (generates data), then at periodic (or continuous, if needed) intervals, the data must be extracted and restored into the RDBS.

• Once in the RDBS, the data is accessible via JDBC (see Section 4.1) to the Java client, in both a read and write mode.

• If the Java client can generate or update the data from the legacy application, the dashed arrows are relevant, indicating the flow of information via the RDBS back into the legacy application.

In such a process, two sets of bi-directional transformations (mappings) of data are necessary: Java client to/from RDBS to/from legacy application. When the Java client can write or update legacy data, concurrency and serializability become major problems, with software needed within the Java client and/or the RDBS. Multiple Java clients may or may not be concurrently supportable, depending on whether a client can write data, and if the legacy application can support simultaneous access. Note that COTS systems that generate data can be integrated in a similar fashion with Java, simply replacing the Legacy Applications component in Figure 5 with a COTS system.

3.4.2 ORBs and CORBA

As introduced in Sections 3.1 and 3.2, object-request brokers and CORBA technologies can be utilized to integrate a legacy application with a Java client. The ORB approach, reported in [47], utilizes the wrapper concept to facilitate the integration. A wrapper is a piece of software that functions as an interface or filter between two distinct software components that must interact. For instance, in Figure 5, the two arrows between the RDBS and legacy application for data mapping can be designed and implemented as a wrapper that transforms legacy data to/from the RDBS. In Figure 6, the ORB approach is reproduced from [47], with a wrapper utilized to facilitate the integration. In this case, the wrapper is coded using an IDL-to-PL compiler, where PL is the programming language of the legacy application. While the IDL-to-PL compiler will map to the legacy application at the data/object level, the software engineer must still provide the needed logic and control flow that promotes the exchange of information via the wrapper according
to the programming interface of the legacy application. Using this approach, concurrent Java clients are possible, providing that the concept is supportable by the legacy application. If the legacy application needs to interface to two or more clients written in different (object-oriented) programming languages, then it will be necessary to design and develop multiple, dedicated wrappers. Integration with a COTS system can be accomplished in a similar fashion using the ORB approach, provided that the COTS system has a robust and versatile programming interface that will allow an IDL-to-PL wrapper to be designed/developed.

3.4.3 Wrapper via RPC and JNI

When the ORB approach is not appropriate, the *Java-client wrapper* approach that takes advantage of JNI (see Section 3.3) and remote procedure calls (RPC) can be utilized, which has also been outlined in [47]. The approach, shown in Figure 7, integrates the wrapper with the Java client across a network. The Java client in Figure 7 is composed of a number of differ-
interactions between the Java layer and native layer occur via the JNI as described in Section 3.3. The Java layer maps from the platform-independent application code to the programming language interactions of the native layer in two stages. In the first stage, the native layer promotes interactions back and forth with the Java layer using a C++ interface which provides a compatible object-oriented context for the exchange. In the second stage of the native layer, there is a mapping from C++ to/from RPC client stubs written in C, which interact across the network to the legacy application. In this approach, it is assumed that the legacy application can interact via RPC over a network to permit the needed interactions with the Java application. Multiple Java clients may be possible, though this may be complicated to achieve if the clients need to synchronize actions and share data. Note that the mapping to the native layer is platform specific, requiring multiple implementations for Java clients on different hardware/OS platforms. This integration is trivially transferable to a COTS system, again provided that an adequate programming interface is available.

3.4.4 Integrating Multiple Legacy/COTS Applications

In this section, a more realistic scenario is provided to illustrate the approaches for integrating two or more legacy/COTS applications into a single, unified system which is accessible by multiple Java clients. In Figure 8, an example is provided that integrates two COTS applications (from TRW and Hughes) and an in-house legacy application, all of which are now accessible by one or more Java clients. In the figure, Java Network Wrappers are shown, whose sole purpose are to provide a mapping from the COTS/legacy application to Java application code. The sole purpose of the Java application code is to provide the means for COTS/legacy data to be Java-available over the network (Internet and/or intranet). The approach shown in Figure 8 is a reformulation of the one shown in Figure 7, upgrading the COTS/Legacy application to a Java-compatible

![Diagram](image_url)
state with the network wrapper. Individually, a consistent Java interface to the COTS/legacy applications is now available for all of the Java clients that require access to one of the applications. If a Java client needs access to multiple COTS/legacy applications in a simultaneous fashion, it is necessary to provide a substantial software component within the client to handle the required collection and synchronization requirements.

In a similar fashion, another approach would extend Figure 6 to provide IDL-to-PL wrappers for the two COTS and one legacy applications. Java clients would then be able to access these applications via a common ORB. If the Java client is sophisticated enough, it is possible to design and develop functionality that would provide access to all three of the applications. Such a solution would need to be prototyped with great care to insure that consistency both within and between the COTS/legacy applications is preserved.

### 3.4.5 Java and Future COTS Applications

The four approaches described in Sections 3.4.1 to 3.4.4 can all be utilized for “future” COTS applications, when those “future” applications continue to use non-Java programming languages. However, the more interesting issue to be addressed concerns the potential acceptability and wide-spread use of Java. If this occurs, it will likely be the case that “future” COTS applications will be coded in Java, as either clients or servers, perhaps even compliant with DOC technologies via CORBA. In this situation, the integration of new Java applications with these “future” COTS will be much more straightforward, facilitating an ease of integration that hasn’t been available with other programming languages. This is where the true potential of Java lies, with the ability to support both legacy and present/past COTS applications, while simultaneously providing an integrated framework for new Java applications with “future” COTS. When coupled with platform independence, business and government entities can write software once that is applicable to multiple configurations, reducing development costs and opening up markets on platforms that had never been targeted. Whether this promise is achieved in practice remains to be seen.

### 3.4.6 Java APIs for Legacy/COTS Applications

The Enterprise API of Java [65] will contain packages/classes for the integration of Java application with legacy/COTS applications. Specifically, the relevant APIs are:

- The JDBC for Database Connectivity (Section 4.1); needed by the data-transformation approach (Section 3.4.1) and the ORB (Section 3.4.2) and multi-application approaches (Section 3.4.4) when the legacy/COTS applications are databases.
- The RMI for remote method invocation between processors (Section 5.5); potentially needed by all four approaches to support access of objects across networks.
- The Java IDL for integration with CORBA (Section 3.2); needed by the ORB (Section 3.4.2) and multi-application (Section 3.4.4) approaches.
- The JNDI (Java Naming and Directory Interface) for platform independent access to native naming and directory services; potentially needed by all four approaches.
- The JNI (Java Native Interface) to allow direct calls to platform specific OS services (Section 3.3); needed by all but the ORB approach to handle platform-specific mappings/calls.

34
These packages, along with the core packages for threading (Section 5.2), exceptions (Section 1.2.4), network services (Section 5.3), and security primitives (Section 5.4), are all critical to allow Java to successfully integrate with legacy/COTS applications.

4 Java and Database Support

Java is expected to make a profound impact on accessing databases over both the Internet and intranets. In particular, there seems a definite synergy between the powerful graphics handling capabilities of Java and the increasing popularity of the client/server architecture. This synergy stems from two trends: client interfaces for database servers are becoming more and more graphical-oriented, and client interfaces are required to be architecturally neutral as database usage becomes fundamental in various data-intensive processing.

The remainder of this section focuses on the support of the entire Java framework for the attainment of object persistence, with and without linking to commercial database management systems (DBMSs). In Section 4.1, the Java Database Connectivity API is presented, which represents the core component of the Enterprise API for integrating Java applications with relational DBMSs. Section 4.2 explores JDBC in greater detail, concentrating on the capabilities, functionality, and usage of the Java APIs for interacting with relational DBMSs. In Section 4.3, emerging commercial products and standards for Java integration with object-oriented DBMSs are considered. Finally, in Section 4.4, examples related to the realization of persistence via object serialization, the JDBC, embedded SQL (JSQL), and OODBMSs are examined.

4.1 Relational Databases: Java Database Connectivity (JDBC)

As part of the Enterprise API [72], Java provides Java Database Connectivity (JDBC) [125], realized in the java.sql [103] package, which enables Java clients to interact with relational databases in a certain standard way. Facilities of this library include: opening and closing database connections, querying metadata information, submitting SQL statements and receiving result sets from the server, and other database-related operations. The JDBC standard [127, 135] is based on the X/Open SQL Call Level Interface which was the same basis for developing the ODBC. The motivation behind the JDBC is that by complying to one low-level (standard) API, applications can be developed that are not concerned with vendor specifics of current or future relational database management systems. Vendor differences are to be addressed by developing a driver specific to each vendor's implementation. The JDBC architecture is shown in Figure 9. In Figure 9, JDBC applications have two layers of APIs, the JDBC API and the JDBC Driver API. There is a software component sitting between the layers, called JDBC Manager. The JDBC API supports application-to-JDBC Manager communications. The JDBC Driver API supports Manager-to-Driver communications. The primary role of the JDBC Manager is to handle communications with different types of drivers, e.g., the JDBC-ODBC bridge driver, the JDBC-Net Manager, and the direct JDBC drivers, as each illustrated in Figure 9.

In Figure 9, the JDBC-ODBC bridge driver translates JDBC method calls into ODBC function calls. Use of this driver allows existing ODBC-compliant legacy systems to be利用ized by a new Java client application without modification. The JDBC-Net Manager uses a published protocol to communicate with a remote database listener, which eliminates the need for using drivers on top of databases, thus enhancing portability. However, a few modifications (e.g., schema caching and tuple look-ahead, etc.) are needed on the client side to achieve portability. Overall, a new Java client can be very lean and fast, and can speak to any server supporting the published
Figure 9: Alternative ways of using JDBC.

protocol. Lastly, the direct JDBC driver communicates with a specific database management system. This direct driver induces a performance benefit by circumventing the ODBC layer.

JDBC is a specification model as opposed to a reference model. Recently, a number of vendors have begun to develop and market JDBC drivers. Each driver must support at least the ANSI SQL 1992 standard in order to be called “JDBC COMPLIANTTM” [127]. The JDBC drivers available in the commercial market can be grouped into four types:

1. JDBC-ODBC bridge which provides JDBC access via most ODBC drivers.
2. Native-API-partly-Java driver which converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMSs.
3. Net-protocol-all-Java driver which translates JDBC calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server.
4. Native-protocol-all-Java driver which converts JDBC calls directly into the network protocol used by DBMSs.

A survey of the currently available JDBC drivers is given in Table 2. Included in the table is a type column, with entries that coincide to the respective category in the above list.

4.2 JDBC API and JDBC Driver API

The JDBC API and JDBC Driver API are the two key APIs needed to successfully exploit database interactions. The JDBC API [126] does not contain actual Java classes, but is composed of a set of Java interfaces (see Section 2.2). A JDBC driver API is the set of Java classes that implements the JDBC interfaces. Specifically, the JDBC API [127, 130] consists of the following four main classes:

- *java.sql.DriverManager* which handles the loading of drivers and provides support for creating new database connections.
- *java.sql.Connection* which represents a connection to a particular database.
- *java.sql.Statement* which acts as a container for executing an SQL statement on a given connection.
- `java.sql.ResultSet` which controls access to the row results of a given `Statement`.

A connection represents a session with a specific database. Within the context of each connection, SQL statements are executed and results are returned. A `java.sql.Statement` object is used for executing a static SQL statement (database query) and obtaining the results that it produces. The `executeQuery()` method is used for handling SELECT statements that return a single `ResultSet`. The `execute()` method is used to handle more esoteric variations such as queries that return multiple results. The `executeUpdate()` method is used to handle INSERT, UPDATE, DELETE, and other simple statements that do not return results.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Type</th>
<th>DBMSs supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agave Software Design</td>
<td>3</td>
<td>Oracle, Sybase, Informix, others via ODBC</td>
</tr>
<tr>
<td>Asgard Software</td>
<td>3</td>
<td>Unisys A series DMSII database</td>
</tr>
<tr>
<td>Borland</td>
<td>4</td>
<td>InterBase 4.0</td>
</tr>
<tr>
<td>Caribou Lake Software</td>
<td>3</td>
<td>Ingres</td>
</tr>
<tr>
<td>Connect Software</td>
<td>4</td>
<td>Sybase, MS SQL Server</td>
</tr>
<tr>
<td>DataRamp</td>
<td>3</td>
<td>Several dozen through ODBC drivers.</td>
</tr>
<tr>
<td>Ensonex, Inc.</td>
<td>3</td>
<td>Several through ODBC</td>
</tr>
<tr>
<td>IBM</td>
<td>2,3</td>
<td>IBM DB2 Version 2</td>
</tr>
<tr>
<td>IBM</td>
<td>4</td>
<td>DB2 for OS/400</td>
</tr>
<tr>
<td>GWE Technologies</td>
<td>4</td>
<td>mysql</td>
</tr>
<tr>
<td>IDS Software</td>
<td>3</td>
<td>Oracle, Sybase, MS SQL Server, MS Access, Informix, Watcom, and others via ODBC</td>
</tr>
<tr>
<td>I-Kinetics, Inc.</td>
<td>3</td>
<td>Oracle, Informix, Sybase, and others via ODBC</td>
</tr>
<tr>
<td>Imaginary</td>
<td>4</td>
<td>mSQL</td>
</tr>
<tr>
<td>InterSoft</td>
<td>3</td>
<td>Essentia</td>
</tr>
<tr>
<td>Intersolv</td>
<td>3</td>
<td>DB2, Ingres, Informix, Oracle, MS SQL Server, Sybase 10/11</td>
</tr>
<tr>
<td>JavaSoft</td>
<td>1</td>
<td>Several dozen through ODBC drivers.</td>
</tr>
<tr>
<td>KonaSoft, Inc.</td>
<td>3,4</td>
<td>Sybase, Oracle, Informix</td>
</tr>
<tr>
<td>NetWay</td>
<td>3</td>
<td>Oracle, Informix, Sybase, MS SQL Server, DB2, others via ODBC</td>
</tr>
<tr>
<td>OpenLink</td>
<td>3</td>
<td>Oracle, Informix, Sybase, MS SQL Server, CA-Ingres, Progress, PostgreSQL, Solid, others via ODBC</td>
</tr>
<tr>
<td>Oracle Corporation</td>
<td>2</td>
<td>Oracle</td>
</tr>
<tr>
<td>SAS Institute Inc.</td>
<td>3,4</td>
<td>SAS, and via SAS/ACCESS, Oracle, Informix, Ingres, ADABAS</td>
</tr>
<tr>
<td>SCO</td>
<td>3</td>
<td>Informix, Oracle, Ingres, Sybase, Interbase</td>
</tr>
<tr>
<td>StormCloud Development</td>
<td>3</td>
<td>Any DBMS that is accessible via ODBC</td>
</tr>
<tr>
<td>Sybase, Inc.</td>
<td>3,4</td>
<td>Sybase SQL Server, SQL Anywhere, Sybase IQ, Replication Server, 25 Legacy DB Servers via Sybase OmniCONNECT</td>
</tr>
<tr>
<td>Symantec</td>
<td>3</td>
<td>Oracle, Sybase, MS SQL Server, MS Access, Watcom, others via ODBC</td>
</tr>
<tr>
<td>Trifox, Inc.</td>
<td>3</td>
<td>ADABAS, DB2, Informix, Ingres, Oracle, Rdb, SQL Server, Sybase, and legacy systems via GENESIS.</td>
</tr>
<tr>
<td>Visigenic</td>
<td>3</td>
<td>Several dozen through ODBC drivers.</td>
</tr>
<tr>
<td>WebLogic</td>
<td>2</td>
<td>Oracle, Sybase, MS SQL Server</td>
</tr>
<tr>
<td>WebLogic</td>
<td>3</td>
<td>Several dozen through ODBC drivers.</td>
</tr>
<tr>
<td>XDB Systems, Inc.</td>
<td>1,3</td>
<td>Many databases through ODBC</td>
</tr>
<tr>
<td>Yard Software GmbH</td>
<td>4</td>
<td>YARD-SQL Database</td>
</tr>
</tbody>
</table>

Table 2. JDBC Drivers.

