The Java™ *Language: An Overview*

Introduction

The Java programming language and environment is designed to solve a number of problems in modern programming practice. It started as a part of a larger project to develop advanced software for consumer electronics. These devices are small, reliable, portable, distributed, real-time embedded systems. When we started the project, we intended to use C++, but we encountered a number of problems. Initially these were just compiler technology problems, but as time passed we encountered a set of problems that were best solved by changing the language.

The companion paper, *The HotJava™ Browser: An Overview*, describes a powerful application that demonstrates the power of the Java language.

Java

> Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.

One way to characterize a system is with a set of buzzwords. We use a standard set of them in describing Java. The rest of this section is an explanation of what we mean by those buzzwords and the problems that we were trying to solve.

> *Archimedes Inc. is a fictitious software company that produces software to teach about basic physics. This software is designed to interact with the user, providing not only text and illustrations in the manner of a traditional textbook, but also providing a set of software lab benches on which experiments can be set up and their behavior simulated. For example, the most basic one allows students to put together levers and pulleys and see how they act. The italicized narrative of the trials and tribulations of the Archimedes' designers is used here to provide examples of the language concepts presented.*

Simple

We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. Most programmers working these days use C, and most programmers doing object-oriented programming use C++. So even though we found that C++ was unsuitable, we designed Java as closely to C++ as possible in order to make the system more comprehensible.

Java omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. These omitted features primarily consist of operator overloading (although the Java language does have method overloading), multiple inheritance, and extensive automatic coercions.

We added auto garbage collection thereby simplifying the task of Java programming but making the system somewhat more complicated. A good example of a common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. By virtue of having automatic garbage collection the Java language not only makes the programming task easier, it also dramatically cuts down on bugs.

The folks at Archimedes wanted to spend their time thinking about levers and pulleys, but instead spent a lot of time on mundane programming tasks. Their central expertise was teaching, not programming. One of the most complicated of these programming tasks was figuring out where memory was being wasted across their 20K lines of code.

Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K bytes; adding the basic standard libraries and thread support (essentially a self-contained microkernel) adds an additional 175K.

Object-Oriented

This is, unfortunately, one of the most overused buzzwords in the industry. But object-oriented design is very powerful because it facilitates the clean definition of interfaces and makes it possible to provide reusable "software ICs."

Simply stated, object-oriented design is a technique that focuses design on the data (=objects) and on the interfaces to it. To make an analogy with carpentry, an "object-oriented" carpenter would be mostly concerned with the chair he was building, and secondarily with the tools used to make it; a "non-object-oriented" carpenter would think primarily of his tools. Object-oriented design is also the mechanism for defining how modules "plug and play."

The object-oriented facilities of Java are essentially those of C++, with extensions from Objective C for more dynamic method resolution.

The folks at Archimedes had lots of things in their simulation, among them, ropes and elastic bands. In their initial C version of the product, they ended up with a pretty big system because they had to write separate software for describing ropes versus elastic bands. When they rewrote their application in an object-oriented style, they found they could define one basic object that represented the common aspects of ropes and elastic bands, and then ropes and elastic bands were defined as variation s (subclasses) of the basic type. When it came time to add chains, it was a snap because they could build on what had been written before, rather than writing a whole new object simulation.

Distributed

Java has an extensive library of routines for coping easily with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the net via URLs with the same ease that programmers are used to when accessing a local file system.

The folks at Archimedes initially built their stuff for CD ROM. But they had some ideas for interactive learning games that they wanted to try out for their next product. For example, they wanted to allow students on different computers to cooperate in building a machine to be simulated. But all the networking systems they'd seen were complicated and required esoteric software specialists. So they gave up.

Robust

Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error prone.

One of the advantages of a strongly typed language (like C++) is that it allows extensive compile-time checking so bugs can be found early. Unfortunately, C++ inherits a number of loopholes in compile-time checking from C, which is relatively lax (particularly method/procedure declarations). In Java, we require declarations and do not support C-style implicit declarations.

The linker understands the type system and repeats many of the type checks done by the compiler to guard against version mismatch problems.

The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays. This allows subscript checking to be performed. In addition, it is not possible to turn an arbitrary integer into a pointer by casting.

