A Quick, Painless Introduction to the Java Programming Language

Norman Matloff
University of California, Davis

March 17, 2004

Contents

1 Why All This Hype over Java?

For the past few years, the press has been full of sensational stories about Java. Like most coverage of computer-related coverage in the press, the paeans to the wonders of Java have been greatly exaggerated.

One of the aspects of the florid coverage of Java in the press is that it is intended as a vehicle for *object-oriented programming* (OOP), said to be "an abrupt change in the paradigms of programming," a completely different philosophy. Don't believe it. True, in OOP one arranges one's code and data structures a bit differently, and true, OOP has the potential for producing somewhat safer, clearer, and more modular code. But programming is programming is programming. One uses the same basic thought processes in any kind of programming. Those of us who've programmed for many years have seen many so-called "abrupt paradigm shifts" come and go, each one heralded with great fanfare, but all of them turning out to be merely "more of the same."[1]

Nevertheless, Java is indeed a rather nice language, for a number of reasons, such as:

- It is arguably cleaner and more OOP-ish (if you like that kind of thing) than C++.
- It is mostly platform-independent, so that a program can usually be written to work in the same manner not matter whether it is running under UNIX, Windows, on a Macintosh, etc.[2]
- It has nice *exception-handling* (i.e. error-handling) facilities.
- It has built-in libraries for GUI development, Web transactions, multithreaded programming, network access and so on.

2 Learning Java

Given these nice features, and given the fact that reportedly Java has overtaken C++ as the language in highest demand by employers, this is a language worth learning.

This is reminiscent of a joke which used to be told by the father of Dr. Dick Walters, professor emeritus at UCD. Prof. Walters' father worked in many European countries, and thus found it necessary to speak many languages. When asked about that, he said, "Well, the first 3 or 4 languages are the hardest."

For learning a programming language, that threshold comes much earlier. Learning a new programming language is easy for anyone who already has reasonable competence in just one language.

But instead of taking a course in Java — a terribly ineffective way to learn any programming language except one's first — or wading aimlessly through a 700-page book on Java, it is easier to read the document you are now holding.

The purpose of this document is to give you a quick but solid foundation in Java. After reading this document carefully, you should be ready to write Java programs, and will have the background needed to learn whatever advanced Java features you need from books in the future, if and when the need arises.

[1] OOP itself is nothing new either. Java was preceded in the OOP world by for example Smalltalk and C++ in the 1980s, and Simula 'way back in the 1960s.

[2] However, contrary to the "write once, run anywhere" claim, Java is not 100% platform-independent.

We assume here that you have some experience with the C language. If you know C++, all the better, but it is not necessary.

3 A 1-Minute Introductory Example

Here is the obligatory "Hello World!" program:

```
public class Hi {   public static void main(String[] Args) {
      System.out.println("hi");
   }
}
```

The basic module unit in Java, and in OOP generally, is the *class*. A class is similar to a C **struct**, except that the latter consists only of data fields, while the former has both data fields and functions, called *methods*. In our example here we have just one class, named Hi, consisting of one method, main(), and no data.

The method main() is of course analogous to main() in C programs. Notice that main() has the familiar command-line arguments, which we have named Args; as in C, it is an array of arrays of characters, i.e. basically an array of strings.[3]

The construct **public** makes these items visible from outside the class. (We will discuss **static** later.)

System.out is one of the built-in classes in Java. The '.' is used to indicate a member of a class, just as is the case for fields within C **struct**s, so System.out refers to the member named "out" which is a member of the System class. In turn, "out" consists of both data — actually, a stream of characters going to the user's screen — and a member method, println(). So, we access println() as System.out.println(). As you've probably already guessed, this method works similarly to C's printf(), though without the formatting (e.g. without %d for printing **int**s).

To execute the class Hi, we would first use a text editor to put the above source code into a file named Hi.java. (There must be a match between the name of the class containing main() and the prefix in the file name.) We then run the Java compiler, typing

```
javac Hi.java
```

from the command line. If there are no compilation errors, a file named Hi.class will be produced. To execute the program, we then type

```
java Hi
```

Had there been command-line arguments for the program, to go in the variables Args above, they would have been typed right after "Hi" on the command line.

[3] Though standalone Java programs have main() functions as their entry points, Java *applets*, which work with Web pages, do not.

The reference to “Hi” here means the class named “Hi”. The **java** program will look for that class in the file whose name’s prefix is the same as the class name, and whose suffix is “class”, i.e. Hi.class.

