Java for COBOL Programmers

Presentation #200

Mike Yawn
Java Architect
HP Commercial Systems Division
19447 Pruneridge Avenue Mailstop 47UA
Cupertino, CA 95014
Phone (408) 447-4367
Fax (408) 447-4441
myawn@cup.hp.com

You've all heard about Java. It's the current Big Thing in software development. It solves once and for all the issue of software portability, so everyone involved in the whole software standards industry can just fold their tents and go home now. Java programs are guaranteed to be free of memory leaks, dangling pointers, uninitialized variables, and array subscripting errors, so everyone involved in software support or maintenance activities, your days are numbered as well. Java will replace all other languages, so if you're a programmer and don't know Java, well, you aren't really a programmer then, are you?

Heard any of this before? I don't just mean the Java hype; have you heard it applied to UNIX (universal software portability), structured programming methodologies (bug-free code), or insert-your-favorite-language here (will replace COBOL and everything else)? Java isn't unique because of the things it attempts to do (though it might be unique in the sheer number of problems it claims to solve all at once). And it isn't unique in being hyped beyond its actual capabilities. But I do believe Java is unique in how well it delivers on each of these promises: portability, reliability, and suitability to real-world problems.

We're going to assume up front that you've bought into the premise that Java can do something for you, while at the same time being sensible enough to know it isn't the answer to every problem. So we'll skip over the sales pitch, and begin the task of making you a Java programmer.

That's not going to happen in an hour. In fact, if you listen to some industry analysts, they'll tell you it's not going to happen at all: that COBOL programmers are fundamentally incapable of grasping object-oriented concepts and becoming Java programmers. That's wrong. Depending on how much you adopt the 'native Java mindset', you may be better or worse at Java than you are at COBOL, and you may be better or worse than Java programmers who came from other backgrounds. A C or C++ programmer may have an advantage over you when it comes to grasping the language syntax quickly. But you may have an advantage over that C or C++ programmer in knowing how to write code that effectively solves a business problem.

So, we will now proceed to turn COBOL programmers into Java programmers in 5 easy steps. Only the first two steps will be covered in detail; the other steps will be your homework assignments.

Step One: Find the Caps Lock key, and disengage it.

OK, I wanted to start with something simple yet profound. Java is a case-sensitive language. While it is theoretically possible to write a Java program in all upper case, or in all lower case, anyone who does so receives a failing grade for the course and is encouraged to leave the ranks of Java programmers immediately.

The use of upper and lower case in Java tells us things. These are conventions; by which I mean the compiler is not going to complain if you use case differently. But the humans who have to read your code will complain; we're used to the visual cues that case provides to tell us what role is being played by the various names we use. Consider the following Java statement:

```
    MyClass myClass = new MyClass();
```

The identifier myclass – if we ignore case and punctuation for just a moment – appears three times in the above statement. It is actually three different identifiers. Only the case and punctuation differ; and these differences tell us something about how the identifier is being used in each case. (The position of the identifiers could also be used as a clue to their function, but in other contexts it might not be as clear from position alone).

The first occurrence – MyClass – is as a type. COBOL doesn't allow user-defined types, so this may be an unfamiliar concept. There is some analogue in a COBOL picture clause, although a picture clause applies to only a single data item, where a type definition can describe a complete complex structure. A type definition in C, C++, Pascal, or Java allows us to describe how a structured variable (like a COBOL 01 level item) will look, without declaring the variable. The type name can then be used over and over again within the program to create as many variables as needed of the same type, without having to repeat the description of the layout. By convention, the names of classes (which are data types) begin with a capital letter, and then each subsequent word in the identifier is also capitalized. While it is legal to use dashes or underscores in a name, the use of infix caps accomplishes the same thing. So if you want your code to look like it was developed by a knowledgeable Java developer, leave out the dashes and underscores and let capitalization work for you. So we know now that the line we're examining is a variable declaration, and the type of the variable is MyClass.

The second occurrence – myClass – is as a variable name, which functions much like the name of a COBOL 01 record. In this case, we're declaring an instance (we'll define that term later) of the class MyClass. By convention, instance names begin with a lower case letter, and then each subsequent word is capitalized as with type names. Using class names and instance names that vary only in the capitalization of their initial letter is a very common idiom in Java; rather than being confusing, it should help you to easily see what the type is of the object without having to refer back to the type declaration. But because this is a convention only, not enforced by the language, you can imagine how ignoring the convention can create confusion.