The folks at Archimedes had their application basically working in C pretty quickly. But their schedule kept slipping because of all the small bugs that kept slipping through. They had lots of trouble with memory corruption, versions out-of-sync and interface mismatches. What they gained because C let them pull strange tricks in their code, they paid for in quality assurance time. They also had to reissue their software after the first release because of all the bugs that slipped through.

While Java doesn't make the QA problem go away, it does make it significantly easier.

Very dynamic languages like Lisp, TCL and Smalltalk are often used for prototyping. One of the reasons for their success at this is that they are very robust: you don't have to worry about freeing or corrupting memory. Programmers can be relatively fearless about dealing with memory because they don't have to worry about it getting corrupted. Java has this property and it has been found to be very liberating.

One reason that dynamic languages are good for prototyping is that they don't require you to pin down decisions early on. Java has exactly the opposite property; it forces you to make choices explicitly. Along with these choices come a lot of assistance: you can write method invocations and if you get something wrong, you are informed about it at compile time. You don't have to worry about method invocation error. You can also get a lot of flexibility by using interfaces instead of classes.

Secure

Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems. The authentication technique s are based on public-key encryption.

There is a strong interplay between "robust" and "secure." For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do have access to. This closes the door on most activities of viruses.

Someone wrote an interesting "patch" to the PC version of the Archimedes system. They posted this patch to one of the major bulletin boards. Since it was easily available and added some interesting features to the system, lots of people downloaded it. It hadn't been checked out by the folks at Archimedes, but it seemed to work. Until the next April 1st, when thousands of folks discovered rude pictures popping up in their children's lessons. Needless to say, even though they were in no way responsible for the incident, the folks at Archimedes still had a lot of damage to control.

Architecture Neutral

Java was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system architectures. To enable a Java application to execute anywhere on the network, the compiler generates an architecture neutral object file format -- the compiled code is executable on many processors, given the presence of the Java runtime system.

This is useful not only for networks but also for single system software distribution. In the present personal computer market, application writers have to produce versions of their application that are compatible with the IBM PC and with the Apple Macintosh. With the PC market (through Windows/NT) diversifying into many CPU architectures, and Apple moving off the 68000 towards the PowerPC, this makes the production of software that runs on all platforms almost impossible. With Java, the same version of the application runs on all platforms.

The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

Archimedes is a small company. They started out producing their software for the PC since that was the largest market. After a while, they were a large enough company that they could afford to do a port to the Macintosh, but it was a pretty big effort and didn't really pay off. They couldn't afford to port to the PowerPC Macintosh or MIPS NT machine.

They couldn't "catch the new wave" as it was happening, and a competitor jumped in ...

Portable

Being architecture neutral is a big chunk of being portable, but there's more to it than that. Unlike C and C++, there are no "implementation dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them. For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32-bit IEEE 754 floating point number. Making these choices is feasible in this day and age because essentially all interesting CPU's share these characteristics.

The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows and the Macintosh.[†]

The Java system itself is quite portable. The new compiler is written in Java and the runtime is written in ANSI C with a clean portability boundary. The portability boundary is essentially POSIX.

Interpreted

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. And since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.

As a part of the bytecode stream, more compile-time information is carried over and available at runtime. This is what the linker's type checks are based on, and what the RPC protocol derivation is based on. It also makes programs more amenable to debugging.

The programmers at Archimedes spent a lot of time waiting for programs to compile and link. They also spent a lot of time tracking down senseless bugs because some changed source files didn't get compiled (despite using a fancy "make" facility), which caused version mismatches; and they had to track down procedures that were declared inconsistently in various parts of their programs. Another couple of months lost in the schedule.

High Performance

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. For those accustomed to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader.

The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally simple. Reasonably

† The Windows and Macintosh versions aren't complete yet.

good code is produced: it does automatic register allocation and the compiler does some optimization when it produces the bytecodes.

In interpreted code we're getting about 300,000 method calls per second on an Sun Microsystems SPARCStation 10. The performance of bytecodes converted to machine code is almost indistinguishable from native C or C++.