Keep in mind that Java is an *interpreted* language. That means that the compiler, **javac**, does not translate your Java source code to the machine language of some real machine. Instead, it produces machine code, called *Java byte code*, for an imaginary machine called the Java Virtual Machine (JVM).[4] The program **java** which we run on the command line emulates the operation of the JVM, thus allowing us to run our application program. So, if for instance we are doing our work on a PC, the program which is really running on that PC is **java**, not our application program.

4 A 30-Minute Example

4.1 Example Program

This program will read in some integers from the command line, storing them in a linked list which it maintains in sorted order. The final list will be printed to the screen. For example, if we type

```
java Intro 12 5 8
```

then the output will be

```
final sorted list:
5
8
12
```

4.2 Source Code

As presented here, the following would be in files named Intro.java and NumNode.java:

```
// ****************** start of file Intro.java *****************

// the overall class name must match the file prefix; i.e. the name of
// this file must be Intro.java

// usage:  java Intro nums

// introductory program; reads integers from the command line,
// storing them in a linear linked list, maintaining ascending order,
// and then prints out the final list to the screen
```

[4] The JVM is not always imaginary. Chips which actually implement the JVM have been built.

```
public class Intro

{  // standalone Java programs must have a main() function, which is the
   // point where execution begins

   public static void main(String[] Args) {

      // note that locals have no public/private/... prefix; also, we
      // are using the fact that array objects, in this case Args, have
      // "length" variables built in
      int NumElements = Args.length;

      for (int I = 1; I <= NumElements; I++)  {
         int Num;
         // need to do C's "atoi()"; use parseInt(), a class method of
         // the Integer class; inverse operation is toString()
         Num = Integer.parseInt(Args[I-1]);
         // create a new node
         NumNode NN = new NumNode(Num);
         NN.Insert();
      }

      System.out.println("final sorted list:");
      NumNode.PrintList();

   }
}


// ****************** start of file NumNode.java ******************

// we have the class NumNode in the file NumNode.java

public class NumNode

{  // by making Nodes variable static, it will be common to all
   // instances of the class
   private static NumNode Nodes = null;

   // the rest of these variables are "local" to each instance of the
   // class

   // valued stored in this node
   int Value;

   // "pointer" to next item in list
   NumNode Next;
```

```
// constructor
public NumNode(int V)  {
   Value = V;
   Next = null;
}

// make the following methods visible externally, e.g. to main(), via
// public; the ones which are class methods rather than instance
// methods also are static

public static NumNode Head()  {
      return Nodes;
}

public void Insert()  {
   if (Nodes == null)  {
      Nodes = this;
      return;
   }
   if (Value < Nodes.Value)  {
      Next = Nodes;
      Nodes = this;
      return;
   }
   else if (Nodes.Next == null)  {
      Nodes.Next = this;
      return;
   }
   for (NumNode N = Nodes; N.Next != null; N = N.Next)  {
      if (Value < N.Next.Value)  {
         Next = N.Next;
         N.Next = this;
         return;
      }
      else if (N.Next.Next == null)  {
         N.Next.Next = this;
         return;
      }
   }
}

public static void PrintList()  {
   if (Nodes == null) return;
   for (NumNode N = Nodes; N != null; N = N.Next)
      System.out.println(N.Value);
}
```

```
}
```

Note that our "main" class — in the sense that it contains main() — is Intro. We also have another class, NumNode, which is used by Intro. The Intro class has one method, main(), and no variables. NumNode has several methods and several variables. We have chosen to put the classes in different files (though in the same directory), but we could have put them in the same file.[5]

Let's have a look at main(), which by the way has as its full name Intro.main. First there is a local variable declared, NumElements. This is the same as in C, but note the item Args.length. This is typical OOP style — everything is an *object*, i.e. a class or instance of a class. Args, recall, is an array. In Java, arrays are objects, and one of the fields in an array object is .length, the number of elements in the array. In our example above,

```
java Intro 12 5 8
```

Args.length would be 3.

Unlike C, in Java the first command-line argument is numbered 0, so here Args[0] = "12".[6]

By the way, speaking of the fact that Java is strongly typed, note that floating-point numerical constants are considered to be of type **double**, not **float**. For example, you may innocently try code like

```
float X,Y;
...
X = 1.5 * Y;
```

and find that the compiler complains. So, you need to convert 1.5 to **float** first, using a cast, i.e.

```
X = (float) 1.5 * Y;
```

Next we have a **for** loop. Local variables can be declared in the midst of code; the **int** variables I and Num here exist only within this loop.[7]

We need to use something like C's atoi() library function to convert the command-line arguments from strings to integers. The parseInt() method in the Integer class (a fancier version of **int**) does this.

