Further Java

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Computer Science Tripos Part 1B
Computer Science Tripos Part 2 (General)
Diploma in Computer Science

Timothy Harris

Michaelmas 2000

Further Java

Computer Science Tripos Part 1B
Computer Science Tripos Part 2 (General)
Diploma in Computer Science

Introduction

The course will develop an understanding of general programming techniques using advanced features of the Java programming language as a vehicle.

On completing the course, students should be able to:

- Understand and apply techniques for building applications from classes, combined using inheritance, composition, nested classes, reflection and interfaces
- Describe and use multi-threaded applications with appropriate synchronisation
- Explain the rôle of abstraction and concurrency in a GUI toolkit, using the AWT as an example
- Explain the components of a distributed program and its implementation with RMI.

An elementary familiarity with Java will be assumed.

Syllabus

- **Programming with objects**. Inheritance and interfaces. Exceptions. Nested classes. Reflection. [3 lectures]

- **Concurrency**. Creating multiple threads. Concurrency control using locks and *wait*/*notify* operations. The *volatile* modifier. [3 lectures]

- **Graphical user interfaces**. The Abstract Windowing Toolkit, Java Beans, Java Studio. [3 lectures]

- **Distributed computing**. Sockets and remote method invocation. [2 lectures]

- **Application programming**. Project management and development environments, Java Workshop and Visual Studio for Java. [1 lecture]

It should be pointed out that these notes do not constitute a complete transcript of all the lectures and they are not a substitute for text books. They are intended to give a reasonable synopsis of the subjects discussed and include fragments of programs that will be used for illustration, but they do not give all the background material.

Separate material will be distributed containing the slides used in the lectures. All of these documents are available on the Laboratory's *Teaching Course Material* web pages. The example programs contained in these notes are available on the *thor* teaching system (in */group/clteach/tlh20/further-java*) and available for download from the Laboratory's web site for use with the Cockroft 4 Linux installation.

Appropriate books

The following reference books are relevant for the course:

- Arnold, K. & Gosling, J. (1997). *The Java Programming Language*. Addison-Wesley (2nd ed.)

- Bracha, G., Gosling, J., Joy, B. & Steele, G. (2000). *The Java Language Specification*. Addison-Wesley (2nd ed.).

- Gosling, J. & Yellin, F. (1996). *The Java Application Programming Interface, vol. 1. Core Packages*. Addison-Wesley

- Gosling, J. & Yellin, F. (1996). *The Java Application Programming Interface, vol. 2. Window Toolkit and Applets*. Addison-Wesley

  The following books take a more tutorial stance. They are less suitable as reference texts, but are more approachable to those with less practical programming experience.

- Eckel, B. (1998). *Thinking in Java*. Prentice-Hall

- Flanagan, D. (1997). *Java in a Nutshell*. O'Reilly (2nd ed.)

- Flanagan, D. (1997). *Java Examples in a Nutshell*. O'Reilly (2nd ed.)

The Java FAQ is a collection of questions that the designers of the Java Programming Language have been frequently asked. The answers provide insight into the design of the language and how and why it differs from other object-oriented languages.

- Kanerva, J. (1997). *The Java FAQ*. Addison-Wesley

The most comprehensive of these is *The Java Language Specification* – it is the official definition of the Java Programming Language, produced by its designers at Sun Microsystems. It serves as a thorough reference to the finer details of the language.

There is a substantial volume of material on the world wide web concerning the Java Programming Language. There are links to selected items from the Laboratory's *Teaching Course Material* web page – in particular many of the Sun specification documents are available free of charge over the web. The Computing Service's UNIX Support web site contains further documentation, and local mirrors of the Sun material, at *http://www-uxsup.csx.cam.ac.uk/java/*.

Beware of text found by casual web browsing or FAQs: some of it is mis-informed and, although of practical use, much of it misses the details or rationale which will be covered in this course.

Programming in Java

This section presents a brief summary of the earlier Java courses.

Hello world

```
// It's that "Hello world" program again!

class HelloWorld {
   public static void main (String [] args) {
      System.out.println ("Hello world!");
   }
}
```

This would be placed in a file HelloWorld.java and compiled with the command

```
javac HelloWorld.java
```

This would create a file HelloWorld.class which could be run with the command

```
java HelloWorld
```

Counting words

Here is a program to count words in files of text:

```
// Count words on named files or standard input

import java.io.*;

class WordCount {

   public static void main (String [] args)
   {
      try {
         if (args.length == 0) count ("Standard input",
                                      System.in);
         else {
            for (int a = 0; a < args.length; a++) {
               try {
                  InputStream is =
                              new FileInputStream (args [a]);
                  count (args [a], is);
               }
               catch (FileNotFoundException e) {
                  System.err.println ("Unable to open " +
                                      args [a]);
               };
            };
         };
      }
      catch (IOException e) {
         System.err.println ("IOException " +
                             e.getMessage ());
      };
   };

   private static void count (String name, InputStream in)
      throws IOException
   {
      int words = 0, lines = 0;
      boolean inWord = false;
      DataInputStream dis = new DataInputStream (in);
      try {
         for (;;) {
            char ch = (char) dis.readByte ();
            switch (ch) {
               case '\n':
                  lines++;
               case ' ':
               case '\t':
                  if (inWord) {
                     words++;
```

```
                    inWord = false;
                 };
                 break;
              default:
                 inWord = true;
                 break;
           };
        };
     }
     catch (EOFException e) {
        if (inWord) words++;
        System.out.println (name + " contains " + words +
           " words in " + lines + " lines.");
     };
  }

}
```

This looks for command line arguments, which it takes to be the names of files to examined. In the absence of such arguments it processes the standard input.

Notation

Java programs are written in a free format using Version 2.0 of the Unicode character set [http://www.unicode.org/]. Comments can be bracketed by /* and */ or prefixed by // in which case they run to the end of the line. A special form of the bracketed comments, starting /**, is used for documentation. In this leading white space and * characters are discarded. Moreover HTML markers (other than <H*n*> and <HR>) may be used together with tags:

`@see`	cross-references to code or general URLs
`@author`	
`@version`	
`@param`	parameters for methods

`@return`	result of method
`@exception`	conditions for exceptions to be raised

Reserved words are written in lower case. Curly brackets { and } are used to enclose blocks of code.

Declarations

There are two main forms of declaration in Java: variables and classes (that is, object types). These share a similar syntax consisting of optional qualifiers and the type name followed by the identifier being declared. The results of methods precede their declarations. Thus the procedure to count words becomes a method:

```
private static void count (String name, InputStream in)
      throws IOException
{
      int words = 0, lines = 0;
      boolean inWord = false;
      DataInputStream dis = new DataInputStream (in);
```

Constants are represented as variables qualified by the keyword final, which renders them immutable. By convention identifers in upper case are used. Exceptions and constructed types are represented as classes and procedures are represented as methods. There are no equivalents of enumerations, sets, subranges or records other than classes. There are no explicit references; arrays and classes are implicitly reference types.

Reserved words

The reserved words in Java are: abstract, boolean, break, byte, byvalue, case, cast, catch, char, class, const, continue, default, do, double, else, extends, false, final, finally, float, for, future, generic, goto, if, implements, import, inner, instanceof, int, interface, long, native, new, null, operator, outer, package,

private, protected, public, rest, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, var, void, volatile and while. Some of these are currently unused but reserved for future developments.

Every object has the following methods which should not be overridden casually: clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString and wait.

Built-in types

byte short int long	8 bits 16 bits 32 bits 64 bits Literals in decimal, octal (prefixed by 0) or hexadecimal (prefixed by 0x) and followed by l (or L) for long if appropriate. So 42 = 052 = 0x2a = 0x2A.
boolean	Literals true and false
char	16 bit Unicode with literals \b, \t, \n, \f, \r, \", \', \\, \u*abc* and \u*abcd*
float double	Literals followed by f or d (the default)
String	Literals enclosed in double quotes, "This is a String".

The type void is used to represent the result of sub-routines that do not return one.

Expressions and assignment

A single equal sign is used for assignment but the assignment statement also returns its value; unwanted values are discarded automatically. The operators in Java in order of decreasing precedence are:

Prec.	Operator	Type	Operation
1	+, -	integral or real	unary plus, minus
	~	integral	bitwise complement
	++, --	integral or real	pre- or post- increment, decrement
	!	boolean	logical complement
	(*type*)		cast
2	*, /, %	integral or real	multiplication, division, remainder
3	+, -	integral or real	addition, subtraction
	+	string	concatenation
4	<<	integral	left shift
	>>, >>>	integral	arithmetic, logical right shift
5	<, <=, >, >=	integral or real	comparison
	instanceof	object	comparison of types

6	==, !=		equal, not equal
7	&	integral or boolean	bitwise or boolean conjunction
8	^	integral or boolean	bitwise or boolean symmetric difference
9	|	integral or boolean	bitwise or boolean disjunction
10	&&	boolean	conditional AND
11	||	boolean	conditional OR
12	? :	boolean	ternary conditional operator
13	=		assignment
	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=		assignment with operation

Operators of precedence 1, 12 and 13 are right associative; all the others are left associative.

Casting converts representation where necessary, so (int) 3.14159 returns the value 3. This particular use applies truncation towards 0.

Automatic conversions widen, narrow and generate string representations of values when necessary in expressions. For example, 1 / (1000 * n + Math.PI) yields a double (being the

type of Math.PI). If this was being assigned to a float, it would have to be narrowed with an explicit cast.

If an object has a toString method, this will be invoked to generate a textual representation for use in string concatenation.

The bitwise and boolean logical operators (&, ^ and |) evaluate both of their operands before combing them; the conditional variants (&& and ||) do not evaluate the right-hand operand when the result is fully determined by the left-hand operand.

The ternary conditional operator `b ? t : f` evaluates the boolean expression b and returns t if it is true and f otherwise.

Sequential control structures

Semi-colons are used to conclude statements rather than to separate them.

if (*boolean*) *statement* else *statement*

The boolean expression to be evaluated is enclosed in round parentheses, the else clause is optional. Only single statements are allowed; blocks must be enclosed in curly braces. Each else binds to the most recent if that does not already have one.

```
if (cholesterol > 6.5) status = “high risk”;
else if (cholesterol >= 5.5) status = “moderate”;
     else status = “acceptable”;
```

switch (*expression*) { … }

The switch statement compares an ordinal value against a series of templates. Labels are introduced by the keyword case and control falls through until a break is reached.

```
dvsn = 0;
switch (marks) {
  case 1:
  case 2:
  case 3:
    clss = 3;
    break;
  case 4:  case 5:  case 6:
    dvsn++;
  case 7:  case 8:  case 9:
    dvsn++;
    clss = 2;
    break;
  case 10:
    clss = 1;
    break;
  default:
    Doom!
    break;
  };
```

If none of the cases match the expression, execution continues after the closing brace.

while (*boolean*) *statement*

The boolean expression is evaluated and the statement executed repeatedly as long as it remains true. So an approximation to $\log_2 n$ could be calculated by:

```
int b = 0;
while (n > 1) {
  b++;
  n /= 2;
};
```

This example also shows how variables can be declared at any point in a program and also the use of increment and operators with assignment.