The third occurrence – MyClass() – is a **method** call. (Methods are java-ese for functions or procedures; think of a COBOL call to an MPE intrinsic). The parentheses tell us that this is a method call. If there were any parameters to be passed to the method, they would be listed inside the parentheses. Normally, method calls follow the same capitalization rules as instances – with a lower-case initial letter. However, in this case we have an exception to the rule. The method MyClass() is a constructor, which is a special-purpose method used to perform initialization when an object is constructed. Constructor names must be identical to the class name that they initialize, so this constructor must start with an upper-case letter since that is how the class is named.

A final capitalization rule worth noting – constants (which are called 'static final' variables in Java) are generally named in all upper case. COBOL 88 level condition names provide a similar capability in some cases. However, Java static final variables can be used anywhere that a variable or constant expression is allowed, while the use of COBOL condition names is more restricted.

Step Two: Learn the key concepts of Object-Oriented Programming

Object-Oriented programming is very different from COBOL programming, except for the fact that it's mostly the same thing. It's one of those "half-full or half-empty?" choices; if you want to emphasize the things that are different, you can make object oriented programming look radically new and different. But if you choose to emphasize the things that are the same, you'll feel very much on familiar ground. One of the things that changes a lot is the terminology; yet the concepts underlying the new terminology all have analogues in procedural programming. Although I've covered the concepts and terminology in previous HPWorld presentations, I feel it's important enough to bear repeating. This time through, we'll try to put even more emphasis on drawing parallels with familiar COBOL concepts. If you can master the new terminology, you're won half the battle.

The three key concepts we'll introduce are encapsulation, inheritance, and polymorphism. We'll drag in some more terminology (such as class and instance referred to earlier) along the way.

Any object-oriented programming class will start with the assertion ‘Everything is an Object’. Understanding what an object is thus becomes the first conceptual hurdle that the COBOL programmer must overcome on the way to object nirvana. Java purists will no doubt cringe at my assertion that an object is like an 01-level data structure in COBOL. This statement is more accurate than not, so we’ll use it as a starting point and refine the definition as we go.

Like a COBOL 01-level structure, a Java object is a collection of data items. There are two key distinctions we’ll introduce right away. The COBOL structure frequently mirrors the actual way data is laid out somewhere, such as a file or database record. The Java Object merely states that the named data items are part of the object, but makes no guarantee of how memory or storage space will be allocated for them, how they will be ordered, etc. The second, and more important, difference is that a Java object contains not just data, but the code that is used to operate on the data. In COBOL, we force a separation between data (in the DATA DIVISION) and code (in the PROCEDURE DIVISION). In Java, we force a union of these elements, putting the code that operates on a particular type of data together with the description of what that data looks like.

We can take this analogy a little further to demonstrate the difference between a 'class' and an 'instance', which are two terms that are each occasionally used interchangeably with the term 'object'. A COBOL 01 level data item plays the role of both ‘class’ and ‘instance’ in object terms, which makes it challenging to distinguish these concepts. A class is like a template, or blueprint, that describes what an object will look like. An instance is an object built from that template or blueprint, with its unique data. Objects that vary only in the values of their data fields are instances of the same class; if objects were built from different blueprints, then they are of different classes. In Java we can use one template to create as many actual ‘instances’ of that type as we want, unlike COBOL where you can only have one of each 01 level record. When the COBOL program is run, data will be loaded into the 01 level entry. If it is a FILE SECTION entry, then the data structure will receive a new set of values each time a new record is read from the file. Each time new values are loaded into the structure, we reuse the instance to hold the new values. Once new values are read in, the old data is lost. In Java, depending on how the program is constructed, you may have the same behavior, or you may create multiple instances of the class for each new set of data, up to the limit of available memory.

We're now ready for our first key concept – **encapsulation**. In the COBOL world, the same concept has been around for years, known as data hiding. The concept is simple – provide a common set of access routines that are used to access a particular data object. Only these routines can be used; no other code is allowed to access the data directly. By doing this, if the underlying data format is ever changed, only the access routines need to be updated; the change will be invisible to anyone using the approved access routines. Data Hiding is frequently a goal of procedural programmers, although many times data hiding is broken either due to a perceived performance penalty or just sloppy programming. Java provides mechanisms to enforce data hiding – the data can be marked ‘private’, and the access methods marked ‘public’, which will prevent any calls made from outside the object from accessing the data except through the supported interface.