When Archimedes was starting up, they did a prototype in Smalltalk. This impressed the investors enough that they got funded, but it didn't really help them produce their product: in order to make their simulations fast enough and the system small enough, it had to be rewritten in C.

Multithreaded

There are many things going on at the same time in the world around us. Multithreading is a way of building applications with multiple threads[†] Unfortunately, writing programs that deal with many things happening at once can be much more difficult than writing in the conventional single-threaded C and C++ style.

Java has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm that was introduced by C.A.R.Hoare[‡]. By integrating these concepts into the language they become much easier to use and are more robust. Much of the style of this integration came from Xerox's Cedar/Mesa system.

Other benefits of multithreading are better interactive responsiveness and real-time behavior. This is limited, however, by the underlying platform: stand-alone Java runtime environments have good real-time behavior. Running on top of other systems like Unix, Windows, the Macintosh, or Windows NT limits the real-time responsiveness to that of the underlying system.

Lots of things were going on at once in their simulations. Ropes were being pulled, wheels were turning, levers were rocking, and input from the user was being tracked. Because they had to write all this in a single threaded form, all the things that happen at the same time, even though they had nothing to do with each other, had to be manually intermixed. Using an "event loop" made things a little cleaner, but it was still a mess. The system became fragile and hard to understand. They were pulling in data from all over the net. But originally they were doing it one chunk at a time. This serialized network communication was very slow. When they converted to a multithreaded style, it was trivial to overlap all of their network communication.

Dynamic

In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment.

† Threads are sometimes also called lightweight processes or execution contexts.

‡ 1974. Hoare, C.A.R. *Monitors: An Operating System Structuring Concept*, Comm. ACM **17**, 10:549-557 (October)

For example, one major problem with using C++ in a production environment is a side-effect of the way that code is always implemented. If company A produces a class library (a library of plug and play components) and company B buys it and uses it in their product, then if A changes its library and distributes a new release, B will almost certainly have to recompile and redistribute their own software. In an environment where the end user gets A and B's software independently (say A is an OS vendor and B is an application vendor) problems can result.

For example, if A distributes an upgrade to its libraries then all of the software from B will break. It is possible to avoid this problem in C++, but it is extraordinarily difficult and it effectively means not using any of the language's OO features directly.

Archimedes built their product using the object-oriented graphics library from 3DPC Inc. 3DPC released a new version of the graphics library which several computer manufacturers bundled with their new machines. Customers of Archimedes that bought these new machines discovered to their dismay that their old software no longer worked. (In real life, this only happens on Unix systems. In the PC world, 3DPC would never have released such a library: their ability to change their product and use C++'s object oriented features is severely hindered.)

By making these interconnections between modules later, Java completely avoids these problems and makes the use of the object-oriented paradigm much more straightforward. Libraries can freely add new methods and instance variables without any effect on their clients.

Java understands interfaces— a concept borrowed from Objective C which is similar to a class. An interface is simply a specification of a set of methods that an object responds to. It does not include any instance variables or implementations. Interfaces can be multiply-inherited (unlike classes) and they can be used in a more flexible way than the usual rigid class inheritance structure.

Classes have a runtime representation: there is a class named Class, instances of which contain runtime class definitions. If, in a C or C++ program, you have a pointer to an object but you don't know what type of object it is, there is no way to find out. However, in Java, finding out based on the runtime type information is straightforward. Because casts are checked at both compile-time and runtime, you can trust a cast in Java On the other hand in C and C++, the compiler just trusts that you're doing the right thing.

It is also possible to look up the definition of a class given a string containing its name. This means that you can compute a data type name and have it easily dynamically-linked into the running system.

To expand their revenue stream, the folks at Archimedes wanted to architect their product so that new aftermarket plug-in modules could be added to extend the system. This was possible on the PC, but just barely. They had to hire a couple of new programmers because it was so complicated. This also added problems when debugging.

Summary

The Java language provides a powerful addition to the tools that programmers have at their disposal. Java makes programming easier because it is object-oriented and has automatic garbage collection. In addition, because compiled Java code is architecture-neutral, Java applications are ideal for a diverse environment like the Internet. For more information send mail to java@java.sun.com.

