Teaching a Course on Understanding Java

Barry Cornelius
Computing Services, University of Oxford
Date: 24th January 2000
http://users.ox.ac.uk/~barry/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

---

1 Abstract

The main aim of this paper is to explain the rationale behind the design of a course that is an introduction to programming using Java. One of the key features of this course is the early introduction of interfaces. The paper explains why it is appropriate to do this. The paper also points out some of the deficiencies in the design of classes that are made by textbooks that are targetted at this kind of course.

2 Introduction

At the University of Durham, there is a first year module entitled *Introduction to Programming*. The module includes 42 hours of lectures. Although it can be taken by any student (whose timetable permits), it is compulsory for students taking any of the degree courses in Computer Science. The module is currently taken by about 125 students.

When the Department of Computer Science chose to move its first programming language from Modula-2 to Java, I took the opportunity to think about how best to introduce the various concepts of Java. The first version of the course that used Java instead of Modula-2 took place during session 1998-99. Some changes are being made to the course for the 1999-2000 session.

The main aim of this paper is to explain the rationale behind the design of the course that is being developed. The paper also points out some of the deficiencies in the design of classes that are made by textbooks that are targetted at this kind of course.

3 The order in which to teach things

During the last 25 years, I've been involved on a number of occasions with switching the first programming language.

```
University of Hull      Algol 60    Pascal
University of Durham    Pascal      Modula-2
University of Durham    Modula-2    Java
```

Every time you find yourself with a lot of new ideas to teach. And I find it exciting to sit down and think about what order to teach the various ideas.

Using Pascal/Modula-2, the topics that I have taught included:

```
terminal I/O
expressions
if statement
for statement
case statement
while statement
arrays
procedures and functions
recursion
records
pointers
linked lists
stacks
queues
trees
```

When moving to Java, *procedures and functions* are called *methods*, and pointers, linked lists, stacks, queues and trees are replaced by other things.

Here is a list of some of the new ideas of Java:

```
creating an object
using an instance method
using a class method
using the wrapper types (Integer, Double, etc)
declaring a class type
declaring a method declaration
declaring an interface
implementing a Core API interface
implementing one of your own interfaces
implementation inheritance from a Core API class
implementation inheritance from one of your own classes
creating a window
adding buttons to a window
using a textfield for input/output
providing menus
using MVC to design programs
using the Collection API: lists, queues, stacks
using the Collection API: sets
using the Collection API: maps
using a throws clause
using a try statement
declaring an exception class
using applets
```

Those are some of the topics that I teach, but of course there are many more that could be taught. The order in which the parts of Java are taught in this course is given in the Appendix to this paper.

4 Introducing objects early?

Although Java is a language in which one can pursue object-oriented programming, it is not a pure object-oriented programming language: in Java some values are represented by values of a primitive type. This complicates the teaching. Students can easily be confused here as the representations of the values of these types are quite different. Andreae et al, in a paper entitled *Surprises in Teaching CS1 with Java* (http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-98-9.abs.html), say that they 'were surprised at just how difficult this was to teach'.

One approach to teaching Java would be to try to ignore the primitive types as they do not fit in comfortably with an *object-first* approach. However, with such an approach, you would have to ignore Java's Core APIs (as these classes make extensive use of primitive types): instead you would have to develop your own classes. And you would have to ignore the `for` and `while` statements as these require values of the type `boolean`.

Although this approach could be adopted, it is not one that I would want to pursue. In my opinion, it is not teaching Java as it really is. One of the key features of Java is its well-stocked library of already written code, i.e., the Core APIs. For this reason, I believe that any course that teaches Java should make extensive use of the Core APIs, and use these from the start of the course. This reinforces the idea that you do not write all the code yourself: there is a wealth of code out there waiting for you to use.

If primitive types have to be taught reasonably early, how do we teach both primitive types and class types? My own view is that they can be taught in the following order:

1. introduce primitive types;
2. demonstrate that operations can be performed on values of a primitive type by using methods (and operators);
3. point out that the primitive types are not sufficient to represent real-world objects;
4. introduce some of the class types of Java's Core APIs as a way of representing other real-world objects.

During this process it is important to use the WWW pages that document Java's Core APIs in order to show what is possible with these class types.