The `java.sql.Statement` class has two important sub-types: `java.sql.PreparedStatement` for executing a pre-compiled SQL statement, and `java.sql.CallableStatement` for executing a call to a stored database procedure. An instance of the `PreparedStatement` class stores a pre-compiled SQL statement and is useful for executing the stored SQL statement multiple times. The class `PreparedStatement` also supports IN parameters. The `CallableStatement`
class extends PreparedStatement so that it is possible to use stored database procedures. The CallableStatement supports OUT parameters.

The java.sql.ResultSet class provides access to a table of data generated by executing a SELECT statement. An instance of the class ResultSet maintains a cursor pointing to its current row of data, which initially is positioned before the first row. The next() method moves the cursor to the next row. The get() method retrieves column values for the current row. In doing so, either the index number of the column, or the attribute name for the column is utilized. Note that two additional classes for metadata interfaces are also contained in the JDBC API: java.sql.DatabaseMetaData and java.sql.ResultSetMetaData. The combination of the two interface enables the client to query metadata information from the database server, e.g., to obtain metadata such as the list of table names, the list of attribute names for a chosen table, or the data types for attributes.

The JDBC Driver API [127, 135] is contained in the java.sql.Driver interface. In the majority of cases, database drivers (java.sql.Driver) simply need to provide implementations of the interface classes provided by the JDBC API. Specifically, each driver must provide implementations of:

java.sql.Connection  java.sql.Statement  java.sql.ResultSet
java.sql.PreparedStatement  java.sql.CallableStatement

In addition, each database driver needs to provide a class which implements the java.sql.Driver interface, which is used by the java.sql.DriverManager class when it needs to locate a driver for a particular database URL.

The JDBC API supports two-tier and three-tier architectures [127] for database access as shown in Figure 10. In the two-tier architectures shown in Figures 10.1 and 10.2, a Java applet or an application talks directly to the database through a JDBC driver that can communicate with the particular DBMS being accessed. In the three-tier architecture shown in Figure 10.3, Java applications make calls to the "middle-tier" functional servers.

Specifically, Figure 10.1 illustrates the case where the JDBC API is included inside a Java applet. In this model, a user downloads a Java applet on-demand and accesses a database on a local or remote machine. The downloaded applet can be either untrusted or trusted with respect to security (see Section 5.4), depending on the circumstances. If the applet is untrusted, (e.g., downloaded from an ad-hoc search of the Internet), then based on security concerns, this untrusted Java applet will be permitted only a limited set of operations in the local host machine. For instance, the untrusted applet may not access local files nor open network connections to other arbitrary hosts. In contrast, a trusted applet can operate just like Java applications to fully exploit the capabilities of the JDBC API. In Figure 10.2, the two-tiered architecture for Java applications using the JDBC API is given. In this approach, it is assumed that the Java application was delivered off-line from a trusted party, and is allowed to read and write local files, open network connections, etc., just like any other Java application.

In the three-tier architecture given in Figure 10.3, Java applications make calls to a "middle-tier" of services on the net instead of directly calling the database server. The middle-tier contains JDBC APIs, which in turn access databases, get the results, and send the results back to the Java application. The calls from the Java application to the middle-tier might be made through a RPC (see Section 3.4.3), an ORB (see Section 3.4.2), the Java Native Interface (see Section 3.3), or RMI (see Section 5.5). The three-tier architecture has several advantages. First, the middle-tier can maintain control over data access by being able to change policies and permissions on the fly without requiring changes to the application. Second, due to an additional level of indirection,
design and implementation flexibilities are available both for developing the application and the middle-level servers. For example, the front-end application may utilize an easy-to-use higher-level API in development which can then be translated into appropriate low-level calls at the middle level. Third, the middle-level server can be developed to optimize performance without conflicting with other competing issues such as the front-end user interface development, e.g., the middle-level is written in C or C++ for performance, with the GUI written using Java. Clearly, the three-tier approach in Figure 10.3 is particularly relevant for integrating with legacy and COTS applications, as has been discussed in Sections 3.4.

4.3 Object-Oriented DBMS

There are a number of commercial products that support the integration of Java with object-oriented database management systems (OODBMS). These products are briefly discussed since they are critical for building homogeneous applications with Java front-ends and OODBMS back-ends or for integrating legacy applications, databases, and COTS with an OODBMS. Sample products include:

- The Versant Java Language Interface developed by Versant Object Technology [136]. This product enables software engineers to utilize Java as a full-fledged programming language for delivering scalable, high-performance, multi-user distributed applications for the Internet and corporate intranets.

- The ObjectStore Persistent Storage Engine (PSE) developed by Object Design, Inc. [131]. Unlike Versant’s product, this product can be seen as a single-user, lightweight version of
the ObjectStore database. The company claims that there is a Java interface interacting with their new ObjectStore version 5.0.

- The Java Relational Binding, by O2 Technology [129], that was initially developed to allow binding of Java applications to a relational database. In the next phase, the company plans to enhance the Java Relational Binding to support binding to the O2 OODBMS.

The main drawback of each of these products is that the Java interfaces are specific to the particular OODBMS platform.

In an effort to produce a standard API for OODBMSs, the ODMG (Object Database Management Group [132]) recently published the Java binding to the ODMG 2.0 [7, 132]. The major components of ODMG 2.0 are: an Object Model that is the common data model to be supported by OODBMSs; an Object Specification Language that is the specification languages for OODBMSs; and, an Object Query Language that is a declarative language for querying and updating database objects. In addition to a Java language binding, this standard also includes C++ and SmallTalk bindings [7]. The ODMG group also has the Object Database Adaptor (ODA) [7] which interfaces to to OMG's ORB, and includes, as shown in Figure 11: translation adaptors, the Basic Object Adaptor (BOA), and the Library Object Adaptor (LOA) The BOA is used for normal object accesses, and is executed via an inter-process mechanism for every dispatch of every method. Alternatively, the LOA allows a direct, considerably faster access to objects. After the first invocation (via the usual ORB mechanism), or through a compile-time optimization, a direct link is established to the object. Then, later access by the client to the object is done directly until the client notifies the ORB that it has released the object. ODA has been proposed to provide the ability to register a subspace of object identifiers and to allow access (including direct access) to the objects as if they had been individually registered.

![Diagram](image)

**Figure 11:** Proposed Architecture for BOA/LOA.

### 4.4 Available Capabilities for Persistence

Java offers four alternative techniques for delivering persistency to instances of a Java application/applet: object serialization, JDBC, JSQL, and OODBMS. In this section, each of these techniques is explained by utilizing examples that illustrate the approach. The examples are based
on the following definition for a sample grocery Item class, which was previously introduced in Section 1.2.2.

```java
class Item {
    private String UPC, Name;
    private int Quantity;
    private double RetailCost;
    protected double WholeCost;

    public Item() { ... };
    public void finalize() { ... };
    public boolean Modify_Inventory(int Delta) { ... };
    public int Get_InStock_Amt() { return Quantity; };

    ...;
}
```

Note that object serialization is also discussed as part of the upcoming material on RMI (see Section 5.5) and JavaBeans (see Section 5.6.4).

4.4.1 Object Serialization

Object serialization [130, 133, 134] is a mechanism of flattening the instance(s) of class(es) into a stream of bytes. The flattened stream of bytes can be stored into a file persistently for later use, or alternatively, can be shipped to another Java Virtual Machine. Java's serialization feature defines interfaces for writing and reading copies of instances into streams, and classes that implement those interfaces. As an example, consider the code given below that flattens an instance defining a grocery Item:

```java
Item i = new Item("11111", "iii", 1, 10.0, 9.9); \ Object Creation
FileOutputStream f = new FileOutputStream("tmp"); \ "tmp" File Creation
ObjectOutputStream s = new ObjectOutputStream(f); \ Linkings to "tmp"
s.writeObject(i); \ Object i is Serialized and Stored in s
s.flush(); \ Flattened Object Stored in s is Flushed to "tmp"
```

Eventually the flattened and stored instance will have to be read in. The built-in feature of deserialization simplifies this task, as shown in the following code:

```java
FileInputStream in = new FileInputStream("tmp"); \ "tmp" File Preparation
ObjectInputStream s = new ObjectInputStream(in); \ s Assigned to the Input Stream
Item i = (Item)s.readObject(); \ s is Read and Constructed to Fit to i
```

In the final step of the above code, the `readObject()` method traverses the instance's references to other instances recursively to complete the deserialization. Object serialization is a convenient way of storing/retrieving persistent data that does not require database functions such as concurrent read/write, query, recovery, etc.

4.4.2 JDBC

As discussed in Section 4.1, the JDBC offers another alternative technique for achieving object persistency, particularly for those applications that require large volumes of objects that are suitable for storage in a relational database. In the example given below, the grocery Item is stored in a relational table and accessed via various system-level calls.
import java.net.URL;
import java.sql.*;

class Select {
    public static void main(String argv[]) {
        try {
            // Create a URL to Access the Database through a JDBC-ODBC Bridge.
            // Use the Subprotocol "odbc" and the Local ODBC DB Source Named "wombat"
            String url = "jdbc:odbc:wombat";

            // Connect the Database at the url with userid "Mitre" and password "ABC".
            Connection con = DriverManager.getConnection(url, "Mitre", "ABC");

            // Execute a SELECT Statement.
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT item-upc, item-name, item-retailcost, item-wholecost FROM items");

            // Step Through the Result Rows.
            while (rs.next){
                String upc = rs.getString(item-upc);
                String name = rs.getString(item-name);
                double retailcost = rs.getDouble(3);
                double wholecost = rs.getDouble(4);
                System.out.println("upc= " + upc + " name= " + name + "
                        retailcost= " + retailcost + " wholecost= " + wholecost);
            }
            stmt.close();
            con.close();
        } 
        catch (java.lang.Exception ex) { ex.printStackTrace(); }
    }
}

The advantage of the JDBC is that it provides a uniform way of accessing relational databases using a standard. Software engineers who are familiar with SQL should be able to easily utilize JDBC. Unfortunately, the problem of impedance mismatch still remains. Unlike object serialization, the tabularized relational ‘object’ that has been retrieved through JDBC will have to be constructed by a software engineer into a corresponding object-oriented instance of Java. A performance delay for the data translation and additional application development cost are inevitable.

4.4.3 JSQL

Java with Embedded SQL (JSQL) [128] is another technique for dealing with persistency by allowing Java applications to interact with relational databases. JSQL is based on the use of JDBC and has been proposed by a consortium of companies including IBM, Oracle, Sybase, and Tandem. JSQL comprises a set of clauses that extend Java applications to include static SQL constructs. A JSQL translator is a utility that transforms those JSQL clauses into the standard Java code that is needed to access the database through a call interface. The output of a JSQL translator is a generated Java application that can then be compiled by the Java compiler. The following code illustrates the use of JSQL for selecting attributes from a table:
import jsql.runtime.*;
  // Create a Connection-Context Class which has Methods for
  // Opening a Connection to a Database Schema, Given a URL,
  // or Other Connection-String, a User Name, and a Password.
#sql context items;

public class Select_Item {

  public static void main(String argv[])
  throws SQLException {
    // Make the Connection to Wombat with Mitre userid and "" passwd.
    items.items(new items("jdbc:odbc:wombat", "Mitre",""));

    // Create Result-Set Iterator Object Containing the Result-Set
    #sql iterator Items (String upc, String name);
    Items item;

    // Set Variable "item" to Contain Result-Set of the Query.
    #sql item = {SELECT ITEM-UPC as "upc", ITEM-NAME as "name" FROM ITEMS};

    // "item" now has Accessor Methods to Obtain Column Values.
    while (item.next()) {
      System.out.println("upc = " + item.upc() + ", name = " + item.name());
    }
  }
}

JSQL can be considered as a static SQL whereas JDBC is thought of a dynamic SQL [128]. A
dynamic interface such as JDBC is a call interface that passes SQL commands as strings
to a DBMS. No analysis or checking on those strings is performed until the database receives
the strings at the execution time. Dynamic SQL is more flexible than static SQL in the sense
that a calling program can compose SQL strings at run-time. On the other hand, embedded
SQL does better for applications that use static, fixed SQL commands. In such an application,
error-checking can be done during the development cycle. Also, the pre-compilation of the SQL
command can permit faster run-time execution. JSQL source programs tend to be smaller than
their equivalent JDBC programs.

