PersonalJava™ Technology White Paper

August 1998

1. Introduction

1.1 Who should read this paper

This paper is an overview of PersonalJava™ technology. It is written primarily for electronic device developers and independent software developers who are evaluating PersonalJava technology. End users of products with PersonalJava technology may also find this paper to be an interesting guide to the products and applications that are rapidly becoming available. For more information on Java™ technology in general, please refer to Sun Microsystems™' collection of white papers (http://java.sun.com/docs/white/index.html). For more information specifically on PersonalJava technology, refer to Sun™'s PersonalJava web site (http://java.sun.com/products/personaljava).

1.2 What is PersonalJava Technology?

For those of you who are new to Java or PersonalJava technology, there are many terms that are used to refer to various components of the technology. Having so many terms can be quite confusing, so let's start with descriptions of some of most commonly used terms.

The **PersonalJava application programming interface (API)** is a collection of packages, classes and methods defined by a high-level specification. Besides the PersonalJava API, there are several other Java technology API's (the Java API, the EmbeddedJava API, etc.) which are tailored for different markets and applications. The definition process for all the API's is an open one, managed by Sun Microsystems. An API does not refer to a specific implementation, but describes the high-level specification that can be implemented in many ways.

The **PersonalJava virtual machine** is an implementation of the Java virtual machine specification described in *The Java Virtual Machine Specification* by Lindholm and Yellin[1]. The specification described in this publication is a high-level description that can be implemented in many ways. The PersonalJava virtual machine is a specific implementation of this specification for the PersonalJava application environment.

The **PersonalJava application environment (AE)** consists of an implementation of the PersonalJava API with the PersonalJava virtual machine. This is actually what gets shipped on a device, *e.g.* PersonalJava applications can run on any device that has the PersonalJava application environment resident on it. The various Java application environments are described in Section 1.3

The **Java Development Kit or JDK ™**is a desktop development environment for application developers. It consists of a Java runtime environment and tools for developing software written in the Java programming language (*e.g.* applet viewer, compiler, etc.). The JDK is used to develop applications to *all* the Java API's. It is freely downloadable from Sun's web site.

The **Java programming language** is an object-oriented language that software developers use to develop applications or applets. There are several books on the market that describe the language and aid in the development of software using it. The Java programming language is used to write applications and

[1] Lindholm, T. and Yellin, Y., *The Java Virtual Machine Specification*, Addison-Wesley, ©1997.

applets for *all* of the Java application environments. The complete manual for it is available on Sun's web site.

A **reference implementation** is source code that Sun ships to its licensees. This is source code that implements the PersonalJava API on a particular operating system *e.g.* Solaris^{TM} or Win32.

1.3 Java Application Environments

There are four Java Application Environments (AEs). A Java AE is also sometimes referred to as a Java "platform." The benefit of having four distinct Java AEs is that each AE is tailored for a specific group of applications. This allows Java technology to be used efficiently from smartcards to mainframes in a well-defined and compatible way. Figure 1 summarizes the four Java AEs and their target applications.

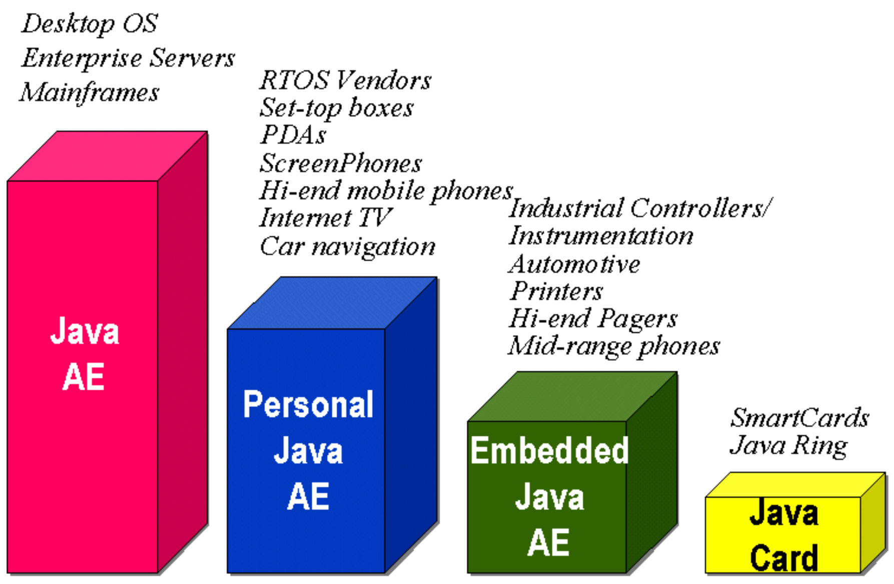


Figure 1

Simply stated, the PersonalJava application environment is the Java application environment optimized for consumer electronics devices. Network connectable devices for home, office, and mobile use are the targeted applications for the PersonalJava AE. Examples of devices implementing the PersonalJava AE are Internet telephones ("webphones"), personal electronic organizers, and set-top boxes. The PersonalJava AE is designed to make electronic devices easy to use. The end users of such devices are assumed to be consumers who have little or no experience using computers.

1.4 Device Manufacturer's Software Challenge

The complexity of software development is becoming an increasingly important problem for consumer electronics device developers. The greater capability and lower cost of hardware are enabling consumer devices to handle larger and more complex system and application software. Conventional software development methods such as "roll your own" are becoming too difficult, unreliable and costly while delaying time-to-market.

Developing for the PersonalJava AE includes all the benefits of developing for the Java AE. The PersonalJava AE provides a solution to the pressures that device developers face today. PersonalJava software is platform independent, allowing application portability. It is possible to upgrade the underlying operating system, CPU, and/or development tools while protecting previous software development investments.

The class packages in the PersonalJava API provide a rich and useful foundation for building high-quality applications quickly. The PersonalJava programming environment is inherently more reliable, secure, and supportable than other software development environments. It includes integral support for the networking and mulithreading required in modern interactive graphical applications.