The Java™ *Language: An Overview*

Introduction

The Java programming language and environment is designed to solve a number of problems in modern programming practice. It started as a part of a larger project to develop advanced software for consumer electronics. These devices are small, reliable, portable, distributed, real-time embedded systems. When we started the project, we intended to use C++, but we encountered a number of problems. Initially these were just compiler technology problems, but as time passed we encountered a set of problems that were best solved by changing the language.

The companion paper, *The HotJava™ Browser: An Overview*, describes a powerful application that demonstrates the power of the Java language.

Java

> Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.

One way to characterize a system is with a set of buzzwords. We use a standard set of them in describing Java. The rest of this section is an explanation of what we mean by those buzzwords and the problems that we were trying to solve.

Archimedes Inc. is a fictitious software company that produces software to teach about basic physics. This software is designed to interact with the user, providing not only text and illustrations in the manner of a traditional textbook, but also providing a set of software lab benches on which experiments can be set up and their behavior simulated. For example, the most basic one allows students to put together levers and pulleys and see how they act. The italicized narrative of the trials and tribulations of the Archimedes' designers is used here to provide examples of the language concepts presented.

Simple

We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. Most programmers working these days use C, and most programmers doing object-oriented programming use C++. So even though we found that C++ was unsuitable, we designed Java as closely to C++ as possible in order to make the system more comprehensible.

Java omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. These omitted features primarily consist of operator overloading (although the Java language does have method overloading), multiple inheritance, and extensive automatic coercions.

We added auto garbage collection thereby simplifying the task of Java programming but making the system somewhat more complicated. A good example of a common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. By virtue of having automatic garbage collection the Java language not only makes the programming task easier, it also dramatically cuts down on bugs.

The folks at Archimedes wanted to spend their time thinking about levers and pulleys, but instead spent a lot of time on mundane programming tasks. Their central expertise was teaching, not programming. One of the most complicated of these programming tasks was figuring out where memory was being wasted across their 20K lines of code.

Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K bytes; adding the basic standard libraries and thread support (essentially a self-contained microkernel) adds an additional 175K.

Object-Oriented

This is, unfortunately, one of the most overused buzzwords in the industry. But object-oriented design is very powerful because it facilitates the clean definition of interfaces and makes it possible to provide reusable "software ICs."

Simply stated, object-oriented design is a technique that focuses design on the data (=objects) and on the interfaces to it. To make an analogy with carpentry, an "object-oriented" carpenter would be mostly concerned with the chair he was building, and secondarily with the tools used to make it; a "non-object-oriented" carpenter would think primarily of his tools. Object-oriented design is also the mechanism for defining how modules "plug and play."

The object-oriented facilities of Java are essentially those of C++, with extensions from Objective C for more dynamic method resolution.

The folks at Archimedes had lots of things in their simulation, among them, ropes and elastic bands. In their initial C version of the product, they ended up with a pretty big system because they had to write separate software for describing ropes versus elastic bands. When they rewrote their application in an object-oriented style, they found they could define one basic object that represented the common aspects of ropes and elastic bands, and then ropes and elastic bands were defined as variation s (subclasses) of the basic type. When it came time to add chains, it was a snap because they could build on what had been written before, rather than writing a whole new object simulation.

Distributed

Java has an extensive library of routines for coping easily with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the net via URLs with the same ease that programmers are used to when accessing a local file system.

The folks at Archimedes initially built their stuff for CD ROM. But they had some ideas for interactive learning games that they wanted to try out for their next product. For example, they wanted to allow students on different computers to cooperate in building a machine to be simulated. But all the networking systems they'd seen were complicated and required esoteric software specialists. So they gave up.

Robust

Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error prone.

One of the advantages of a strongly typed language (like C++) is that it allows extensive compile-time checking so bugs can be found early. Unfortunately, C++ inherits a number of loopholes in compile-time checking from C, which is relatively lax (particularly method/procedure declarations). In Java, we require declarations and do not support C-style implicit declarations.

The linker understands the type system and repeats many of the type checks done by the compiler to guard against version mismatch problems.

The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays. This allows subscript checking to be performed. In addition, it is not possible to turn an arbitrary integer into a pointer by casting.