So, although some may argue that my course is not an *object-first* course:

- using the classes of Java's Core APIs, it introduces objects early;
- it does this by giving examples that use classes like `java.awt.Point`, `java.lang.String` and `java.util.StringTokenizer`;
- it does this by referring the students to the descriptions of these classes given in the WWW pages documenting the Core APIs.

5 Handling control statements and method declarations

At http://gee.cs.oswego.edu/dl/html/javaInCS.html, Doug Lea gives details of his experiences of teaching Java back in 1996. He says that 'we probably over-exploited how easy it is to learn basics; sometimes forgetting that it still takes time to gain minimal *programming maturity* in any language'. Although this could be interpreted in a number of ways, it may be that he was trying to say that there is a lot of ideas that a student has to come to terms with besides those of creating and understanding class declarations.

One of the key aspects of a class declaration is a method declaration. And there is a lot of complexity in writing both the header and the block of a method declaration. In my view, it is wrong to explain how to write class declarations at the start of a course and hope that the students can take on the idea of method declarations at the same time.

So in this course, the following order is used to teach the bread-and-butter topics:

- input from the keyboard and output to the screen
- if statements
- for statements
- method declarations
- other control statements

I'm not ashamed that you will find these taught in this order in many books that are teaching a language other than Java.

6 Interfaces and classes

6.1 The role of interfaces and classes

Both Ada and Modula-2 separate out the interface from the implementation:

```
Ada         package specification    package body
Modula-2    definition module        implementation module
```

Although a similar idea can be achieved in C++, it is not so cleanly done. In Java, interfaces can be used:

```
Java        interface declaration    class declaration
```

So when I introduce class declarations, I also introduce interface declarations. So the students are encouraged to develop an interface before developing the class. To begin with, I was scared about doing this as there are no books on Java that teach interfaces at the same time as classes. But my experience with teaching Modula-2 told me that this was the right thing to do.

When a student gets to writing their own class declarations, they are already familiar with the WWW pages of some of the classes of Java's Core APIs, and so they are aware of what classes often provide. And they are also already familiar with method declarations. As a result, they are more in a position to concentrate on the actual class declaration. Although class declarations can be used for a number of different purposes, this part of the course introduces interface and class declarations as a means for introducing a new type.

I stress that there are two main characteristics to a type:

- a type has a set of values associated with it;
- a type has a set of operations that are permitted on these values.

So, we can use a class declaration to introduce a new type because the fields of the class declaration can represent the values of the type, and the methods of the class declaration can provide the operations.

After exploring some of the various possibilities with class declarations, these lectures end by introducing the interface declaration. Here the example that was used to introduce class declarations is reworked in terms of an interface (called `Date`) and a class (called `DateImpl`) that implements the interface:

```
public interface Date;
{
   public int getDay();
   ...
   public void setDay(int pDay);
   ...
}

public class DateImpl implements Date
{
   private int iYear;
   private int iMonth;
   private int iDay;
   ...
   public int getDay()
   {
      return iDay;
   }
   ...
   public void setDay(final int pDay)
   {
      iDay = pDay;
   }
   ...
}
```

The lectures stress that code should be written in terms of the interface and that the name of the class should only be used when you want to create an object (of the class that implements the interface):

```
Date tTodaysDate = new DateImpl(2000, 1, 24);
...
private static boolean iIsLeap(final Date pDate)
{
   final int tYear = pDate.getYear();
   return (tYear%400==0) || (tYear%4==0 && tYear%100!=0);
}
```

The course points out that:

- The interface is better at expressing the contract between the client and the supplier of the class.
- The use of an interface helps to delineate in the client the code that is dependent just on the interface from the code that is dependent on the implementation.
- Although there will be some client classes that need to create `DateImpl` objects, it is possible that the code of some client classes can be written solely in terms of `Date` rather than `DateImpl`. Such classes need not be re-compiled and re-tested if `DateImpl` is changed. The extreme of this is to declare an interface and a class for a factory class which is responsible for creating `DateImpl` objects. In this way, no other class of the program needs to be re-compiled if `DateImpl` is changed.

Although these lectures on interfaces and classes emphasize that one of their primary roles is to represent a new type, additional lectures introduce some of the other ways in which class declarations can be used.