Encapsulation is the process of sealing up your data in an impenetrable capsule, and then providing the necessary ‘ports’ (methods) to allow data in and out in an approved fashion. Think of it like windows at the DMV. If you want access to your vehicle registration information, you have to present your request to a worker at a window rather than just jumping over the counter and rummaging through the records yourself. Thus if the DMV changes its records system, you don’t need to change the way you use the DMV.

We move to our next new concept, **inheritance**. Inheritance is an unfamiliar solution to a familiar problem. Frequently, our data structures have some degree of variance in them – the layout of the data items is not quite the same in all cases. In COBOL, we address this through the REDEFINES clause, which allows us to provide alternate schemas, if you will, for part or all of a data structure.

[Note: there are at least three different reasons for using REDEFINES in COBOL. One use is to cause the same memory area to be reused for data structures that may be otherwise unrelated. If the programmer knows that only one of these data areas is in use at any time, we can avoid allocating memory for each one individually and instead have them share a block of memory. This isn't an issue in Java, since memory is allocated dynamically when needed rather than statically at program load time, and the garbage collector will recover memory from an object once it is no longer needed. A second use is to force a type conversion, e.g., redefining an X type (character) field as a 9 type (numeric). In Java, there are explicit type conversion methods you must use to accomplish this. The third use is to provide multiple layouts for a particular data record or portion of a data record; it is this third use that corresponds to Java inheritance.]

The basic premise behind this use of REDEFINES is that for part of the data structure, there are alternate versions. Object-oriented languages such as Java provide a similar capability through inheritance. Take the portion of the structure that is invariant, and create a class to represent it. This is called the base class. Now create a new class, which will be a subclass of the base class. Add to the subclass the items that are unique to this variant. If there is more than one variant, you can create more than one subclass to represent them. For example, consider the following COBOL representation of an Employee record with alternate forms of recording payroll information:

```
01 EMPLOYEE-RECORD
      05 EMPLOYEE-NUMBER            PIC X(08).
      05 FIRST-NAME                 PIC X(20).
      05 LAST-NAME                  PIC X(20).
      05 PAY-METHOD                 PIC X.
      05 SALARY-INFO.
            10 ANNUAL-SALARY        PIC S9(6)V99 USAGE COMP.
      05 HOURLY-INFO REDEFINES SALARY-INFO.
            10 HOURLY-RATE          PIC S9(4)V99 USAGE COMP.
            10 HOURS-REGULAR        PIC S9(4)V99 USAGE COMP.
            10 HOURS-OVERTIME       PIC S9(4)V99 USAGE COMP.
            10 OVERTIME-RATE        PIC S9(4)V99 USAGE COMP.
      05 COMMISSION-INFO REDEFINES SALARY-INFO.
            10 COMMISSIONED-SALES   PIC S9(8)V99 USAGE COMP.
            10 COMMISSION-RATE      PIC S9(2)V99 USAGE COMP.
```

There are several ways to accomplish this in Java. For starters, here is a base class with three subclasses. Note that we're only showing the data here; in a real Java implementation of these classes, the methods used to access the data would also be part of the appropriate classes. (There are valid reasons to create a data-only class in Java, but if you have data that will be accessed by more than one program it is best to provide the access methods as part of the class.)

```
// Employee.java
public class Employee {
      private String employeeNumber;
      private String firstName;
      private String lastName;
}
// SalariedEmployee.java
public class SalariedEmployee extends Employee {
      private float annualSalary;
}
// HourlyEmployee.java
public class HourlyEmployee extends Employee {
      private float hourlyRate;
      private float hoursRegular;
      private float hoursOvertime;
      private float overtimeRate;
}
// CommissionedEmployee.java
```

```
public class CommissionedEmployee extends Employee {
      private float commissionedSales;
      private float commissionRate;
}
```