[5] If so, we would have needed to delete the word **public** in the declaration of the NumNode class, and place NumNode in the latter portion of the file. Java only allows one public class per file, and that one must come first.

[6] Recall that it is "12", not 12. Args is an array of strings, as in C.

[7] You may wish to use this sparingly, though. Some debugging tools will not display the values of such variables.

4.3 Creating Objects

Now consider the line

```
NumNode NN = new NumNode(Num);
```

If you know C++, the **new** construct is similar in Java. If you don't know C++, the way to understand this is that it is similar to calling malloc() in C. For example, in C

```
int *z;
...
z = malloc(sizeof(int));
```

would create a new **int** in memory, and point z to it. Here in this loop in Intro, we are creating a new *instance* of the class NumNode. This new instance of NumNode will be named NN. We say that NN is an *object* of class NumNode.

You might wonder why "NumNode" appears on both sides of this assignment statement. On the left side, we are declaring NN to be of type NumNode. On the right side, we are recognizing that NN will be produced by calling the *constructor* for the NumNode class, with the parameter Num. The constructor of a class has the same name as the class. As we will see later, here the constructor will be initializing one of the fields in this instance of NumNode to Num. For the beginner, this statement might be clearer if split into two statements:

```
NumNode NN;  // declares NN to be a "pointer" to objects of type NumNode
NN = new NumNode(Num);  // creates such an object and points NN to it
```

4.4 Class/Instance, Public/Private

Now compare the next line,

```
NN.Insert();
```

with the one a couple of lines later:

```
NumNode.PrintList();
```

Both of them execute methods in the NumNode class. However, Insert() is an *instance method*, while PrintList() is a *class method*. An instance method acts on a specific instance of the class, in this case NN, while a class method applies to the class in general. Class methods are designated as such by declaring them

to be **static**, as seen in the declaration of Insert() later in the code. When calling a class method, the name of the class is used as a prefix, such as with NumNode.PrintList() in our example here. If on the other hand we call an instance method, we use the name of the particular instance, in this case writing NN.Insert().

Again, an instance method is applied to the particular instance. With NN.Insert(), the Insert() method is applied specifically to NN, i.e. NN is what will be inserted into our linked list. In effect, when calling an instance method, the instance (NN here) becomes an implicit parameter to the call, in addition to any explicit arguments the method might have.

A class method, on the other hand, applies to the class as a whole, not an instance of the class, and indeed we do not even have to have any instances around to be able to call it. This was the case when we called Integer.parseInt(), without creating any instances of Integer. By contrast, for the instance method Insert() here, we did need an instance of NumNode to apply it to.

In fact, since we never do create an instance of class Intro, i.e. we don't have any statement which performs "new Intro()", that means that any methods we might declare in Intro would have to be static.

Now, let's discuss some related details in the NumNode class itself. Note first that the declaration of NumNode,

```
public class NumNode
```

has an *access modifier* **public**, recognizing the fact that we are accessing it outside the class. (We are accessing it from Intro.)

We have not specified access modifiers for our variables Value and Next. For reasons which will be explained later, these two variables are still accessible from Intro. For example, we could within Intro access NN.Value (say, printing out NN.Value), even though Value is in NumNode. But if we do not want Value to be accessible from within Intro — in the spirit of modularity, "data hiding" and other tenets of modern software engineering — we would declare it **private**:

```
private int Value;
```

Beginners should not worry about **public**/**private** differences. Just make everything **public** or unspecified. (Switch from unspecified to **public** if the compiler complains.) Later, after you become more proficient, you can start using **private** for the sake of the encapsulation tenets of good software engineering.

On the other hand, the qualifier **static** is something even beginners must know, because it is used to specify whether a method/variable is to be a class method/variable or an instance method/variable. We state **static** in the former case but omit it in the latter.

We have already explained the difference between class and instance methods. The distinction between class and instance variables, while similar, has important differences. To learn about them, let's look at the three data members in the class NumNode: Nodes, Value and Next. Again, all three of them act like fields in a C struct, but there is an enormous difference between Nodes on the one hand, and Value and Next on the other.

The best way to understand this difference is to first get a feel for what roles these three variables play in our program. Remember, the program builds up a linked list. Each node in that list will contain some number

(which had originally been obtained from the command line array Args), stored in the variable Value. Each node in the list (except the last node) will be linked to the next node, and the variable Next serves in this role. The variable Nodes will in essence serve as a pointer to the head of the linked list.