These lectures also introduce UML's class diagrams as a way of expressing the dependencies of interfaces/classes.

6.2 Supporters of the use of interfaces

One of the areas where some useful work was done in the 1990s on understanding how complex systems are built was in recognizing the importance of the use of *pattern*s. Currently, the principal book in this area is 'Design Patterns: Elements of Reusable Object-Oriented Software' by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. In their book, they say:

'This ... leads to the following principle of reusable object-oriented design: *Program to an interface, not an implementation.* Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class [or an interface in Java]. You will find this to be a common theme of the design patterns in this book.'

Another book that covers similar material is 'Patterns in Java: Volume 1' by Mark Grand.

In his book 'UML Distilled', Martin Fowler writes:

‘One of the great qualities of object-oriented development is that you can vary the interfaces of classes independent of the implementation. Much of the power of object development comes from this property. However, few people make good use of it. Programming languages use a single construct, the class, which contains both interface and implementation. When you subclass, you inherit both. Using the interface as a separate construct is rarely used, which is a shame.’

On the WWW, at http://www.chimu.com/publications/javaStandards/, you will find a document entitled ‘ChiMu OO and Java Development: Guidelines and Resources’. In this document, the ChiMu Corporation say:

‘Use interfaces as the glue throughout your code instead of classes: define interfaces to describe the exterior of objects (i.e., their Type) and type all variables, parameters, and return values to interfaces. The most important reason to do this is that interfaces focus on the client’s needs: interfaces define what functionality a client will receive from an Object without coupling the client to the Object’s implementation. This is one of the core concepts to OO.’

In this document, Mark Fussell has a lot of useful ideas. It’s well worth reading.

6.3 Non-supporters of the use of interfaces

Apart from the book that I’m currently preparing, I’ve found no books on teaching Java as a first language thapt teach this role of interfaces. Often the material on classes and interfaces are presented in the following order: class, implementation inheritance, dynamic binding, abstract class, interface. Thus, interfaces are described as a deviant form of abstract class which is itself a rather strange form of superclass.

In an informal analysis of when specific aspects of Java are taught by various books, I obtained the following results. When a result appears as *n:p*, *n* gives the page number and *p* gives the page number expressed as a percentage. So the percentage figure gives a measure of how far through the book the aspect is taught.

```
  Arnow  Bishop    Deitel  Garside  Horstmann  Koffman   Lewis   Smith Cornelius
How many pages are in the main part of the text?  How many in all?
725,805 479,508 1228,1344  609,660    569,624  683,824 559,857 433,474 ?470,?500
When is the user-declared class introduced?
  90:12   80:17    326:27   170:28     324:57    56:08  140:25   66:15    206:44
When are interfaces introduced?
 659:91  269:56    427:35   347:57         no   440:64  338:60  269:62    223:47
```

7 Minimal public interfaces

7.1 What is meant by *minimal public interface*?

When teaching interfaces and classes, I stress that it is important that each interface-class has:

- methods called `equals`, `hashCode`, `toString`, and (if appropriate) a method called `compareTo`;
- a constructor that initializes the object from a parameter of type `String`;
- get methods;
- set methods (if appropriate);
- some means for cloning an object.

Some of these (e.g., `equals`, `hashCode`, `compareTo`) are needed if objects of the class are to be stored in a collection. When we produce a class, we may be uncertain as to what clients will want to do with objects of the class. I don’t believe we should change the class later or produce a subclass later. Instead, we should provide these things at the outset.

7.2 The need to provide `equals` and `hashCode`

I find it incredible that so many books on Java introduce classes as a means of representing real-world objects and yet do not provide each class with the ability to find out whether two objects of the class have the same value.