The folks at Archimedes had their application basically working in C pretty quickly. But their schedule kept slipping because of all the small bugs that kept slipping through. They had lots of trouble with memory corruption, versions out-of-sync and interface mismatches. What they gained because C let them pull strange tricks in their code, they paid for in quality assurance time. They also had to reissue their software after the first release because of all the bugs that slipped through.

While Java doesn't make the QA problem go away, it does make it significantly easier.

Very dynamic languages like Lisp, TCL and Smalltalk are often used for prototyping. One of the reasons for their success at this is that they are very robust: you don't have to worry about freeing or corrupting memory. Programmers can be relatively fearless about dealing with memory because they don't have to worry about it getting corrupted. Java has this property and it has been found to be very liberating.

One reason that dynamic languages are good for prototyping is that they don't require you to pin down decisions early on. Java has exactly the opposite property; it forces you to make choices explicitly. Along with these choices come a lot of assistance: you can write method invocations and if you get something wrong, you are informed about it at compile time. You don't have to worry about method invocation error. You can also get a lot of flexibility by using interfaces instead of classes.

Secure

Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems. The authentication technique s are based on public-key encryption.

There is a strong interplay between "robust" and "secure." For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do have access to. This closes the door on most activities of viruses.

Someone wrote an interesting "patch" to the PC version of the Archimedes system. They posted this patch to one of the major bulletin boards. Since it was easily available and added some interesting features to the system, lots of people downloaded it. It hadn't been checked out by the folks at Archimedes, but it seemed to work. Until the next April 1st, when thousands of folks discovered rude pictures popping up in their children's lessons. Needless to say, even though they were in no way responsible for the incident, the folks at Archimedes still had a lot of damage to control.

Architecture Neutral

Java was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system architectures. To enable a Java application to execute anywhere on the network, the compiler generates an architecture neutral object file format -- the compiled code is executable on many processors, given the presence of the Java runtime system.

This is useful not only for networks but also for single system software distribution. In the present personal computer market, application writers have to produce versions of their application that are compatible with the IBM PC and with the Apple Macintosh. With the PC market (through Windows/NT) diversifying into many CPU architectures, and Apple moving off the 68000 towards the PowerPC, this makes the production of software that runs on all platforms almost impossible. With Java, the same version of the application runs on all platforms.

The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

Archimedes is a small company. They started out producing their software for the PC since that was the largest market. After a while, they were a large enough company that they could afford to do a port to the Macintosh, but it was a pretty big effort and didn't really pay off. They couldn't afford to port to the PowerPC Macintosh or MIPS NT machine.

They couldn't "catch the new wave" as it was happening, and a competitor jumped in ...

Portable

Being architecture neutral is a big chunk of being portable, but there's more to it than that. Unlike C and C++, there are no "implementation dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them. For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32-bit IEEE 754 floating point number. Making these choices is feasible in this day and age because essentially all interesting CPU's share these characteristics.

The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows and the Macintosh.†

The Java system itself is quite portable. The new compiler is written in Java and the runtime is written in ANSI C with a clean portability boundary. The portability boundary is essentially POSIX.

Interpreted

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. And since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.

As a part of the bytecode stream, more compile-time information is carried over and available at runtime. This is what the linker's type checks are based on, and what the RPC protocol derivation is based on. It also makes programs more amenable to debugging.

The programmers at Archimedes spent a lot of time waiting for programs to compile and link. They also spent a lot of time tracking down senseless bugs because some changed source files didn't get compiled (despite using a fancy "make" facility), which caused version mismatches; and they had to track down procedures that were declared inconsistently in various parts of their programs. Another couple of months lost in the schedule.

High Performance

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. For those accustomed to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader.

The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally simple. Reasonably

† The Windows and Macintosh versions aren't complete yet.

good code is produced: it does automatic register allocation and the compiler does some optimization when it produces the bytecodes.

In interpreted code we're getting about 300,000 method calls per second on an Sun Microsystems SPARCStation 10. The performance of bytecodes converted to machine code is almost indistinguishable from native C or C++.