4.4.4 OODBMSs

A fourth technique for addressing persistency is to connect a Java application directly to an
OODBMS. Unlike relational databases, the object-oriented model of a Java application (i.e.,
the classes, packages, etc.) can be directly mapped into the schema of an OODBMS model,
thereby eliminating the impedance mismatch regarding instances of the Java-to-relational case.
The example code segment given below illustrates the interface of ObjectStore [131] with Java:

    Database db = Database.open("items.db", DatabaseReadWrite);

    Transaction t = Transaction.begin(Transaction.readOnly);

    // Use the "AllItems" Database Root to Access Objects in the
    // Database which is an Entry Point into a Database.
    SetOfItem items = (SetOfItem) db.getRoot("AllItems");

43
// Get the Enumeration Set of Items.
ItemEnumeration ei = items.elements();

while(i.hasMoreElements()) {
    Item i = ei.nextElement();
    System.out.println("UPC = " + i.getUPC + " Name = " + i.getName);
}

t.commit();
db.close();

Note that some conversion could still be needed if the data modelings in Java and the OODBMS are not carefully performed. This could occur if a modeling construct in Java is utilized that does not have an analog in the OODBMS, or vice versa. The situation can be further aggravated when the vendors of the OODBMS continuously develop and use proprietary interface tools to the OODBMSs. One way of averting the trend is to conform to a standard such as ODMG as discussed in Section 4.3.

5 Designing/Developing Java Applications

The goal of this section is to provide an examination into the type of software engineering and programming that is needed to successfully utilize Java and its major APIs for significant and productive application design and development. Specifically, this section explores Java APIs that involve: graphical user interfaces or GUIs in Section 5.1, multi-processing via threads in Section 5.2, networking for inter-process communication and Internet access of URLs in Section 5.3, security and access control techniques in Section 5.4, and remote method invocation which is crucial for DOC in Section 5.5. While the APIs are the major tools of Java available to software engineers, the utilization of these tools is facilitated by integrated development environments (IDEs) that are available in the shareware and commercial marketplace, which are examined in Section 5.7. Of particular interest is JavaBeans and associated IDEs for plug-and-play, visual development, that allows software engineers to combine components into significant applications that are rapidly prototyped, as examined in Section 5.6.

5.1 Java and GUIs

The Abstract Window Toolkit (AWT) [66] supplied by Java provides primitives for the design and development of graphical user interfaces (GUIs) in a platform-independent fashion. With AWT 1.1, an application or applet with a GUI component can be written once and run anywhere. AWT 1.1 is a part of Java Foundation Class (JFC) API for providing graphical user interfaces for Java programs. JFC [26, 66, 74], which is still in development, extends the original AWT by adding a comprehensive set of graphical interface libraries that is completely portable and delivered as part of the Java platform. JFC will include many of the key features of the Netscape's Internet Foundation Classes [60], and has been jointly developed by SUN, Netscape, and IBM.

5.1.1 The AWT API

The AWT API [6, 27, 66, 67] provides the components, containers, and layout managers for the creation of a graphical user interface. A component in AWT is a building block that is used to develop the GUI for an application, and is supported by the Component class. AWT has
the following components which the software engineer directly interacts with: Canvases which allow the software engineer to create custom components, Checkboxes for check boxes and radio buttons, Menu, MenuBar, MenuItem, Choice, Button, Scrollbar, List, TextArea, TextField, and Label.

Components in AWT are placed and grouped within a container; that is supported via the Container class. AWT provides two types of containers, Window and Panel. The Window class has two subclasses Frame and Dialog which has FileDialog as a subclass. The subclasses of Window provide windows which can contain or hold components. The Frame class is a fully functioning window with its own title and icon. On the other hand, the Dialog class is a pop-up window which is dependent on Frame. The Panel class is a pure container which groups components within an area of an existing window.

Layout managers provide the ways of organizing the components in the containers. Each container is given a layout manager that decides where each component should be displayed. There are five kinds of layout managers in the AWT: BorderLayout, CardLayout, FlowLayout, GridLayout, and GridBagLayout. BorderLayout is the most simple manager, and the default manager for all Window objects. This layout manager places the components into one of five areas (i.e., North, South, East, West, or Center) of the container as dictated by the software engineer. CardLayout is used to hold several components for an area and displays only one component at a time. FlowLayout is the default layout manager for every Panel, and places the components from left to right until no more components will fit in a row, and then starts to fill a new row. GridLayout places components that are each the same size in a set number of rows and columns as defined by a software engineer for the application. GridBagLayout is the most flexible layout manager, providing versatility to software engineers by supporting the placement of components within a grid of cells, with some components spanning more than one cell.

The AWT API in Java is based on the “peer” model, which means that each Java Component class creates a native component instance, so that it can support the native look-and-feel of the underlying operating system on which the component is actually being displayed. Every AWT Component class has an equivalent peer class in the platform-specific implementation, which means that every Component object (instance) has a peer object (instance) that controls the Component’s look-and-feel on a specific platform. A peer is created just prior to the time that its corresponding Component object is drawn for the first time.

In addition to the GUI components that have been described to this point, the AWT API also provides primitives for manipulating graphical data. The Graphics class provides the typical methods needed to create and draw on the screen. Primitives for the Graphics class include methods for drawing lines, rectangles, ovals, arcs, polygon, and so on. Java uses the RGB (Red, Green, and Blue) color model. The Graphics class provides two methods for manipulating colors: getColor which returns the Graphics object’s current color as a Color object and setColor which sets the Graphics object’s color by passing it a Color object. The Graphics class also defines methods that allow software engineers to display text: drawBytes which draws a text using an array of bytes representing characters, drawChars which draws an array of characters, drawString which draws a string of characters, setFont which selects a font, and getFontList which returns a list of fonts. For displaying images, two methods are provided in the Graphics class: the getImage method which gets the image from the given URL object, and the drawImage method which draws the image at the given coordinates.
5.1.2 Java Foundation Classes

The Java Foundation Classes, JFC [26, 66, 74], is an enhanced and extended graphical user interface library of AWT 1.0. In JFC, all of the components are JavaBean compliant (see Section 5.6), which means that they have a consistent API and a standard event handling mechanism. This standard property and event model lends itself well to component reuse and interoperability. To resolve the limits of the peer model, which has been adopted in AWT 1.0, JFC contains the lightweight UI framework, which does not require components to be in a native window. This means that an application can be written entirely in Java and not carry any overhead from the native windowing system. By using the lightweight UI framework, applications can have universal look-and-feel.

The event model supported by AWT 1.0 required that all events pass up the inheritance hierarchy, forcing objects to catch and process by subclassing, and overriding either the handleEvent or action methods. This model has two major problems: an excessive number of classes is created since each individual component can be subclassed to specifically handle its target events; and, complex if-then-else logic is required in the top-level object to determine which object triggered an event. In the new event model of JFC, the callback-style delegation event model, events are class specific and can be sent to or delegated directly to the object that can handle the request. Only objects interested in a particular event need to deal with the event and no super-event handler is required.

In addition to the change in event model, there were also a number of other changes in JFC to broaden the choices for software engineers, including:

- JFC supports printing by adding the PrintJob class to encapsulate printing functionality.
- JFC supports the transfer of data via a clipboard, enabling dynamic data types to be created, registered, and transferred from both within and across process pace boundaries.
- JFC handles dynamic changes in the desktop color scheme.
- JFC offers mouseless operation for keyboard navigation and accelerators.
- JFC adds a ScrollPane class as a container that implements automatic scrolling and supports ‘when needed’, ‘always’, and ‘never’ modes for its scrollbars.

Also, the JFC fixes the problems related to enhanced graphics and images in AWK 1.0 as part of the AWK 1.1 release.

JFC has only been partially released to date, with many new and extended features planned for the next release, including:

- Drag-and-drop programming that enables objects to be dragged from one application to another, thereby promoting reuse.
- New high-level components such as tree view, list view, table view, toolbar, pane splitter, tabbed folder, etc., that offer more choices to software engineers.
- Pluggable look-and-feel that allows a separation of the GUI style from the function of the AWT components, which enables an application’s visual interface to be individualized.
- A two-dimensional graphics API that extends the existing graphics capabilities.
The Screen Reader that creates an off-screen representation of the GUI components enabling the
information to be provided via text to speech and/or Braille terminals.

The Screen Magnifier which allows the software engineer to adjust the magnification of the
screen 1 to 10 times the normal size for visually impaired users of applications.

These planned features of JFC demonstrate the potential breadth of the language for handling a
wide variety of application in different domains.

There are several visual interface builders which support the lightweight UI framework of JFC.
Sun [74] has SwingSet which provides the following components: Label, Bordered Pane, Progress
Bar, Tool Tip, Button, Radio Button, Checkbox, Tool Bar, Slider, Combo Box, Menu, Tree,
Scroll Bar, List Box, Tabbed Pane, and Table. Marimba Inc. [117] has developed Bongo which
is a visual interface builder for Java. DTAI Incorporate [57] has the Gadget Windowing Toolkit
(GWT) which is distributed as freeware. Note that when using these tools, software engineers
can only design the user interface aspects of the application. The remaining functionality can be
designed using interactive development environments (see Section 5.7).

5.2 Java and Threads

Threads, as introduced in Section 1.2.4, allow a single Java application to have multiple active
threads of control in a single application, which may all simultaneously share application resources
while performing different tasks. In fact, a thread can perform an independent task, and can also
simultaneously share access to application's objects with other threads, which represents one of
the most useful features of multithreading in Java and one of its greatest pitfalls. If a software
engineer is not careful, two or more threads may modify the same piece of data in an interleaved
way such that the state of the object becomes corrupted, causing what is termed a race hazard.

Despite this potential problem, a significant advantage to threads is that they allow interactive
applications to be designed and developed with versatile and dynamic behavior, which benefits
end-users. Single-threaded systems usually provide an illusion of multiple threads either by using
interrupts or by polling, which mixes displayed information with user input, and leads to complex
code that is difficult to maintain. When languages like Java provide primitives for multi-threading,
the sharing of control within a process can be directly supported without needing to be specially
handcrafted and customized for each new polling case. Threads are used extensively in Java
applications, particularly in GUIs (see section 5.1) to allow multiple windows to be open and
active. The remainder of this section examines the creation and use of threads in Java.

5.2.1 Creating Threads

Java threads are implemented by the Thread class [96], which is part of the java.lang package [65].
Threads in Java are defined as independent classes that can be compiled separately, which is contrary to what is supported by Ada95 tasks [29]. The Thread class implements a
system-independent definition of Java threads, with the actual implementation of concurrency
provided by a platform-specific implementation. To create a thread, a simple declaration is util-
ized:

```
Thread worker = new Thread();
```

After a thread is created, it is available to be configured and run. Configuring a thread in Java
involves setting its priority and name. The thread name can be passed to the constructor or
supplied explicitly by invoking the \texttt{worker.set\_name} method. Once the name has been set, the method \texttt{worker.start} can be utilized to initiate the running of the thread. When the \texttt{start} method is invoked, it first spawns a new thread of control based on the data in the \texttt{Thread} object. Then, the JVM invokes the new thread's \texttt{run} method, making the thread active. The thread finishes its execution when the \texttt{run} method exits, or when it receives an explicit \texttt{stop} message. The implementation of \texttt{Thread.run()} must be extended by the software engineer to represent the required application functionality. The \texttt{run} method cannot throw any exception since it overrides a method that does not throw any. This means that all of the exceptions that may be thrown in the new implementation must be caught within the \texttt{run} method. Note that the methods \texttt{set\_name}, \texttt{start}, and \texttt{run} are defined in the \texttt{Thread} class.

Every thread belongs to a \texttt{Thread\_group} class \cite{116} for security reasons. A thread group can be contained within another thread group, providing the ability to create and maintain a hierarchy of threads within a Java application. A thread can modify any other thread in its group, but cannot change threads that are in other groups. Thread groups are also used to assign a maximum priority to any thread in the group. The group identification can be given to the constructor when the thread is created; if not, the group is the same one as the creator's thread group.

5.2.2 Thread Scheduling

In any machine, one of the runnable threads with the highest-priority will run next, and all threads at that priority will get some processor time. Lower-priority threads might be able to run at some point, depending on the state of the machine, but there is no guarantee that this will occur. A thread is blocked if it is sleeping or executing a system or thread function that is blocked. When a thread blocks, Java picks the next highest-priority runnable thread and lets its run. The Java runtime environment can even suspend the highest-priority thread to let another thread at the same priority run. A thread's priority is equal to that of the thread that created it, and it can be modified with the \texttt{set\_priority} method defined on the \texttt{Thread} class. Understanding priorities is critical to successfully using threads, since priorities will dictate the runtime behavior of an application. In general, the continuously running portion of an application should run in a lower-priority thread than the threads dealing with rarer events such as user input. This will allow the JVM to switch control to higher-priority threads on the basis of user actions (e.g., menu selection, mouse movement, etc.). Two static methods of \texttt{Thread} that are useful to software engineers to manipulate priorities and, therefore, application behavior are \texttt{sleep} and \texttt{yield}: \texttt{sleep} puts the thread to sleep during a specified time period, while \texttt{yield} yields so that another thread (maybe the same one) can run.

5.2.3 Synchronization

When using multiple threads within a single application that need to all access the same object, the software engineer must synchronize their actions. This is supported in Java through the definition of \texttt{synchronized} methods on an object. When a thread invokes a \texttt{synchronized} method on an object, that object is \texttt{locked} until the method returns. Another thread invoking a \texttt{synchronized} method on that same object will block until the lock is released. Synchronization forces mutually-exclusive execution. When a synchronized method is invoked on an object that is already locked by that same thread, the method executes, but the lock is not released until the outermost synchronized method returns. For example, suppose that a bank application has a class \texttt{Account}:

\begin{verbatim}
class Account {
    private double balance;
}
\end{verbatim}
public Account(double initialDeposit) {
    balance = initialDeposit;
}

public synchronized double getBalance() {
    return balance;
}

public synchronized void deposit(double amount) {
    balance += amount;
}

Whenever there is data that can be updated by more than one synchronized method, such as the balance member variable in the example above, it is best to define it as private, so that it is only accessible by specific methods. Whenever there are two threads and two objects with locks, there is a potential for deadlock. Java neither detects nor prevents deadlock; the software engineer must be responsible for avoiding them during design and implementation. Note that a synchronized method can be overridden in a subclass, with the new method synchronized or not.