The key is to note from this that we will have many (Value, Next) pairs — one for each node in the list — but only one Nodes variable. This is exactly where the idea of class variables versus instance variables comes in. Each instance of the class will have different data in its instance variables, but there is only one copy of each class variable, no matter how many instances of the class exist. You can see from this that in our application here, the appropriate setup is to make Value and Next instance variables but make Nodes a class variable. Again, this is accomplished by using the qualifier **static** in the declaration of Nodes but not doing so for Value and Next.

Also, look at the initialization of Nodes:

```
private static NumNode Nodes = null;
```

The initialization of Nodes will occur the first time the class is loaded. It is NOT the case that Nodes will be re-initialized to null each time a new instance of NumNodes is created, which would be a disaster, making the program operation completely incorrect — the list would alternate, first empty then having one node, then empty again, then having one node, then empty, etc. Making Nodes a class variable here ensures correct operation of the program.

Among other things, this means that main() needs to be declared **static**, as we did. The reason for this is that when we execute Intro, we are not creating any instances of it, so that main() must be a class method. If we had any other methods in Intro, we would have to make them **static** too.

4.5 "Pointers" in Java

By the way, there are no explicit pointers in Java, as can be seen for example in the declaration of Next:

```
NumNode Next;
```

In C or C++, this probably would have been

```
NumNode *Next;
```

The creators of Java designed it this way, out of a concern that many program bugs in C/C++ are due to pointer errors. They feel that the Java way is clearer and less error-prone.

Nevertheless, it's important to understand that a variable declared to be of class type does indeed serve as a pointer. Suppose we have a class, A, and a declaration

```
A X;
```

X is basically declared as a pointer to A. As of yet, it points to nothing. That will change when we execute something like

```
X = new A();
```

which allocates space for an instance of the A class and points X to it (similar to calling malloc() in C), or when we execute something like

```
X = Y;
```

assuming Y already points to an instance of the class A.

4.6 Setting Up Arrays

Be careful when setting up arrays of objects. For example, suppose we have a class DEF. Then:

```
DEF ABC[];  // declare ABC to be an array of DEFs
...
ABC = new DEF[10];  // now the JVM knows the array length will be 10
for (int J = 0; J < 10; J++)  // create the objects
   ABC[J] = new DEF();
```

If we did not have that last loop, our first reference to an element of ABC, say

```
ABC[0].X = 12;
```

(assuming DEF has a integer member X), would result in a null pointer runtime error, since ABC[0] would be pointing to nothing.

Arrays of scalars are simpler to declare and use, e.g.

```
int[] UVW = new int[10];
...
UVW[5] = 88;
```

4.7 Java's "this" Construct

If you have not used C++ before, Java's **this** construct, seen here for example in the statement

```
Nodes = this;
```

needs explanation. The keyword **this** refers to the instance on which the given method was called. The above line, which was part of the Insert() operation in our application here, handling the case in which the list currently happens to be empty. We want Nodes, the head of the list, which had been null, to now consist of (or "point to") the instance of NumNode which we are now working on (NN in Intro).

Since **this** means the current instance of the class, we could write lines like

```
if (Value < Nodes.Value) {
```

as

```
if (this.Value < Nodes.Value) {
```

if we wanted to. Of course, it is less typing for us to not do it this way, but it will help cement your understanding of the **this** construct if you take a minute now to make sure you see why the alternative writing is equivalent.

5 Exception Handling in Java

Suppose I type a mistake in my input to Intro, say as

```
java Intro 1w 5 8
```

I accidentally typed "1w" instead of "12". When Integer.parseInt() is called on this string from Intro, that method will object that it is not a number, and will give me an error message something like this:

```
java.lang.NumberFormatException: 1w
at java.lang.Integer.parseInt(Integer.java, Compiled Code)
at java.lang.Integer.parseInt(Integer.java, Compiled Code)
at Intro.main(Intro.java, Compiled Code)
```

Java methods allow a **throws** construct, which tells the JVM what to do if something goes wrong in this function. The methods in the Java class libraries make use of this construct, and you can do so with the methods you write too. Each use of **throws** is associated with one or more *exception* types, i.e. error codes, which once again are class objects.

The error message above tells us that the Integer method parseInt(), called from within Intro.main, is the one which encountered the error, and that the error was caused by the string "1w". The exception was of type NumberFormatException. What is that, really?