The PersonalJava programming environment provides a great degree of independence between design elements. The architecture allows software modules to be purchased from outside sources and quickly and reliably integrated into products. There is no need to incur the delay and expense of creating readily available software modules from scratch. In evaluating the "Buy vs. Build" question, the "Buy" option is becoming more attractive than ever. As developers move from in-house custom software development to commercial off-the-shelf software modules for common functions, they have more time to focus on creating value-added, differentiating features for their products.

1.5 Content Developer's Opportunity

The proliferation of a wide array of different and incompatible consumer devices has always been a problem for software developers. These fragmented and proprietary environments have limited the market size of any single device platform. Time to market pressures force content developers to pick and choose individual target devices, thereby limiting their available market. This highly-segmented market for consumer software has not been a particularly attractive opportunity for content developers until now.

The PersonalJava application environment solves these problems by enabling the same application software to run on consumer devices of different brands, models, and types. Since the same application software can be run on all devices without re-compilation, the market for off-the-shelf consumer software is finally large enough to be economically attractive for independent software developers.

For example, a developer could write an electronic program guide (EPG) to the PersonalJava API that can be used on many different television set-top boxes without code modification or recompilation. Similarly, scheduling software for personal electronic organizers could be used on a variety of different models and brands. More specialized applications such as interactive electronic banking could be dynamically downloaded and used on a large number of different devices. In addition to full applications, developers may create components or modules that can be used by both device developers and other content developers.

Since PersonalJava applications are upward compatible to the Java application environment, the market for PersonalJava applications is further expanded. Most desktop computers today have the ability to run Java applets or applications. By adhering to the development guidelines defined by the PersonalJava application environment, the same applications that run on a small consumer device can run on a desktop computer.

Another important advantage for software developers is the availability of many powerful programming tools for the Java application environment. The same tools that are used to develop Java applications can be used to develop PersonalJava applications. These tools range from the Java Development Kit (JDK^{TM}) available for free from Sun's web site to a variety of powerful, professional graphical application builders available from third parties. The JavaCheck^{TM} tool, available for free from Sun, is used to verify an application's conformance to the PersonalJava API specification. PersonalJava applications can be developed on desktop computers without access to the target device. Another tool, the PersonalJava emulation environment allows applications to be executed and tested. These excellent tools let developers focus their creativity and imagination on their applications rather than solving niche programming problems on a particular device.

1.6 Design of the PersonalJava API

The design of the PersonalJava API is a collaborative effort between Sun Microsystems and many device and content developers in the industry. This open approach to defining the PersonalJava API has allowed it to benefit from the knowledge and experience of developers who are actually using the PersonalJava API in product designs. As the PersonalJava API continues to be enhanced, this open collaborative process will also continue.

Because of the cost pressures on consumer devices, minimizing the memory footprint has been a key consideration in the design of the PersonalJava API. All features that have been included in the PersonalJava API have been carefully weighed against this requirement. The memory requirements of the PersonalJava API have been further reduced by rewriting portions of the underlying implementation to use less memory.

The PersonalJava API is a feature level subset of the Java API. Certain packages were designed to be "optional" to allow device manufacturers to pull in or leave depending on the needs of the device. This allows the manufacturer to use the minimum amount of space for the PersonalJava application environment.

Where necessary, certain features have been added or modified in the PersonalJava API from the original Java API to make it more useful in the consumer environment. For example, the PersonalJava API pays special attention to the needs of portable devices by providing an optimized version of the Abstract Window Toolkit (AWT) targeted to small displays. Likewise, the PersonalJava API is designed to enable applications to be fully functional using a touchscreen, joystick, remote control or pen for input rather than a full keyboard.

1.7 Overview of Remaining Chapters

Chapter 2 gives examples of devices that utilize PersonalJava technology. The History of Java Technology in Chapter 3 shows how PersonalJava technology is actually the evolution of Java technology back to its original purpose. Chapter 4 explains the pressures faced by consumer device developers today. Chapter 5 describes the challenges faced by the independent software developer specifically when writing software for consumer devices. Chapter 6 delves into the details of specific features of the PersonalJava API.

2. Examples of Devices Implementing the PersonalJava Application Environment

There are many examples of consumer devices that are enabled and enriched by PersonalJava technology. One important, high-visibility application is television set-top boxes. Set-top boxes may receive their signal from a variety of sources such as cable, satellite, or terrestrial broadcast. In addition to the traditional functions of outputting audio and video, many new software-based features are now desired. Examples include on-screen intelligent electronic program guides, personalized information tickers, games, shopping, web access, and billing information. The PersonalJava application environment is an ideal standard for providing an open and secure environment where the same programs and data can be deployed across a variety of different set-top boxes.

Another important category of products is the Internet telephone or “webphone.” Webphones can be either traditional counter-top phones or can be portable wireless (cellular) units. Big or small, wired or wireless, webphones provide universal access to important information and communication services. Compared to the traditional PC, webphones are a convenient and trouble-free way for people to get connected.

The PersonalJava application environment drastically simplifies the software development of webphone products. Commercial software modules for parsing and displaying HTML, the language of web pages, are available. Since many web sites contain Java applets, having the PersonalJava application environment on the webphone allows users to more fully experience the rich content available from the Internet.

Other common examples of devices implementing the PersonalJava application environment include personal electronic organizers, game consoles, and set-top boxes.

It is important to remember that the PersonalJava application environment provides an open platform where imaginative new capabilities can be realized by device manufacturers, independent software developers, and even end users.

3. History of Java Technology

It is no coincidence that the list of desirable properties for a consumer electronics software environment mirrors the feature set of the PersonalJava API. A brief look at the history of Java technology shows why it is such an excellent fit for consumer electronics devices.

3.1 Catching the Next Wave of Computing

The history of Java technology goes back quite a few years. In December 1990, Sun initiated “Project Green” to develop a prototype product for the coming wave of consumer-computer convergence devices. A team of Sun researchers moved to an off-site location and began development of this new type of convergence device. The prototype, which became known as the Star7, was conceived as a portable home controller. The Star7 was designed to be inexpensive yet able to perform a wide variety of useful functions.