Synchronization avoids method interference, but it is not enough to coordinate or communicate among threads. For these purposes, the wait and notify methods defined on the Java Object are utilized. The wait method stops the execution of a thread until some condition holds. The notify method is defined to tell waiting threads that something has occurred. The methods wait and notify are used as follows:

    synchronized void doWhenCondition() {
        while (!condition) {
            wait();
            // Do Actions for Method after Condition has been Met.
        }
    }

    synchronized void changeCondition() {
        // Change Some Value used in a Condition Test.
        notify();
    }

Notice that wait and notify are used inside synchronized methods. When wait suspends the thread, it atomically releases the lock on the object, avoiding race hazards. The condition testing for the wait should always be in a loop. The notify method is invoked after the modification to data that other threads may be waiting for. In Java, multiple threads can be waiting for the same condition to become true. The notify method, when invoked, will awake exactly one thread. If multiple threads must be awoken, the software engineer can use the notifyAll method. The wait method is overloaded on Object, with other versions available that allow the software engineer to specify an elapsed time that the thread has to wait before it can resume execution.

5.2.4 Suspending, Interrupting, and Ending Execution

At different times in an multi-threaded application, in order to achieve the desired behavior, a software engineer may need to suspend a thread to wait for an event to occur, interrupt a thread
when condition(s) are met, and end a thread’s execution. In Java, threads can be suspended to check or wait for some condition, and can then be either resumed or stopped as needed. The interrupt concept in Java threads is similar to Ada95’s abort [29], and is utilized when a software engineer wants to give the running thread some control over when it will handle an event. In this situation, the thread itself checks for a condition and finishes its own execution. Threads that are interrupted while sleeping or waiting throw the InterruptedException. To end the execution of a thread, the stop method throws a ThreadDeath object to the target thread. ThreadDeath is a subclass of Error. The top-level error handler then kills the thread without printing any message. A thread can also directly invoke its destroy method. But, this action is drastic since it kills the thread dead with no cleanup, and can leave other threads waiting forever.

Every Java application has at least one thread, the one that is started when main is executed. Once the main thread is executing, the Java application can create other threads. The application ends when its last thread finishes executing. In Java, there are two kinds of threads: user and daemon. When the last user thread finishes, the remaining deamon threads are stopped and the application is done. The methods setDaemon(true) and getDaemon are used to set and test this characteristic. Deamon-ness is inherited from the thread that creates a thread, and cannot be changed after the thread starts running.

5.2.5 Using the Runnable Interface

In addition to creating threads as presented in Section 5.2.1, a software engineer can customize thread concepts by providing his/her own implementation of the Runnable interface, which is part of the java.lang API. The Runnable interface declares a single method run(). The Thread class implements the Runnable interface, since it has a run method that can be overridden. A Runnable object can be executed in its own thread by passing it to the Thread constructor, and as described in Section 5.2.1, can be given a name and a thread group. To illustrate the Runnable interface, the following example taken from [1] is provided.

class WriteWord extends Thread {
    String word;
    int delay;
    WriteWord(String whatWord, int time) {
        word = whatWord;
        delay = time;
    }
    public void run() {
        try {
            for (; ;) {
                System.out.print(word + " ");
                sleep(delay);
            }
        } catch (InterruptedException e) {
            return;
        }
    }
    public static void main(String[] args) {
        new WriteWord("He", 30).start();
        new WriteWord("She", 10).start();
    }
}

class RunWord implements Runnable {
    String word;
    int delay;
    RunWord(String whatWord, int time) {
        word = whatWord;
        delay = time;
    }
    public void run() {
        try {
            for (; ;) {
                System.out.print(word + " ");
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
            return;
        }
    }
    public static void main(String[] args) {
        Runnable ping = new RunWord("He", 30);
        Runnable pong = new RunWord("She", 10);
    }
}

50
In the example, the same class is implemented in two ways: as a subclass of Thread in WriteWord and as the Runnable interface in RunWord. Regardless of the implementation approach chosen, the end result achieved by WriteWord and RunWord classes is equivalent.

5.3 Java and Networking

Computers running on the Internet or intranets communicate with one another using TCP (transmission control protocol) and UDP (user datagram protocol) which provide four layer of access: application layer (HTTP, ftp, telnet, ...), transport layer (TCP, UDP, ...), network layer (IP, ...), and link layer (device driver, ...). Java applications communicate over the network at the application layer, using the classes in the java.net package [98], but the classes (protocols/layer) actually chosen depend on the use of TCP or UDP. TCP provides reliable point-to-point communication, ensuring that messages are received in the same order that they were sent. On the other hand, UDP is not a connection-based protocol, and while it does not have the same reliability, it provides a faster link.

Most every computer today has at least one physical connection to the network. Data that arrives at a computer may be intended for different applications or different threads of the same applications. The computer knows which application to forward the data to through the use of ports. Data transmitted through the Internet has a 32-bit IP address to identify the computer, and a 16-bit port identification. No two applications have the same port number. Port numbers ranging from 0 to 1023 are restricted for system services, and they are called well-known ports. The remainder of this section reviews the networking capabilities of Java by examining its support for: Internet access via URLs, point-to-point communications between two processes via sockets, and datagrams that provide freedom and versatility at the expense of a guaranteed, in order delivery of messages.

5.3.1 URLs

Universal resource locators, URLs, are references to resources on the Internet, such as, the http addresses that have been cited throughout this white paper. URLs have two main components: the protocol that is to be used to fetch the resource (e.g., http, ftp, etc.), and the complete address of the resource. The complete address of the resource has the following format:

- Host name: The name of the machine the resource lives on.
- Filename: The pathname to the file on the machine.
- Port number: Port number to connect to (optional).
- Reference: Reference to a named anchor within the resource (optional).

Within Java applications, URL objects [104] can be created to represent URL addresses. The methods that are defined on the URL object can be utilized to retrieve information about the pointed resources, e.g., getProtocol, getHost, getPort, getFile, and getRef. A URL can also be created from a base URL and a relative URL.

In Java, the simplest way of creating a URL object is by providing a String with the URL in its constructor:

```java
URL gamelan = new URL("http://www.gamelan.com/");
```

This represents an absolute URL, since it refers to a complete http address. A relative URL has only enough information to reach the resource relative to another, previously defined URL:
URLgamelanNetwork = new(gamelan, "Gamelan.network.html");

There are other overloaded formats for the URL constructor where the port number and the reference can also be given. If the arguments to the constructor refer to a null resource or an unknown protocol, the constructor will throw a MalformedURLException.

After creating a URL object, the URL's openStream method can be called to get a stream and read the contents of the URL. The openStream method returns a java.io.InputStream object which can be read using the normal InputStream methods. The URL's openConnection method allows a Java application connect with a URL over the network:

URL yahoo = new URL("http://www.yahoo.com/");
yahoo.openConnection();

The openConnection method creates and initializes a new URLConnection object, connects to the URL, and returns the URLConnection object. After this call has been made, the Java application is able to read from or write to the connection.

5.3.2 Sockets

A socket is one endpoint of a two-way communication link between two processes (e.g., a client and a server) running on the network. TCP provides this reliable, point-to-point communication channel. Each side of the connection, the client and the server, binds a socket to its end of the connection, allowing it to then read and write to the socket. The java.net package provides the Socket[101] and ServerSocket[102] classes for each side of the connection, respectively. The URL class is built on top of the socket abstraction. While sockets in Java do not provide any new concepts to the traditional Unix sockets[45], the API realization of sockets provides a higher-level abstraction, hopefully resulting in an easier use of the protocol. The Socket and ServerSocket classes are platform-independent implementations of the client and the server ends of the socket, and reside on top of platform-dependent implementations, hiding the details of any particular system from the Java application. Using these classes instead of the native code, Java applications can communicate over the network in a platform-independent fashion.

The steps for connecting from the client side to the server using a socket are:

1. open a socket,
2. open an input stream and an output stream to the socket,
3. read from and write to the stream according to the server's protocol,
4. close the streams, and
5. close the socket.

When executing the server program, the server creates a new ServerSocket with a particular port number and waits, listening for some request. This is done with the accept method that waits until a client starts up and requests a connection on the specified port for the server. When the accept method successfully establishes a connection with the client, it returns a new Socket object which is bound to a new local port, as shown below:
Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept();
} catch (IOException e) {
...
}

The server can then communicate with the client over the new socket on a port that is different from the one it was originally listening to for connections. The server can continue to listen for client connection requests on the original port through the ServerSocket by using different threads to deal with each client. All client-server pairs must have some protocol by which they speak to one another.

5.3.3 Datagrams

The UDP protocol provides a mode of network communication whereby applications send packets of data, called datagrams, to one another. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. Datagrams are often used for broadcast messages, where one message can be sent out on the network to be received by one or more destinations. The java.net package provides two classes to work with datagrams: DatagramPackets [100] are sent through DatagramSockets [99]. Datagram sockets can be created with a DatagramSocket constructor with no arguments:

    socket = new DatagramSocket();

This declaration does not bind the socket to any local available port. There is another constructor that allows the program to specify the desired port to which to bind the socket.

Packets are created differently depending on their purpose. Those that are used for receiving a datagram require only two parameters: a byte array and its length.

    packet = new DatagramPacket(buf, 256);
    socket.receive(packet);

The receive method waits forever until a datagram arrives. Given that the arrival is not guaranteed, this method is usually within a loop with a timeout. Packets that are to be sent through the datagram socket also require the address and the port of the destination application:

    packet = new DatagramPacket(buf, buf.length, address, port);
    socket.send(packet);

Note that ports are limited resources and should always be closed after the end of the communication.

5.4 Java and Security

While applets solve many of the important problems in client/server and network centric computing (code-once, use-many), they also raise concerns about security. Protective measures can go from a firewall between the Internet and the company's intranet to obtaining software only from trusted sources to the use of anti-virus programs to check all of the software. There is a trade-off between the level of security achieved and the cost of the procedure, since it is impossible to find a security system that is 100% secure.

53
Java provides transparent, general, and open security mechanisms which do not require any knowledge or action on the part of the software engineer. The sandbox is Java's basic security mechanism, which forces downloaded applets to run in a confined portion of the system, and allows the software engineer to customize a security policy. One result of this approach is that the security policy is hard coded as part of the application, providing little or no flexibility either to modify the policy or to have discretionary access control. The Java language/environment has features that assist in protecting the integrity of the system and preventing several common attacks. Mainly, Java programs are not allowed to define their own memory pointers or to access physical memory directly. Studies have shown that 40% to 50% of all bugs are caused by errors in memory management. In this way, Java has more reliable code. The remainder of this section contains four parts to review the support for security in Java, including: core security concepts; the functionality and operation of security managers; advanced security features such as digital signatures, keys, message digests, and access control lists; and, discretionary access control techniques that offer versatility from a user-security perspective.

5.4.1 Core Security Concepts

An applet's actions are restricted to its sandbox, an area of the Web browser dedicated to that applet. The applet may do anything it wants within its sandbox, but cannot read or alter any data outside it. The sandbox model supports the running of untrusted code in a trusted environment so that if a user accidentally imports a hostile applet, that applet cannot damage the local machine. To implement sandboxes, the Java platform relies on three major components: the class loader, the bytecode verifier, and the security manager. Each component plays a key role in maintaining the integrity of the system, assuring that only the correct classes are loaded, that the classes are in the correct format, and that untrusted classes will neither execute dangerous instructions nor access protected system resources. Java's Protected Domains constitute an extension of the sandbox, and determine the domain and scope in which an applet can execute. Two different protected domains can interact only through trusted code, or by explicit consent of both parties.

The class loader forms the first line of defense in the Java security model by determining how and when applets can load classes, and is responsible for: fetching the applet's code from the remote machine, creating and enforcing a namespace hierarchy, and preventing applets from invoking methods that are part of the system's class loader. An executing Java environment permits multiple class loaders, each with its own namespace, to be simultaneously active. Namespaces allow the JVM to group classes based on where they originated (e.g., local or remote). This delineates and controls what other portions of the runtime environment the applet can access and modify. Java applications are free to create their own class loaders. In fact, the JDK provides a template for a class loader to facilitate customization. Before a class loader may permit a given applet to execute, its code must be checked by the bytecode verifier. The verifier insures that the applet's code, which may not have been generated by a Java compiler, adheres to all of the rules of the language. In fact, in order to do its job, the verifier assumes that all code is meant to crash or penetrate the system's security measures. Then, the verifier insures that the code conforms to language specifications and that the applet does not forge pointers, circumvent access restrictions, or access objects through an illegal cast. Using a bytecode verifier means that Java validates all untrusted code before permitting execution within a namespace. Thus, namespaces insure that one applet cannot affect the rest of the runtime environment, and code verification insures that an applet cannot violate its own namespace.
5.4.2 Security Managers

The security manager enforces the boundaries around the sandbox by implementing and imposing the security policy for applications. All classes in the Java packages cooperate by asking the security manager for permission to perform certain operations. SecurityManager is an abstract class of the java.lang API, and provides the programming interface and partial implementation for all Java security managers. By default, an application has no security manager, so all operations are allowed. But, if there is a security manager, all operations are disallowed by default. Existing browsers and applet viewers create their own security managers when starting up.

When there is a security manager, each operation or group of operations will have its own checkXXX method. There are checkXXX methods for operations on sockets, threads, files, networking, windows, etc. Generally, the application code does not use them, given that the Java runtime does it at a lower level. However, they are necessary when overriding some methods. To write a security manager, it is necessary to create a subclass of SecurityManager and override most or all of its methods: class MyPolicy extends SecurityManager { ... } . Once a new security manager is created, it can be installed with the setSecurityManager method from the System class. The security manager will remain active until the end of the application. A method that opens a file for reading invokes the checkRead method of the security manager. A method that opens a file for writing invokes the checkWrite method. If the security manager approves the operation, the checkXXX method returns, otherwise, it throws a SecurityException.

5.4.3 Advanced Security Features

The Java Security API [76], which is part of the java.security package [77], has classes that support digital signatures, message digests, key management, and access control lists. Users can rely on the default package, or purchase and/or create one of their own. Thus, an organization can select a security level commensurate with their risk tolerance. The remainder of this section reviews the capabilities of the Java Security API.