It is important to realize that the subclasses, such as HourlyEmployee, will each contain all of the data fields named in the base class - thus, there is an employeeName field in the HourlyEmployee class, even though it doesn't show up in the declaration of HourlyEmployee. The clause 'extends Employee' on the class declaration line tells us we need to look in the Employee class definition to find additional data and methods that are also part of this subclass. Inheritance is a powerful mechanism for representing similar yet distinct objects. However, it tends to be overused by beginning object-oriented programmers. If you look hard enough, you can find some commonality between just about any two objects, but that doesn’t mean that they should be part of the same inheritance hierarchy. Frequently, two distinct objects merely need to share a small number of data items or methods. There are other ways besides inheritance to accomplish this. The simplest way is through **aggregation**. Objects can contain other objects, as well as simple data types, as members. So related groupings of fields within a data record can be used to assemble an object, and then these objects can themselves be put together to create other objects. Here's the sample example reworked to use aggregation rather than inheritance.

```
public class SalaryInfo {
      private float annualSalary;
}
public class HourlyInfo {
      private float hourlyRate;
      private float hoursRegular;
      private float hoursOvertime;
      private float overtimeRate;
}
public class CommissionInfo {
      private float commissionedSales;
      private float commissionRate;
}
public class Employee {
      private String employeeNumber;
      private String firstName;
      private String lastName;
      private SalaryInfo salaryInfo;
      private HourlyInfo hourlyInfo;
      private CommissionInfo commissionInfo;
}
```

In this implementation, we’ve avoided creating three separate classes of Employee, instead focusing on the part of the record that is actually different – the pay information. However, the implementation above has changed some of the basic implementation choices from the original code. For example, in the original code, an employee had either Salary, Hourly, or Commission info, where in this example an employee can have all three. (Of course, two of the items could be empty for any given employee). We can mirror the original design more closely by bringing inheritance back into the picture, but at the pay information level rather than at the Employee level. This design will combine inheritance and aggregation in the same object. Create a new base class called PayInfo, and have our SalaryInfo, HourlyInfo, and CommissionInfo classes extend the new base class. Then, in our Employee class, rather than three separate fields for the possible info types, we can create a single PayInfo field that will be able to contain an instance of any of the three subclasses.

```
public class PayInfo {
      /* no shared data or methods in this class */
}
```

```
public class SalaryInfo extends PayInfo{
      private float annualSalary;
}
public class HourlyInfo extends PayInfo {
      private float hourlyRate;
      private float hoursRegular;
      private float hoursOvertime;
      private float overtimeRate;
}
public class CommissionInfo extends PayInfo{
      private float commissionedSales;
      private float commissionRate;
}
public class Employee {
      private String employeeNumber;
      private String firstName;
      private String lastName;
      private PayInfo payInfo;
}
```

Up to this point, we've only been looking at the data part of each object. It's important to remember that each of these objects can also contain code, and achieving code reusability is one of the strongest reasons for using an object-oriented design. Code can be handled exactly like data -- methods can be added to the base class, where they will automatically be inherited by all the subclasses, or the methods can be added to the subclasses. Let's look at possible ways this could work. First, we could add a calculatePay() method to the base class (Employee). But the logic to compute pay for the three different classes of employees becomes cumbersome; we'd need some sort of if-then test to determine which method to use. Also, any changes to any of the three algorithms, or the addition of a new class of Employee, requires changes to be made to the base class. A second alternative would be to put the calculatePay() logic in the subclasses. So we could add a calculateSalariedPay() to the SalariedEmployee class, calculateHourlyPay() to the HourlyEmployee class, and so on. This puts the logic at the appropriate level, but creates a new problem. Since the methods don't have the same name, the calling programs that need to cause the calculations to occur must have explicit knowledge of each employee class, and what the corresponding pay method is named. Ideally, we'd like to be able to keep the name calculatePay() for all three methods -- and in fact, Java allows us to do this, providing two different means of doing so. This makes it transparent to the caller what class of Employee we're dealing with, since the external interface used to invoke the calculation is the same in all cases.

The first way to do this is to declare an abstract method in the base class. An abstract method is essentially a placeholder; it names the method and defines its parameters, but does not implement the method. Any class that includes an abstract method is incomplete, and cannot be ***instantiated*** (instantiation is the process of creating an instance. As all programmers know, any noun can be verbed). Only the subclasses that provide true implementations of the abstract method can be instantiated. Here's one way these methods could be implemented:

```
public class Employee {
      /* data items as before, not shown */
      abstract float calculatePay();
}
public class SalariedEmployee extends Employee {
      /* data items as before, not shown */
      float calculatePay() {
            /* assumes semi-monthly pay */
            return annualSalary / 24;
      }
}
public class HourlyEmployee extends Employee {
      /* data items as before, not shown */
      float calculatePay() {
            return (hourlyRate * hoursRegular) +
                  (overtimeRate * hoursOvertime);
      }
}
/* CommissionedEmployee is left as an exercise for the reader */
```

An abstract method is a good design choice when the following conditions are met:

1. There is no need to ever instantiate the base class.
2. Every subclass will have a different implementation of the method.
3. All of the classes that need to implement the method are descended from the same base class.