To keep the Star7 inexpensive, it was built from commonly available components. Many of the original parts were taken from other consumer electronics devices. The bill of materials for constructing a Star7 was quite similar to many of the consumer devices that are being developed today. Some of the main hardware components in the Star7 included:

- 5” 16-bit color 240x240 LCD display
- Double-buffered graphics controller
- Touchscreen input (no mouse)
- SPARC^{TM} RISC CPU
- 200 Kbps, 900 MHz spread-spectrum wireless networking

- Infrared communications
- Multimedia audio codec
- Miniature speakers
- PCMCIA card slots

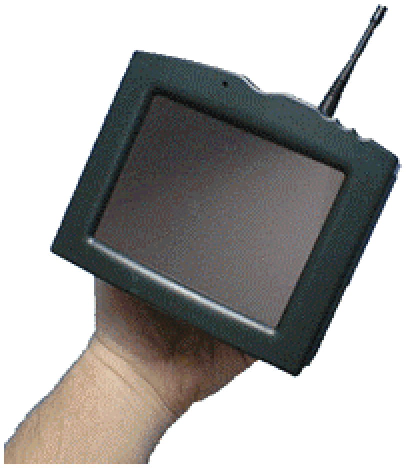


Figure 2: Star7

On this hardware platform, a complete set of system software was developed. A version of UNIX® that would run in less than 1 MB, including drivers, was ported to the Star7. An embedded file system was created to read from and write to the Star7's flash memory. A small, fast graphics library was developed which included support for true-color, anti-aliased graphics, alpha-channel compositing, and sprite animation.

The Star7 implemented many impressive features. For example, the touchscreen input pad understood an easy to learn set of gesture commands. The Star7 was able to communicate with a base station as well as other Star7s to exchange programs and information. A friendly on-screen agent named Duke was created to assist the user.

The Star7 included a number of built-in application functions such as:

- TV/VCR remote control
- Electronic TV program guide
- Distributed whiteboard
- Checkbook balancing
- Appointment book

Most importantly though, the Star7 was envisioned to have expandable and dynamic applications software. The Star7 would be able to handle new functions by simply loading new software over the wireless network. The loading of new software would be seamless and dynamic. Software modules could be loaded and unloaded on-the-fly without going through a traditional software installation process. The ability to dynamically load and unload software during runtime is particularly important for an inexpensive consumer device because it allows the device to make the best use of its limited memory. For example, when used as a remote control, the Star7 was able to download a custom user interface from the device that was being controlled.

From the perspective of this history of Java technology, the Star7 is important because it defined the original objectives for software on a small, networked consumer device. The next section discusses these software development issues in more detail.

Figure 3: Duke Holding the Star7

3.2 Star7 Software Difficulties

The C++ programming language was originally used to develop software for the Star7. Given the advanced capabilities planned for the Star7, it is probably no surprise that significant software problems arose during implementation. It was these difficulties of using C++ that served as the impetus for what has now become the Java programming language.

Despite having a team of seasoned programmers, serious problems continued to hamper the progress of the Star7's C++ software. The development tools were failing and programming bugs were becoming increasingly difficult to track down, eliminate, and avoid.

An even more complicated situation was dealing with the goal of having various devices share programs and data in a secure, reliable way over a network. Portability was a huge problem. Different processor instruction sets, data sizes, and compiler behavior made it extremely difficult to achieve this goal. Even if these obstacles were solved, security remained a big issue since there was no way to know what potentially harmful things might occur as a result of running a certain set of code.

3.3 Programming Language Requirements

James Gosling, the originator of Java technology, realized that they needed a better solution. He began working on a language called "Oak" that would enable their team to write the necessary software for the Star7.

The key requirements of the new language were:

- Networked - to communicate data and programs with other devices
- Secure - to prevent unauthorized access to sensitive or valuable information
- Reliable - so consumer devices will not fail or need to be re-booted sporadically
- Platform independent - for binary compatibility between a wide variety of devices
- Multithreaded - to easily allows more than one activity to occur simultaneously
- Dynamic - to allow loading and unloading of various software without a local hard disk
- Small code size - necessary for small-memory, low-cost consumer devices
- Simple and familiar - make it easy for programmers by keeping it similar to C/C++

It is important to remember that the developers of the Star7 did not originally intend to create a new programming language. Oak was a necessary tool to solve a particular set of programming problems on an advanced consumer electronics device. The design of Oak was much more practical than academic. Rather than adding lots of new features and concepts, Gosling shamelessly borrowed the best proven ideas from existing programming languages.

Gosling describes the Java programming language as a fusion of four kinds of programming:

Object Oriented	like Smalltalk
Numeric	like FORTRAN
Systems	like C
Distributed	like nothing else

To accomplish the goal of familiarity, the syntax of Oak was kept as similar to C++ as possible. The most problematic parts of C++ were removed from the language and explicit support was added for important features. In the creation of Oak, the designers were careful not to add unnecessary or error-prone features. A feature would be included only if there were several real examples to demonstrate its necessity.

3.4 Emergence of the World Wide Web

Although the Star7 was a bit ahead of its time, the emergence of the World Wide Web provided a fortuitous opportunity for its unique software. In a sense, the web is much like the environment imagined for the Star7: networked, open, dynamic, innovative, and unconstrained.

Initially, Web content was static and relatively unsophisticated, consisting primarily of simple text and graphics. The browser did little more than format pages from the server for display on the client. Interactivity was very limited since any unique computation had to occur on the server and be sent to the browser client as a single page of results. One problem with this approach is that it placed a large burden on the server, making the interactivity very slow. For many applications this method of having the server do all the work and send results back is so unwieldy as to be impractical.

To illustrate this point, imagine a web site that continuously graphs the real time price of a stock over the last hour. Originally, this would have been difficult and slow since the server would have to continuously generate a new picture of the graph and send the entire picture to the browser for each display update. Since the client browser was "dumb," it could not simply receive the number of the most current price and update the graph by itself. For this application, and many others, the ability to run a small program on the client browser would be of enormous benefit.