Documentation similar to UNIX "man pages" for the Java class libraries is available on the Web at `http://java.sun.com/j2se/1.3/docs/api/index.html`. If we check the Integer class there, we will see documentation on parseInt(). It tells us that parseInt() indeed "throws NumberFormatException." Moreover, the documentation includes a Web link to the details on that exception type. If we hadn't already realized that our problem was the w in "1w", we would see it now.

The response of the Java interpreter to the discovery of this error is to kill the program. However, Java gives us the chance to avoid this drastic remedy. We define a new method called ConvertArg:

We replace our old call to Integer.parseInt() in Intro to

```
Num = ConvertArg(Args[I-1]);
```

and define the method (in Intro.java) as follows:

```
static int ConvertArg(String Arg)

{  DataInputStream In = new DataInputStream(System.in);
   InputStreamReader Ins = new InputStreamReader(In);
   BufferedReader Inb = new BufferedReader(Ins);
   while (true)  {
      try  {
         int N = Integer.parseInt(Arg);
         return N;
      }
      catch (NumberFormatException NFE) {
         System.out.println("bad command-line argument--" + NFE);
         System.out.println("enter number again, or type q for quit");
         try  {
            Arg = Inb.readLine();
         }
         catch (IOException IOE) {
            System.out.println("failed to read correction--" + IOE);
            System.exit(1);
         }
         if (Arg.equals("q"))
            System.exit(1);
      }
   }
}
```

Let's ignore the first three lines for the moment, and go straight to the **while** loop. The key point is that we surround our call to Integer.parseInt() in a **try** block, which is paired with a **catch** block. What we are

specifying is that the JVM try to execute Integer.parseInt(), but if an NumberFormatException occurs, the **catch** block will be executed. As you can see, our code there will give the user a chance to rectify the error, inputting the integer again (or quitting) via the keyboard.

Note that if there is a NumberFormatException, we have assigned it to the variable we've named NFE. NFE will then contain the error message the system itself would have given us, and we are printing out both that message and our own elaboration. By the way, in Java string concatenation is done with the '+' operator.

We are using BufferedReader.readLine() to read in the user's response. Yet our keyboard input stream class is System.in, not BufferedReader. Thus we needed to convert from one to the other, which we accomplished via the intermediate conversion to InputStreamReader; see the first three lines before the **while** loop which we skipped over earlier. This seems a bit cumbersome, but unfortunately Java has no direct analog to C's very flexible scanf().

Note too that BufferedReader.readLine() throws an IOException, so we need to catch it too.

If we are too lazy to catch an exception (not advisable), we can do something like in this example:

```
public static void main (String Arg[]) throws NumberFormatException,
     IOException
```

Here we are saying to the Java compiler, "Yes, yes, I know that there are points in my code within this method main() at which I should provide for exceptions of the type NumberFormatException and IOException, but I am too lazy to do so." If we don't even do this much, the compiler will complain.

If we set up this ConvertArg function, we will need a line

```
import java.io.*;
```

at the beginning of the file, in order to access the various I/O classes used here. In many programs you will need to use **import** statements. These are like C's **#include** statements, but a little more convenient, thanks to a combination of Java's OOP and interpreted natures, which allow both code and data in one *package*. Compare this to the C/C++ setting, in which the **#include**s are used to get the data, while the corresponding functions are linked in separately.

6 I/O in Java

Java has a rather intimidatingly rich set of I/O mechanisms. However, it is much more manageable if you keep in mind that there are two main categories:

- Subclasses of InputStream and OutputStream. These deal with I/O on a byte-by-byte basis, and are useable in all file and networks contexts, whether text or "binary" data.
- Subclasses of Readers and Writers. These are only for the case of text files. They provide more convenience, e.g. methods which will read an entire line of text, rather than byte-by-byte.

You may wish to avoid Readers and Writers during your early stages of learning Java, though really they are not that complex.

The basic entities in Java I/O are known as *streams*. If you read from a file, for example, the sequence of bytes from the file is a stream.

One stream may be *chained* to another. This means that some operation has been performed on the first stream, with the result being the second stream. You may find for example that you wish to take advantage of methods in a stream class C but the stream you have, say MyStream, is of class A. You may need to chain A to some class B and then chain B to C. The chaining is done via constructors of the classes involved, e.g.

```
B My2ndStream = new B(MyStream);
C My3rdStream = new C(My2ndStream);
```

You would then be able to apply the mdthods of C to My3rdStream, a converted version of your original stream MyStream.