If the first two conditions are not met, a better alternative may be to provide a method implementation in the base class, and override it in the subclasses that require a different implementation. This taps in to the power of inheritance; allowing the most-frequently-used variant of the method to be defined in the base class and be automatically inherited by subclasses. Any subclasses that require something different merely implement their own calculatePay() method, which will then be used in place of the base class version for instances of that subclass. Here is sample code showing overriding the base class:

```
public class Employee {
      protected float annualSalary;
      public float calculatePay() {
            return annualSalary / 24;
      }
}
public class SalariedEmployee extends Employee {
      /* nothing needed; base class method works fine for us */
}
public class HourlyEmployee extends Employee {
      private float hourlyRate;
      private float hoursRegular;
      private float hoursOvertime;
      private float overtimeRate;
      public float calculatePay() {
            return (hourlyRate * hoursRegular) +
                  (overtimeRate * hoursOvertime);
      }
}
```

We've taken the data items and calculatePay() method from the SalariedEmployee class and moved them to the base class. This means that the SalariedEmployee class has no need to declare these data items and methods, since they will be inherited automatically. It has been assumed in writing these examples that we're just looking at a small subset of the data items and methods of each class; if in fact this was the entire class, then we would no longer have a need for the SalariedEmployee class at all, since the base class is now effectively a SalariedEmployee. In the HourlyEmployee subclass, we override calculatePay() with a method to handle hourly employees. We add the private fields needed by this class. Note that in the base class, we have changed annualSalary from private to protected. Private data is not accessible by subclasses; protected data is. If we truly need to keep SalariedEmployee, it should be allowed to access the annualSalary field. If instead we decide that the SalariedEmployee class can be dropped from our design, we can leave annualSalary private, since none of the other subclasses really need to access it.

If the third condition for using an abstract method is not met, you may want to use an ***interface*** rather than an abstract or overridden method. An interface is similar to an abstract class, in that it specifies a method name and parameter types without providing the implementation. The difference is that interfaces can be attached to any class, thus they work separately from the inheritance mechanism. If we made the calculatePay() method part of an interface, we could specify the interface on the Employee base class, which would automatically cause the interface to be required for the subclasses as well. But we could also attach the interface to classes that are not derived from Employee, such as a Contractor class, for example.

```
public interface Payee {
      public float calculatePay();
}
public class Employee implements Payee {
      /* data items as before */
      /* can either implement calculatePay() here, then
         override in subclasses as needed, or declare
         abstract here and implement in all subclasses
       */
}
public class Contractor implements Payee {
      /* data items appropriate for this class */
      /* calculatePay() for contractor implemented here */
}
```

An interface doesn't provide any code. Instead, it is an indicator to users of the Employee and Contractor classes that those classes implement all methods defined in the Payee interface. In this example, the interface includes only a single method, but interfaces can be and frequently are much more complex.

We've spent a lot of time on inheritance, but it is the central concept around which much of the software reuse capability of object-oriented languages is built. Don't expect everything to be crystal clear if this is your first exposure to the concept. Only after seeing lots of examples, and trying to implement a few things yourself, will you begin to think in ways that make inheritance seem like a natural tool for program design.

Our final new concept in the object-oriented world is **polymorphism**, which we've actually shown a little of without explaining it. We have hinted around the problem that in an inheritance hierarchy such as we have created for the Employee class, users of the class don't want to have to be concerned with whether they are dealing with a salaried, hourly, or commissioned employee. Or more precisely, in the cases where it matters, we want to be able to determine which type of Employee we are dealing with, but in cases where it doesn't matter, we don't want to be bothered.

We've shown a number of ways that the EMPLOYEE-RECORD structure from the original COBOL listing could be implemented in Java. Now let's see an example of how it could be used:

```
public class PayThem {
      EmployeeList workers;
      Employee employee;
      Payroll  payroll;

      public static void main(String[] args) {
            boolean moreEmployees = true;
            float amountToPay;

            /* conveniently ignore how EmployeeList and
             * Payroll objects are created and where they
             * get their data from; that's outside the
             * scope of this example
             */
```

```
                while (moreEmployees) {
                      employee = workers.getNext();
                      amountToPay = employee.calculatePay();
                      payroll.add(employee, amountToPay);
                }
          }
    }
```

This code assumes the existence of some other classes and methods that we haven't actually developed, but that are shown just to demonstrate how our Employee class (and its subclasses) might be used. The code is far simpler than would be required for any real payroll application, and exists only to show the use of our Employee class.