Just as it had been for the Star7, running programs across the Web raises the same daunting problems of binary compatibility, operating system compatibility, reliability, and security. To demonstrate how Oak could successfully solve these problems for the Web, an experimental web browser called WebRunner was developed in 1994. WebRunner, the predecessor of today's HotJava^{TM} browser, allowed applets to be downloaded from the server and executed from within the browser. When Oak was introduced for the Web, it was renamed the Java programming language. WebRunner was written entirely in the Java programming language.

Being able to execute programs on the client's browser rather than the server made many more Web applications practical. Applications such as the stock ticker graph are simple and efficient when implemented with Java technology.

Once on the Internet, the popularity of Java technology spread rapidly. The Java application environment and the HotJava browser were announced in April 1995. The following month, Netscape licensed Java technology and built it into the Netscape 2.0 browser. Sun released the Java Development Kit (JDK) in January 1996 and has continued to enhance the functionality of the JDK ever since.

3.5 Returning to the Roots of Java Technology

As a good object oriented language should, the Java programming language makes it easy to create libraries of sharable and reusable software. This is important for programmers, since powerful applications with complex user interfaces can be developed with minimal effort. The downside of creating this additional functionality is that it takes up memory space. For example, the “core” or “required” classes take up about 8.8 Mbytes of memory in JDK 1.1. On systems that have a local hard drive, the space for the core classes is negligible. However, for memory-constrained consumer devices, these extensive class libraries may take up valuable memory and may not be needed for the types of applications the device is designed for.

Stated simply, one of the main goals in designing the PersonalJava API is to slim-down the Java API to fit the needs of consumer devices. As the set of “core classes” in the Java API has grown, many of the new specialized functions are not needed on all types of devices. The PersonalJava API specifies a set of classes that are specifically targeted towards small networked consumer devices. The PersonalJava 1.0 API specification was released in September 1997.

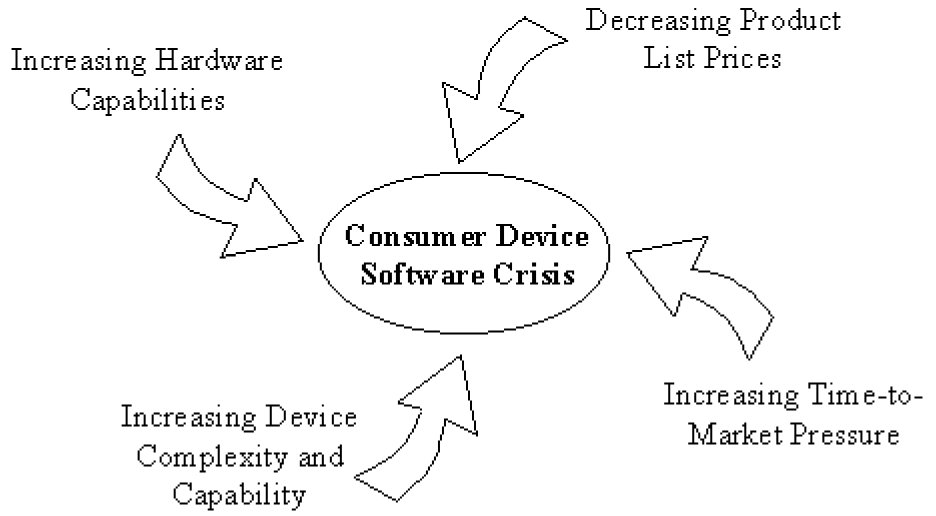
In summary, Java technology was originally invented to solve the problems of networked consumer electronics devices. Although its first large-scale deployment was on desktop computers for connecting to the Web, there is a resurgence of interest in using Java technology for its original purpose. The PersonalJava application environment is the return of Java technology to its roots. It is the Java application environment optimized for networked consumer devices.

4. Consumer Device Market Pressures

This chapter describes the challenges facing device manufacturers today and shows how PersonalJava technology can help them meet these challenges, not just today, but going into the future.

4.1 Market Forces

External market forces are shaping the landscape for development of new consumer devices. Microprocessors are becoming much more powerful as their prices drop. Memory prices continue to decrease as their capacities increase. More functions are combined into each integrated circuit, lowering overall cost and decreasing product size. The complexity and capability of consumer devices continues to grow as their list prices decline. This highly competitive environment increases the time-to-market pressure on device developers.


Increasing Hardware Capabilities
Decreasing Product List Prices
Consumer Device Software Crisis
Increasing Time-to-Market Pressure
Increasing Device Complexity and Capability

4.2 Effect on Consumer Device Software

These external market forces are causing a crisis in developing software for consumer devices. New consumer devices can handle larger, more complex software applications. The days of providing a few simple built-in functions are gone. Many more applications are required now. Consumers expect to go beyond a few fixed functions and run an expanding variety of applications. Being able to attract software developers to develop for a device is important for the success of the device. Purchasing portions of a consumer device's internal software is often more attractive than building everything from scratch.

The next section discusses the many benefits of implementing the PersonalJava application environment in consumer electronics devices. At a high level, the PersonalJava application environment allows developers to decrease their time to market, provides an attractive environment for software developers, and decreases development costs.

5. Consumer Device Software Requirements

The software requirements for consumer devices are different from those of desktop computing systems. The PersonalJava application environment has been optimized for delivering software on consumer platforms. This section highlights some of the benefits of using the PersonalJava application environment on consumer electronics products.

5.1 Small Memory Footprint

One criterion for a successful consumer electronics product is to achieve high unit volumes since profit margins are often quite slim. Product pricing can have a dramatic effect on the number of units a product will sell. Most highly successful consumer products retail for no more than a few hundred dollars. In order to meet these aggressive price targets, consumer electronics manufacturers must minimize the cost of every component in their system. Although memory capacity has steadily increased while its price has declined, keeping memory sizes to a minimum remains an important consideration in keeping a rein on system costs.

With the pressures of consumer electronics manufacturers in mind, the designers of the PersonalJava application environment have made a considerable effort to minimize the both the ROM and RAM footprints. The most obvious memory savings was achieved by recognizing that consumer device applications do not require the functionality of the full JDK specification. The designers of the PersonalJava application environment removed classes that were only useful on a desktop environment. In addition, certain classes were deemed to be optional and may be included or left out at the discretion of the device designer.