An infinite loop simply uses the constant value true as the boolean expression. So the sum of an infinite series could be calculated by:

```
while (true) {
  float next = sum + f (n);
```

```
  if (next == sum) break;
  sum = next;
  n++;
};
```

do *statement* while (*boolean*)

The do statement is similar to a while, but always executes at least once:

```
do {
  n++;
  prev = sum;
  sum = prev + f (n);
} while (sum != prev);
```

for (*initialization*; *boolean*; *increment*) *statement*

The brackets enclose an initialization expression, which can consist of several statements (including variable declarations) separated by commas, a boolean expression and an increment expression which can also consist of several statements. The statement is equivalent to:

```
initialization;
while (boolean) {
  statement;
  increment;
};
```

For example, the terms of a series might be added as follows:

```
sum = 0.0f;
for (int t = n; t >= 0; t--)
  sum += 1 / (1000 * t + Math.PI);
```

Compound initialization and increment expressions could be used to calculate the approximation to $\log_2 n$:

```
int i, b;
for (i = n, b = 0; i > 1; b++, i /= 2);
```

This example leaves no work to be done in the body of the loop so it is omitted. All of the expressions in the for construct are optional and the default boolean value is true. Thus the idiomatic way to write an infinite loop in Java is:

`for (;;)` *statement*

Transfer of control

A break statement transfers control to the end of the smallest enclosing loop or switch statement.

A continue statement skips to the end of the smallest enclosing loop and re-evaluates the boolean expression controlling its execution.

Statements can be labelled by prefixing them with an identifier followed by a colon. Both break and continue can be followed by the label of an enclosing block (not necessarily the smallest such), in which case control transfers to the appropriate point in that block.

return is used to conclude the execution of a method and is followed by an expression if the method returns a result.

Importing identifiers

The naming scheme in Java aims to provide unique identifiers across the Internet. A fully qualified name consists of a package name, a class name and a method name, separated by full stops. The package name itself can consist of several components, starting with an Internet domain name (although the components are written in decreasing order of significance). The prefix java is used for standard packages.

Public names can always be used in fully qualified form without being specifically imported. However, the import statement allows shorter names to be used when this does not give rise

to any ambiguity. Importing a class allows that class name to be used in unqualified form. All the classes in a package can be imported by using an asterisk as the class name:

```
import uk.ac.cam.cl.tlh20.teaching.introduction.*
```

This simply saves typing. The actual package must still be available via the CLASSPATH environment variable. This is a list of directories (separated by colons) and the actual classes will reside in a subdirectory whose path is derived from the components in the package name.

The default package (with no name) and java.lang.* are always imported implicitly.

Exceptions

Exceptions in Java are just classes derived from the Exception type. In fact Exception is a sub-type of Throwable which also includes the Error type to represent system failures.

Signatures of methods include a list of exceptions that may arise during the execution of the corresponding code. This consists of the word throws followed by a comma-separated list of exceptions.

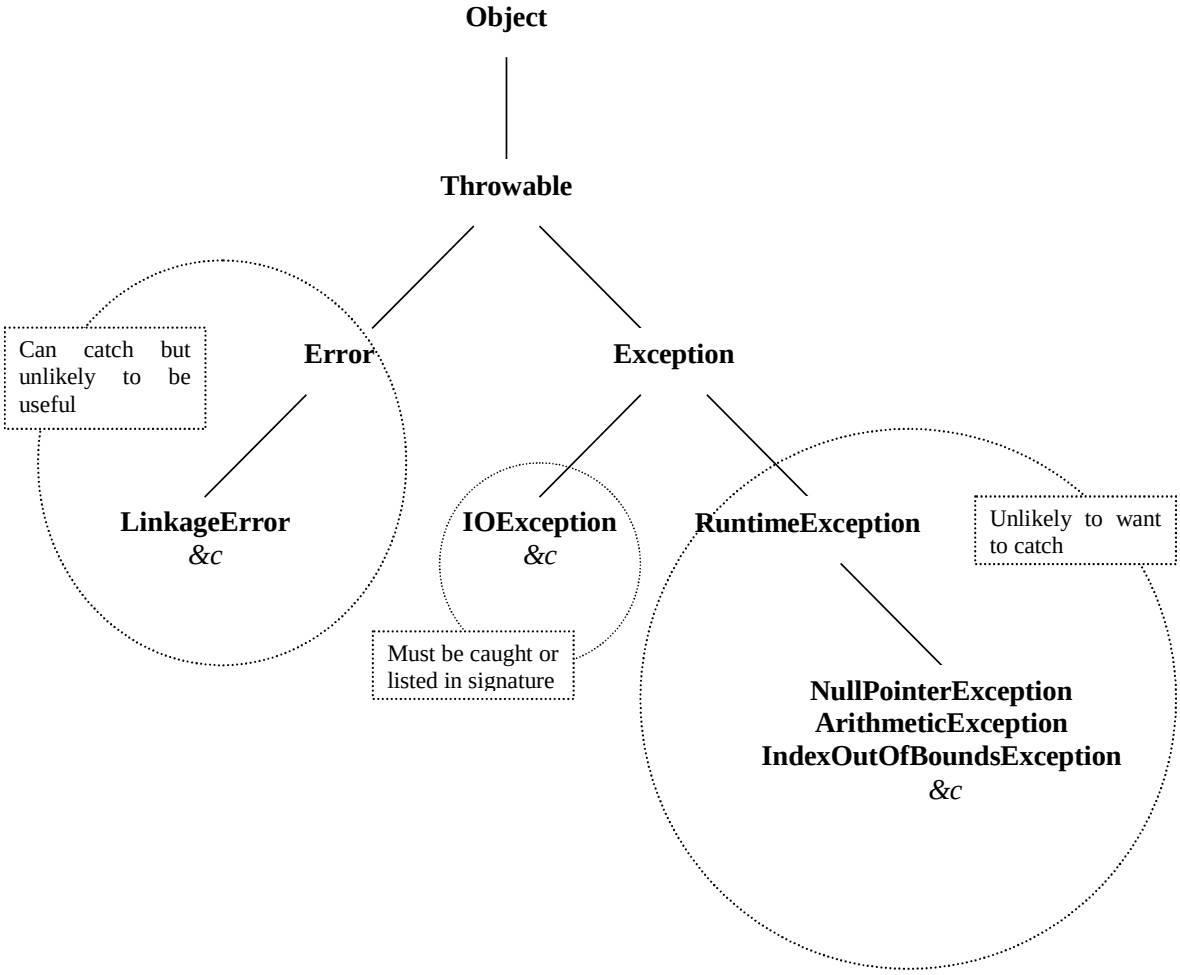
An exception is raised in a statement consisting of the keyword throw followed by an instance of the appropriate exception type. This might well be created dynamically using new.

Exception handling takes place in a unified try statement:

```
try {
   main code to be executed
}
catch (ExceptionOne e) {
   deal with exceptions of type ExceptionOne possibly calling
   methods like e.getMessage()
}
catch (ExceptionTwo e) {
```

```
   deal with exceptions of type ExceptionTwo
}
finally {
   code to restore invariants always executed, even after catching exceptions
};
```

The hierarchy of exception classes has Throwable at its root, with Error and Exception as the immediate descendants, so catching Throwable t will pick up any exception. RuntimeException is derived from Exception.



Instances of Error, RuntimeException and their subclasses are known as *unchecked exceptions* and relate to major failures. All other exceptions are *checked*, which means that they must be caught or specified in throws clause of any methods in which they might arise.

Compound data types

Previous sections have described *primitive types* such as *integer* and *reference types* that refer to instances of classes – i.e. objects – these are the main data types in Java. One further addition is arrays or, more strictly, references to open arrays. An array type is written as the element type followed by square brackets, []. They are reference types, so space has to be allocated with new:

```
int [] ai = new int [10];
```

This allocates a vector of 10 consecutive locations capable of holding integers and returns their address. Individual elements of the array are accessed as ai [0] up to ai [9].

Every array has a field length that indicates how many entries it has. Thus, in the example above, ai.length would be 10. Here is another example:

```
static int dot (int [] a, int [] b) {
    int sum = 0;
    for (int i = 0;
         (i < a.length) && (i < b.length);
         i++) {
      sum += a[i] * b[i];
    };
    return sum;
  };
```

Another example is the array of strings that is passed to the main method of a Java program. This contains the arguments typed on the command line that invoked the program (excluding the actual program name itself). Look back at the word-counting program above.

It is also possible to initialise an array to a particular set of values:

```
int [] primes = {2, 3, 5, 7, 11, 13, 17};
```

The brackets can be written after the identifier being declared:

```
int ai [] = new int [10];
int primes [] = {2, 3, 5, 7, 11, 13, 17};
```

Multi-dimensional arrays

Multi-dimensional arrays are just arrays of arrays:

```
int [] [] mi = new int [3] [3];
```

These can also be initialised:

```
int [] [] factors = {{2}, {3}, {2, 4}, {5}, {2, 3, 6}, {7}};
```

factors [4] [2] would be 6. The abbreviation factors [4, 2] is not available. For example:

```
static void printMatrix (int [] [] m) {
    for (int i = 0; i < m.length; i++) {
      for (int j = 0; j < m[i].length; j++) {
        System.out.print (m[i][j] + "\t");
      };
      System.out.println ();
    };
  };
};
```

When allocating a multi-dimensional array, it is possible to omit trailing dimensions:

```
int [] [] factors = new int [7] [];
```

The resulting arrays has 7 elements, each of type int []. These could then be allocated and initialised (possibly with different lengths) by assigning to them.

Objects and methods

Objects can be regarded as references to records containing data fields and also methods. The word class is used for an object type and object will usually refer to an instance of that type.

The syntax for a class definition is:

```
[modifiers] class Name [extends Parent] {
 [modifiers] Type field = expression;
 …

 [modifiers] Name (T1 a1, T2 a2, …) {
      instance constructor code…
 };

 [modifiers] [Result] method (T1 a1, T2 a2, …)
                                  throws E1, E2, … {
      method code…
 };

 …

 static {
      class initialisation code…
 }
};
```

By convention, class names have an initial capital letter but field and method names start in lower case.

It is possible to have several methods with the same name but different signatures. When one is invoked, the system looks at the pattern of actual arguments and calls the corresponding method. This can be used to simulate default values for parameters.

Code within methods can refer to fields in the object simply by name or as *this.name* if there is ambiguity (as might arise if field names were the same as argument names in a method).

A method with the same name as the class and returning no result (not even void) can be provided as a constructor to initialise objects in the class. Indeed, there can even be several of these taking different patterns of arguments. This can be invoked when an instance of the class is created using new.

All classes have some methods provided by default, but which it might make sense to override by redeclaring them. These include:

- public boolean equals (Object o)
  // compare contents rather than address

- public native int hashCode ()
  // calculate a hash value for the object

- public String toString ()
  // represent the object as text

- protected native Object clone ()
  // manufacture a copy of the object

- protected void finalize ()
  // tidy up before garbage collection

Here is an example of the use of classes to model complex numbers:

```
class ComplexExample {
  public static void main (String [] args) {
    Complex one = new Complex (1.0f),
              i = new Complex (0.0f, 1.0f),
            two = one.add (one),
              m = i.multiply (i);
    System.out.println ("one = " + one);   // (1.0, 0.0)
    System.out.println ("  i = " + i);     // (0.0, 1.0)
    System.out.println ("two = " + two);   // (2.0, 0.0)
    System.out.println ("  m = " + m);     // (-1.0, 0.0)
  };

};

class Complex {
  private float real, imaginary;

  Complex (float real) {
    this.real = real;
    this.imaginary = 0.0f;
  };

  Complex (float real, float imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  };

  Complex add (Complex c) {
    return new Complex (this.real + c.real,
                        this.imaginary + c.imaginary);
  };

  Complex multiply (Complex c) {
    return new Complex (this.real * c.real –
                         this.imaginary * c.imaginary,
                       this.real * c.imaginary +
                         this.imaginary * c.real);
  };

  public String toString () {
    return "(" + this.real + ", " + this.imaginary + ")";
  };
};
```

Including a *toString* method makes it simpler to print instances of the class. This is used by the debugger.