It is polymorphism that gives us the ability to declare a data type of type Employee, and then assign to it instances of the subclasses SalariedEmployee, HourlyEmployee, or CommissionedEmployee. Or, in the combined inheritance/aggregation example, we could declare a variable to be of type PayInfo, and assign any of the three PayInfo subclasses to the variable without getting a type mismatch error. The interesting part of polymorphism is that we separate the compile-time declaration from the run-time type. So at compile time, we can just specify the data type as PayInfo or Employee. But at run time, when we call the calculatePay() method of an Employee, we will get the right code depending on the actual instance of the object that is being operated on.

We've explored many different implementations of the Employee class and its subclasses. What changes do we have to make to our PayThem class depending on whether we are using inheritance, aggregation, abstract classes, or interfaces? None whatsoever. The PayThem class will work equally well with any of the implementations we have shown. This is the benefit of encapsulation -- the caller doesn't have to be concerned with the implementation details of the methods being called. Encapsulation could not have been this complete without polymorphism, which allows us to have an Employee data declaration in PayThem that can handle instances of any of Employee's subclasses. And it is inheritance that allowed us to create the hierarchies of base classes and subclasses, distributing data and methods between them in any number of ways depending on our needs.

Step Three: Learn the language syntax

The remaining steps make up your homework assignment. Now that you have a reasonably good theoretical background to build on, you're ready to tackle Java, The Language. There are so many good resources available in this area that I won't try to cover the same content. Some of my favorite resources are Thinking in Java (Bruce Eckel, Prentice Hall PTR) and Java in a Nutshell (David Flanagan, O'Reilly). The definitive source is the Java Language Specification, available in book form or available online (http://java.sun.com/docs/books/jls/index.html), but it's rather dry reading. Another good source is the Java Tutorial, also available either as a book or online (http://java.sun.com/docs/books/tutorial/index.html).

Are there any conceptual hurdles for the COBOL programmer that won't be addressed in these works, which are primarily focused on programmers coming from a C/C++ background? Perhaps a few. Without getting to deeply into the details, here's a very high-level overview over some of the things that it may be assumed you already know:

- Memory for objects is dynamically allocated when they are created, whereas storage for all your COBOL 01 level 'objects' is allocated at program load time. This is especially convenient in the case of arrays, where you can decide what size array you need at run time, rather than having to compile it in as a constant value. (Sure, OCCURS .. DEPENDING ON gives you some of this capability, but the Java syntax is more flexible).
- Control flow is quite different syntactically, since there is no PERFORM and no GOTO. However, the basic control structures in Java -- the for loop, the while loop, the case statement -- are so widely used

in other languages that you'll probably already find them familiar. If not, it shouldn't take long to adapt.

Step Four: Learn (some of) the core classes

The real power of Java comes from how many truly useful things are already programmed for you in the core classes and the many additional APIs. For a parallel, it's quite nice that COBOL already knows how to sort, so you don't have to write your own sort routines. And it has string search and replace code so that you don't have to write that code yourself, either. Add to this the subsystems that MPE provides for keyed file access, database access, form design and control, and there is a tremendous library of code available to the COBOL programmer that is ready-to-use.

Java provides an even larger standard library that saves you from having to code many commonly needed objects. Just a very brief sampling:

- The java.io package provides Files and the various classes required to create, read, write, and perform other common operations on them.
- The java.net package provides support for Sockets, InetAddresses, URLs, and the code needed to deal with objects of these types. Opening a remote URL and reading the data from a system across the Internet is just as easy (and shares much syntax with) opening a local file.
- The java.rmi package provides Remote Method Invocation, making it quite simple to write distributed applications.
- The java.sql package provides access to SQL databases, when used with a JDBC driver.
- The java.text package provides localization capabilities.
- The java.util package includes collection types such as Arrays, BitSets, HashTables, LinkedLists, Sets, Stacks, Trees, and Vectors. The collection types support common operations such as sort, binary search, min, max, reverse order, shuffle, and copy.
- The java.util.zip package provides the ability to create, read, and write ZIP and GZIP format files.
- The java.awt and javax.swing hierarchies provide everything you need to build robust Graphical User Interfaces.