Digital Signatures and JAR files

If a particular publisher is trusted, and a signed applet from that publisher has arrived over the Internet and been authenticated, then the Java Security Manager could allow that applet out of the sandbox, and treat it as an application. The first task of any security system is to be able to assure that who or whatever is on the other side of a connection is who or what the user expected to be there, i.e., the host that they have connected to is the host they contacted and not an impostor, or the module that they have loaded is really the one they expected to run and not a substitute. This is of particular concern in downloaded environments where there is a constant threat of a Trojan Horse.

The man-in-the-middle or middleman is a type of attack to which all network-based systems might be vulnerable, and proceeds in a number of steps. First, a client application requests some service from a legitimate server. Unknown to both client and server, an attacker application observes this request and waits for the server to respond. When it does, the attacker intercepts the server's response and replaces it with one of its own, one that the client may assume came from the original server. The way to prevent this type of attack is to ship code contained within a digital shrink-wrap. Java environments can achieve this results using signed applets. To do this, a supplier bundles Java code (and any related files) into a JAR (a Java Archive). The supplier then signs the file with a digital signature. The client can verify the authenticity of the supplier
by verifying the signature.

JavaKey [78] is the SUN security provider command-line tool whose primary use is to generate digital signatures for archive files. In order to generate a signature for a particular file, the signer must first have a public/private key pair associated with the file, along with one or more certificates that authenticates its public key. JAR files also provide increased performance. Current Web technology transmits files serially; using JAR files, a Web page can download everything it needs in a single request.

**Key Management**

The Java Security API provides support for integrated key management in Java programs and applets. All keys have three characteristics:

**An Algorithm:** The key algorithm is usually an encryption or asymmetric operation algorithm (such as DSA or RSA) which will work with those algorithms and with related algorithms (such as MD5 with RSA, SHA-1 with RSA, Raw RSA, etc.).

**An Encoded Form:** This is an external encoded form for the key that is used when transmitting it to some other party. The key is encoded according to a standard format (such as X.509 or PKCS#8).

**A Format:** This is the name of the format of the encoded key.

Keys are generally obtained through key generators, certificates, or the various Identity classes used to manage keys. There are no provisions yet for the parsing of encoded keys and certificates. An identity certificate is a guarantee by a principal that a public key is that of another principal. The KeyPairGenerator class is used to generate pairs of public and private keys. Key generation is an area that sometimes does not lend itself well to algorithm independence. There are therefore two ways to generate a key pair: in an algorithm-independent manner and in an algorithm-specific manner.

**Message Digests**

Using public key encryption algorithms to encrypt messages can be quite slow. As a result, cryptographers have developed a way to generate a short, unique representation of your message, called a message digest, that can be encrypted and then used as your digital signature. The MessageDigest class provides the functionality of a message digest algorithm, such as MD5 or SHA. Message digests are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value. Like other algorithm-based classes in the Java Security API, MessageDigest has two major components: the Message Digest API, or the methods called by applications needing message digest services, and the Message Digest SPI (Service Provider Interface), or the interface implemented by providers that supply specific algorithms.

**Access Control Lists**

Every authenticated principal will have a level of accessibility: highly trusted resources should be granted more access than those of more dubious origin. Access Control Lists (ACL) are data structures used to guard access to resources, and allow users to define read/write permissions based on users and groups. ACLs simplify the effort required for programs to maintain and access these lists. An ACL can be thought of as a data structure with multiple ACL entries. Each ACL
entry contains a set of positive or negative permissions associated with a particular principal (an individual user or a group). Individual permissions (either positive or negative) override the groups’ permissions. The java.security.acl package provides the interfaces to the ACL and related data structures (ACL entries, groups, permissions, etc.), and the sun.security.acl classes provide a default implementation of the interfaces.

5.4.4 Discretionary Access Control

Object-oriented design and implementation techniques rely on the public interface of a class to collect all of the permissible operations (or methods) needed by all potential users. Thus, methods placed in the public interface are available to all potential users regardless of their intended responsibilities within the application. Unless a security mechanism is designed and implemented, there is no way to prevent access by any user to a method in the public interface. For example, in a health care application, a method placed in the public interface to allow a Physician (via a GUI tool) to prescribe medication on a patient can't be explicitly hidden from a Nurse using the same GUI tool. Rather, the software engineer is responsible for insuring that such access does not occur, since the object-oriented programming language cannot inherently enforce the required security access. Moreover, in applications like health care, there must be a balance between the ability to access critical health data (such as patient drug allergies) with the ability to protect sensitive and personal information.

The current public interface provided by most object-oriented languages is the union of all privileges (methods) needed by all users of each class. This allows methods intended for only specific users to be available to all users. In a discretionary access control (DAC) approach, a customizable public interface is promoted, that appear differently at different times for specific users. There has been extensive work at The University of Connecticut in discretionary access control for object-oriented models [10, 23, 24]. Recent efforts at UConn have proposed extensible and reusable DAC enforcement mechanisms that utilize inheritance, generics, and exception handling for the automatic generation of code for DAC policies [11, 12, 13, 14] and for detailing client/server software architectures for DAC [15]. The goal of these efforts has been to minimize the amount of knowledge a software engineer must have on DAC by having mechanisms that are self-contained, class libraries, which supply all of the required code to define and enforce the desired security policy.

All of these DAC efforts, which used C++ as a basis to demonstrate the approach, are applicable to any object-oriented programming language that supports the required constructs. For example, the DAC approaches that utilize inheritance and exceptions are realizable in Java, while the ones that use generics are not. DAC for Java must be explored in order to provide a versatile capability to software engineers to insure that security policies are definable and enforceable. The Java security primitives, in particular the sandbox, security managers, and ACL, can be exploited to realize and support DAC.

5.5 Java and Remote Method Invocation

The remote method invocation (RMI) [94] API of Java is utilized for method invocations between client and server processes, when both are written in Java. Through RMI, it is possible to invoke methods on objects that are actually running under the control of a remote JVM. RMI is critical for DOC, as noted in Section 3.4.6. This has been recognized by others, namely: “This RMI facility, along with the CORBA IDL compiler libraries, make Java a very attractive platform for client/server applications.” [36, page 986]
The use of either RMI or CORBA allows access of objects over a network. Objects and object methods on a remote host can be accessed as though they resided on the local host. While IDL provides client/server access using the CORBA protocol, which allows access to any other server which uses the CORBA protocol, RMI provides a very robust and transparent interface to remote objects, but it is specific only to servers written in Java. "The RMI system also provides a naming service that allows servers to bind object references to URLs such as rmis://fohost.com/ObjectName. A client passes a URL to the naming class's lookup() method, which returns a reference to an object implementing the appropriate interface." [36, page 987]

The designers of Java intended to keep RMI simple to use. Calling a method on a remote machine has all the simplicity of a remote procedure call (RPC) in C++. In Java, a remote method call is syntactically the same as a local method call; the implementation of the object which determines whether the method call is local or remote. The client objects implement "stub" code, and the server objects implement "skeleton" code. By using stubs and skeletons, which are strictly defined, Java insures that RMI code remains easy to implement and platform independent [44]. RMI uses object serialization to convert objects to streams of characters before transmitting them over the network. In Section 5.6.4, when discussing persistence as supported in JavaBeans, more detail on object serialization is provided.

Perhaps one of the biggest boons to software engineers is that remote Java objects, just as local Java objects, are reclaimed automatically by the system's garbage collector. As C/C++ software engineers know, keeping track of dynamic objects can sometimes be rather tricky, and compounding the problem by making the objects remote would make the task daunting. That task is virtually eliminated by the garbage collector [44].

5.6 JavaBeans

The JavaBeans API [68] is a set of classes that enable software engineers to perform component-based design and development to facilitate and promote software reuse. A Java Bean is a customizable, extendible software component designed to be manipulated in a visual interactive development environment (VIDE). Although visual manipulation is their primary purpose, Beans can also be used without a VIDE, by directly writing programs "line-by-line". JavaSoft's ultimate goal for JavaBeans is to have reusable software components that are compact (for efficient Internet transmission), easy to create and use, easy to integrate into graphical application development packages, fully portable, and that support distributed computing [44]. By using Beans to design and develop a Java application, reuse is promoted, with Beans used as is or by extension.

While Java is completely object-oriented, and as such, the utilization of the APIs does substantially promote reuse, the purpose of the JavaBeans API is to provide a higher level API that embodies larger components, each of which represents functionality that is spread across multiple APIs. For example, one might combine a GUI component (dialog box) with a network component (URL) into a Bean that inputs the URL from the user and automatically opens a WWW browser using the URL. Once this component has been built using the separate APIs (awt and net), other software engineers can use this Bean without having to resort to the underlying APIs. "JavaBeans builds on the existing design of Java by specifying a rich set of mechanisms for interaction between objects, along with common actions that most objects must support, such as persistence and event handling." [36, page 1022]

Java Beans are similar to Microsoft Visual Basic's VBXs or OCXs. Java Beans can be simple components such as buttons and scroll bars or more complex components such as calendars and spreadsheets, even built up from other Java Beans. VIDEs for Java extend development
capabilities even further by allowing entire applications to be developed without writing any
code. While most Java Beans such as buttons and list boxes are intended to be visible at run-
time, they can also be objects which are not visible at run-time, such as timers or containers for
other objects. With the widespread development of Java Beans by third parties, it should soon
be possible for an expert in a particular field to have a rich enough set of Beans at their disposal
to be able to create a complex application that would have otherwise required collaboration with
one or more software engineers. One could argue that in the future, JavaBeans will facilitate
programming in a wide range of domains for non-software engineers. The experts in the domain,
not the software engineers, will create the software. However, the software engineers would still
be needed to write the domain-specific Beans.

The ease of integration of Java Beans in applications is facilitated by a set of features specific
to JavaBeans. These features are related to the actions that are taken to create and use Beans,
rather than involving Beans for specific domains. The JavaBeans API [69] provides support for:

- **introspection**, which allows a JavaBeans' VIDE to analyze the structure and operations of
  a bean,

- **customization**, which allows a user utilizing a JavaBeans' VIDE to tailor the appearance
  and behavior of a Bean to suit the needs of an application,

- **events** for a simple communication metaphor that can be used to connect Beans together,
  thereby building larger Beans with greater functionality,

- **properties**, which are characteristics of Beans, and are used for both customization and for
  programmatic use, and

- **persistence**, which allows new Beans that are built by customizing or combining existing
  Beans to be stored by the VIDE so that they can be used and reused in the future.

In the remainder of this section, some of these features are discussed in greater detail, to provide
insight into the use and function of the JavaBeans API. Note that properties are similar to
metadata, since they contain data about other data.

### 5.6.1 Introspection

In the `java.lang` API, the `Object` and `Class` classes have a substantial set of methods defined
that are accessible to software engineers for obtaining information about any system-defined or
user-defined class in Java. For instance, `Class` has methods that can be invoked to return, for
a specific user or system class, a list of its public methods, a list of its member variables, a list
of its declared constructors, and so on. This is **introspection**, and is critical for VIDEs for Java
Beans to provide the ability to supply this information to users automatically. Java Beans are
different from other Java classes, since they contain methods that follow specific rules involving
the definition and use of a Bean; e.g., getting and setting property values. In addition, there are
introspection functions that are specific to Beans and not available to other classes. These are
available in the `Introspector` class [73] of the `java.beans` API [69].

### 5.6.2 Properties

A Java Bean's *property* is a data member for which accessor and mutator functions are defined.
An accessor is a function which can read or get the value of the data member, and a mutator
is one which can change or set the value of the data member. For example, if there is a data
member called title, then for title to be an actual property, two functions must be created:
the accessor called get_Title and the mutator called set_Title. These functions must be defined
for each Bean so that the introspection activity needed by VIDEs can be supported. Functions
can also be created which cause other Beans to be notified automatically whenever the value of a
particular property changes. Notified Beans are also allowed either to permit or deny the change.

5.6.3 Customization

After the properties have been created for a Bean, property editing functions can be created
and attached to the Bean. These functions can be very simple or they can be very complex,
providing a significant amount of functionality to allow users to change the value of a property.
For example, to change the color of an object, it may be desirable to provide a complete color
wheel on the screen for the user. A property sheet, provided automatically by the VIDE, can
then be used to view and edit all of a Bean’s properties. The property sheet is essentially the
same as the property lists found in Microsoft’s Visual C++ or Visual Basic IDEs. When editing
the Bean, all of its properties will appear automatically in the property sheet. The user can then
select a property from the property sheet, and the property’s editing function is automatically
called.

5.6.4 Persistence and Serialization

Persistence is the saving and loading of objects from some persistent storage mechanism (e.g.,
a database or a raw disk file), and is achieved in Java through serialization. Serialization is the
conversion of an object into a string or some other data format that facilitates its storage to disk
or its transfers across a network. The conversion back to an object from a serialized format is
known as deserialization. Since Java already supplies methods for the serialization of intrinsic
types, it is left to the software engineer to develop methods to serialize and deserialize a custom
object; these methods are called respectively write and read. In most cases, the serialization of
an object will consist merely of calling the write methods on all of the object’s data members.
If one of the data members is itself a custom object, there must also be a write method for that
object as well.

Persistence is mostly applicable to Java applications, not applets. Applets are usually prohib-
ited from writing to the local host’s disk. A serialized object from an applet can be passed back
over the network to the remote host, but this is at a cost of increased complexity. The designers
of Java did not intend applets to be or to use persistent objects. Persistence and serialization
are critical for VIDEs, since Beans must be stored and available for later use (reuse). Persistence
involves the ability to store/retrieve a Bean, which in turn may be composed of other Beans.
Serialization converts the Bean into a form that is storable. All of the properties and charac-
teristics of a Bean must be converted and stored. If VIDEs are to support team-oriented design
and development, then database systems are critical to provide the underlying framework for
sharing, concurrency, storage, retrieval, and recovery of Beans. Without an underlying database
system, two software engineers may change the functionality of a Bean that is shared between
them, resulting in an inconsistent state when the “last” version of the Bean is saved.

60
5.6.5 Packaging

To facilitate the transmission and reception of Java class files, a new compressed-file format has been developed. Although the ZIP file format is very popular and widely supported, it is not universal. Thus the Java archive (JAR) format was created. "JAR files allow you to package classes, images, sounds, and other binary files, in a cross-platform manner." [44] An applet loaded into a web browser can be in the JAR format, and it will be extracted automatically by the browser. JAR files also support standard Java security measures such as certificates.