Another technique for conserving memory is more subtle. One of the benefits of a good object oriented programming language like the Java programming language is that an object's private internal implementation be completely hidden from its external appearance. Consequently, as long as the public fields (variables) and methods remain the same, it is possible to re-implement the data structures and methods that form the internal implementation of a class. Complete compatibility can be maintained as long as all the public interfaces remain identical.

As anyone who has ever written even a simple computer program knows, there are many ways to implement a given program function. To maximize the benefit for consumer electronics manufacturers, much of the underlying implementation of the PersonalJava application environment has been optimized to use memory more efficiently. All classes in the PersonalJava API are still fully compatible with the Java API since the public class interfaces have been kept identical.

The table below shows the operating targets for each of the four JAEs.

	Java	**PersonalJava**	EmbeddedJava	JavaCard
ROM	4-8 MB	**< 2MB**	< 512 KB	16 KB
RAM	> 4 MB	**1 MB**	< 512 KB	512 B
Processor	32-bit; >100MHz	**32-bit; >50MHz**	32-bit; >25MHz	8-bit, 300 KIPS

Table 1. JAE Hardware Targets

Note that the RAM and ROM targets listed in the table only refer to the requirements of the Java application environment itself, which is comprised of the Java virtual machine and the required class packages. Each particular system will require a variable amount of additional memory for the operating system, application software, and application data.

One thing to keep in mind when calculating the total memory required for a particular device is that PersonalJava applications tend to be much smaller than their corresponding processor-specific counterparts. Many microprocessors today have a 32-bit instruction size. In contrast, Java virtual machine instructions are composed of compact bytecodes. Many Java virtual machine instructions use only 8 bits.

This difference between the size of the bytecodes in Java software and processor-specific instructions is further multiplied when "fat binaries" are taken into account. Binary code generated from Java programs runs unmodified on all Java virtual machines. To partially achieve a similar effect, multiple versions of complied code are combined together to form a single "fat binary" file which can run on a few different processors. The obvious disadvantage of this method is that it only runs on a limited number of processors and in the process multiplies the memory requirements by the number of processors supported.

Even though class files implementations of the Java and PersonalJava APIs are relatively compact, a mechanism for lossless compression is also provided. The Java Archive (JAR) format allows multiple class files to be combined into a single file while allowing the archive to be compressed. In addition to saving memory storage, JAR files can be transferred very efficiently over the Internet.

In summary, although the cost of memory is continually declining, the PersonalJava application environment does an excellent job of keeping memory utilization to a minimum, which in turn reduces the cost of electronic devices.

5.2 Optimizations in the PersonalJava Virtual Machine

One of the primary design goals of the PersonalJava virtual machine is the reduction of static memory footprint and runtime memory usage. The core classes of the PersonalJava API are typically preloaded with JavaCodeCompact into an alternate format that requires mostly ROM and a small amount of RAM storage. The PersonalJava virtual machine also drastically reduces the memory footprint of preloaded and dynamically loaded classes, native stack usage and Java stack usage. This results in a system with reduced ROM and RAM requirements, without introducing any perceivable degradation in performance.

5.3 Security

Security for programmable electronic devices is more important now than ever before. With all sorts of electronic devices being networked together, programs and information are flowing at unprecedented speeds. Fifteen years ago "sneaker-net" (carrying a floppy disk) was a common way to get a few files from one place to another. With the growth of networking and modem connectivity, people could go to a bulletin board or shared server to exchange the files. However, because of the difficulty and slow transfer speeds, a relatively small amount of information was exchanged compared to now. Today, with high-speed networks, everyday users are accessing thousands of remote files without necessarily even realizing it. A quick look at the cache directory of any web browser is a good reminder of the amount of information that can be seamlessly transferred between electronic devices today.

5.3.1 Components of Security

One of the strongest advantages of the Java and PersonalJava application environments is the excellent security. Strong security is achieved through a two-pronged approach based on digital signatures and the Java "sandbox". A digital signature lets you know the true identity of the provider of a file. Digital signatures prevent people from distributing information using a false identity and also allow recipients to know that received information has not been tampered with. The assumption is that if you know where a program comes from, then you can judge whether to trust it. Avoiding security problems amounts to avoiding content from providers who are not known to be trustworthy.

In addition to supporting the use of digital signatures, the Java application environment provides a "sandbox" in which an applet can execute safely without any risk to the host system. Programs are restricted to playing within the sandbox, a safe virtual program area that does not allow activity that could interfere with the system or compromise security. Programs executing in the sandbox may be restricted from activities such as the following:

- reading files
- writing files
- creating files
- modifying files
- accessing a web site other than its own
- reading restricted system properties

- launching other programs
- shutting the system down

For most cases, a digital signature is not necessary and trust is not an issue since Java applets are not allowed to do any harm. This ability to restrict program activities and prevent problems strongly differentiates the PersonalJava application environment in the consumer software space.

There are occasions when the sandbox is too restrictive. For example, an applet might want to save a small amount of information in local storage. Using "signed applets" a digital signature can be used to authenticate a particular Java applet for which some sandbox restrictions can be temporarily relaxed. The combination of the Java sandbox and use of digital signatures is a powerful security system. No other general purpose programming language can match the strength and versatility of the security system provided by the Java application environment. The proliferation of Java applets on the Web is working proof of the success of this security system.

5.3.2 Security Holes Eliminated in the Java Application Environment

Given the large number of software viruses, there are obviously a lot of clever people out there looking for security holes in software systems. At a very low level, the Java virtual machine eliminates many of the mechanisms that could be used by programs to violate a system's security.

One of the cornerstones of the Java application environment's runtime security is called "bytecode verification." Essentially, bytecode verification performs a number of checks on an application or applet before it is run. These checks ensure that the application is well behaved and not involved in any tricky programming shenanigans. There is no way to disable bytecode verification. All software must be checked before it is used by the Java virtual machine. The following list shows some examples of analyses that are part of the bytecode verification process:

- Class files are structurally correct
- Arguments to all Java virtual machine instructions are of the correct type
- Every branch instruction branches to the beginning of an instruction
- Execution cannot branch outside the ends of the code
- Every method has a correct signature
- No operand stack overflow or underflow can possibly occur
- All local variable usage is valid
- Final classes are not subclassed
- Final methods are not overridden

Compared to other binary machine languages, instructions for the Java virtual machine contain important additional type information. For example, there is a different store instruction for an `int` and a `float` even though they both just store a 32-bit value from the top of the stack to a local variable. This extra type information allows the Java virtual machine to submit the program to an additional level of scrutiny. All bytecode verification occurs only once at link time so that runtime performance is not hindered by these security checks.