Modifiers

The modifiers restrict the use of classes, fields and methods:

- **abstract** means that the methods in the class have no code. It simply specifies the signature of a parent from which working classes can be derived. Abstract methods can only be declared in an abstract class.

- **final** means that a field or method can not be overridden. This prevents the behaviour of a class being perverted in a derived type. A final field is effectively a constant. A final class can not be sub-typed.

- **native** means that the code of a method is being provided in a language other than Java.

By default, identifiers have the package modifier. Each file of code can start with an optional package declaration to name the package to which it belongs.

bpublic, **protected**, **package** and **private** limit the scope of identifiers as follows:

	Modifier applied to identifier			
Identifier visible in	**public**	**protected**	**package**	**private**
Class in which it is defined	✓	✓	✓	✓
Another class in the same package	✓	✓	✓	
Derived class in another package	✓	✓		
Anywhere else	✓			

- **static** means that the field or method of the class should be available even if no instance of the class has been created. Static methods can be used to provide subroutine libraries like Math.sqrt or the main method of a class that is invoked as a program.

Static variables are shared by all instances of a class and can be initialised by including something that looks like a method with a static modifier but no name or signature in the class.

- **synchronized** means that the method acquires a lock associated with its instance.

- **transient** fields are omitted from serialised version of instances.

- **volatile** fields can be safely accessed asynchronously in multi-threaded programs.

Exceptions

Recall that exceptions are just classes derived from Throwable. By convention any such class has two constructors, one taking no arguments and one taking a string argument giving some sort of explanation. This string is returned by an instance's getMessage method.

Inheritance

Java provides single inheritance through the **extends** *Parent* part of a class signature. New methods can be added and old methods overridden by providing a new method with the same name, return type and arguments.

Suppose a class for hash tables had been written in a file Table.java as follows:

```
package table;

public class Table {
  private class Entry {

    String key, value;
    Entry next;

    Entry (String key, String value, Entry next) {
      this.key = key;
      this.value = value;
      this.next = next;
    };

    public String toString () {
      return "(\"" + key + "\" => \"" + value + "\") " + next;
    };

  };

  private Entry [] table;

  public Table (int size) {
    table = new Entry [size];
  };

  public String toString () {
    String s = "";
    for (int i = 0; i < table.length; i++)
      s += "[" + i + "] = " + table [i] + "\n";
    return s;
  };

  public void store (String key, String value)
                  throws DuplicateException
  {
    try {
      retrieve (key);
      throw new DuplicateException (key);
    }
    catch (MissingException e) {
      int h = key.hashCode() % table.length;
      table [h] = new Entry (key, value, table [h]);
    };
  };
```

```
 public String retrieve (String key)
                    throws MissingException {
    Entry e = table [key.hashCode() % table.length];
    while (e != null) {
      if (key.equals (e.key)) return e.value;
      e = e.next;
    };
    throw new MissingException (key);
  };

};
```

Actually, this needs a couple of supporting classes to declare the new exceptions:

```
package table;

public class DuplicateException extends Exception {

    public DuplicateException (String key) {
                    super ("Key \"" + key + "\" already in use");
    };

}


package table;

public class MissingException extends Exception {

    public MissingException (String key) {
                    super ("Key \"" + key + "\" not found");
    };

}
```

A test program, TableTest.java could then use these as follows:

```
import table.*;

class TableTest {
  public static void main (String [] args) {
    TryTable table = new TryTable (2);

    table.tryStore ("lcp", "Larry");
    table.tryStore ("pr", "Peter");
    table.tryStore ("acn", "Arthur");
    table.tryRetrieve ("pr");
    table.tryRetrieve ("jcw");
    table.tryStore ("jwc", "Jenni");
    table.tryStore ("lcp", "Lawrence");
    table.tryRetrieve ("jwc");
  };

};

class TryTable extends Table {
  TryTable (int size) {
    super (size);
  };

  void tryStore (String key, String value) {
    try {
      store (key, value);
      System.out.println ("Successfully stored (" +
                          key + ", " + value + ")");
    }
    catch (DuplicateException e) {
      System.out.println ("Failed: " + e.getMessage ());
    };
  };


 void tryRetrieve (String key) {
    try {
      System.out.println ("Successfully retrieved (" +
                          key + ", " + retrieve (key) + ")");
    }
    catch (MissingException e) {
      System.out.println ("Failed: " + e.getMessage ());
    };
  };

};
```

Note how the identifier *super* is used to refer to the parent class of *this*. Just as the first action of a constructor might be to call *super.constructor*, the last action of a *finalize* method might be to call *super.finalize*.

Programming environment

This section describes the basic facilities for compiling and running Java programs, using both the basic commands in the Java Development Kit (JDK) and a simple development environment working within emacs.

More elaborate facilities are provided by Sun's Java workshop on Thor. However, that program is now somewhat old and most people find it more convenient to use the JDK directly and the emacs editor for source-code development. Microsoft's Visual Studio for Java is available on the Cockroft 4 PCs running Windows NT.

Java programs come in two forms: applications and applets. Both are compiled into machine-independent byte codes in the same way. The former are run from the console through the Java interpreter and the latter are run within a World-Wide Web browser or the applet viewer.

Applications must have a principal class that includes a method with the signature:

```
public static void main (String [] args)
```

Applets must have principal class that is public, extends the class java.applet.Applet and includes a method with the signature:

```
public void paint (java.awt.Graphics g)
```

that is responsible for drawing to the screen.

Command line tools in JDK

Note that versions of the JDK differ slightly between the Solaris machine hammer.thor*, the Linux machines* belt.thor *and*

gloves.thor *and the Cockroft 4 Linux installation. They are all now JDK 1.2.x versions (aka Java 2). The behaviour of multi-threaded programs in particular is likely to vary because of the differences in the operating systems involved.*

The following programs are available in /usr/java/bin on Thor:

appletviewer *url*

Read the HTML document at the specified URL and display any applets included in it.

java *class*

Run the Java program stored as the main method of the specified class.

javac *file*

Compile the Java source code in the specified file. The –d switch can be used to redirect the resulting class file to a particular directory. If package names are used, the components of the name will be mapped into a directory structure under the named directory.

javadoc

Generate HTML documenting a Java package or source file.

jdb *class*

Invoke the Java debugger for the specified class. Useful commands include:

- stop at *class:line*
- stop in *class:method*

- run *arguments*
- locals
- up
- print *identifier*

Solaris (as run on hammer.thor) uses native threads if the environment variable THREADS_FLAG is set to native. However, this interferes with single stepping, so the environment variable should be cleared before running jdb. On that machine you can also obtain some status information about the JVM, without invoking jdb, by interrupting Java with Ctrl-\.

emacs support with LJW

A collection of emacs macros are available in /opt/gnu/share/emacs/site-lisp on *hammer.thor* to support Java programming. There are four files:

- ljw.el - the main menus for Lucian's Java Workshop.
- jde-run.el - support for executing Java programs.
- jde-db.el - support for debugging Java programs.
- anders1-java-font-lock.el - syntax highlighting.

Just put the command (load "ljw") in the .emacs file in your home directory to enable the system.

Whenever a Java source file is edited, this colours keywords using the font lock system and helps retain the format of the

code whenever tab is used. The compiler can be invoked and the program run within emacs. This uses the following keys:

- f5 - compile the source code in the current buffer.
- f6, f7 & f8 - parse the compiler's output and jump to the first, previous and next error respectively.
- f9 - run the program in the current buffer. The first time this is run, the system will prompt for details about how the source code is to be handled.

There are also menu entries under the Java heading to run the debugger, set breakpoints and run the program. These link the source code of the program and jdb so it is possible to specify breakpoints by pointing at them and see where execution has reached at any time. Two more function keys are relevant:

- f3 - step into the method at the current breakpoint.
- f4 - single step past the current breakpoint.

WWW support on Thor

It is possible to export HTML files and Java classes on Thor. Create a directory public_html in your home directory; this will then be available via the URL http://www.thor.cam.ac.uk/~crsid/ on the World-Wide Web (where crsid is your user identifer). Further sub-directories are addressed in the obvious way.

Here is a revised version of the Hello world program that can be run either as a program directly from the Java interpreter or as an applet within the applet viewer:

```
public class HelloWorld extends java.applet.Applet {

   public static void main (String [] args) {
      System.out.println ("Hello console world!");
   };

  public void paint (java.awt.Graphics g) {
    g.drawString ("Hello applet world!", 50, 25);
  };

};
```

This would need a simple HTML wrapper to be invoked. Both the compiled class file and the HTML would be placed in the public_html directory for export on WWW.

```
<html>
<head>
<title>HelloWorld test</title>
</head>
<body>
<applet code="HelloWorld" width=500 height=300>
Sorry!  Your browser does not support Java applets.
</applet>
</body>
</html>
```

The text between the <applet> and </applet> tags is only displayed if the browser does not support Java applets (or has them disabled).

Programming with objects

Here is a program that models integer and floating point numbers as objects:

```
public class NumberExample {

  public static void main (String [] args) {
    Number r = new Real (3.14159f),
           i = new Integer (42),
           s = r.add (i),
           j = i.add (new Integer (37));
    System.out.println ("r = " + r);  // r = 3.14159: Real
    System.out.println ("i = " + i);  // i = 42: Integer
```

```
    System.out.println ("s = " + s);  // s = 45.14159: Real
    System.out.println ("j = " + j);  // j = 79: Integer
  };

};

abstract class Number {
  abstract Number add (Number n);
};

class Integer extends Number {
  int i;

  Integer (int i) {
    this.i = i;
  };

  Number add (Number n) {
    return n instanceof Integer
           ? (Number) new Integer (i + ((Integer) n).i)
           : (Number) new Real (i + ((Real) n).r);
  };

  public String toString () {
    return i + ": Integer";
  };
};

class Real extends Number {
  float r;

  Real (float f) {
    r = f;
  };

  Number add (Number n) {
    return n instanceof Integer ? new Real (r + ((Integer) n).i)
                                : new Real (r + ((Real) n).r);
  };

  public String toString () {
    return r + ": Real";
  };
};
```

Interfaces

This example used an abstract class to describe the interface to objects in the class. Java also has a quite separate construction called an interface which is used to specify constraints on classes.

An interface is rather like an abstract class, but all of its methods are implicitly abstract. If it declares and fields, they must be static and final (that is, constants). Interfaces are classes and there can be an hierarchy of inheritance just as for classes. Interfaces often have names ending -able or -ible to indicate the constraint being imposed.

Any other class can implement the interface by adding **implements** *Interface1, Interface2* after any **extends** clause. It must then provide code for all the methods specified in the interface.

This is a bit like multiple inheritance. Although a class in Java can only inherit actual code of methods from a single superclass, it can satisfy the specifications in several different interfaces.

If an abstract class implements an interface then any derived class is required to implement the interface, but doesn't have to say so explicitly. If the derived class doesn't provide the methods for the interface, then it must be declared abstract so that it can not be instantiated.