Since most VIDEs directly support JAR files, the sharing of Beans among developers is promoted. The JAR standard insures that, regardless of the platform being used, if a Bean or any other class file is downloaded from a remote site, it will be immediately usable at the local site. This may not seem to be a significant issue, but when one considers the wide abundance of formats for archived and/or compressed files (e.g., TAR, GZ, Z, ZIP, ZOO, SEA, etc.), one can understand the importance of a file format standard, especially for a language (Java) that is intended to provide true platform independence!

5.7 IDEs for Java

There are currently a wide variety of development environments which support the creation of Java applets and applications. The integrated development environment (IDE) is the simpler of two primary types, providing an environment in which to edit Java source code and compile and run the application. The visual integrated development environment (VIDE) is far more sophisticated and allows drag-and-drop creation of applications which have complex GUIs. Some environments tend to encompass both categories, such as Borland International’s JBuilder for visual Java development, that supports "...100% Pure Java, JavaBeans, JDK1.1, JFC, AFC, RMI, CORBA, JDBC, ODBC, and all major database servers, and will provide developers with a flexible open architecture that may incorporate third-party tools, add-ins, and JavaBean components." [61]

5.7.1 IDEs and VIDEs

There is currently a large number of GUI-based IDEs available for Java, created by both large, well-known companies, and small start-ups. While Microsoft’s Visual J++ is probably the most widely marketed, all the IDEs provide similar baseline capabilities. Figure 12 shows the IDE mode of Visual J++ (which is a typical IDE) being used to develop a Java application. On the left is a hierarchical view of the classes in the project, and the methods and variables in each class; on the right is the primary edit window where the class files are edited; and, on the bottom is a transcript window where the results of each compilation are shown. Many IDEs such as Visual J++ are also VIDEs and support visual application development.

VIDEs are typically used in three ways: to create applications consisting of one or more components, to create individual components, and to modify existing components. The standard "component" is the Java Bean, as discussed in Section 5.6. Although a new Java Bean can be created from existing Beans using a VIDE, creating a Java Bean which requires functionality that is not supplied by other Java Beans requires writing Java code manually.

Figure 13 shows Symantec’s Visual Cafe being used to develop a Java application. The display is divided into four areas, one across the top third of the screen, and three more aligned horizontally in the middle of the display. The topmost area is the primary Visual Cafe window. There is a toolbar with all of the "widgets" available to be dragged-and-dropped onto the application window, which is the middle window. The application window contains two objects: the small
square to the left is the object for the menu that the application will have and the square on the right is the object for the “file open” dialog box associated with the application. These objects are placed on the application window automatically when it is first created. The window to the left is the project window, consists of three windows:

1. **Frame1** is the application being created. Menu and file dialog objects can be seen listed under this item.

2. **AboutDialog** is the “about box” found on most windowed applications.

3. **QuitDialog** is the ubiquitous “are you sure you’d like to quit” dialog box.

Finally, the rightmost window on the display is the property list. Clicking on a property's value in the right column allows the value to be changed. The difference in complexity of different property editing functions should be noted. For instance, the font values can be changed individually in the property list by simply selecting a value from a drop-down list. Alternatively, clicking in the empty box to the right of the word “Font” causes a small square containing an ellipsis “...” to appear. Selecting the ellipsis in this case causes a complex font selection dialog box to be displayed.

### 5.7.2 Comparison of VIDEs

Three VIDEs were tested under Microsoft Windows 95, each having its own peculiarities: Penumbra Software’s Super Mojo, Microsoft’s Visual J++, and Symantec’s Visual Café. All three IDEs
Figure 13: Symantec’s Visual Café.

were tested on a 133 MHz Pentium-class PC with 16 MB RAM running Microsoft Windows 95. Penumbra Software’s Super Mojo is written entirely in Java, and should run directly on any other platform that supports Java, although this was not tested. Super Mojo had the poorest performance, occasionally requiring 5 and 10 seconds between the time a button was pressed or a menu item was selected before the appropriate action would be taken. Visual J++ and Visual Café performed much better, comparable to any other Windows 95 program executing native machine instructions, but these IDEs were compiled specifically for this platform.

Visual J++, as shown in Figure 12, is a very complex program which seems almost infinitely configurable, but sometimes it can be difficult to find the particular setting being sought. One of Visual J++’s drawbacks is that it uses its own libraries instead of using the standard Java Development Kit class files. In fact, Visual J++ did not include a particular class file, which is part of the standard Java Development Kit, needed for a program which was written just for testing purposes. This may be due to the early release version of J++, but no documentation could be found either to support or refute this. Nevertheless, since this is a Microsoft product running under Microsoft’s operating system, it is no surprise that it performed very well. Two caveats, though, are (1) it is unlikely that Visual J++ will ever run under an operating system other than Windows 95 or Windows NT, and (2) upgrades to the Java Development Kit must be obtained directly from Microsoft as a Visual J++ upgrade.

Unlike Visual J++, which starts up in IDE mode, Symantec’s Visual Café starts up in visual design mode (see Figure 13). Although Café is not as rich in features as Visual J++, it is intuitive to use. Café makes it very simple to drag-and-drop components onto an empty applet window and then connect them with just a click of the mouse, allowing useful applets to be created literally in
minutes. Visual Café can also be configured to use any Java compiler. This VIDE does not seem to have a mode which operates as a simple IDE, although it does allow the source code associated with each object to be modified directly. There was also a menu option in Visual Café which was labeled “Insert Component into Library”, but it was expressly disabled in the trial version. This option presumably allows one to insert a customized component such as a Java Bean into the standard set of components that this application presents to the user. Visual Café has already been ported to other platforms.

Penumbra Software’s Super Mojo, shown in Figure 14, is the most visually complex of the three, and starts up with three separate windows open: the “Mojo Designer” visual development window, the “Mojo Coder” source code editing window, and the “Mojo Visual Script” window, which allows the developer to associate “scripts” with components. Exactly what these scripts are and what they can do could not be determined, as the on-line help was not functioning in the beta version of the software that was tested. There was also a menu item titled “Wizards” which had no sub-items.

5.7.3 Available IDEs and VIDEs

Information on IDEs and VIDEs for Java are widely available on WWW. Most of the IDEs and VIDEs, even the commercial ones, are available as shareware or trial versions, so it is easy to try them all out. The following is a short list of some of the more popular IDEs:

- Microsoft Visual J++
  http://www.microsoft.com/visualj/
• Sun Java Workshop
  http://www.sun.com/software/Products/Developer-products/java/Workshop/

• Borland JBuilder
  http://www.borland.com/jbuilder/

• Asymetrix SuperCede
  http://www.supercede.com/

• Symantec Visual Café
  http://www.symantec.com/

• Parts for Java by ObjectShare
  http://www.objectshare.com/p4j/default.htm

• Powersoft PowerJ

• IBM Visual Age for Java
  http://www.software.ibm.com/ad/vajava/

• JavaSoft JDK tools
  http://www.javasoft.com

• StepSoft Javelin

• Super Mojo
  http://www.penumbrasoftware.com

A good comparative discussion of a number of different IDEs/VIDEs is also available online [62], including links to sites offering everything from freeware IDEs to commercial IDEs.

6 Summary and Recommendations

This section summarizes the material in this white paper, and provides a framework for assessing the Java language/environment and its potential impact. The discussion is divided into three parts. In Section 6.1, the pros and cons of Java are reviewed on an issue-by-issue basis, effectively providing a view of the present state of the language/environment, its major problems, and its greatest potential. Included in Section 6.1 is a rundown of Java against C, C++, Ada95, Smalltalk-80, and Visual Basic 5, based on numerous features of programming languages. In Section 6.2, future capabilities of the Java language and platforms are provided, including future APIs for Java, JavaOS, and dedicated micro-processors for Java. In Section 6.3, a summarization of Java recaps the pros and cons of Section 6.1 and provides a recommendation on Java into the 21st century.

6.1 Pros and Cons of Java

Clearly, from the discussion in Sections 1 to 5 of this white paper, Java is an evolving programming language/environment with substantial potential, targeted for a wide-range of diverse domains. But, Java remains immature, without a standard in place, that must evolve and stabilize before
it can take its position as the replacement language for C++ , Ada95 , Eiffel , Visual Basic , etc. Whether a business or government entity decides to commit resources to a technology (Java) that remains largely unproven and untested is not a topic that this white paper can directly address. What can be addressed are the positive features that champion Java (PROs), the significant drawbacks that illuminate Java's immaturity (CONs), and the UNKNOWN characteristics of Java that must be proven in the coming 1 to 5 years before Java can be considered a stable and successful technology. The remainder of this section presents fifteen issues that summarizes, in part, the material presented in Sections 1 to 5, identifying the PROs, CONs, and UNKNOWN facets of each issue. When necessary, the explanation and critique differentiates between the short run (within 2 years) and the long run (3 to 5 years) to frame the discussion. This section also contains a comparison/contrast of Java to C, C++, Ada95, Smalltalk-80, and Visual Basic, to clearly identify the different features that each language supports.

6.1.1 Unclear Standardization

The standardization of Java is an unsettled issue that has the potential to seriously impact the potential of the language. Many companies and government facilities will not accept a language unless it has the backing of a standards organization. Thus, for a language like Java that is still rapidly evolving, standardization becomes a paramount issue. As recently reported [19], Sun has submitted a request for a Java standard to the Joint Technical Committee (JTC) of the International Organization for Standardization and International Electrotechnical Commission. Sun is attempting to exert significant control over the Java standards process by maintaining control over which Java technologies go forward in the standards. Other companies object since Sun wants to retain copyrighting, distribution, and trademarks for Java. IEEE, HP, Apple, Compaq, TI, Lucent, Microsoft, and Intel all oppose Sun's proposal. In fact, JavaSoft's standards proposal for Java, pending before ISO, apparently had a "no vote with comments" by 18 of 27 ISO members (countries) [33]. Of the other nine countries, six voted yes with comments, while three abstained. JavaSoft must respond within sixty days, with a final decision by ISO slated for the November 1997 timeframe. The unstable situation of standardization for Java is a CON of the language in the present, and remains an UNKNOWN in the long run.

6.1.2 Lack of Generics

The Java language does not support generics, which is a key software engineering concept for promoting and achieving reuse. The interfaces feature (see Sections 2.2 and 2.4) of Java only supports design-level generics. In Java, a single interface can be written to represent a shared set of actions that are intended for use by multiple classes. For example, the Runnable interface for threads (see Section 5.2.6) is intended to allow objects the ability to run their own threads, and could be implemented when a class in an application needs this ability. But since interfaces do not have implementations, the sharing is only in the specification and not in the code that is required. If a software engineer uses the same interface X times, s/he must write X pieces of code to implement the X versions of the interface. The commonality of the reused interface is a plus, but this is more than offset by the requirement to reimplement similar code over and over again. Thus, the interface doesn't offer a gain in productivity, but only provides a consistent and common means to share design concepts. While this has the potential to enhance productivity, such a gain is difficult to measure. The lack of generics in Java is considered a CON from a software-engineering practices perspective since it is so limited, and a CON from a software reuse/productivity perspective.

66
6.1.3 Microsoft Infusion into Apple

In mid-August 1997, Microsoft announced that it was infusing $150 million into Apple. One of the side agreements involved the two companies cooperating on Microsoft's Java, as reported [118], which may be an effort to capture the Java market and wrest control of Java from Sun. This may have the side effect of splintering Java into two camps - Sun vs. Microsoft. Such a split has a number of potential, negative repercussions, including: hampering of standardization, producing incompatible versions of Java, and frustrating the user community. The result may be an adverse impact on the acceptability and use of Java by business and government entities. To counter Microsoft's move, IBM, Sun, and Netscape have announced an alliance [3] to cooperate on the Java language/environment, to insure that Java runs smoothly on all platforms, including Windows 95. Depending on one's perspective, this issue has the potential to be a PRO by forcing Sun to give ground on the standards issue or a CON by splitting Java into two competing camps. The evolving nature of the Microsoft/Apple and the IBM/Sun/Netscape partnerships means that this is an UNKNOWN issue to be carefully monitored since it may impact on Java.

6.1.4 Security Flaws and Java

Over the past few years, there have been announcements in the mainstream press whenever a major security flaw has been found in one of the popular browsers (Netscape Navigator and Internet Explorer) or on-line services. In most situations, the problems have not been flaws in Java per se, but there is the potential that a flaw in a browser (or other software application) may result in an applet (Java code) executing and causing damage in some way. For example, the recent bug in Internet Explorer, according to a published report [119], "allows a Java applet to open a network connection to a server other than the one from which it originated, enabling a hostile user to remotely steal image files from a site, according to News.com. Such a connection is not allowed under the Java security model, which has led the discoverer of the flaw to believe that the problem rests in Microsoft's browsers and not in Java itself." Regardless of where the problem lies, the fact that other software can cause a Java applet to perform a non- permissible operation must be taken into consideration, especially for entities that maintain and use sensitive and confidential data. Java across the Internet is powerful, but there must be assurance related to security. This issue is classified as a CON, since it must clearly be watched with browsers in such widespread use.

6.1.5 Complexity of APIs

The Java language as presented in Sections 1.2 and 2 can be viewed as quite simple, from programming language and software engineering perspectives. But simple does not equate with easy from a software design and development perspective. While the language itself is simple, the APIs are quite complex and require a sophisticated software engineer with significant amount of expertise, working in a dedicated fashion in order to successfully utilize the APIs. For example, both the threading (Section 5.2) and networking (Section 5.3) APIs are complex. The threading capability in Java is an example of an API that is part of the syntax/semantics of another programming language (Ada95). This makes the Java language simpler by placing the complexity of threading into a separate package. However, one can argue that the result is that threads are harder to use in Java than in Ada95, where explicit tasking primitives clearly outline and restrict actions to achieve the desired functionality in a program. Thus, there is a tradeoff between a simpler language (Java) vs. language constructs that are easier to use (Ada95).
But, in order to fully and successfully utilize Java, a software engineer must embrace and learn multiple APIs. Some APIs are easy to learn and use, like java.awt for GUIs. Others, like the Java Native Interface (JNI - Section 3.3) require significant time and effort. One reference involving JNI [16] clearly illustrates that native OS access to and from Java is a complex task that requires a great deal of skill. One must be careful and not be fooled by the simplicity of the Java language. It may be easy and quick to learn the basics of Java (see Sections 6.1.7 and 6.1.8), but this may be more than offset by the complexity involved when using advanced features for threading, networking, IDL, etc. Whether it is more efficient to use these advanced features in Java vs. another object-oriented programming language is an unresolved issue. However, it is the nature of distributed computing and programming across the Internet to be difficult, time-consuming, and require significant, dedicated effort. This is a by-product of platform independence! Platform independence in Java does not equate to easy-to-develop software in Java. The simplicity of the Java language is a PRO, which one can argue is offset by the complexity of the APIs, a CON, from a usability perspective.