Both the InputStream/OutputStream and Readers/Writers categories of I/O classes include classes for *buffered* I/O. Say you are sending data from your machine to another machine on the Internet. The application program you've written may produce data byte-by-byte, but you would not want it to go through the network that way, as there is heavy overhead each time data is sent. Buffered I/O saves up data so that it would be sent in groups of bytes, even though it appears that your program is sending the data a single byte at a time.

As a beginner you may wish to avoid buffering, especially since your operating system will probably do some buffering for you anyway. But as you gain expertise, truely efficient I/O will require that you use buffering.

7 Strings in Java

In C or C++, a *string* simply means an array of **char**, but in Java we have the String class. Not only does this mean that functions corresponding to C's strlen(), strcat() etc. are now methods built in to the String class, but also the storage of the string itself is different: Each character now takes up two bytes, instead of one, in order to accommodate non-ASCII characters, i.e. foreign languages. Another variable stored in this class is the length of the string.

So if for example we have:

```
String ABC;

ABC = "xyz";
```

then the storage is 10 bytes rather than just three: Six bytes for the three characters and four bytes for the length.

One often must convert between an instance of String and byte[]. For example, file operations cannot use the former (since any file is just a sequence of bytes), while methods such as System.out.println() cannot

really use the latter. To convert from String to byte[], use String's getBytes() method, while the opposite conversion can be done by simply using String's constructor.

For example:

```
String ABC;
byte[] B;

ABC = "xyz";
B = ABC.getBytes();

B[0] = 'q';
S = new String(B);  // S is now "qyz"
```

8 Debugging Java Programs

Hopefully you are already a pro at debugging C or C++ programs. Note that being a pro means that you do not use printf() calls as your debugging tool! Use a good debugging tool; it will save you lots of time! If you are a student, you may find my debugging tutorial at `http://heather.cs.ucdavis.edu/~matloff/debug.html` to be useful. It gives tips on debugging, and cites some good debugging tools (mainly UNIX/C/C++ oriented).

Now, what is the situation for Java debuggers, or if you prefer, Integrated Development Environments (IDEs, consisting of debuggers, editors, etc. packaged in an integrated manner) for Java? There are many IDEs available for Java, but most are commercial, moderately to highly expensive, and take up huge amounts of disk space.

Listed below are a few debuggers and IDEs. The criteria used for choosing them are that they are

- free
- small
- easy to use for small- to moderate-sized projects

Here is the list, ordered from best to worst in my view:

- JSwat:

  This is a very nice debugging tool, which you can use either with your favorite editor or with the JIPE Java IDE. See my introduction at `http://heather.cs.ucdavis.edu/~matloff/jswat.html`.

- BlueJ:

  BlueJ, available at `http://www.bluej.org`, is nice and small. It's minimalist in terms of its editor (e.g. no autoindent). The variables and their values are automatically displayed (though, unfortunately, **static** variables are not fully given), which is nice, and the system is very small and easy to

use. It also allows debugging individual classes on a standalone basis. You may wish to read my quick introduction to BlueJ before you use it, at `http://heather.cs.ucdavis.edu/~matloff/bluej.html`.

- JDB:

  This is the basic debugger which comes with Java distributions. It is quite primitive, but if you use it through the DDD interface (Version 3.3 or newer) it is quite usable.

  My introduction to JDB is at `http://heather.cs.ucdavis.edu/~matloff/jdb.html`, and my introduction to DDD is at `http://heather.cs.ucdavis.edu/~matloff/ddd.html`.

- GDB:

  Yes, you can use GDB! See Section A.2 below.

9 Classpaths and Packages

In this section, we discuss a bit more on usage of the Java compiler, interpreter and other aspects you need to know about to get started with Java. However, we will not go into details here.

9.1 Classpaths

If you have written C/C++ programs on UNIX, you are probably aware of the -I and -L command-line options to compilers, and non-UNIX systems have similar provisions. The -I option tells the compiler directories in which to look for **#include** files, in addition to the standard directories it searches; -L does the same kind of thing for linking in libraries.

In Java, the corresponding idea is that of a classpath. The default classpath consists of the Java system directory (as defined by the location of the programs **javac** and **java**), and the current working directory, i.e. the one from which the program **javac** or **java** is run. The -classpath option for the Java compiler, **javac**, will tell the compiler where else to look, and the same option for the Java interpreter, **java**, works at runtime. (The latter also allows you to abbreviate the option as -cp.) There is also a CLASSPATH environment variable which can be used.

If you need to add more than one directory to the classpath, concatenate the names, separated by colons. As in Unix, a dot, '.', means the current directory.