Here is an example of a program that uses an interface *Sortable* to specify a constraint on objects that will make their instances comparable. A general purpose sorting routine can then be written for arrays of objects that implement the interface.

```
public class SortExample {
   public static void main (String [] args) {
    try {
      sortints ();
      sortnums ();
      sortmixed ();
    } catch (IncompatibleTypeException e) {
      System.out.println ("Incompatible types: " +
                          e.getMessage ());
    };
  };

  static void sortints () throws IncompatibleTypeException {
    Sortable [] d = {new SortableInt (3),
                          new SortableInt (5),
                          new SortableInt (2),
                          new SortableInt (4),
                          new SortableInt (3),
                          new SortableInt (5)};
    print (d);
    sort (d);
    print (d);
  };

  static void sortnums () throws IncompatibleTypeException {
    Sortable [] d = {new Integer (3), new Integer (5),
                     new Integer (2), new Real (4.0f),
                     new Real (3.0f), new Real (5.0f)};
    print (d);
    sort (d);
    print (d);
  };

  static void sortmixed ()
                   throws IncompatibleTypeException {
    Sortable [] d = {new SortableInt (3),
                          new SortableInt (5),
                          new Integer (2),
                          new Integer (4),
                          new Real (3.0f),
                          new Real (5.0f)};
    print (d);
    sort (d);
    print (d);
  };

  static void print (Object [] data) {
    System.out.print ("[");
    if (data.length > 0) System.out.print (data [0]);
    for (int i = 1; i < data.length; i++)
      System.out.print (", " + data [i]);
    System.out.println ("]");
};
```

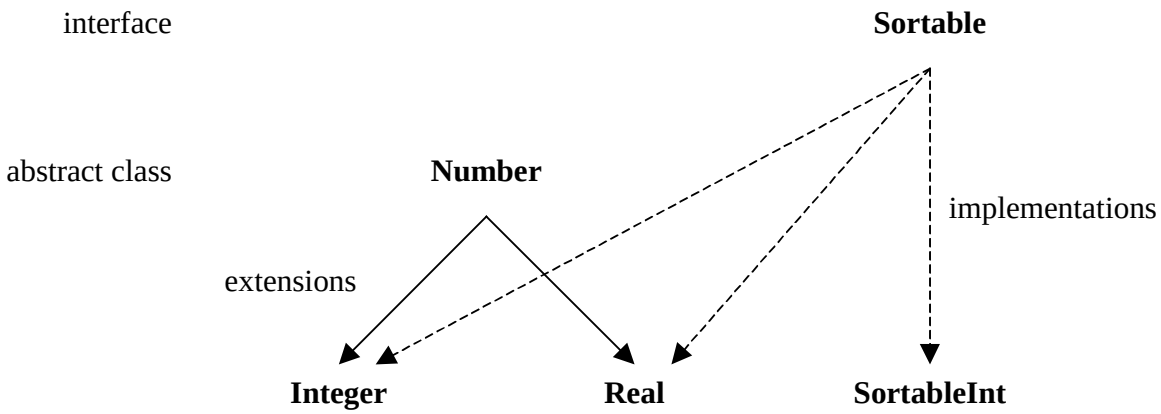
```
static void sort (Sortable [] data)
                  throws IncompatibleTypeException
  {
    for (int i = 1; i < data.length; i++) {
      for (int j = i;
           (j > 0) && (data [j] .compare (data [j-1]) < 0);
           j--) {
        // if (data [j] .compare (data [j-1]) >= 0) break;
        Sortable d = data [j];
        data [j] = data [j-1];
        data [j-1] = d;
      };
    };
  };
};

class IncompatibleTypeException extends Exception {
  IncompatibleTypeException (Object a, Object b) {
    super ("Can not compare " + a.getClass().getName() +
                  " with " + b.getClass().getName());
  };
};

interface Sortable {
  public int compare (Sortable s) throws IncompatibleTypeException;
};

class SortableInt implements Sortable {
  int i;
  SortableInt (int i) {this.i = i;};
  public String toString () {return i + "";};
  public int compare (Sortable s)
                  throws IncompatibleTypeException
  {
    if (s instanceof SortableInt)
      return i - ((SortableInt) s).i;
    else throw new IncompatibleTypeException (this, s);
  };
};

abstract class Number {
  abstract public Number add (Number n);
};
```

```
class Integer extends Number implements Sortable {
  int i;
  public Integer (int i) { this.i = i; };
  public Number add (Number n) {
    return n instanceof Integer ? (Number) new Integer (i +
                                                  ((Integer) n).i)
                                : (Number) new Real (i +
                                                  ((Real) n).r);
  };
  public String toString () {return i + ": Integer";};
  public int compare (Sortable s)
                  throws IncompatibleTypeException
  {
    if (s instanceof Integer)
      return i - ((Integer) s).i;
    else if (s instanceof Real)
      return i - ((Real) s).r > 0 ? +1 : -1;
    else throw new IncompatibleTypeException (this, s);
  };
};

class Real extends Number implements Sortable {
  float r;
  public Real (float f) {r = f;};
  public Number add (Number n) {
    return n instanceof Integer ?
                 new Real (r + ((Integer) n).i)
               : new Real (r + ((Real) n).r);
  };
  public String toString () {return r + ": Real";};
  public int compare (Sortable s) throws IncompatibleTypeException {
    if (s instanceof Integer)
      return r - ((Integer) s).i > 0 ? +1 : -1;
    else if (s instanceof Real)
      return r - ((Real) s).r > 0 ? +1 : -1;
    else throw new IncompatibleTypeException (this, s);
  };
}
```

The resulting output would be:

```
[3, 5, 2, 4, 3, 5]
[2, 3, 3, 4, 5, 5]
[3: Integer, 5: Integer, 2: Integer, 4.0: Real,
3.0: Real, 5.0: Real]
[2: Integer, 3.0: Real, 3: Integer, 4.0: Real,
5.0: Real, 5: Integer]
[3, 5, 2: Integer, 4: Integer, 3.0: Real, 5.0: Real]
Incompatible types: Can not compare Integer with SortableInt
```

This can be pictured as follows:



The abstract class Number could have been made to implement Sortable. There would still have been no code provided for the methods such a compare although any derived class would have been obliged to do so (or be declared abstract itself).

Interfaces are used throughout the standard class libraries:

- Input and output – *Cloneable, Serializable, DataInput* and *DataOutput*
- Concurrency - *Runnable*
- Graphical user interfaces - *KeyListener, MouseListener, WindowListener* and many more.

Some of these (such as *Cloneable* and *Serializable*) do not actually have any methods. Implementing the interface is just part of the class's specification.

Choosing between inheritance and interfaces

There are three common idioms for combining classes in a program.

The simplest is *composition*. A new class definition simply includes fields for objects of some other class. If these are made private, they will be hidden from users of the new class. This may be particularly appropriate if the internal representation is likely to change subsequently.

The second method is to extend an existing class to form a new one that *inherits* all the fields and methods of the old class and may add new ones. The behaviour of the old class may be modified by overriding existing methods. It is also possible to overload methods by using the same name as an existing method but a different signature; this does not mask the old method.

Finally, *interfaces* provide a mechanism for specification of particular aspects of behaviour while the implementation is deferred to particular classes.

Reflection

Reflection or *introspection* is the mechanism by which programs obtain type information about objects at run time. Every class defined in the source of a program gives rise to an instance of a Class object in the executing program and, for that matter, an associated .class file of compiled code.

Every instance of an Object (that is, every instance of every class) has a public final native method getClass() returning its Class. This is also available as the static field class of every class. Alternatively the static method Class.forName(String) yields the Class whose name is given.

Classes have many methods that can be used to investigate objects:

- String getName () was used in the code of IncompatibleTypeException in the program above to determine the names of the two classes whose values could not be compared.

- Field [] getFields () can be used to investigate data fields

- Method [] getMethods () can be used to investigate methods

- Class getSuperclass () gives information about the class hierarchy

- Class [] getInterfaces () returns an array of interfaces that the class implements

Separate compilation

Library classes that are intended for use in other programs should be made public and have any externally visible fields and methods also marked as public. Public classes have to be stored in a file that has the same name as the class (with a .java extension), so there can only be one public class in any file of code. When a Java program is running, the directories listed in the CLASSPATH environment variable will be searched for library classes.

Each class belongs to a package. The package can be specified in an optional first line of the file which simply consists of the keyword **package** followed by the package name. By convention, this is written entirely in lower case. Several classes can be grouped in a single package and their fields and methods are then visible across the package if no other visibility modifier has been specified. The **import** statement

allows identifiers to be used in unqualified form. In the absence of any explicit specification otherwise, classes belong to an anonymous package.

The exact scheme for interpreting package names depends on the particular system being used, but often the components in a package name reflect an hierarchical directory structure nested within one of the directories specified on the CLASSPATH. That is how it works on the *thor* and *Cockroft 4 Linux* installations, in which the CLASSPATH names directories separated by ':' symbols.

Concurrency

Java supports threads to lightweight concurrency within a program running in a single address space. This is vital for good interactive response and for distributed computing. It also allows programs to exploit multi-processor architectures.

The steps are as follows:

- Specify an environment and a body of code whose execution constitutes the asynchronous activity. This can be done by deriving a subclass of Thread with new fields to hold the environment and overriding the run () method to provide the code. Alternatively, any class satisfying the Runnable interface (that is, having a run () method) will suffice.

- Make an instance of Thread by calling **new** and providing appropriate arguments as desired. If a subclass of Thread with a run () method is being instantiated, no further arguments are necessary, but a ThreadGroup or a name may be specified. If the code to be executed is in another Runnable class, it should be passed as an argument to

the constructor for Thread together with either or both of the other arguments.

- Initiate activity by invoking the start () method of the new Thread. This will call the run () method to do the actual work.

- The activity will continue until either run () concludes or the Thread's stop () method is called.

- Calling a Thread's join () method blocks the calling thread until activity in the called Thread has finished.

- Activity in a Thread can be paused and restarted by calling its suspend () and resume () methods. The current thread can pause for a fixed time by calling the static Thread.sleep () method.

- Mutual exclusion can be established by placing the relevant data in the fields of a class and only accessing them via **synchronized** methods.

- A Thread calling the wait () method of an object will be suspended until some other Thread calls the object's notify () or notifyAll () method. Any mutual exclusion locks held by the Thread will be temporarily released; this does not happen when a thread is blocked for synchronization, or by sleep () or suspend (), or waiting for an input or output operation to complete.

- Each object (that is, each instance of any class) has a *monitor* which is a list of Threads that are blocked because either they are waiting for synchronized mutual exclusion or they are waiting explicitly.

- Activity in a Thread can be interrupted in two ways. Calling the Thread's interrupt () method causes a flag to be set, which can be tested by calling the Thread's isInterrupted ()

method. In any case, the InterruptedException will be thrown when the thread next calls wait, which might also be called by other methods such as sleep which also raise the exception. The Thread can catch this and proceed in whatever way is appropriate. Calling the Thread's stop () method causes the ThreadDeath error to be thrown immediately at the current point of execution. This may also be caught in order to tidy up but should then be re-thrown. Otherwise ThreadDeath is not checked.

- The use of stop (), suspend (), resume () and destroy () is now deprecated to reduce the likelihood of deadlock in threaded programs.