There are 60 packages in the Java 2 Platform (formerly known as JDK 1.2), so we've only scratched the surface with the list above.

Step Five: Write Java code!

Programming is an activity we learn by doing. No matter how well you think you understand what you've read, when you try to do it yourself you'll find that you have questions that haven't been answered. And while you could no doubt find the answers by digging through books or papers, frequently the best way to answer a question is to write some code and see what happens.

Start small; try to find a self-contained task that isn't performance critical and isn't on a tight deadline, so that you can take the time to experiment with different approaches to the problem. I say self-contained because interoperability with other languages can be more of a challenge with Java than with other languages, although we'll address a little of that in the postscript.

To develop code for the 3000, you have several possibilities. A traditional development model of writing code on a text editor and then compiling on the 3000 is of course still available. You can also use a tool like Whisper Programmer Studio (www.whispertech.com) to develop in a graphical environment on the PC, and compile either locally on the PC or remotely on the HP 3000. You can also use PC-based development tools such as Symantec Visual Café to develop code on the PC and then deploy on the 3000. You can deploy by moving the files via FTP or similar means, or you can use Samba/iX to mount your HP 3000 source directory directly on the PC and edit and compile the code that way.

Java for the 3000 and Java for the 9000 can be found at http://www.hp.com/java.

Also available from the MPE side of the above site is a Java Class Library for accessing TurboIMAGE databases, and the HP Driver for JDBC that provides access to both ALLBASE and IMAGE/SQL. Additional class libraries for access to MPE intrinsics and interoperability with VPLUS user interface code may also be available in beta form by the time you read this.

PostScript: Interoperability between Java and other languages

'Start small and self contained' is great advice for getting started, but if you decide Java is going to be part of your programmer's toolkit, pretty soon you're going to need to interoperate with your existing code. Java code can call 'native' code (in particular, C or C++ code) using the Java Native Interface (JNI). The JNI also provides a means for a C or C++ program to create a Java Virtual Machine and execute Java code within that VM. The JNI was covered at some length during my HPWorld presentations (also reprinted in Interact magazine) during the past two years, so I won't repeat the information.

Another common interoperability requirement is to be able to read from files that have been created by other languages. Generally, these files are record structured files, whereas the Java File object only knows about bytestream files. So a key technique to allow you to interoperate is knowing how to convert an arbitrary stream of bytes into separate data fields as you would find in a record structured file. Following are two examples of how this can be done. The first code sample is from an under-development VPLUS class library. The code in question represents the VPLUS comarea field. Although this field is typically passed from caller to callee, rather than read in from a file, the methods used to extract individual fields from the overall structure, which is defined as a Java byte array, would be directly applicable to the problem of reading from bytestream files.

This is just a fraction of the code from ComArea.java. The conversion code itself is pretty ugly; there is no method call that converts byte array data into numeric types, so we have to use a brute force approach.

```
//ComArea.java
package com.hp.vplus;

/** This class represents the VPLUS Communications Area */
public class ComArea {

   private byte[] data;

   /** This constructor is invoked from the C language
     * vdriver code via JNI, and is used to transfer the
     * contents of the comarea into this class' private data
     * area.
     */
   public ComArea(byte[] data) {
      this.data = data;
   }

   // cstatus: bytes 0-1
   public void setCStatus(short value) {
      data[0] = (byte)(value >>8);
      data[1]=(byte)((value <<8) >>8); }
   public short getCStatus() {
      return (short)((data[0] <<8) + data[1]); }

   // language: bytes 2-3
   public void setLanguage(short value) {
      data[2] = (byte)(value >>8);
      data[3]=(byte)((value <<8) >>8); }
   public short getLanguage() {
       return (short)((data[2] <<8) + data[3]); }

   // comarealen: bytes 4-5
   public void setComAreaLen(short value) {
      data[4] = (byte)(value >>8);
      data[5]=(byte)((value <<8) >>8); }
   public short getComAreaLen() {
      return (short)((data[4] <<8) + data[5]); }
```

Obviously, there are lots more fields in the comarea, but you see enough to get the pattern.

The second example is similar. In this case, the code comes from the TurboIMAGE class library, and is code used to break up a TurboIMAGE data buffer into its constituent fields. In this case, the code was implemented in C rather than Java to provide the best possible performance; the C code is then invoked using the Java Native Interface described earlier. Also, the ComArea class was written specifically to handle the VPLUS comarea, but the TypeConverter code was written to be reuseable anywhere. This is largely due to the fact that we can't code the offsets and field names of fields in a TurboIMAGE data record ahead of time, since the class library is designed to be usable with any TurboIMAGE database.