9.2 Packages

Java's **package** concept can be used to group files together.[8] It interacts with the classpath in the following way, illustrated with our Intro/NumNode example above. Suppose Intro.java is in the directory /a/b/c/ti and NumNode.java is in /x/y/tn. We would think of ti as a package within /a/b/c and tn as a package within /x/y. At the top of Intro.java we would include a line

[8] In our example here, we will have only one source file per package, but typically a package will contain multiple files.

```
package ti;
```

and put a similar statement,

```
package tn;
```

in NumNode.java.

Also, in Intro.java, we would have a statement

```
import tn.NumNode;
```

To compile Intro.java, we would go to the ti directory and type

```
javac -classpath .:/x/y Intro.java
```

This tells **javac** that during its compilation of Intro.java, if it sees a reference to a class not defined in that file, it should look in /x/y (and the current working directory). In this case, such a class is NumNode. Our **import** statement told **javac** to find NumNode in the package tn, and the -classpath option in our **javac** command line above tells **javac** to look for that package in /x/y. In other words, **javac** will look for NumNode in /x/y/tn.

It may surprise you that we cannot now run Intro by typing

```
java Intro 12 5 8
```

from within the /a/b/c/ti directory. The **java** interpreter will respond with a “wrong name” error, admonishing us that the program’s name is ti.Intro, not Intro!

Instead, to run Intro we would type (from any directory)

```
java -cp /a/b/c:/x/y ti.Intro 12 5 8
```

Or, since the default classpath includes the current directory, we could go to the directory /a/b/c and type

```
java ti.Intro 12 5 8
```

Packages can have hierarchical directory structures. In the example above, the directory /x/y/tn could have a subdirectory uv and a class Z, in which case Z would be referred to as tn.uv.Z.

9.3 Access Control Revisited

Recall that in our Intro/NumNode example, we did not specify access modifiers for the variables Value and Next. We will now return to this issue. There are two main points to consider:

- All methods and variables of unspecified access status in a class in a package are accessible from all other classes within that package.
- All classes in a program which are not explicitly part of a package are treated by Java as belonging to the *unnamed package*.

There are two implications of this:

- In the original version of Intro/NumNode, which does not have explicit packages, the classes Intro and NumNode belong to the unnamed package. Thus Value and Next are accessible from Intro.
- In the newer version, with packages ti and tn, Value and Next are not accessible from Intro.

10 Jar Files

Often a number of Java files will be packed together in a manner similar to the UNIX .tar files. The Java analog is .jar files, which are created and unpacked using the jar command and the xf option, similar to the UNIX tar.

The program **java** also has a -jar option, allowing you to execute a program without unpacking the .jar file.

In referencing a .jar file via a classpath, the .jar file name must be included in the path, not just the directory containing it.

11 Inheritance

Since inheritance is one of the fundamental ideas in OOP, and since usage of Java's specialized classes and advanced features depends on understanding inheritance, we will at least introduce it here.

Consider again our linked-list example above. Actually, Java has a built-in library class named LinkedList. This would have saved us the trouble of writing the code in our NumNode class, and would have given us better protection, in the sense of encapsulation, data-hiding and so on.[9]

However, the LinkedList classs doesn't have any method comparable to our NumNode.Insert(). This is not surprising, since we assumed an ordered list, while LinkedList doesn't. So, we could create a new class, calling it for example OrderedLinkedList, which would be an extension of LinkedList. The beginning of the declaration would look something like this:

```
public class OrderedLinkedList extends LinkedList {
```

[9] Programs using this class need to **import** java.util.*.

The effect of the **extends** keyword is that OrderedLinkedList would consist of all variables and methods in LinkedList, plus whatever new variables and/or methods we declare here in OrderedLinkedList, in our case for example an Insert() method. So, if we have an instance of OrderedLinkedList called, say, OLL, then we could not only call Insert() via OLL.Insert(), but also access the LinkedList method addLast(), which adds a node to the end of the list, via OLL.addLast().

By the way, the code for Insert() would be a little simpler than what we have above, because here we could make use of the methods in LinkedList.

Methods in a class can be extended — we use the word *overridden* — too. Suppose that class B extends class A, and that A includes a method Print(), which prints out all the variables in A. Suppose we wish Print() to also print out a variable X which was added in B too. Then we could redeclare Print() in B, with it looking something like this:

```
void Print()

{  super.Print();
   System.out.println("B = "+X);
}
```

The construct "super" refers to the parent class, so super.Print() refers to the original version of Print() in A. Thus the above code calls that original Print() to print out the original variables (which, remember, are in B) and then we print out X too.