Summing a series in parallel

```
public class ThreadExample {
  final static int count = 5;

  public static void main (String [] args) {
    Evaluator [] workers = new Evaluator [count];
    for (int i = 0; i < count; i++) {
      workers [i] = new Evaluator (i);
      workers [i].start ();
    };
    int sum = 0;
    for (int i = 0; i < count; i++) {
      try {
                  workers [i].join ();
                  sum += workers [i].result;
      }
      catch (InterruptedException e) {
                  System.out.println ("Interrupted while
waiting for thread " + i);
      };
    };
    System.out.println ("Sum = " + sum);
  };

};

class Evaluator extends Thread {
  int argument, result;
  Evaluator (int a) {argument = a;};
  public void run () {result = argument * argument;};
};
```

Synchronised access to a buffer

```
public class BufferExample {
  public static void main (String [] args) {
    Buffer b = new Buffer ();
    Thread c = new Consumer (b);
    c.start ();
    try {
      for (int i = 1; i <= 7; i++) b.put (i);
    }
    catch (InterruptedException e) {
      System.out.println ("Producer interrupted!");
    };
    c.stop ();
  };
};

class Buffer {

  int value;
  boolean valid = false;

  synchronized void put (int i) throws
                              InterruptedException {
    while (valid) wait ();
    value = i;
    valid = true;
    notify ();
  };

  synchronized int get () throws
                             InterruptedException {
    while (! valid) wait ();
    valid = false;
    notify ();
    return value;
  };

};
```

```
class Consumer extends Thread {

  Buffer buffer;

  Consumer (Buffer b) {buffer = b;};

  public void run () {
    for (;;) {
      try {
        System.out.println ("Found " + buffer.get ());
      }
      catch (InterruptedException e) {
        System.out.println ("Interrupted while consuming!");
      };
    };
  };

};
```

The Hamming problem

```
public class HammingExample {

  static final int [] primes = {2, 3, 5};
  static final int LIMIT = 100;

  public static void main (String [] args) {
    Minimum min = new Minimum (primes.length);
    Broadcast bc = new Broadcast (primes.length);
    Thread [] hammers = new Thread [primes.length];
    for (int p = 0; p < primes.length; p++) {
      hammers [p] = new Hammer (min, bc, primes [p]);
      hammers [p] .start ();
    };
    try {
      for (;;) {
                  int next = min.get ();
                  if (next > LIMIT) break;
                  bc.put (next);
                  System.out.println (next);
      };
    }
    catch (InterruptedException e) {};
    for (int p = 0; p < primes.length; p++) {
      hammers [p] .interrupt ();
    };
  };

};
```

```
class Minimum {

  static final int INFINITY = 0x7fffffff;

  int value;
  int samples;
  int remaining = 0;

  Minimum (int s) {samples = s;};

  synchronized void put (int v)
                   throws InterruptedException {
    while (remaining == 0) wait ();
    if (v < value) value = v;
    remaining--;
    notify ();
  };

  synchronized int get ()
                  throws InterruptedException {
    value = INFINITY;
    remaining = samples;
    notifyAll ();
    while (remaining > 0) wait ();
    return value;
  };

};

class Broadcast {

  int value;
  int clients;
  int remaining = 0;

  Broadcast (int c) {clients = c;};

  synchronized int get () throws InterruptedException {
    while (remaining == 0) wait ();
    remaining--;
    notify ();
    return value;
  };
```

```
synchronized void put (int v) throws InterruptedException {
      value = v;
      remaining = clients;
      notifyAll ();
      while (remaining > 0) wait ();
    };

  };

  class Cell {
    int number;
    Cell rest = null;
    Cell (int i) {number = i;};
  };

  class Hammer extends Thread {
    Minimum min;
    Broadcast bc;
    int prime;
    Hammer (Minimum m, Broadcast b, int p) {min = m; bc = b;
  prime = p;};
    public void run () {
      Cell first = new Cell (1);
      Cell last = first;
      try {
        for (;;) {
          min.put (first.number);
                    int next = bc.get ();
          last.rest = new Cell (next * prime);
          last = last.rest;
          if (first.number == next) first = first.rest;
        };
      }
      catch (InterruptedException e) {};
    };
  };
```

Thread groups and priority

Each thread has a priority used in scheduling. Higher priority threads are usually scheduled before lower priority ones. A thread inherits its priority from the thread that created it, but can change it by calling its own setPriority () method.

The ThreadGroup class gathers together a collection of threads, which can then have a collective maximum priority imposed and can be suspended or resumed together.

Graphical interaction

The applet version of the *Hello world* program showed above simply extended the java.applet.Applet class by overriding its paint method to draw the String “Hello world!” on the applet’s underlying java.awt.Graphics object. The paint method is called whenever the applet’s window needs to be re-drawn, for example when it has been obscured and revealed.

This is just one example of the call-back system used in window systems and graphical user interfaces. The programmer provides routines which will be called by the program’s environment in response to occlusion and exposure in the window system or to input from the keyboard and mouse.

The underlying class for graphical input and output is java.awt.Component. The line of descent passes from this via Container and Panel to Applet. The paint method overridden for an Applet is actually inherited from Component. It should be able to recreate the image on screen completely, which may involve retaining some data structure in private data fields.

When the paint method is called, it is provided with a java.awt.Graphics object as an argument. This has methods to draw lines, polygons, arcs, filled areas and text.

Graphical input is handled differently in versions 1.0 and 1.1 of Java.

Java 1.0 input

In Java 1.0, input is handled via further methods in a Component (and hence in an Applet). These are overridden by the programmer and then called by the underlying run-time system in response to mouse and keyboard input.

Here is a program that allows polygons to be drawn:

```
import java.applet.*;
import java.awt.*;

public class AppletExample extends Applet {

  private Polygon poly = null;

  public boolean mouseDown (Event e, int x, int y) {
    poly = new Polygon ();
    poly.addPoint (x, y);
    repaint ();
    return true;
  };

  public boolean mouseDrag (Event e, int x, int y) {
    poly.addPoint (x, y);
    repaint ();
    return true;
  };

  public void paint (Graphics g) {
    if (poly != null) g.drawPolygon (poly);
  };
};
```

This works, although with subtly different behaviour on the appletviewer, Netscape and Internet Explorer. However, the overridden methods are now deprecated.

Java 1.1 input

Java 1.1 has a different event model in which different events are delivered to objects that implement different interfaces. The MouseListener interface handles the delivery of mouse button pushes and releases, the MouseMotionListener interface handles mouse movement and so on. The objects behind these interfaces will usually need to share state with the main Applet but this is easily arranged by extending the Applet with the required methods and implementing those interfaces. It can then act as a listener itself.

Here is a Java 1.1 version of the polygon drawing program:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class NewAppletExample extends Applet
                              implements MouseListener,
MouseMotionListener {

  private Polygon poly = null;

  public void init () {
    this.addMouseListener (this);
    this.addMouseMotionListener (this);
  };

  public void paint (Graphics g) {
    if (poly != null) g.drawPolygon (poly);
  };

// Methods for the MouseListener interface:

  public void mousePressed (MouseEvent e) {
    poly = new Polygon ();
    poly.addPoint (e.getX (), e.getY ());
    repaint ();
  };

  public void mouseReleased (MouseEvent e) {};
  public void mouseClicked (MouseEvent e) {};
  public void mouseEntered (MouseEvent e) {};
  public void mouseExited (MouseEvent e) {};

// Methods for the MouseMotionListener interface:
```

```
  public void mouseDragged (MouseEvent e) {
    poly.addPoint (e.getX (), e.getY ());
    repaint ();
  };

  public void mouseMoved (MouseEvent e) {};

};
```

This uses no deprecated features but, unfortunately only runs under the appletviewer, and not with Netscape Navigator 3.01 or Microsoft Internet Explorer 3.02 which complain about security violations.

Java 1.1 input with inner classes

It is slightly tedious to have to write all of the method signatures for the interfaces even when no bodies are being provided because the events in question are not interesting. Java 1.1 avoids this by providing ready-made classes that satisfy the listener interfaces but with null event handlers. These can be sub-classed to provide just the behaviour required.

However, these new classes would normally be disjoint from the Applet and so would not be able to share state. Java 1.1 also introduces a new feature called *inner classes* to solve this. Inner classes allow overriding methods to be provided when a class is being instantiated. The syntax is:

> **new** *ClassName* (*constructor arguments*)
> {*overriding methods*}

Here is the polygon drawing program with inner classes:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class NewAppletExample2 extends Applet {

  private Polygon poly = null;
```

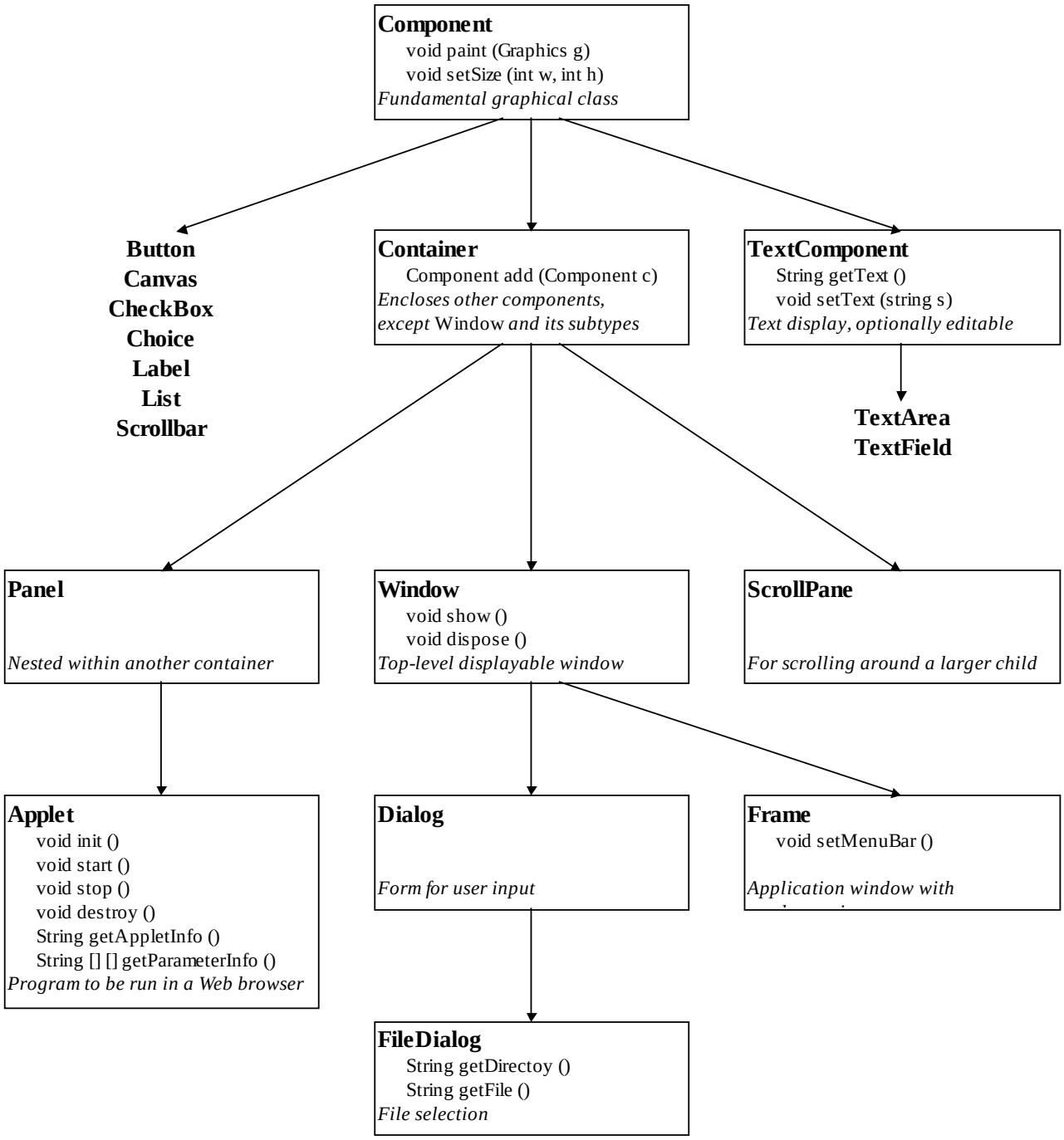
```
  public void init () {
    this.addMouseListener (new MouseAdapter () {
      public void mousePressed (MouseEvent e) {
                  poly = new Polygon ();
                  poly.addPoint (e.getX (), e.getY ());
                  repaint ();
      };
    });
    this.addMouseMotionListener (new MouseMotionAdapter () {
      public void mouseDragged (MouseEvent e) {
                  poly.addPoint (e.getX (), e.getY ());
                  repaint ();
      };
    });
  };

  public void paint (Graphics g) {
    if (poly != null) g.drawPolygon (poly);
  };

};
```

This is the preferred way of handling events. Unfortunately it fails to work not only with the older Web browsers but also with the appletviewer for PCs under Windows which can not handle inner classes. However, it does work with Sun's appletviewer for Solaris and PCs under Linux.