When Archimedes was starting up, they did a prototype in Smalltalk. This impressed the investors enough that they got funded, but it didn't really help them produce their product: in order to make their simulations fast enough and the system small enough, it had to be rewritten in C.

Multithreaded

There are many things going on at the same time in the world around us. Multithreading is a way of building applications with multiple threads[†] Unfortunately, writing programs that deal with many things happening at once can be much more difficult than writing in the conventional single-threaded C and C++ style.

Java has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm that was introduced by C.A.R.Hoare[‡]. By integrating these concepts into the language they become much easier to use and are more robust. Much of the style of this integration came from Xerox's Cedar/Mesa system.

Other benefits of multithreading are better interactive responsiveness and real-time behavior. This is limited, however, by the underlying platform: stand-alone Java runtime environments have good real-time behavior. Running on top of other systems like Unix, Windows, the Macintosh, or Windows NT limits the real-time responsiveness to that of the underlying system.

Lots of things were going on at once in their simulations. Ropes were being pulled, wheels were turning, levers were rocking, and input from the user was being tracked. Because they had to write all this in a single threaded form, all the things that happen at the same time, even though they had nothing to do with each other, had to be manually intermixed. Using an "event loop" made things a little cleaner, but it was still a mess. The system became fragile and hard to understand. They were pulling in data from all over the net. But originally they were doing it one chunk at a time. This serialized network communication was very slow. When they converted to a multithreaded style, it was trivial to overlap all of their network communication.

Dynamic

In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment.

† Threads are sometimes also called lightweight processes or execution contexts.

‡ 1974. Hoare, C.A.R. *Monitors: An Operating System Structuring Concept*, Comm. ACM **17**, 10:549-557 (October)

For example, one major problem with using C++ in a production environment is a side-effect of the way that code is always implemented. If company A produces a class library (a library of plug and play components) and company B buys it and uses it in their product, then if A changes its library and distributes a new release, B will almost certainly have to recompile and redistribute their own software. In an environment where the end user gets A and B's software independently (say A is an OS vendor and B is an application vendor) problems can result.

For example, if A distributes an upgrade to its libraries then all of the software from B will break. It is possible to avoid this problem in C++, but it is extraordinarily difficult and it effectively means not using any of the language's OO features directly.

Archimedes built their product using the object-oriented graphics library from 3DPC Inc. 3DPC released a new version of the graphics library which several computer manufacturers bundled with their new machines. Customers of Archimedes that bought these new machines discovered to their dismay that their old software no longer worked. (In real life, this only happens on Unix systems. In the PC world, 3DPC would never have released such a library: their ability to change their product and use C++'s object oriented features is severely hindered.)

By making these interconnections between modules later, Java completely avoids these problems and makes the use of the object-oriented paradigm much more straightforward. Libraries can freely add new methods and instance variables without any effect on their clients.

Java understands interfaces— a concept borrowed from Objective C which is similar to a class. An interface is simply a specification of a set of methods that an object responds to. It does not include any instance variables or implementations. Interfaces can be multiply-inherited (unlike classes) and they can be used in a more flexible way than the usual rigid class inheritance structure.

Classes have a runtime representation: there is a class named Class, instances of which contain runtime class definitions. If, in a C or C++ program, you have a pointer to an object but you don't know what type of object it is, there is no way to find out. However, in Java, finding out based on the runtime type information is straightforward. Because casts are checked at both compile-time and runtime, you can trust a cast in Java On the other hand in C and C++, the compiler just trusts that you're doing the right thing.

It is also possible to look up the definition of a class given a string containing its name. This means that you can compute a data type name and have it easily dynamically-linked into the running system.

To expand their revenue stream, the folks at Archimedes wanted to architect their product so that new aftermarket plug-in modules could be added to extend the system. This was possible on the PC, but just barely. They had to hire a couple of new programmers because it was so complicated. This also added problems when debugging.

Summary

The Java language provides a powerful addition to the tools that programmers have at their disposal. Java makes programming easier because it is object-oriented and has automatic garbage collection. In addition, because compiled Java code is architecture-neutral, Java applications are ideal for a diverse environment like the Internet. For more information send mail to java@java.sun.com.