Note that only instance methods, not class methods, can be overridden.

12 Advanced Stuff

12.1 What Else Is There to Java?

In a word, "Plenty!" Here are just a few of the things we haven't covered in this introduction:

- Advanced class types, such as **abstract** and **interface**.
- The Java libraries for GUI programming, notably AWT and Swing, and their usage for Web applications. To many people, Java **is** a Web applications language, so our omission of these here is an outrage. But again, we have tried to keep this short and simple.
- The Java libraries for multithreaded programming, networking, etc.

Even character strings are different in Java, with powerful capabilities. The char type uses 2-byte Unicode, and is thus usable on non-English alphabets.

12.2 How to Learn The Advanced Stuff

First of all, there are of course books. As a rule, I suggest getting at least three or four books on any new subject one is learning, in this case Java, if you can afford it.

Second, there are online "man pages" for built-in Java classes, at `http://java.sun.com/j2se/1.3/docs/api/index.html`.

Finally, there is the Web! For example, just plugging "Java UDP" into a Web search engine will give you tutorials, examples and documentation on the use of the UDP network protocol in Java.

A How to Obtain and Install Java

A.1 One Approach: Download From Javasoft

For example, you may download Java from Javasoft, at `www.javasoft.com`. You'll want the Java Development Kit (JDK, also known as SDK), standard edition. The latest version is JDK 1.4, but I suggest 1.3 for now.

The download file will be an executable, with a file name suffix .bin (for Unix) or .exe (for Windows). Run the file. In the Linux case, after running the file, an .rpm file will be produced; then run the **rpm** command with the -i option. You may need to add the location of Java executable files, say /usr/local/java/jdk1.3/, to your path.

A.2 Another Approach: Use GCC

Another alternative, convenient if you have Linux on your machine, is to use GCC. Yes, the GCC compiler, traditionally used for C and C++, can now compile Java. This has the additional advantage of producing real native machine code for your machine! Normally one must sacrifice execution speed when writing in Java, as it is usually interpreted. But with GCC, you compile to real machine code. Another advantage is that this means you can use the GDB debugging tool, which is better than JDB.

To compile our example program in Section 4.2, do the following:

```
% gcj -c -g NumNode.java
% gcj -g --main=Intro Intro.java NumNode.o -o intro
```

There are like the usual GCC commands, except for `-main=Intro`, which states in which class a function **main()** is to be the entry point for execution of the program.[10] I ran this on my PC, so the executable file **intro** really is Intel machine code. I run it the same way as I would for compiled code from C aor C++:

```
% intro 5 12 8
final sorted list:
5 8 12
```

[10] We did choose to compile the two source files one at a time. We found this necessary in order to ensure that GDB kept track of the source files correctly.

To debug with GDB, I'd type:

```
% gdb intro
```

Then I'd give GDB a couple of commands before getting to the debugging task:

```
(gdb) handle SIGPWR nostop noprint
(gdb) handle SIGXCPU nostop noprint
```

These tell GDB not to stop or print announcements to the screen when UNIX *signals* are generated by Java's garbage collection operations. Such actions would be a nuisance, and may interfere with our ability to single-step using GDB.

Now, I could, for instance, set a breakpoint at the beginning of the **Insert()** method, and then run:


```
(gdb) handle SIGPWR nostop noprint
(gdb) handle SIGXCPU nostop noprint
```

These tell GDB not to stop or print announcements to the screen when UNIX *signals* are generated by Java's garbage collection operations. Such actions would be a nuisance, and may interfere with our ability to single-step using GDB.

Now, since the first apparent casualty of the bug was for the number 12 to disappear, let's set a breakpoint at the beginning of the **Insert()** method, and then run:

```
(gdb) b NumNode.java:17
Breakpoint 1 at 0x8048b3e: file NumNode.java, line 17.
(gdb) cond 1 this.Value==12
(gdb) r 5 12 8
Starting program: /debug/intro 5 12 8
[New Thread 16384 (LWP 11965)]
[New Thread 32769 (LWP 11988)]
[New Thread 16386 (LWP 11989)]
[Switching to Thread 16384 (LWP 11965)]
Breakpoint 1, NumNode.Insert() (this=@80b0c00) at NumNode.java:17
17            if (Nodes == null)  {
Current language:  auto; currently java
```

As can be seen here, threads are set up even for programs which are not explicitly threaded.

We might want to make sure we had set the condition properly:

```
(gdb) p Value
$1 = 12
```

We would then use GDB as usual, stepping through our code, printing out values of variables, and so on.