More elaborate interaction

There is a large hierarchy of graphical classes descended from Component. Some of the key classes and their methods are shown in the diagram below.

The full range of events presented in Java 1.1 handle actions, adjustments, components, containers, focus, items, keys, mice, text and windows. Each of these event classes has a listener interface presenting suitable call-back methods. Many of the classes also have adapters to simplify the use of inner classes.


Component
void paint (Graphics g)
void setSize (int w, int h)
Fundamental graphical class
Button
Canvas
CheckBox
Choice
Label
List
Scrollbar
Container
Component add (Component c)
Encloses other components,
except Window and its subtypes
TextComponent
String getText ()
void setText (string s)
Text display, optionally editable
TextArea
TextField
Panel
Nested within another container
Window
void show ()
void dispose ()
Top-level displayable window
ScrollPane
For scrolling around a larger child
Applet
void init ()
void start ()
void stop ()
void destroy ()
String getAppletInfo ()
String [] [] getParameterInfo ()
Program to be run in a Web browser
Dialog
Form for user input
Frame
void setMenuBar ()
Application window with
FileDialog
String getDirectoy ()
String getFile ()
File selection

The general pattern for a graphical program is as follows:

- The main class will extend Applet. It may also provide a main method so that it can be invoked either as an applet or directly from the command line. Different processing of the arguments will be needed for the two cases - applets use getParamater () and programs simple look at the elements of the String [] passed as a parameter to main.

- A complete window on the screen is modelled by a class extending Frame. This sets up the general layout of the window including a menu bar with pull-down menus and a main working area for the application. Call-backs relaying events from the menu items may well be directed to the working component.

- The main working are may well extend Component and provide call-backs for interactive input arising either directly or from menu items in the surrounding frame.

GUI components

Here is a simple program to square a number that uses a GUI:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class GUIExample extends Applet {

  public void init () {

    final TextField number = new TextField ();
    final Button square = new Button ("Square");
    final TextField result = new TextField ();

    number.setColumns (10);
```

```
    square.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent ae) {
        try {
          int n = Integer.decode (number.getText ()).
                                                  intValue ();
          result.setText (n * n + "");
        }
        catch (NumberFormatException nfe) {
          result.setText ("Format error");
        };
      };
    });
    result.setColumns (10);
    result.setEditable (false);

    this.add (number);
    this.add (square);
    this.add (result);

  }

}
```

Here is the hash table program equipped with a GUI:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class TableApplet extends Applet {

  public void init () {

    final TryTable table = new TryTable (2);

    final TextField key = new TextField ();
    final Button store = new Button ("Store");
    final Button retrieve = new Button ("Retrieve");
    final TextField value = new TextField ();
    final TextArea log = new TextArea
      ("Hash table testing\n", 10, 50,
TextArea.SCROLLBARS_VERTICAL_ONLY);

    key.setColumns (15);
    store.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent ae) {
        log.append (table.tryStore (key.getText (),
                                    value.getText ()) + "\n");
      };
    });
    retrieve.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent ae) {
```

```
        log.append (table.tryRetrieve (key.getText ()) +
                                                    "\n");
      };
    });
    value.setColumns (15);
    log.setEditable (false);

    this.add (key);
    this.add (store);
    this.add (retrieve);
    this.add (value);
    this.add (log);

  };

};

class TryTable extends Table {

  TryTable (int size) {
    super (size);
  };

  String tryStore (String key, String value) {
    try {
      store (key, value);
      return "Successfully stored (" + key + ", " +
                                          value + ")";
    }
    catch (DuplicateException e) {
      return "Failed to store with duplicate key (" +
                                         key + ")";
    };
  };

  String tryRetrieve (String key) {
    try {
      return "Successfully retrieved (" + key +
                              ", " + retrieve (key) + ")";
    }
    catch (MissingException e) {
      return "Failed to retrieve with key (" + key + ")";
    };
  };

};
```

The complete drawing program

Here is a more elaborate version of a drawing program that uses several of these additional facilities:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;


public class Drawing extends Applet {
  public void init () {
    drawing (getParameter ("title"),
                getParameter ("width"),
                getParameter ("height"));
  };

  static private String appletInfo =
    "Simple sketching program\nPeter Robinson\nOctober 1997";

  public String getAppletInfo () {
    return appletInfo;
  };

  static private String [] [] parameterInfo = {
    {"title", "String", "Title for window banner"},
    {"width", "int", "Width of drawing area"},
    {"height", "int", "Height of drawing area"}
  };

  public String [] [] getParameterInfo () {
    return parameterInfo;
  };

  public void paint (Graphics g) {
    g.drawString ("Use the other frame!", 20, 30);
  };

  public static void main (String [] args) {
    System.out.println (appletInfo);
    drawing (args.length > 0 ? args [0] :
                   "Drawing example program",
                args.length > 1 ? args [1] : null,
                args.length > 2 ? args [2] : null);
  };
```

```
 static void drawing (String title, String width, String height) {
    try {
      new DrawingFrame (title == null ? "Drawing example" : title,
           width == null ? 300 :
                   new Integer (width).intValue (),
           height == null ? 250 :
                   new Integer (height).intValue ());
    }
    catch (NumberFormatException e) {
      System.out.println (
   "Usage: java DrawingExample [title [width [height]]]");
    };
  };

};
```

```
class DrawingFrame extends Frame {

  private static String [] [] menus = {
    {"File", "New", null, "Load", "Save",
                  ll, "Print", null, "Quit"},
    {"Colour", "Black", "Blue", "Red", "Green"},
    {"Fill", "On", "Off"}
  };

  private int width, height;

  DrawingFrame (String title, int width, int height) {
    super (title);

    this.width = width;
    this.height = height;
    ScrollPane pane = new ScrollPane
          ScrollPane.SCROLLBARS_ALWAYS);
    this.add (pane, "Center");
    DrawingCanvas canvas = new DrawingCanvas (this,
                  * width, 2 * height);
    pane.add (canvas);

    MenuBar menubar = new MenuBar ();
    this.setMenuBar (menubar);

    for (int c = 0; c < menus.length; c++) {
      Menu m = new Menu (menus [c][0]);
      menubar.add (m);
      for (int r = 1; r < menus [c] .length; r++) {
           if (menus [c][r] == null) m.addSeparator ();
           else {
             MenuItem i = new MenuItem (menus [c][r]);
             m.add (i);
             i.setActionCommand (menus [c][r] .toLowerCase ());
             i.addActionListener (canvas);
           };
      };
    };
    this.pack ();
    this.show ();

  };

  public Dimension getPreferredSize () {
    return new Dimension (width, height);
  };

};
```

```
class DrawingCanvas extends Canvas
          implements ActionListener {

  private Frame frame;
  private int width, height;
  private Vector drawing = new Vector ();

  class Splodge implements Serializable {
    Color color;
    boolean filled;
    Polygon polygon;
    Splodge (Color c, boolean f, Polygon p) {
      color = c;  filled = f;  polygon = p;
    };
  };

  Color color = Color.black;
  boolean filled = false;
  Polygon polygon;

  void XORpolygon () {
    Graphics g = this.getGraphics ();
    g.setXORMode (this.getBackground ());
    g.setColor (color);
    if (filled) g.fillPolygon (polygon);
    else g.drawPolygon (polygon);
  };


class DCMouseAdapter extends MouseAdapter {
      public void mousePressed (MouseEvent e) {
           polygon = new Polygon ();
           polygon.addPoint (e.getX (), e.getY ());
           XORpolygon ();
      }
      public void mouseReleased (MouseEvent e) {
           XORpolygon ();
           polygon.addPoint (e.getX (), e.getY ());
           drawing.addElement (new Splodge (color,
                               filled, polygon));
           repaint ();
      }
  }
```

```
class DCMouseMotionAdapter extends MouseMotionAdapter {
    public void mouseDragged (MouseEvent e) {
         XORpolygon ();
         polygon.addPoint (e.getX (), e.getY ());
         XORpolygon ();
    }
}

DrawingCanvas (Frame frame, int width, int height) {
  this.frame = frame;
  this.width = width;
  this.height = height;
  this.addMouseListener (new DCMouseAdapter ());
  this.addMouseMotionListener (new DCMouseMotionAdapter ());
}

public Dimension getPreferredSize () {
  return new Dimension (width, height);
};

public void paint (Graphics g) {
  g.setPaintMode ();
  for (int d = 0; d < drawing.size (); d++) {
    Splodge s = (Splodge) drawing.elementAt (d);
    g.setColor (s.color);
    if (s.filled) g.fillPolygon (s.polygon);
    else g.drawPolygon (s.polygon);
  };
};

public void actionPerformed (ActionEvent e) {
  String s = e.getActionCommand ();
  if (s.equals ("new")) {
    drawing.removeAllElements ();
    this.repaint ();
  }
  else if (s.equals ("load")) load ();
  else if (s.equals ("save")) save ();
  else if (s.equals ("print")) print ();
  else if (s.equals ("quit")) System.exit (0);
  else if (s.equals ("black")) color = Color.black;
  else if (s.equals ("blue")) color = Color.blue;
  else if (s.equals ("red")) color = Color.red;
  else if (s.equals ("green")) color = Color.green;
  else if (s.equals ("on")) filled = true;
  else if (s.equals ("off")) filled = false;
};
```

```
private void load () {
  FileDialog fd = new FileDialog (frame,
                oad drawing", FileDialog.LOAD);
  fd.show ();
  String f = fd.getFile ();
  if (f != null) {
    try {
         FileInputStream s = new FileInputStream (f);
         ObjectInputStream o =
                           w ObjectInputStream (s);
         Vector v = (Vector) o.readObject ();
         o.close ();
         drawing = v;
         this.repaint ();
    }
    catch (Exception e) {System.out.println (e);};
  };
};

private void save () {
  FileDialog fd = new FileDialog (frame,
                           "Save drawing",
                           FileDialog.SAVE);
  fd.show ();
  String f = fd.getFile ();
  if (f != null) {
    try {
         FileOutputStream s = new FileOutputStream (f);
         ObjectOutputStream o = new ObjectOutputStream (s);
         o.writeObject (drawing);
         o.flush ();
         o.close ();
    }
    catch (Exception e) {System.out.println (e);};
  };
};

private static Properties preferences = new Properties ();

private void print () {
  Toolkit toolkit = this.getToolkit ();
  PrintJob printjob = toolkit.getPrintJob (frame,
                           "Print drawing",
                           preferences);
  if (printjob == null) return;
  Graphics sheet = printjob.getGraphics ();
  Dimension canvassize = this.getSize ();
  Dimension pagesize = printjob.getPageDimension ();
  sheet.translate ((pagesize.width - canvassize.width) / 2,
                (pagesize.height - canvassize.height) / 2);
  sheet.drawRect (-1, -1,
                canvassize.width + 1,
                canvassize.height + 1);
```