There are many other restrictions enforced by the Java virtual machine to maintain security. One of the more well-known features of the Java application environment is that it prevents a program from accessing memory outside of its allowed range. This is quite unlike C/C++ in which illegal memory access can easily occur. For example, in C, it is up to the programmer to know the correct length of an array at any given time. If the wrong length is used, other data can be accessed. The Java application environment makes arrays both simpler and more reliable. The Java virtual machine automatically keeps track of the length of all arrays at all times. It checks during runtime that a program does not try to access beyond the valid range of an array. Array bounds checking is one simple way that the Java application environment prevents programs from doing things that they should not.

In C, it is possible to access random memory locations by simply reading the contents of a computed pointer address. In the Java programming language, there is no way to directly calculate addresses and access memory directly. Restricting programs to the memory locations they are supposed to have access to is a required element of security.

One last example of a potential security hole that the Java application environment does not allow is the access of allocated but uninitialized memory. In the Java application environment, all fields of all objects and arrays are guaranteed to be initialized to a known value. Unless otherwise specified by the program, all numeric primitive fields are initialized to zero, all characters to Unicode /u0000 (null), and all Boolean values to false. The Java compiler will report an error if a local variable is used in a program without initialization. This is unlike the C programming language where a programmer could request a large block of memory that had been previously used by another application and directly read its contents.

In summary, security is critically important for all sorts of electronics systems today. The Java and PersonalJava application environments are unmatched in their ability to provide a secure computing environment. This excellent security is one of the reasons why it is so widely used in networked applications and across the World Wide Web.

5.4 Consumer Graphical User Interface

The user interface design for consumer electronics devices differs significantly from that of desktop computer applications. First of all, consumer electronics devices utilize a much wider variety of input methods, display types and display sizes. More importantly though, consumers have little patience for learning how to operate complicated new devices. Rather than struggle with a labyrinthine user interface, many consumers will simply give up in frustration.

The PersonalJava application environment provides an excellent solution for creating effective graphical user interfaces for consumer devices. The PersonalJava API makes use of a subset of the JDK's Abstract Window Toolkit (AWT) specifically tailored for consumer device needs. This version of the AWT is tailored towards creating user interfaces that are familiar, non-intimidating, and easy to learn. It does not assume that the users of consumer devices are familiar with desktop computers. Use of complex, computer-oriented graphical constructs like layered menus and horizontal scrolling is de-emphasized. This AWT, dubbed "Personal AWT", enables the creation of top-notch consumer graphical user interfaces with a minimal amount of design and programming effort.

Consumer electronics devices utilize a wide variety of input methods. A programmer cannot assume that a mouse and keyboard will always be available. The Personal AWT is designed for handling the diverse input methods encountered in consumer electronics devices. For example, a consumer might interact with device using specialized buttons, a remote control, a virtual keyboard, a joystick, a pen, a touchscreen, or even speech. With the Personal AWT, programmers can write applications that transparently support whichever input mechanism is available on a particular device. For situations where more than one input method is available, a preferred input method can be specified by the programmer. The AWT greatly simplifies the user interaction problem for consumer software.

5.5 Software Portability

Software is portable if it can be used on different systems without change. Historically, portability has been a significant problem for software developers. It is not unusual to have employees whose sole function is porting software to different platforms. The easiest form of portability to achieve is source code portability. Given the number of small differences in implementations of programming languages and the subtle differences between similar libraries on different platforms, it can take a lot of effort to write portable source code.

Due to the relative difficulty of achieving object (binary) code portability, it has been little more than an academic subject prior to the development of Java technology. One of the traits which has made Java technology so compelling is that it has both source code and object code portability.

Portability makes the job of software developers much easier. It gives them independence from a particular operating system, microprocessor, and software development environment. If a device designer decides to switch to a new operating system for a next-generation device, all of the legacy Java and PersonalJava applications will run on it. Furthermore, portability makes it much easier to find useful software modules for a project. Not having to write everything from scratch can save a lot of time and effort. The ability to develop software independent of the final target device allows developers to start earlier and take advantage of the most powerful development tools.

Object code portability enables network downloadable applications. Without object code compatibility, it would not be practical to write software that can run on any client machine. Software portability also benefits independent software developers greatly since the total available market for their products and services becomes much greater.

5.5.1 Platform Independent Bytecodes

One of the most well known slogans of Java technology is the “Write Once, Run Anywhere”™ mantra. The key underlying technology behind this binary portability is Java’s bytecode instruction format. Java bytecodes provide a hardware independent instruction set format for representing programs. Since the bytecodes are not targeted to any processor, there is a virtual machine which defines an abstract model of how the bytecode instructions are to be interpreted

At the Java bytecode instruction level, the Java and PersonalJava application environments are identical. They both fully implement all the instructions that are allowed by the Java virtual machine specification. The main difference is that PersonalJava applications refer to fewer system classes than Java applications. The Java bytecode instruction format is very stable. One of the key reasons why Java class object files are so portable is the stability of the Java virtual machine instruction set.

5.5.2 Multithreading

Most microprocessor-based systems give the appearance of being able to perform several tasks simultaneously. This might range from the simple ability to respond immediately to pressed buttons while a display is being updated, to a more complex situation of running several related but independent software applications at the same time.

In reality, a microprocessor’s CPU performs one task a time. By rapidly switching between a number of different tasks, the microprocessor creates the appearance of doing the tasks simultaneously. Virtually all microprocessor-based systems today utilize some form of task switching.