Suppose a book introduces a class called `Date`. Then, if it does mention `equals`, often you will find it declared with a parameter of type `Date`:

```
public boolean equals(Date pDate)
{
   return iYear==pDate.iYear &&
          iMonth==pDate.iMonth && iDay==pDate.iDay;
}
```

If a client attempts to use objects of this class with objects of the Collections API (or `Hashtable` or `Vector`), they are in for a shock. None of the following methods will work with that declaration of `equals`:

`Hashtable`	`contains`, `containsKey`, `get`, `put`, `remove`

Vector	contains, indexOf
List	contains, remove, indexOf
Map	containsKey, containsValue, get, put, remove
Set	add, contains, remove

Instead, you need to teach that for the `equals` method to be useful it needs to have a parameter of type `Object`. For example, if we are providing an interface called `Date` and a class called `DateImpl`, we could declare the following:

```
public boolean equals(final Object pObject)
{
   if ( ! (pObject instanceof DateImpl) )
   {
      return false;
   }
   return iYear==((DateImpl)pObject).iYear &&
          iMonth==((DateImpl)pObject).iMonth &&
          iDay==((DateImpl)pObject).iDay;
}
```

If you declare `equals` properly, you need also to declare `hashCode`. There are warnings about this in the documentation of some of these classes. For example, the WWW pages that document `java.util.Hashtable` states that 'to successfully store and retrieve objects from a hashtable, the objects used as keys must implement the `hashCode` method and the `equals` method.'

By this, it means that a class should override the methods called `hashCode` and `equals` that are declared in `java.lang.Object`. So you must declare methods with the following headers:

```
public boolean equals(Object pObject);
public int hashCode();
```

Here is a possible coding for the `hashCode` method for `Date` and `DateImpl`:

```
public int hashCode()
{
   return iYear*416 + iMonth*32 + iDay;
}
```

The informal analysis of when specific aspects of Java are taught by various books gave the following results:

```
  Arnow  Bishop    Deitel  Garside  Horstmann  Koffman   Lewis   Smith Cornelius
Does the book suggest that equals always be declared?
   lost      no        no       no     339:60       no    lost    lost    219:47
Does the book teach the proper declaration of equals?
   lost      no        no       no     340:60       no      no   wrong    221:47
Does the book teach the proper declaration of hashCode?
     no      no        no       no         no       no      no  311:72    223:47
```

7.3 The need to provide a cloning operation

Few books explain that, when producing an interface and class, it is desirable to provide a cloning operation. For example, when creating a `PersonImpl` object, we might let a client supply a `DateImpl` object that is the person's date-of-birth.

```
Date tBirthDate = new DateImpl(2000, 1, 24);
Person tSomePerson = new PersonImpl("Joe", tBirthDate);
```

If we use:

```
public class PersonImpl
{
   private String iName;
   private Date iDateOfBirth;
   public PersonImpl(final String pName,
                     final Date pDateOfBirth)
   {
      iName = pName;
      iDateOfBirth = pDateOfBirth;               // share
      ...
```

then the `PersonImpl` object is sharing the `DateImpl` object supplied by the client. If we do not trust the client, it would be safer for the constructor to use:

```
      iDateOfBirth = new DateImpl(pDateOfBirth);  // clone
```

Note that you need not do this copying if the objects of the class (in this case, the class `DateImpl`) are immutable.

The classes of the Core APIs use two different ways of producing a copy of an object:

- a class sometimes provides a method called `clone` that overrides `java.lang.Object`'s `clone`;
- a class sometimes provides a suitable constructor.

It is best to provide a method called `clone` as this can be used when inheritance is involved. However, getting the code of a `clone` method completely right is difficult. See the article by Bill Venners at http://www.artima.com/innerjava/webuscript/cloning.html

Because it is hard to get right, it is also difficult to teach. Instead, I cheat by just providing a constructor that can be used for cloning:

```
public DateImpl(final Date pDate)
{
   iYear = pDate.getYear();
   iMonth = pDate.getMonth();
   iDay = pDate.getDay();
}
```

7.4 The need to provide `compareTo`

If the interface and class that you are providing is for a type where there is a natural order for the values of the type, the interface and class should also provide a means for finding out whether one value of the type is less than another value. In fact, there are some parts of the Collections API (e.g., `TreeSet` and `TreeMap`) that work better if your class implements the interface `Comparable` (from the package `java.lang`).