```
    sheet.setClip (0, 0, canvassize.width, canvassize.height);
    this.print (sheet);
    sheet.dispose ();
    printjob.end ();
  };

}
```

The parameters are supplied to the applet via <param> tags within the <applet> block of HTML:

```
<applet code="DrawingExample" width=500 height=300>
  <param name=title value=”Drawing example applet”>
  <param name=width value=250>
  <param name=height value=150>
Sorry!  Your browser does not support Java applets.
</applet>
```

Java Beans

Beans are library classes whose methods conform to a special naming convention. :

- The classes are general purpose and may be tailored to a specific use by giving particular values to properties. The convention dictates that a property called xyz will be set by a setXyz () method and the value will then be available through a getXyz () method. (Boolean properties can also be checked by an isXyz () method.)

- The classes may well be used in GUIs, in which case they use the Java 1.1 input model with listeners. A listener for the AlphaEvent would be installed with a call to the addAlpha|EventListener () method and cancelled by the removeAlphaEventListener () method.

- Other methods specific to the class have whatever names they like, but should be public.

GUI design systems can then use reflection to determine the specification of beans and control their composition on screen.

Java Studio (available on *hammer.thor* as /opt/java/java-tools/bin/js) is one such system, allowing GUIs to be designed interactively.

Swing

The Swing library provides a higher level mechanism for building GUIs. Swing is built on top of AWT and its components are beans. Indeed, most of the components provided by AWT are available as Swing components simply by prefixing their names with J. There is a whole new class hierarchy descended from JComponent, which is itself derived from Container. However, there is an important difference: any JComponent can be nested within another JComponent. So, for example, scrolling can be added to a component simply by wrapping it in a JScrollPane.

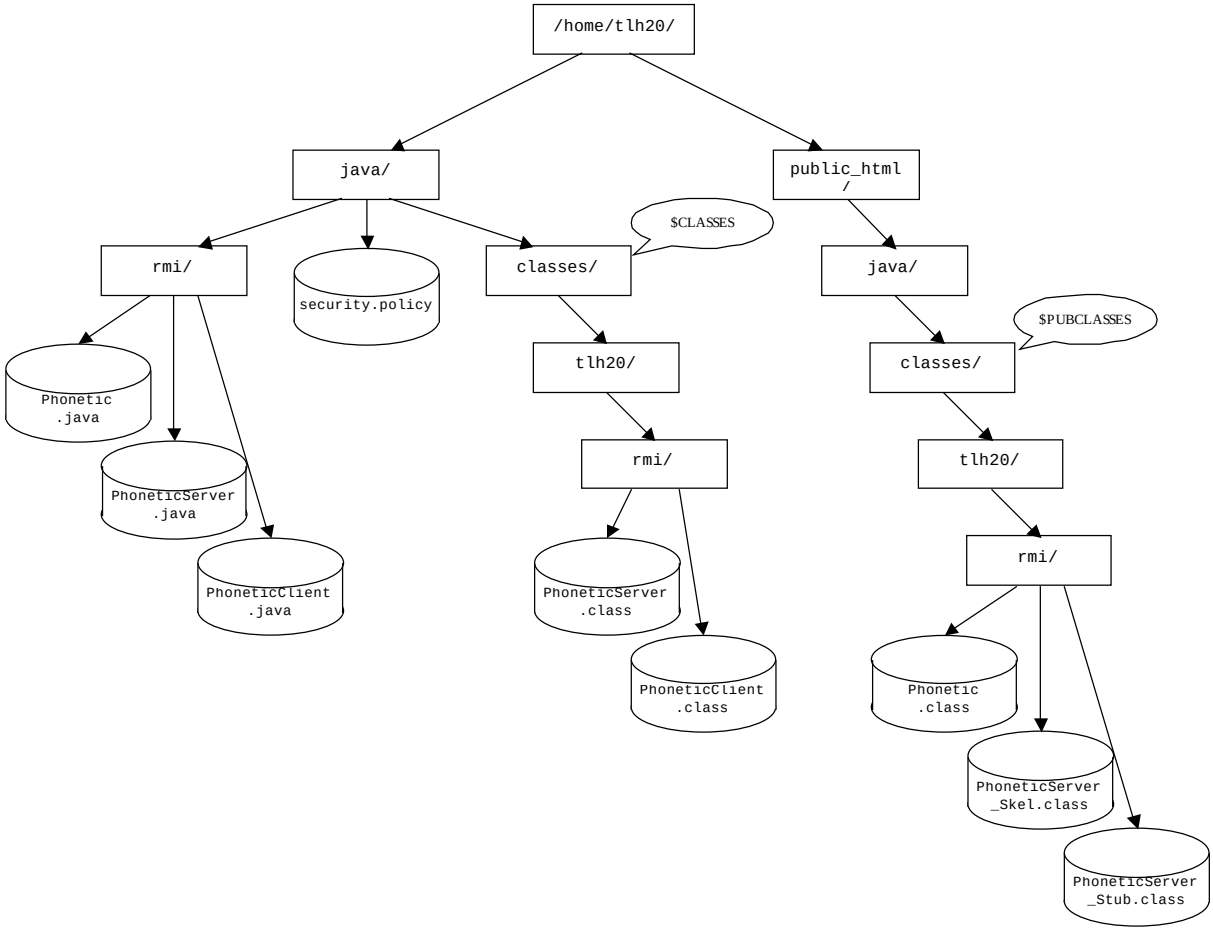
The other important feature of the Swing library is that its components separate the structure of an application's window (its *model*) from its appearance on the screen (its *view*). The actual interpretation of a model as a view is controlled by a look-and-feel manager and, by default, conforms to that of the window system being used. A single program will have a different look-and-feel depending on where it is run. Alternatively, a completely new idiom can be enforced.

Graphical input is also managed in a standard way so pop-up tool-tips for buttons and keyboard alternatives for mouse actions are provided automatically.

Distributed computing

Java Remote Method Invocation (RMI) allows type-safe communication between Java programs running on different machines.

It is convenient to package names and a particular directory structure for such programs. The diagram below shows how this might be achieved on Thor.



On *hammer.thor* the steps are as follows:

- Set up environment variables for the Java compiler and interpreter to locate the compiled class files both in the local filing system and on the Web.

```
$ export CLASSES=/home/tlh20/java/classes/
$ export PUBCLASSES=/home/tlh20/public_html/java/classes/
$ export CLASSPATH=$CLASSES:$PUBCLASSES
$ export CODEBASE=http://hammer.thor.cam.ac.uk/~tlh20/java/classes/
```

- Write a public interface extending java.rmi.Remote that specifies the signatures of the remote methods [say, Phonetic.java]. Each method should declare

java.rmi.RemoteException in its throws clause. Any arguments and results must implement Serializable. This is also a good place to specify a unique name for the service and the machine through which it will be made available.

```
package tlh20.rmi;

import java.rmi.*;

public interface Phonetic extends Remote {

  public final static String URL =
      "rmi://hammer.thor.cam.ac.uk/~tlh20/java/classes/";
  public final static String NAME = "tlh20-Phonetic-1.1";

  public String [] spell (String s) throws RemoteException;
  public String [] spell (String s, boolean b)
                                        throws RemoteException;

}
```

- Compile it [giving, say, Phonetic.class] and put the compiled class file in a package sub-directory of the public directory to make available through the WWW server.

```
$ javac –d $PUBCLASSES Phonetic.java
```

- Write a server program extending the java.rmi.UnicastRemoteObject class and implementing the interface [say, PhoneticServer.java]. The constructor will have to be written explicitly to accommodate the possibility of RemoteException being thrown. The main method must install a security manager.and then export the service via the RMI registry.

```
package tlh20.rmi;

import java.net.*;
import java.rmi.*;
import java.rmi.server.*;

public class PhoneticServer
                  extends UnicastRemoteObject
                  implements Phonetic {

  public static void main (String [] args) {
    try {
      System.setSecurityManager (new RMISecurityManager ());
      PhoneticServer s = new PhoneticServer ();
      Naming.rebind (Phonetic.URL + Phonetic.NAME, s);
      System.out.println (Phonetic.NAME + " server running");
    }
    catch (Exception e) {
      System.out.println ("Exception: " + e);
    };
  }

  public PhoneticServer () throws RemoteException {
  //  super ();
  }

  public String [] spell (String s)
                  throws RemoteException
  {
    return spell (s, false);
  }

  private final static String [] [] WORDS = {
    {"alfa", "aesthetic"},
    {"bravo", "bdellometer"},
    {"charlie", "ctenoid"},
    {"delta", "djibbah"},
    {"echo", "ewe"},
    {"foxtrot", "fnese"},
    {"golf", "gnome"},
    {"hotel", "heir"},
    {"India", "iero"},
    {"Juliet", "Jugoslavia"},
    {"kilo", "know"},
    {"Lima", "pounds, shillings and pence"},
    {"Mike", "mnemonic"},
    {"November", "Nzima"},
    {"Oscar", "oesophagus"},
    {"papa", "pneumonia"},
    {"Quebec", "Qatar"},
    {"Romeo", "’rithmetic"},
    {"sierra", "sdeath"},
    {"tango", "tsar"},
```

```
    {"uniform", "Uigur"},
    {"victor", "veldt"},
    {"whiskey", "write"},
    {"x-ray", "xylophone"},
    {"yankee", "yggdrasil"},
    {"zulu", "zucchetto"}
  };

  public String [] spell (String s, boolean silly)
                              throws RemoteException {
    System.out.println ("Spelling '" + s +"'");
    String source = s.toUpperCase ();
    int length = s.length ();
    String [] reply = new String [length];
    for (int i = 0; i < length; i++) {
      try {
          int w = (int) source.charAt (i) - (int) 'A';
          reply [i] = source.substring (i, i+1)
                 + " as in " + WORDS [w] [silly ? 1 : 0];
      }
      catch (ArrayIndexOutOfBoundsException e)
      {reply [i] = "?";}
      catch (StringIndexOutOfBoundsException e)
      {reply [i] = "?";};
    };
    return reply;
  }
}
```

- Compile it [giving, say, PhoneticServer.class].

```
$ javac –d $CLASSES PhoneticServer.java
```

- Run the RMI compiler rmic on the server class [producing, in this case, PhoneticServer_Skel.class and PhoneticServer_Stub.class]:

```
$ rmic –d $PUBCLASSES tlh20.rmi.PhoneticServer
```

Be careful that there are not any old versions of the class accessible on the CLASSPATH.

- Create a file specifying the security policy [called, say, security.policy] in the current directory.

```
grant {
  permission java.net.SocketPermission
                      "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission
                      "*:80", "connect";
  permission java.util.PropertyPermission
                      "java.rmi.server.codebase", "read";
  permission java.util.PropertyPermission
                      "user.name", "read,write";
};
```

- Run the server with the codebase defined to be the URL of the Java class sub-directory of your home URL and the security policy the file just created.

```
$ java -Djava.rmi.server.codebase=$CODEBASE \
       –Djava.security.policy=security.policy \
       tlh20.rmi.PhoneticServer
```

The service is advertised through the RMI Registry (essentially a name look-up service) running on the same machine as the server.

- Write a client program [say, PhoneticClient.java].