In the past, a processor might have had a relatively small number of different tasks to handle. Devising a way to handle a few I/O requests on a particular device was not too problematic. Today, the demand for software concurrency is much greater. A web browser is a good example of an application that performs several software tasks concurrently. The browser software might be performing an http GET request for information, playing an audio clip, updating download progress, animating several different graphical image sequences, and checking for additional user input --- all at the same time.

Two of the most common ways of handling concurrency on small electronic devices are using processor interrupts and multitasking operating systems. One of the problems with these solutions is that they tend to be platform specific. Different processors have different interrupt capabilities. An interrupt service routine written for one device is unlikely to run on another. Similarly, the calls to initialize and execute multiple tasks are likely to be different on different operating systems. More interaction between software tasks and finer grain control usually results in even more complicated, non-portable software. A portable method for software concurrency is a critical component of software portability, especially for network downloadable software.

Among programming languages, the Java programming language is somewhat unique in that support for threads ("threads of control") is built directly into the language. This allows programmers to solve many concurrency problems more simply than ever before. Most importantly, threads in the Java programming language are portable. The same multithreaded executable program will run on a variety of devices regardless of the underlying operating system.

The PersonalJava application environment fully supports these threads. For small electronic devices, the ability to write, debug, and test multithreaded software on a larger development system can save a lot of time and effort. Writing software that is portable and independent of the processor and operating systems has numerous benefits for developers.

5.5.3 Word Sizes

One source of compatibility problems is that various microprocessors have different native word sizes. An integer might be 16 bits on one processor and 32 bits on another. Mathematical operations on one processor might yield different results on processors with smaller or larger word representations. To avoid these incompatibilities, the designers of the Java application environment fully specified the meaning of each of the primitive data types. Each primitive data type is always the same regardless of the underlying microprocessor that is running the code.

In the Java programming language, all numeric types are signed. This eliminates some incompatibilities that occur in C/C++ programming, such as between signed and unsigned characters. Since all primitive types are fully specified and always the same, there is no need for the `sizeof()` operator which is used in C/C++ to determine the size of a type on particular machine.

5.6 Reliability

As mentioned in the earlier sections that discuss the history of Java technology and the development of the Star7, C and C++ leave much to be desired for creating reliable software systems. Much of the original motivation for the Java programming language came from the unreliability of software created with C++.

In the design of the Java programming language, many of the most problematic aspects of C/C++ were eliminated. Even if a feature was not inherently bad, common misuse by programmers motivated finding more reliable solutions. In the following paragraphs, features that have been removed are discussed first. Additional Java programming language features are covered subsequently.

In the Java programming language, there is no mechanism for directly manipulating memory addresses. Although objects are referenced through pointers, it is not possible to perform arithmetic on pointers. This eliminates a number of often difficult-to-find bugs that can occur in C/C++.

Although `goto` is a reserved word in the Java programming language, it serves no useful purpose other than to prevent programmers from abusing it. A `goto` in a program often indicates that it should be reorganized. There are a few valid cases when `goto` should be used in C/C++ programming, such as breaking out of nested loops. Instead, the Java programming language provides an explicit mechanism, the `break` and `continue` statements. These statements satisfy the valid use of `goto` without allowing the abuse.

The Java programming language has no preprocessor statements such as `#include`. It also does not allow any typedef statements. These constructs can contribute towards software that is needlessly difficult to understand and maintain. The inherent structure and capabilities of the Java programming language makes these troublesome features unnecessary.

In general, the Java programming language strives to be clear and complete in its specification. Issues that are designated as "implementation specific" in other languages have been fully specified in the Java programming language. Small changes such as creating an explicit Boolean primitive type greatly increases program reliability. Unlike C/C++, a Boolean is not the same as a number. In the Java programming language, a Boolean cannot be cast to or from other primitive types. Things that do not make sense, such as adding or subtracting the values of true and false, are caught by the compiler. Common and difficult to find errors in C/C++ such as the difference between `&` (bitwise and) and `&&` (logical and) are identified immediately by the compiler.

In the Java programming language, the bounds of an array are the same as C/C++, ranging from zero to the length minus one. One very important difference though is that in the Java programming language, all array accesses are checked against the array bounds automatically. Referencing elements outside of an array is a common problem in C/C++ programs which is caught and identified explicitly by the Java virtual machine. Since the Java virtual machine automatically keeps track of the length of arrays, the statement:

```
for (int i = 0; i < myarray.length; i++)
```

operates on every element of the named array. The `length` variable can be queried at any time and cannot be overwritten by the program. Making arrays foolproof increases software reliability.

One last, important addition to the Java programming language, is explicit exception handling. In a program, there is a normal flow of events. When something goes wrong, an exception occurs. Examples of exceptions include trying to open a file that does not exist and dividing an integer by zero. In C/C++ exception handling is left up to the programmer. There are conventions that indicate unusual conditions but no standard signaling mechanism exists. More importantly, checking for and handling exceptional conditions is not required. Letting a programmer assume that an exception is "impossible" is a risky proposition. By comparison, the Java compiler alerts the programmer when exceptions are possible but have not been "caught." Since exceptions are part of the class hierarchy, many exceptions can be handled together by catching a parent class.

One of the most important benefits of the exception handling feature is that it allows programmers to structure their code better. Exception handling code is grouped separately from the normal flow. This makes it easy to follow the main algorithm without getting bogged down in all the unexpected conditions.

Less clutter helps reduce programming bugs too. Overall, having explicit support for exceptions makes software written in the Java programming language much more reliable.

5.7 Performance

The raw computing performance of most consumer electronics devices is rarely a problem. Most modern microprocessors provide a surprising level of performance at a very low price. For most modern consumer electronics products, the majority of a device's processor cycles are spent idle, waiting for input from the user. Even when a processor is not waiting for the user, faster computation will not necessarily make the device more responsive if the processor is not the bottleneck. For example, if a device is receiving information from the network, the delivery rate of that data might well be the limiting factor.

For most networked consumer devices, PersonalJava applications perform very well. For example, since for a given functionality, Java programs tend to be very small in size, they can be transferred more quickly over the network. In addition, the PersonalJava programming environment defines a Java Archive (JAR) format to further streamline the transfer process. The JAR format helps systems run faster by compressing programs and combining multiple program files into one JAR file. As a result, the transfer of the Java application incurs the overhead associated with only one file transfer. This can significantly reduce the number of individual transfer requests the network and the two end nodes must process.