This interface is simply:

```
public interface Comparable
{
   public int compareTo(Object pObject);
}
```

In order for `Date` and `DateImpl` to do this, we need to change the header of the interface `Date` to be:

```
public interface Date extends Comparable
```

and to add the following declaration to the class declaration for `DateImpl`:

```
public int compareTo(final Object pObject)
{
   DateImpl tDateImpl = (DateImpl)pObject;
   int tResult = iYear - tDateImpl.iYear;
   if (tResult==0) {
      tResult = iMonth - tDateImpl.iMonth;
      if (tResult==0) {
         tResult = iDay - tDateImpl.iDay;
      }
   }
   return tResult;
}
```

The `Comparable` interface became part of Java when Java 2 was released in December 1998. Although most of the books that teach Java as a first language were prepared before then, a few have since been updated for Java 2. None of the ones that I've seen discuss the possibility of a class implementing the `Comparable` interface.

7.5 Supporters of minimal public interfaces

I've nicked the phrase *minimal public interface* from a book called 'Object-Oriented Design Heuristics' by Arthur Riel. He says: 'If the classes that a developer designs and implements are to be reused by other developers in other applications, it is often useful to provide a common minimal public interface. This minimal public interface consists of functionality that can be reasonably expected from each and every class.'

Bill Venners has also written an article on the topic of minimal public interfaces. It is entitled 'The canonical object idiom'. You will find it at http://www.javaworld.com/javaworld/jw-10-1998/jw-10-techniques.html

7.6 Non-supporters of minimal public interfaces

I've found no books on teaching Java as a first language that stress the importance of always providing a minimal public interface.

8 A turning point

The previous sections of this paper have described the contents of the first 20 lectures. This is a turning point in the course. Although there are a few important aspects of the Java language that are still to come, most of the important ones have now been introduced. The remaining lectures of the course are mainly concerned with introducing two important APIs: the Swing API and the Collections API.

Although both of these APIs came into being when Java 2 was released, they are both considered by Sun to be part of the Core APIs of Java.

One of the main aims of the second half of the course is to become familiar with these two APIs. They are both quite large: the course's aim is to introduce the basic ideas giving the students sufficient knowledge to look at the other parts of the API on their own.

Note the emphasis in this part of the course is on using the code of existing libraries: by taking a ride on the work of others, Java gives even the beginner the possibility of easily building reasonably powerful pieces of code.

9 The Swing API

The course looks at the Swing API in two takes. In the first take, the students are introduced to the following ideas:

- creating a new window using `JFrame`;
- putting GUI components such as `JButtons` and `JTextFields` into the pane of a window;
- introducing an object to listen for the event of a click of a button;
- grouping components together using a `Box` (in preference to a `JPanel`);
- providing a form to allow multiple values to be input;
- using a `JDialog` to force the user to complete the form.

In order to use the Swing API, a programmer is required to be familiar with the idea of declaring a class that implements an interface. In this course, the lectures on the Swing API appears immediately after the lectures that introduce interfaces and classes.

In later lectures, the classes `JMenuBox`, `JMenu` and `JMenuItem` are introduced. These three classes are needed to build a menu system. Because `JMenuItems` are another form of button, responding to a click of a menu item can be handled in a similar way to that for buttons (which were discussed in earlier lectures).

10 The Collections API

The Collections API provides a useful set of interfaces and classes that can be used for building data structures that represent collections of data. The emphasis in this course is on how to make effective use of this API rather than on how to implement the various data structures.

The course describes and explains the differences between the capabilities of the three main interfaces called `List`, `Set` and `Map` (and subinterfaces called `SortedSet` and `SortedMap`), and the six classes called `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap` and `TreeMap`. The use of the `Iterator` interface is also described.

Once again, the stress is to write the code of clients in terms of the interface, only mentioning the name of a class when you want to create a collection object.

Although later lectures show how to implement the `List` interface by (a) an array and (b) a doubly linked list, this only appears after extensive use in earlier lectures of the `List`, `Set` and `Map` interfaces.

11 The use of MVC and the Observer pattern

Whilst introducing the Collections API, the course explains and demonstrates the advantages of using the Model-View-Controller pattern to decouple a data structure from any user interface. The simpler form in which the view and the controller are combined (sometimes called Model-UserInterface) is also discussed. By having the class that implements the model derived from `java.util.Observable`, it is easy to provide multiple views of the model (the class for each view implementing the `java.util.Observer` interface). These ideas are illustrated by two examples.