### 6.1.6 Performance, Portability, Look-and-Feel

Since its inception, Java has suffered from a poor-performance reputation. A Java application may run significantly different (and perhaps slower) on different platforms. There have been many approaches that attempt to solve this problem. One solution approach has been a customization of the JVM by a third party. For example, Kaffe [115] is a just-in-time compiler and interpreting JVM for Java bytecode that is supposed to provide a performance (runtime) improvement over Sun's JVM. Another solution approach has been a major company producing its own version of Java. For example, Microsoft's version of Java runs better on the Windows 95 platform than Sun's corresponding version. This is to be expected since Microsoft, as the developer of Windows 95, can exploit its greater knowledge of the OS in developing its own JVM. A third solution approach being promoted by Sun involves its upcoming 1.2 release of Java, which includes the Hotspot JVM and just-in-time compilation.

Performance problems in Java are related to portability concerns of the language, both in the ability to fully port the code across platforms and the look-and-feel of the same code on different platforms. Portability is a definite problem in Java and has many causes [33]:

- Changes to the Java runtime environment resulting in interoperability problems.
- GUI differences across platforms resulting in look-and-feel differences.
- Tuning of the JVM for performance yields uneven application behavior across platforms.

Sun claims that these problems will all be fixed in Java 1.2, but at this time, clearly it is not a resolved issue. When using Java, it is crucial to realize that true 100% portability is not attainable in practice. From the discussion in Section 3, programming interactions (via JNI) and integration with legacy/COTS applications have a strong potential to introduce platform specificity, thereby eliminating 100% portability. Performance, portability, and look-and-feel differences are all CONS for Java. If one considers the alternatives (or lack thereof) for platform-independent software design and development, perhaps all three are very tolerable. However, this is a significant UNKNOWN issue in the long run, since it is unclear whether true platform independence at reasonable performance can be achieved in practice.
6.1.7 Similarity to C/C++/OO

As indicated in Sections 1.2 and 2, the (non-)object-oriented capabilities of Java are very similar to C++, resulting in a short learning curve for the core language if one knows C++. Most of the statements, operators, precedence, variable declarations, and so on, are either identical or similar to C/C++. Most of the object-oriented features including class, inheritance, and exceptions, have strong syntactic and semantic ties to C++. At the University of Connecticut, Java has been utilized for two offerings of a team-oriented design/development course, and all 20 juniors and seniors have picked up the language quickly with only a one-semester background in software engineering, object-oriented design, and C++. Many of the object-oriented features of Java are superior to C++, particularly from a software-engineering-practices perspective. For example, packages and the ability to more carefully control access to classes both within and external to packages (see Sections 2.1 and 2.2) are definitely superior to C++, which doesn’t even have packages. However, while the language can be learned quickly, as noted in Section 6.1.5, learning the APIs take time, especially the complex APIs for threading, networking, RMI, JNI, JDBC, etc. In that same course mentioned above, while the students can do GUIs and remote database access via URLs in their semester timeframe, using threading and other advanced features has not been successfully accomplished. The similarity of Java to C++ which promotes ease of learning, the superior object-oriented capabilities of Java vs. C++, and the comparable object-oriented abilities vs. Ada95, are all PROS.

6.1.8 Teaching Perspective

As object-oriented design and programming has become dominant in the past 10 years, the requirement to teach the concepts to working software engineers is a paramount concern. From a programming-languages perspective, Smalltalk is superior on object-oriented concepts, but its lack of popularity would make it a poor choice in a continuing education/retraining situation. While Ada might have been the choice in the late 1980s, its prominence has drastically diminished. C++ has often been the language of choice, but since C++ has acquired most of the “problems” of C, from a software-engineering-practices perspective, it is a poor alternative. The emergence of Java has a number of points in its favor as a teaching language:

1. Platform independence means that an instructor doesn’t have to be concerned about hardware/OS differences, especially for the examples that are used to teach object-oriented design and programming.

2. Availability of free compilers on different hardware/OS configurations simplifies laboratory assignments and makes it easy for a student to use the platform that s/he is most familiar with.

3. Java is easy to learn if students know C++.

4. Java enforces the software-engineering principles of abstraction and modularization (i.e., class and package) and forces software engineers to clearly identify all resources that must be imported in for use within a class/package.

5. Java enforces and provides various degrees of hiding that demonstrate how members (variables/methods) can be shared between classes of the same package while invisible outside the package.
6. Java provides the means via APIs for extensibility to a wide range of programming domains. Hence, an instructor can teach software engineers object-oriented design and programming regardless of whether they will use the language for applications in business, engineering, embedded systems, GUIs, databases, etc.

At the University of Connecticut, while we are using Java in project courses, it is not the language of choice yet for freshman or core software engineering courses. As Java stabilizes and matures, we expect to transition the curriculum appropriately. The teaching perspective of Java is a PRO of the language.

6.1.9 Breadth of Domains

As indicated in the previous subsection, Java can be used by software engineers in a wider variety of diverse domains than other available programming languages. Ada was designed to be a broad language, with real time and embedded features to tackle a wider range of programs/domains than was available in a programming language prior to 1983. But while Ada95 added object-oriented features, its ability to support Internet based, client-server design/development is lacking, which is to be expected for a language specification that dates back to the 1980s. On the other hand Java, with its rich set of current and planned APIs, is targeting a much larger domain of potential applications. The Enterprise APIs, the ability to use JVM in consumer products (planned microprocessors from Sun), Embedded API, Commerce API, Management API, Media API, Personal API, etc., all support the ability to use Java in multiple domains, which is needed for wider acceptance. C++, which can also be used in multiple domains, requires software engineers to write significant amounts of detailed code to serve as an interface to the target domain. While C++ is an alternative for any domain, its lack of platform independence is a major drawback. The breadth of domains is a PRO for Java.

6.1.10 Breadth of Language

The breadth of domain is achieved by the breadth of the Java language, especially as related to the package concept. With packages in Java, it is possible to provide significant new functionality to support an emerging domain or market segment. Ada95 also has this capability, but its limited use in the market is a hindrance. The APIs listed by Sun [65] provide a rich set of current and future tools for software engineers. One example of a planned API is for Java Card, that supports the smart card technology. A smart card is a credit size card that contains a microprocessor that could have multiple uses, e.g., connection to networks and telecommunication services, storing of health care or credit history data, etc. The potential for Java to be utilized in such a domain demonstrates the versatility of the language. Clearly, the breadth of Java from an API perspective is a PRO. Note that future APIs are discussed in Section 6.2.

6.1.11 Programming Language Interactions

As discussed in Section 3.3, there are many different ways to integrate different programming languages into a single application. Programming-language-to-Java translators, while attractive, in practice have difficulty in achieving 100% or total translation. PERC (COBOL-to-Java) and C2J++ (C++-to-Java) both have the asset of converting code to Java with the potential drawback that the automatic conversion process may be lacking in completeness. Environments that compile other programming languages directly to bytecode (AppletMagic and Kawa) suffer from two problems: completeness, and the utilization of Java libraries within the source code (in Ada95,
Scheme, respectively), which makes the source code non-compatible with other compilers for the language. For translators, if there is a need for intervention by a software engineer, such tweaking of code has a significant impact. At a system level, the Java Native Interface (JNI) provides the means for Java programs to call programs written in other languages, and vice versa, with the caveat that platform independence is lost. While JNI is a useful technology to allowing programs written in different languages to interact, it requires precision and substantial knowledge by the software engineer to utilize effectively.

From a translation perspective, this issue is a CON for Java if the translator in use isn't 100%. However, this issue has the potential to be a PRO when the translator is 100%, or the bytecode generator works for all possible source programs. The unclear market demand for translators to Java when coupled with the incompleteness of translation means that this issue is UNKNOWN in the long run. From a programming-language-interaction perspective via the JNI, this issue is a PRO, since it provides a framework that can be utilized by software engineers.

6.1.12 Client/Server and DOC

Distributed object computing (DOC) via client/server applications has evolved to be a norm for many domains. Java's Enterprise API [72], which consists of JDBC, RMI, Java IDL, JNDI, and JNI, holds much potential for supporting the integration of databases, legacy applications, COTS systems, and newly developed Java clients/servers that bind everything together. In Section 3.4, the various integration strategies and design alternatives for legacy applications and current/future COTS was detailed. Many of these approaches rely on CORBA in addition to the Java Enterprise API. As noted in Section 3.2, there are many products (VisiBroker, Joe, OrbixWeb, etc.) available that involve Java and CORBA, with more expected. If all of the promises of Java and CORBA work as advertised in the coming years, then this has the potential to be one of the major PROs of Java with respect to legacy applications and current COTS systems. Additionally, "future" COTS that are written in Java, will facilitate integration with new Java software in a platform-independent manner, opening up new potential markets (see Section 3.4.5). This strong "promise" is accompanied by an even stronger UNKNOWN. One must realize that as a technology, CORBA is still in its infancy, and has neither stabilized nor come into widely accepted use. Java for DOC/CORBA is combining a very immature technology (Java) with one that has yet to fully stabilize (CORBA), which is always a dangerous proposition. It will likely be 2 to 5 years before a clear and precise understanding of Java and its relationship to DOC/CORBA emerges.

6.1.13 Database and Persistence

Persistence is achievable in Java via three different approaches, as discussed in Section 4: object serialization (see Section 4.4.1), JDBC for integration with relational DBMSs (see Sections 4.1, 4.2, 4.4.2, and 4.4.3), and emerging Java integration with object-oriented DBMSs (see Section 4.3 and 4.4.4). All three approaches are PROs for Java. Object serialization, provided by the Java language, is a elegant technique for storing, transmitting, and recovering instances that have been converted into streams of characters (or other data format), that is useful for Java applications that do not require database management. Integration with relational and object-oriented DBMS is important, both for developing new Java clients that integrate with existing databases, and for supporting wrapper and ORB approaches which are needed for integration with legacy/COTS applications. This issue also represents a significant UNKNOWN for two reasons. First, the JDBC API has not been totally completed, and there are many planned and
proposed features that must be evaluated in the short term. Second, Java's interactions with object-oriented DBMSs is in its infancy, with products and technologies just beginning to emerge from vendors.

6.1.14 JavaBeans

JavaBeans (Section 5.6) and its support for component-based, plug-and-play design and development, also is a PRO of Java. Visual development has the potential to allow seasoned software engineers and/or domain experts (without computer science & engineering background) to work cooperatively or independently to design major applications on a component basis. To illustrate the non-software engineers potential of JavaBeans, consider the recent announcement [70] by IBM [71] on its new product: "IBM today announced at Summer Internet World the release of the first products to emerge from its Java frameworks project known as San Francisco. The San Francisco products, consisting of the Foundation and Utilities Layer, Common Business Objects and General Ledger, are scheduled for first shipment to customers on August 15, according to an IBM statement. The products offer developers pre-tested, reusable 100% Pure Java components for creating applications for business processes."

JavaBeans provides the tools and capabilities to support the development of such applications. Rather than building applications from scratch, the JavaBeans component-based approach promotes reuse by facilitating the combination of existing components (Beans) into new applications. This is an important facet of Java, but it is still an emerging capability. As the Java language continues to evolve and stabilize, from a standardization perspective (Section 6.1.1) and by the addition of new capabilities (Section 6.2), the role of JavaBeans will continue to increase. Thus, at some level, JavaBeans is an UNKNOWN, with much promise, but with its widespread acceptance and use unclear.

6.1.15 Interactive Development Environments (IDEs)

IDEs, as reviewed in Section 5.7, are a PRO of Java, both today and in the future. More than any other language, the proliferation of commercial and shareware IDEs for Java is exploding. IDEs for JavaBeans component-based design and development offers powerful capabilities that can contribute to the popularity and success of a programming language. The list of IDEs given in Section 5.7.3 is impressive, especially for a language (Java) that is less than two years old. Everything from standard to visual IDEs are available, with minimal or no cost. This lack of cost is a definite incentive for colleges and universities, who often switch to a language based on cost, availability of tools, and the perceived future potential. Lack of cost is also important to government and businesses; it is arguable that Ada83's lack of popularity today is more a result of the high cost of validated compilers ($50,000 or more in early 1980s) and not its capabilities as a programming language. IDEs definitely can play a major role in popularizing a programming language, and Java has a strong and growing advantage in this area.

6.1.16 Comparison to Other Languages

In Section 2, when discussing the object-oriented features of Java, when relevant, comparisons were made to C++ and/or Ada95. In Section 2.5, a detailed comparison to Smalltalk was provided. A discussion of the PROs and CONs would not be complete if it did not comprehensively examine the different ways that Java measured up against other programming languages. In this
subsection, Java, C, C++, Ada95, Smalltalk-80 (ST-80), and Visual Basic 5 (VB5) are rated
from four different perspectives:

- Programming Capabilities: Environment and compilation features, along with available
  programming development tools and capabilities.

- Object-Oriented Features: Encapsulation, hiding, inheritance, generics, etc., as reviewed in
  Section 2.

- Programming Constructs: Detailed constructs such as support for exceptions, inline func-
  tions, parameter passing methods, etc.

- Critical Support Issues: Ancillary issues such as compiler/runtime performance, IDEs, avail-
  able code, etc.

Tables 3, 4, 5, and 6, review the six programming languages based on these four perspectives,
respectively.

In Tables 3, 4, and 5, the following terms are utilized to rate the degree to which each
programming language attains the particular capability:

- YES: Directly supported by the language.
- MAYBE: Possibly or indirectly supported by the language.
- NO: Very difficult or not supported by the language.

If the support of a concept/capability by a language is unclear, then ‘????’ is placed in the table
entry.