Garbage collection can also have an impact on performance. The responsibility of garbage collection requires the Java virtual machine to handle all memory allocation and cleanup. If a system runs out of memory then it is the job of the garbage collector to clean up and provide more free space to the running applications.

Like many aspects of the Java application environment, garbage collection performance will undoubtedly improve with time. The Java virtual machine specification simply requires that the memory management be automatic. This leaves plenty of room for improved algorithms and implementations of garbage collection techniques.

In case a device developer needs to write some non-Java code for their device, a well-defined mechanism is provided for running native code. In both the Java and PersonalJava application environments, "native methods" can be called from the Java Native Interface (JNI). For example, if a developer wanted to optimize a particularly time consuming operation by accessing special hardware capabilities, then a small C program can be called from within the Java application to accomplish this task.

6. The PersonalJava API Specification

The PersonalJava 1.1 API is derived from the Java 1.1 API, then supplemented by a few APIs specifically for consumer device applications. For simplicity, the PersonalJava 1.1 API specification only describes the ways in which it is different from the Java 1.1 API specification. The PersonalJava 1.1 API specification can be found at http://java.sun.com/products/personaljava/spec-1-1/pJavaSpec.html.

Note that an API is a high-level description of the packages, classes, and methods. An API can have many underlying implementations that conform to the same specification. In fact, Sun's underlying implementation for many of the classes in the PersonalJava application environment are optimized versions of their counterparts in the Java application environment. The optimizations are necessary to minimize the memory usage on devices implementing the PersonalJava application environment.

This chapter summarizes the main features of the PersonalJava 1.1 API specification. This is only a description of the package-level differences between the PersonalJava 1.1 API and the Java 1.1 API. For a detailed account of the differences at the class and method levels, refer to the specification.

6.1 PersonalJava 1.1 API Packages

Forming the core of the PersonalJava 1.1 API is a set of API's which are referred to as the "Required" packages. In addition, there are also modified, optional, additional and unsupported packages relative to the Java 1.1 API. These packages are briefly described below. Again, for a detailed account of the differences at the class or method level, refer to the specification.

Package	**Type**	**Description of Package**
java.applet	Required	Contains the `Applet` class and interfaces that enable the creation of applets. An applet is a small, embeddable Java program that often runs in a browser.
java.awt	Modified	Abstract Windowing Toolkit
java.awt.datatransfer	Required	Contains classes and interfaces that support a generic inter-application transfer mechanism. It also provides support for cut-and-paste data transfer on top of that mechanism.
java.awt.event	Required	Defines classes and interfaces used for event handling in the Abstract Windowing Toolkit (AWT).
java.awt.image	Required	Provides the infrastructure to support image processing. Most of the classes are not normally used by ordinary applications that have only simple image manipulation requirements.
java.beans	Required	Contains the classes and interfaces that constitute the JavaBeans API. This API provides a framework for defining reusable, embeddable, modular software components. It is useful to programmers who are developing "beanbox" tools to manipulate beans, and to programmers who are writing their own beans.
java.io	Modified	Contains the classes and interfaces to support input and output streaming. The classes and interfaces required for file operations is optional.
java.lang	Required	Contains the classes that are fundamental to the design of the Java language.
java.lang.reflect	Required	Contains the classes and interfaces that, along with `java.lang.Class`, comprise the Java Reflection API. The Java Reflection API allows a Java program to inspect and manipulate itself.
java.math	Optional	Contains classes for arbitrary-precision integer and floating-point arithmetic.
java.net	Required	Contains the classes and interfaces that provide a powerful and flexible infrastructure for networking.
java.rmi	Optional	Contains classes and an interface to implement remote method invocation.
java.rmi.dgc	Optional	Contains classes and an interface to implement distributed garbage collection
java.rmi.registry	Optional	Contains a class and interfaces to implement a registry of remote objects.
java.rmi.server	Optional	Contains classes and interfaces to implement the server side for remote method invocation.
java.security	Optional	Contains classes and interfaces to manage various forms of

		security.
java.security.acl	Unsupported	Contains interfaces to manage an access control list.
java.security.interfaces	Optional	Contains interfaces to manage digital signature algorithm keys.
java.sql	Optional	Contains classes and interfaces to implement Java Database Connectivity (JDBC)
java.text	Required	Contains the classes and interfaces that are useful for writing internationalized programs that handle local customs, such as date and time formatting and string alphabetization, correctly.
java.util	Required	Contains the classes and interfaces that form a set of "utilities" upon which several of the other packages depend. Examples of classes in this package include `Hashtable`, `Vector` and `Enumeration`.
java.util.zip	Optional	See specification.
com.sun.awt	Additional	See specification.
com.sun.lang	Additional	See specification.
com.sun.util	Additional	See specification.

Table 2. Packages in PersonalJava 1.1 API

6.2 Optional Packages

It is important to note that the PersonalJava 1.1 API has several optional packages. These give the device developer maximum flexibility in including only those packages that are important for a particular type of device. For example, a pager manufacturer may decide that a pager will never use remote method invocation (RMI). In this case, to minimize the footprint required by the PersonalJava application environment on the pager, the manufacturer can decide to leave out the `java.rmi`, `java.rmi.dgc`, `java.rmi.registry`, and `java.rmi.server` packages.

7. Conclusion

Security, portability, and reliability are just a few of the features of the PersonalJava application environment that make it uniquely suitable for consumer electronic devices. The PersonalJava API is not simply a stripped down version of the Java API, it is an optimized version tailored for network-connectable consumer devices. The "Write Once Run Anywhere" capability of the Java application environments enables many new applications by putting into play economies of scale. Webphones and powerful new set-top boxes are just two of the many devices that will be ushered into the home by PersonalJava technology.

For additional information about the Java and PersonalJava technologies, refer to the following sources:

Sun's Java web page: http://java.sun.com
Sun's Java White Papers: http://java.sun.com/docs/white/index.html
Sun's PersonalJava web page: http://java.sun.com/products/personaljava