The course develops a case study for which different user interfaces are developed and, also, different data structures are used. Both the user interface and the data structure are easily switched because of the use of interfaces and MVC.

12 Implementation inheritance

Often books on Java teach the following topics in one chapter:

- deriving classes from base classes
- polymorphism (dynamic binding)
- abstract classes
- interfaces

These topics are taught in this order because it seems to be an obvious order: first introduce the advantages of deriving classes from base classes; then explain polymorphism (dynamic binding); the explain that an abstract class is a special form of base class; and finally introduce an interface as a special form of an abstract class. However, a chapter that has these topics introduces a lot of new ideas and explaining them all at once will overload the reader.

Although there is some logic to teaching interfaces in this way, it is better to make a clearer separation between implementation inheritance and interfaces. I think it is important to teach interfaces early, and to leave implementation inheritance until later.

Although the lectures on implementation inheritance appears late in this course, some of the basic ideas are introduced in earlier lectures:

- When `toString`, `equals` and `hashCode` are introduced, you can also introduce the idea that, by default, classes are derived from `Object`; that you can use methods of the class `Object` on other objects; and that methods of a superclass can be overriden.
- When discussing the classes of the Swing API you can demonstrate the use of implementation inheritance and explain the usefulness of organising classes in a class hierarchy;
- When using `java.util.Observable`, you need to produce a class that is derived from a class of the Core APIs and explain why this is being done.

It is only towards the end of this course that the students learn how to derive a class from one of their other classes. These lectures also introduce the idea of abstract classes.

13 Applets

Because the full details of implementation inheritance are taught late, the material on applets also appears late. However, because all the ground work has been covered, there is little that needs to be taught about applets: just the role of the `init`, `start`, `stop` and `destroy` methods.

14 Exceptions

Some books seem to include exceptions as an afterthought: often the whole topic is taught in the last chapter. As with implementation inheritance, my course gradually introduces the various aspects of exceptions:

- Exceptions first need to be mentioned when discussing the manipulation of `Strings` and when values are read from the keyboard. So the students first meet the exceptions called `StringIndexOutOfBoundsException` and `IOException`. A distinction can be made at this stage between unchecked exceptions and checked exceptions. So, the throws clause for `IOException` on the main method is also explained.
- When method declarations are introduced, you can explain why a throws clause is needed with a method declaration as well as the main method.
- The lectures on control statements can introduce the use of a try statement to handle an exception, e.g., to handle `NumberFormatException`.
- The lectures on using files can introduce the need for a finally clause.

Although in Java it can be tempting to overuse exceptions, in this course it is recommended that they are only used for untoward situations. For this reason, the idea of introducing your own exceptions is briefly only mentioned at the end of the course. A good article on the topic is *Designing with exceptions* by Venners. You will find it at http://www.javaworld.com/javaworld/jw-07-1998/jw-07-techniques.html

15 Keyboard input

Often, books that teach Java provide their own package of classes. One reason for doing this is that it makes it easy to provide a class that facilitates the reading of a value of some primitive type from the keyboard.

Although this approach has its advantages, it is also inconvenient for students:

- if a student has their own computer, they have to obtain the package and install it;
- if they read more than one book, they have to become familiar with more than one package;
- they can easily fall into the trap of always using a book's package and never shifting to using the classes defined in Java's Core APIs.

In this course, the horrors of Java's keyboard input are introduced early:

```
BufferedReader tKeyboard = new BufferedReader(new InputStreamReader(System.in));
String tHeightString = tKeyboard.readHeightString();
double tHeight = Double.parseDouble(tHeightString);
```

Note the use of both instance methods and class methods and the need to use the wrapper classes of the primitive types.

16 Teaching devices

- A consistent style is used for the layout of the code that are presented in the course.
- A naming convention is used for the identifiers used in the code. This convention uses a lower-case letter as a prefix to indicate where an identifier is declared, i.e., `i` is used for a private member of a class, `p` is used for a parameter, and `t` is used for a temporary (i.e., a local) variable of a block.
- The course uses syntax diagrams to explain the syntax of the language. To make things easier, some simplifications are made to the syntax.
- The course uses UML's class diagrams to provide a visual representation of the dependencies that exist between the interfaces/classes of a program.
- The course presents tables containing a description of the some of the constructors and methods of commonly-used classes.
- The lectures of the course often refers the student to the WWW pages that document the Core APIs.
- Each chapter of the notes supporting the course has a collection of tips (i.e., hints about programming pitfalls and debugging hints).
- Each chapter of the notes supporting the course has a section entitled *curios, controversies and cover-ups*.
- Each chapter of the notes supporting the course has a set of exercises.