```
package tlh20.rmi;

import java.rmi.*;

public class PhoneticClient {

  public static void main (String [] args) {
    if (args.length > 0) {
      try {
          System.setSecurityManager (new RMISecurityManager ());
          Phonetic p = (Phonetic)
              Naming.lookup (Phonetic.URL + Phonetic.NAME);
          String [] results = args.length > 1
                   ? p.spell (args [0], true)
                  : p.spell (args [0]);
          for (int r = 0; r < results.length; r++)
            System.out.println (results [r]);
      }
      catch (Exception e) {
          System.out.println ("Exception: " + e);
      };
    } else System.out.println ("Usage: java " +
                  "PhoneticClient word [silly]");
  };

};
```

- Compile it [giving, say, PhoneticClient.class].

```
$ javac –d $CLASSES PhoneticClient.java
```

- Run the client program with some security policy (quite possibly the same).

```
$ java –Djava.security.policy=security.policy \
       tlh20.rmi.PhoneticClient "Phonetics is phun"
```

If the convention suggested above of putting the service name and base URL in the interface specifying the service, it will be necessary to have the associated class file available when the client (or server) is run.

Remote Method Invocation on Thor

Java RMI must be used carefully if it is to avoid adversely affecting other users in a shared environment. Care must also be taken to avoid opening any security loopholes through its use. The following guidelines circumscribe the use of network objects on the Computing Service's Unix teaching system, Thor.

- The RMI registry is run automatically as part of the system on hammer.thor.cam.ac.uk, but not on belt or gloves. It should not be run by individual users. Operations staff should be consulted if it is not available.

- All classes exported through RMI have to be on a path descending from a known URL. Users should make sure that any classes that they with to export are in a subdirectory, say java/classes, of the public_html directory in their home directories. The URL must be passed to the Java interpreter by defining the symbol java.rmi.server.codebase.

- The name-space in the RMI Registry is flat, so users should make sure that any services exported through it are prefixed by their CRSIDs. It also makes sense to append a version number to the name. The entire name can be specified conveniently in the interface for the remote service.

- All programs using RMI, both servers and clients, should only run under the control of a logged-in user while reporting their operation to standard output or to a graphical user interface. It is strongly advisable that they should also copy this logging output to a file for subsequent analysis in the event of a failure.

- The user running such a program is, of course, responsible for its actions and any resources that it consumes. It is therefore prudent to limit the external operations of the program severely.

Remote Method Invocation on Cockroft 4 Linux

If you use RMI on the Cockroft 4 Linux machines then you need to start the RMI registry yourself. This is done by executing the program *rmiregistry* from a shell. If you wish to use the example programs on those machines then you need to convert the codebase and RMI URLs to refer to the machine in question. Typically the machine's address is of the form 'pc???.cl.pwf.cam.ac.uk'. If you wish to use your own Linux machine then you will need to make similar changes to the examples and configure a web server to distribute the code.

In either case, you should take the same care when exporting services through RMI as you would on *hammer.thor*.

Class libraries

The standard Java development kit includes extensive class libraries. The Application Programming Interfaces are documented in two volumes in Sun's Java Series describing the core packages and the Abstract Window Toolkit (AWT) respectively. Further details and examples are given in the class libraries volume in the same series. Most of this material is available on-line at http://www-uxsup.csx.cam.ac.uk/java/jdk-1.2.2/docs/api/packages.html. There is a lot of it; read and enjoy.

Text input

Reading and parsing textual input is not particularly obvious. For general conversion from a String to a value, use the static methods decode in the classes Byte, Short, Integer, Long, Boolean, Float, Double and so on to manufacture an instance of the class and then call its intValue or realValue method. So:

```
int i = Integer.decode (“42”) .intValue ()
```

assigns the value 42 to i.

For more complicated examples, the StreamTokenizer class can be used to pick words and numbers off the standard input:

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

public class IOExample {

  public static void main (String [] args) {
    StreamTokenizer st = new StreamTokenizer
          (new InputStreamReader (System.in));
    try {
      for (;;) {
           int t = st.nextToken ();
           try {
             switch (t) {
             case StreamTokenizer.TT_EOF:
               throw new EOFException ();
             case StreamTokenizer.TT_EOL: break;
             case StreamTokenizer.TT_NUMBER:
               System.out.println ("Number = " +
                                   (int) st.nval);
               break;
             case StreamTokenizer.TT_WORD:
               System.out.println ("Word = " +
                                   st.sval);
               break;
             default:
               System.out.println ("Character = '" +
                                   (char) t + "'");
               break;
             };
           }
           catch (NumberFormatException e) {
             System.out.println ("Format error: " +
```

```
                                        e.getMessage ());
            };
        };
      } catch (EOFException e) {
        System.out.println ("End of file");
      }
      catch (IOException e) {
        System.out.println ("IO exception: " + e.getMessage ());
      };
    };
}
```

Foundation classes

A number of competing libraries known as *foundation classes* are being released:

- The Abstract Window Tookit, AWT, is the basic, low-level kit.
- Microsoft has developed the Abstract Foundation Classes, AFC, which offer special support for things like DirectX.
- Netscape's Internet Foundation Classes, IFC, encapsulates AWT. It is available, it works and it is being phased out.
- Sun and Netscape are collaborating on the Java Foundation Classes, JFC, which will work with AWT. The Swing libraries for GUIs are part of this.

None are particularly stable yet.

Development environments

There appear to be at least 20 development environments available for Java. Some of the more significant ones are:

- Borland JavaBuilder.

- IBM's ADK.
- Microsoft's Visual Studio for Java.
- Sun's Java Workshop - which is available for Sun/Solaris, PC/Windows and (allegedly, soon, …) PC/Linux. It is available as /opt/java/java-tools/bin/jws on Thor.
- Symantec Café.

None are particularly stable yet.

Exercises

This is a practical course and the only way to understand its material is to write lots of programs. Here are some ideas to try.

Hello world

Type in the Hello world program, and compile and run it.

Once the program has worked, try damaging it by making small changes -- putting a reserved word in lower case, misspelling a name, omitting an import and so on. Compile the program and understand the error messages. (It is much easier to understand the diagnostic messages when you know what the errors are. This practice should help when you make unintentional errors in the future!)

Word counting

Repeat this procedure with the word counting program and practice using the emacs environment and the symbolic debugger.

Now modify the program to accumulate further statistics such as the number of lines, sentences and paragraphs. You may need to think a little about how these are defined.

Summing a series

Write a program to sum the series $\sum_{n=0}^{\infty} 1/(1000n + \pi)$ and print the result.

This should work by computing successive terms from $n = 0$ upwards and accumulating their sum until the action of adding in the term makes no difference to the total. This will probably involve a **while** loop of some sort. Print out the number of terms and the partial sum.

Then recompute the answer by accumulating the sum of the same terms running back down from the limit just discovered to zero; this will probably involve a **for** loop. Are the answers the same? Why? What answer would a mathematician have given for the sum?

Pascal's triangle

Write a program to print out Pascal's triangle, that is, a table of binomial coefficients: $\binom{n}{r} = \frac{n!}{r!(n-r)!}$

Caesar cipher

Given an integer *n* as a key, Caesar's cipher encodes each letter in the alphabet as the letter *n* after it, where all the letters are considered as cyclic so A follows Z.

Write a program to encode text using this algorithm.

Code breaking

Write a program to help an interceptor decode a message that has encoded using the simple cipher program just written. This should accumulate statistics on the frequency of occurrences of various letters (ignoring their case). Gather data by running it on various pieces of plain text and then run it on an encoded message and try to deduce the key that must have been used from the shift in the distribution.

You could even automate this inference by finding the key that gives the least mean square difference between the two frequency distributions. There is a dictionary of some 25 000 English words in /usr/dict/words on Thor which can be used for calibration.

8 queens problem

This involves placing 8 queens on a chess board in such a way that no one of them can take any of the others. That is, there may only be at most one queen in any row, column or diagonal line of squares.

The main data structure should be an array to store the board recording the presence or absence of a queen in each square:

```
boolean [] [] board = new boolean [8] [8]
```

Initially all the values would be set **false** to show the complete absence of queens.

Clearly each row must contain precisely one queen. It makes sense to structure the program around a class containing the board and a method that takes a row number as an argument and tries in turn to place a queen in each column in that row. If it manages to do this without conflicting with any of the queens in earlier rows (either vertically or diagonally), it calls itself recursively to place a queen in the next row. When it finds itself called past the last row, it knows that a solution has been found and can print it out.

Big numbers

Design a data structure based on a linked list for storing arbitrarily large natural numbers; each cell in the list should contain one decimal digit. Encapsulate this in a class with a constructor to create such a number from a positive integer and methods to add two such numbers and to convert one to text.

Primes

Calculate how many primes there are less than a million. A useful technique is the sieve of Eratosthenes: Write down a list of all the numbers between 2 and a million. The first number is prime, so strike out all of its multiples in the list. The next number now left in the list must also be prime, so strike out all of its multiples and so on. Finally you are left with a list containing only prime numbers. The arithmetic is simple; the real problem is devising a data structure and algorithm that balance the time and space requirements.

Sorting

Rewrite the sorting program used to illustrate interfaces with a different algorithm. QuickSort or Shell's sort are likely to be promising.

Implement a new Sortable type to hold Strings and combine these parts to make a program that sorts the lines of text in a file into alphabetic order.

Spelling checker

Write a program to help with checking spelling in text files. This should separate out individual words in the source text file and sort them by incrementally storing them in a binary tree. A separate routine can then walk over this tree, meeting all the words in the original file in alphabetical order, and compare them with words in a dictionary stored as a separate text file, drawing the user's attention to any discrepancies.

Reflection

Write a method that takes an arbitrary Object as an argument and uses reflection to produce an approximation to the its source as Java code.

Caching functions

Write an abstract class to model functions. This should have an apply method that takes an int argument and returns an int result. Derive a subclass that implements a particular function.

Derive another subclass that caches values calculated by functions. This should have a constructor that takes an existing function as its argument and caches its values by saving pairs of arguments and results in a table.

Test it by calculating Fibonacci numbers naïvely and with caching.

Lazy lists

A list can be considered as an object with two methods: *head*, which yields the first element of the list, and *tail*, which yields a new list consisting of all the elements except the first one. In the normal scheme of things, both of these methods would have to be able to raise an exception if they were invoked on an empty list. This requirement goes away if we restrict our attention to *streams*, that is, lists with infinitely many elements. However, there may be some difficulty representing the contents of an infinite list.

Lazy lists handle this by representing a list as a pair consisting of the first value and a function that returns a new list for the tail (which will itself be a pair...). Devise a representation for lazy lists as objects in Java and write a program to generate a stream of primes as a lazy list.

Parallel primes

Another technique for generating primes is the sieve of Eratosthenes mentioned above. Implement a parallel version of this algorithm, passing the natural numbers along a chain of threads, each of which filters out multiples of a particular prime. Any number reaching the end of the chain must be a new prime which can be printed. A new thread must then be added to the chain to filter out multiples of that prime as well. (Observant readers will have noticed that this is more-or-less what was happening in the lazy list approach.)

Four-function calculator

Implement a four-function calculator as a graphical applet.

Dining philosophers

Write a simulator for the dining philosophers problem:

> Five philosophers spend their time *thinking* and *eating*. They share a common, circular table, surrounded by five chairs, each belonging to one philosopher. In the centre of the table is a bowl of spaghetti and the table is laid with five forks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two forks that are closest to her (between her and her left and right neighbours). A philosopher may only pick up one fork at a time. When a hungry philosopher has both her forks at the same time, she eats without releasing her forks. When she has finished eating, she puts down both of her forks and starts thinking again.

Each philosopher should be represented by a separate thread and each fork by a separate instance of a synchronized class. Your applet should display the results of the simulation in a graphical form, where there should be a control to switch the philosophers between a naïve algorithm resulting in deadlock and a more intelligent one.