First, we'll look at the Java portion of the class. Since the actual method code is implemented in C, the Java code merely defines constants and method ***signatures*** (parameter types and return types)

```
//: TypeConverter.java
package com.hp.turboimage;
import java.io.*;

/** This class provides conversion routines that
  * can take an array of bytes and convert it to a
  * java datatype that most closely matches the
  * IMAGE type for that data, and vice versa.
  *
  * @version 2.1.1  April 1998
  * @author         Mike Yawn
  */
public class TypeConverter {

static {
   System.loadLibrary("TurboIMAGE");
}

   /** Specifies IEEE format for conversion of reals */
   public static final int IEEE_FORMAT = 1;
   /** Specifies HP (MPE/V) format for conversion of reals */
   public static final int HP_FORMAT   = 2;

   /** Return 2-byte byte array as a short */
   public native static short bytesToShort(byte[] in);

   /** Return 4-byte byte array as an int */
   public native static int bytesToInt(byte[] in);

   /** Return 8-byte byte array as a long */
   public native static long bytesToLong(byte[] in);

   /** Return 4-byte byte array as a float */
   public native static float bytesToFloat(byte[] in, int format);

   /** Return 8-byte byte array as a double */
   public native static double bytesToDouble(byte[] in, int format);

   /** Return short as a 2-byte byte array */
   public native static byte[] shortToBytes(short in);

   /** Return int as a 4-byte byte array */
   public native static byte[] intToBytes(int in);

   /** Return long as an 8-byte byte array */
   public native static byte[] longToBytes(long in);

   /** Return float as a 4-byte byte array */
   public native static byte[] floatToBytes(float in, int format);

   /** Return double as an 8-byte byte array */
   public native static byte[] doubleToBytes(double in, int format);

   /** Return String as a packed byte array */
   public native static byte[] stringToPackedBytes(String in, int len);

   /** Return array of packed bytes as a String */
   public native static String packedBytesToString(byte[] in);

}
```

Now we'll look at the C code. Like the ComArea code, this quickly gets repetitive, so we'll only show two of the conversion routines. The C code is mostly dealing with the requirements of the Java Native Interface, so unless you are familiar with JNI syntax it won't make much sense yet. But if you have legacy integration with Java in your future, the JNI is something you'll be seeing much more of.

```
//: TypeConverterImpl.c
#include "com_hp_turboimage_TypeConverter.h"

typedef union {
    char    asBytes[2];
    short   asShort;
} ByteArray2;

typedef union {
    char    asBytes[4];
    int     asInteger;
    float   asFloat;
} ByteArray4;

typedef union {
    char        asBytes[8];
    long long   asLongLong;
    double      asDouble;
} ByteArray8;

typedef union {
    char    asByte;
    struct {
       unsigned int n1:4;
```

```
      unsigned int n2:4;
   } asNibbles;
} Byte;

int i;

/* constants and declarations used for floating point conversion
 * omitted, since we aren’t showing that code anyway
 */

// bytesToShort
JNIEXPORT jshort JNICALL
Java_com_hp_turboimage_TypeConverter_bytesToShort(JNIEnv    *env,
                                                  jclass     this,
                                                  jbyteArray in) {
   ByteArray2 conv;
   jbyte *input = (*env)->GetByteArrayElements(env, in, 0);
   for(i=0; i<2; i++)
      conv.asBytes[i] = input[i];
   (*env)->ReleaseByteArrayElements(env, in, input, 0);
   return conv.asShort;
}

// bytesToInt
JNIEXPORT jint JNICALL
Java_com_hp_turboimage_TypeConverter_bytesToInt(JNIEnv    *env,
                                                jclass     this,
                                                jbyteArray in) {
   ByteArray4 conv;
   jbyte *input = (*env)->GetByteArrayElements(env, in, 0);
   for(i=0; i<4; i++)
      conv.asBytes[i] = input[i];
   (*env)->ReleaseByteArrayElements(env, in, input, 0);
   return conv.asInteger;
}

// many other conversion routines omitted for brevity . . .
```

Full source for the type conversion and comarea classes, if you’re interested in studying them further, is available for download. Since the site is currently being reorganized, I can’t provide an exact URL, but a quick browse of HP’s Java site should locate them, or send me an email if you can’t find them.