17 Appendix: the order in which the material is taught

1. Examining the structure of Java programs
2. Constructing simple programs
3. Using objects and classes from Java's Core APIs
4. Reading values from the keyboard
5. Using arithmetic expressions to represent formulae
6. Using if statements to make decisions
7. Using for statements to repeat statements
8. Using methods to organize programs
9. Using other forms of control statements
10. Reading from and writing to files
11. Using interfaces and classes to produce new types
12. Looking at other examples of interfaces and classes

13. Providing a GUI: buttons and textfields
14. Manipulating collections: lists, queues and stacks
15. Decoupling the model from the user interface
16. Using enumerations, sets and maps
17. Providing a GUI: menus
18. Using arrays and implementing the List interface
19. Deriving a class from another class
20. Writing applets (for use with the WWW)
21. Looking briefly at other topics: (a) starting another thread, (b) declaring your own packages, (c) the factory pattern, (d) the JDBC API, (e) client-server computing, (f) generic types, (g) standardization, (h) finding information about Java.

18 References

- Andreae et al, 'Surprises in Teaching CS1 with Java',
  http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-98-9.abs.html
- David Arnow and Gerald Weiss,
  'Introduction to Programming Using Java: An Object-Oriented Approach (Java 2 Update)',
  Addison-Wesley 2000, 0-201-61272-0.
- Robert Biddle and Ewan Tempero, 'Learning Java: Promises and Pitfalls',
  http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-97-2.abs.html
- Judy Bishop, 'Java Gently (2nd edition)',
  Addison-Wesley, 1998, 0-201-34297-9.
- ChiMu Corporation (Mark Fussell), 'ChiMu OO and Java Development: Guidelines and Resources',
  http://www.chimu.com/publications/javaStandards/
- Barry Cornelius, 'Understanding Java',
  Addison-Wesley (in preparation).
- Harvey Deitel and Paul Deitel, 'Java: How to Program (3rd edition)',
  Prentice-Hall, 1999, 0-13-012507-5.
- Martin Fowler (with Kendall Scott), 'UML Distilled',
  Addison-Wesley, 1997, 0-201-32563-2.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides,
  'Design Patterns: Elements of Reusable Object-Oriented Software',
  Addison-Wesley, 1995, 0-201-63361-2.
- Roger Garside and John Mariani, 'Java: First Contact',
  International Thompson, 1998, 1-85032-316-X.
- Mark Grand, 'Patterns in Java: Volume 1',
  John Wiley, 1998, 0-471-25839-3.
- Cay Horstmann, 'Computing Concepts with Java Essentials',
  John Wiley, 1998, 0-471-17223-5.
- Elliot Koffman and Ursula Wolz, 'Problem Solving with Java',
  Addison-Wesley, 1999, 0-201-35743-7.
- Doug Lea, 'Some Questions and Answers about using Java in Computer Science Curricula',
  http://gee.cs.oswego.edu/dl/html/javaInCS.html
- John Lewis and William Loftus, 'Java Software Solutions',
  Addison-Wesley, 1998, 0-201-57164-1.
- Arthur Riel, 'Object-Oriented Design Heuristics',
  Addison-Wesley, 1996, 0-201-63385-X.

- Michael Smith, ‘Java: an Object-Oriented Language’, McGraw-Hill, 1999, 0-07-709460-3.
- Bill Venners, ‘Cloning’, http://www.artima.com/innerjava/webuscript/cloning.html
- Bill Venners, ‘Designing with exceptions’, http://www.javaworld.com/javaworld/jw-07-1998/jw-07-techniques.html
- Bill Venners, ‘The canonical object idiom’, http://www.javaworld.com/javaworld/jw-10-1998/jw-10-techniques.html