Table 3 contains a rating of the languages based on their programming capabilities. Criteria
used for the rating include:

Supplies Run-Time Environment: Is a run-time environment provided as a standard part
of a typical installation? This criterion indicates that the system is either interpreted or
compiled into an intermediate code.

GUI Development Environment: Is a GUI-based IDE provided? This criterion similar to
interface builders for Java such as Swingset and GWT as discussed in Section 5.1.2.

Internet Programming: Can the language be used for Internet programming? That is, does
the language directly support the development of applications that are Internet based?
While C supports this criterion, there are no explicit primitives in the C language that
facilitate this in an analogous fashion to Java’s APIs.

GUI Programming: Can GUIs be created easily with the language? Are their packages,
libraries, or tools that facilitate the construction of GUIs for applications? This criterion is
definitely subjective, since compilers like Borland C++ do provide tools for GUIs.

Standard Libraries: This criterion is the degree to which standard libraries are provided with
the language—useful libraries, classes, or objects which may be found with each standard
implementation.

Automatic Documentation: Does the language have an automatic documentation feature?
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Java</th>
<th>C</th>
<th>C++</th>
<th>Ada95</th>
<th>ST-80</th>
<th>VB5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supplies Run-Time Environ.</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>GUI Development Environ.</td>
<td>YES</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Security</td>
<td>YES</td>
<td>NO</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>MAYBE</td>
</tr>
<tr>
<td>Internet Programming</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>MAYBE</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>GUI Programming</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Drag &amp; Drop Programming</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Standard Libraries</td>
<td>YES</td>
<td>NO</td>
<td>MAYBE</td>
<td>YES</td>
<td>MAYBE</td>
<td>YES</td>
</tr>
<tr>
<td>Automatic Documentation</td>
<td>YES</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>NO</td>
</tr>
<tr>
<td>Open System</td>
<td>YES</td>
<td>NO</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>MAYBE</td>
</tr>
</tbody>
</table>

Table 3. Programming Capabilities of Languages.

In addition to these criteria, the rating also considers support for security (Section 5.4), drag & drop programming (JavaBeans, Section 5.6), and whether open system is promoted. Overall, Java appears to have an advantage, but one must be careful in interpreting the table. Clearly, Java, as the newest language of the set, has been targeted specifically for all of these criteria, so perhaps the comparison in this case is somewhat biased.

Table 4 evaluates the six languages based on their support of the key object-oriented features, which were discussed in detail in Section 2, and include the criteria: encapsulation, hiding, (multiple) inheritance, generics, simple object integration, and overloading. Simple object integration refers to the ease of integrating an class/object from a third-party developer into an application. As indicated in earlier portions of the white paper, Java's lack of generics and multiple inheritance are two of its shortcomings. Also, unlike C++ and Ada95, operators (like equality, addition, multiplication, etc.) cannot be overloaded in Java; overloading in Java is limited to constructors and methods.

In Table 5, the six languages are rated on the basis of the various constructs that are often found across all programming languages. While most of the criteria are self-explanatory, a number require additional explanation:

**Low-level Programming:** Does the language support low-level access to the hardware?
Arbitrary Pointers: These are the canonical pointer variables as one would find in C.

Function Variables: Is it possible to store a function in a variable?

Variable Aliasing: This is the same as variable references.

Remote Linking: Linking to objects on remote hosts.

Overall, Java is comparable, with perhaps a slight edge, over the other languages when rated based on programming language constructs. As mentioned earlier, Java's advantages are more likely traced to the fact that it is a new language; if one was designing a new language, features from other languages and new features would be included to make it as versatile and attractive as possible.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Java</th>
<th>C</th>
<th>C++</th>
<th>Ada95</th>
<th>ST-80</th>
<th>VB5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Records</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Inline Functions</td>
<td>MAYBE</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>MAYBE</td>
</tr>
<tr>
<td>Exceptions</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>MAYBE</td>
</tr>
<tr>
<td>Low-Level Programming</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>MAYBE</td>
<td>NO</td>
<td>MAYBE</td>
</tr>
<tr>
<td>Arbitrary Pointers</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Function Variables</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>MAYBE</td>
<td>NO</td>
</tr>
<tr>
<td>Variable Aliasing</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Pass-by-Value</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Pass-by-Reference</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>???</td>
<td>YES</td>
</tr>
<tr>
<td>Dynamic Linking</td>
<td>YES</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Remote Linking</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>MAYBE</td>
</tr>
<tr>
<td>Concurrent Threads</td>
<td>YES</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Inter-Process Communic.</td>
<td>YES</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>YES</td>
<td>YES</td>
<td>MAYBE</td>
</tr>
<tr>
<td>Dynamic Memory Allocation</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Explicit Deallocation</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Auto. Garbage Collection</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Enumerated Types</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Array Bounds Checking</td>
<td>YES</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 5. Programming Language Constructs.

In Table 6, a comparison of the six languages is given based on a number of critical support issues, that while not as quantifiable as some of the earlier ratings, are nevertheless, important, when attempting to understand the tradeoffs of a language. In Table 6, the terms GOOD, FAIR, and POOR are utilized to indicate the degree to which the language supports the specific capability. Criteria that require explanation from Table 6 include:

Compiler Performance: Even if a system is interpreted, sometimes it is possible to compile a program into a standalone executable file, e.g., this is supported in Visual Basic. This criterion indicates the relative performance of the compilers for each language.
Amount of Provided Code: The amount of code, classes, libraries, and/or objects are provided in a standard implementation.

Amount of Available Code: The amount of code that is available for download or purchase for the language. This criterion may be more subjective than the other ones, and represents, for example, that Smalltalk code is more difficult to find than Java or C/C++ code.

Platform Availability: The number of platforms for which the language is available.

High-level: This criterion represents the amount of results that can be accomplished with a small amount of code.

Other criteria such as maturity and textbook availability are also somewhat subjective. Overall, Java is comparable to C, C++, and Ada95, with an edge over Smalltalk-80 and Visual Basic 5. Java does currently suffer from compilation and runtime performance, particularly when the same code executes at significantly different rates on different platforms.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Java</th>
<th>C</th>
<th>C++</th>
<th>Ada95</th>
<th>ST-80</th>
<th>VB5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler Performance</td>
<td>POOR</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>???</td>
<td>GOOD</td>
</tr>
<tr>
<td>Runtime Performance</td>
<td>POOR</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>FAIR</td>
<td>FAIR</td>
</tr>
<tr>
<td>Amount of Provided Code</td>
<td>GOOD</td>
<td>POOR</td>
<td>FAIR</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
</tr>
<tr>
<td>Amount of Available Code</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>FAIR</td>
<td>POOR</td>
<td>GOOD</td>
</tr>
<tr>
<td>Platform Availability</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>POOR</td>
<td>POOR</td>
<td>POOR</td>
</tr>
<tr>
<td>Third Party IDEs</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>POOR</td>
<td>POOR</td>
</tr>
<tr>
<td>Source Code Portability</td>
<td>GOOD</td>
<td>POOR</td>
<td>POOR</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
</tr>
<tr>
<td>Cost</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>FAIR</td>
<td>POOR</td>
<td>POOR</td>
</tr>
<tr>
<td>Maturity</td>
<td>POOR</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>FAIR</td>
</tr>
<tr>
<td>Textbook Availability</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
<td>POOR</td>
<td>GOOD</td>
<td>GOOD</td>
</tr>
<tr>
<td>High-Level</td>
<td>GOOD</td>
<td>POOR</td>
<td>FAIR</td>
<td>GOOD</td>
<td>GOOD</td>
<td>GOOD</td>
</tr>
</tbody>
</table>

Table 6. Critical Support Issues of Languages.

6.2 Future of the Language

The future of Java is greatly embodied in its planned APIs and true attainment of platform independence. There are many APIs that are in the planning stages, with specifications for their functionality released. A select set are described below:

• **Personal Java** [107]: This API is intended for small scale applications in the personal, commercial, consumer market. Sample networked applications include smart phones, remote controls, touch screens, etc. There will be strict limitations on memory for the runtime environment of application and code (2M ROM and 1-2 RAM).

• **Embedded Java** [106]: This API is intended for embedded computing applications, such as mobile phones, process control, network routers, etc., that often have real-time constraints. Embedded applications are much larger than those envisioned for PersonalJava, and often require resources of a PC, Unix workstation, or more. Embedded computing also requires a different set of development and testing tools.
• Java Card [105]: This API supports the smart card technology, which is essentially a computer the size of a credit card. The potential applications for smart cards are numerous, including storage of all personal data, health care data for emergency use, and so on.

These three APIs truly demonstrate the breadth and potential of Java for support many different domains.

From a hardware/OS platform perspective, Sun and JavaSoft are investing a great deal in the future potential of Java. To provide a better framework for the execution of Java applications, JavaSoft is planning an entire OS customized for Java, called JavaOS [108], which is intended to allow the execution of Java applications on microprocessors in a variety of consumer applications. Specifically, “JavaSoft has made mention of JavaOS being able to run with as little as 512K of ROM and 256K of RAM in an embedded environment. Likewise, an entire JavaOS system running on a networked computer requires only 3M of ROM and 4M of RAM. These last figures include space for JavaOS, the HotJava Web browser, and a cache for downloading Web content and applets.” [36, page 1029] To complement JavaOS at the hardware level, Sun is planning on designing and producing a set of microprocessors in three distinct classes: picoJava, microJava, and ultraJava. The picoJava microprocessor is the core specification from which all other microprocessors will be produced, and will be released as a licensable standard for other chip manufacturers. To target various aspects of the consumer market, microJava will be produced which “...builds application-specific I/O, memory, communications, and control functions onto the picoJAVA core.” [36, page 1031] The target cost for microJava processors will be in the range of $25 to $100. To address larger needs, the ultraJava microprocessor “...is designed to encompass the very fastest Java processors available. [It] includes support for advanced graphics [and] is primarily targeting high-end 3D graphics and multimedia applications.” [36, page 1031]

6.3 Recommendations

Table 7 summarizes the sixteen PROs, nine CONs, and eight UNKNOWNs of Java, for the fifteen issues reviewed in Section 6.1. The PROs include: the Microsoft/Apple partnership which may force Sun to change its stance related to standards; the ease and simplicity of the Java language (non-APIs); the similarity to C++ and strong support of OO for Java, which is excellent for teaching to software engineers; the breadth of using Java as embodied in the APIs and across multiple, varied domains; the huge potential of Java for integration with programming languages, CORBA, and legacy/COTS applications; programming-language translators and bytecode generators that ease the transition to Java; the Java Native Interface for programming-language interactions at the OS level; database support via object serialization and Java's integration with relational and object-oriented DBMS; and, the future potential of IDEs, with and without JavaBeans, for supporting design and development. The PROs of Java are very strong, and provide a solid core of capabilities and features for a programming language to be useful in a large number of situations and across a wide-range of domains.

The CONs are framed as: Sun's position regarding standardization; the lack of generics in Java which severely limits software reuse; the Microsoft/Apple partnership which may split Java into two competing and perhaps incompatible factions; the complexity for software engineers to learn and use advanced APIs; the varying performance of the “same” code across platforms; look-and-feel differences and other idiosyncrasies that impact on portability; and, programming-language translators that are not total in their mappings to Java. More troubling, are the great number of UNKNOWNS, related to: the unresolved status of standardization; the long-term impact of the Microsoft/Apple partnership; whether true 100% platform independence can be
<table>
<thead>
<tr>
<th>Issue/Subissue</th>
<th>PRO</th>
<th>CON</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standardization</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Generics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW Practices</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SW Reuse</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microsoft/Apple</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Sun/Standards</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Splitting Java</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Ease of Use</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Java</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Advanced APIs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Portability</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Similarity C++/OO</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Teaching OO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth of Domain</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Breadth of APIs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming Languages</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client/Server/DOC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java and Legacy</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Java and COTS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Java/Future COTS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Database/Persistence</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Obj. Serialization</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Java/Relation DBMS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Java/OO DBMS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JavaBeans</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IDEs</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 7. Summarizing PROs and CONs of Java.

achieved; the unproven promise of integration of Java with programming languages, databases, CORBA, and legacy/COTS applications; and, the evolving status of Java's integration with relational and object-oriented DBMSs. As shown in Table 7, some of these UNKNOWNS accentuate the CONs of Java while other juxtapose the PROs.

In summary, the recommendation on Java as a programming language is: **Caution, Caution, Caution**! Clearly, Java is a rapidly evolving language without a standard, and with a potential battle brewing between its major promoter (Sun) and Microsoft. Java measures up well against C, C++, Ada95, Smalltalk-80, and Visual Basic 5, as presented in Tables 3 through 6, but it is difficult to rank programming languages, since they all have their own strengths and weaknesses. Further, while Java claims to have platform independence, in practice, there are many portability issues that are yet to be resolved. Even though Sun is claiming that most of the portability and performance problems of Java will be minimized with the release of Version 1.2 of Java [33], such promises must be taken at face value; only time can determine if Java will continue to mature into a viable and robust, cross-platform technology. However, when one considers the alternatives
(or lack thereof) for design and development of platform independent software, and the massive number of resources on WWW related to Java (e.g., just key in Java into your favorite search engine), one would expect that despite all of the **PROs** and **CONs**, Java is here to stay and will likely dominate from now until 2010.

Despite this prediction, it is important to place the potential of Java in perspective. In the history of computing, the successes and failures of products has often been dictated by availability rather than superiority. C gained popularity rather than PL/I, since there was a conscious decision to uncouple the programming language from the hardware/OS, and make it available free of charge. IBM contributed, in part, to C’s rise to prominence, when it made the business decision to limit PL/I to IBM and Amdahl platforms. Despite the fact that PL/I is an excellent programming language, it has disappeared from the programming landscape. Unix became popular based on its lack of cost and the fact that it was the first OS available across hardware platforms. Unix’s popularity is clearly not based on its terse commands nor its superiority from an OS perspective. System managers tolerate Unix, but in doing so, leave themselves and their enterprises open to security breaches and violations, since Unix was never intended to be utilized as it is today. One can argue that Sun is in the position to either have Java soar into prominence or disappear as yet another programming language that flashed and burned. While Java has exceptional promise, its success or failure might be determined by a poor business decision (Sun’s standard position to date - Section 6.1.1) or the moves of a strong competitor (Microsoft’s infusion into Apple - Section 6.1.3). That is why **Caution** is so strongly recommended!
References


