Applications II

Agenda

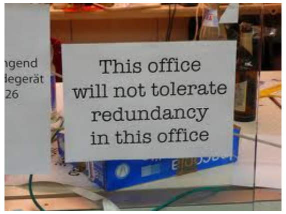
Utilities

- File-compression (Huffman’s algorithm)
- Cross-referencing

Simulation

- Discrete event simulation
- Carwash simulation
- Call bank simulation

File compression



Compression reduces the size of a file

- to save **space** when *storing* the file
- save **time** when *transmitting* it

Many files have low information content. Compression reduces **redundancy** (unnecessary information).

Compression is used for

text:	some letters are more frequent than others
graphics:	large, uniformly colored areas
sound:	repeating patters

Redundancy in text

removal of vowels

Yxx cxn xndxrstxnd whxt x xm wrxtxng xvxn xf x rxplxcx xll thx vxwxls wxth xn 'x' (t gts lttl hrdr f y dn't kn whr th vwls r).

Run-length encoding

Compression by counting repetitions.

Compression of **text**:

The string

```
    AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD
```

may be encoded as

```
    4A3BAA5B8CDABCB3A4B3CD
```

Using an escape character ('\\'):

```
    \4A\3BAA\5B\8CDABCB\3A\4B\3CD
```

Run-length encoding is normally not very efficient for text files.

Run-length encoding

Compression of (black and white raster) **graphics**:

000000000000011111111111111000000000	13 14 9
000000000001111111111111111110000000	11 18 7
000000001111111111111111111111110000	8 24 4
000000011111111111111111111111111000	7 26 3
000001111111111111111111111111111110	5 30 1
000011111110000000000000000001111111	4 7 18 7
000011111000000000000000000000011111	4 5 22 5
000011100000000000000000000000000111	4 3 26 3
000011100000000000000000000000000111	4 3 26 3
000011100000000000000000000000000111	4 3 26 3
000011100000000000000000000000000111	4 3 26 3
000001111000000000000000000000001110	5 4 23 3 1
000000011100000000000000000000111000	7 3 20 3 3
011111111111111111111111111111111111	1 35
011111111111111111111111111111111111	1 35
011111111111111111111111111111111111	1 35
011111111111111111111111111111111111	1 35
011111111111111111111111111111111111	1 35
011000000000000000000000000000000011	1 2 31 2

Saving:

(19*36 - 63*6) bits = 306 bits

corresponding to 45%

Fixed-length encoding

The string

`ABRACADABRA` (11 characters)

occupies

11 * 8 bits = 88 bits	in byte code	
11 * 5 bits = 55 bits	in 5-bit code	
11 * 3 bits = 33 bits	in 3-bit code	(only 5 different letters)

`D` occurs only once, whereas `A` occurs 5 times.

We can use short codes for letters that occur frequently.

Variable-length encoding

If `A` = `0`, `B` = `1`, `R` = `01`, `C` = `10`, and `D` = `11`, then

`ABRACADABRA`

may be encoded as

`0 1 01 0 10  0 11 0 1 01 0` (only 15 bits)

However, this code can only be decoded (decompressed) if we use delimiters (for instance, spaces)

The cause of the problem is that some codes are *prefix* (start) of others. For instance, the code for `A` is a prefix of the code for `R`.

Prefix codes

A code is called a **prefix code** if there is no valid code word that is a prefix of any other valid code word.

A prefix code for the letters `A`, `B`, `C`, `D`, and `R`:

`A` = `11`, `B` = `00`, `C` = `010`, `D` = `10`, `R` = `011`.

The string `ABRACADABRA` is encoded as

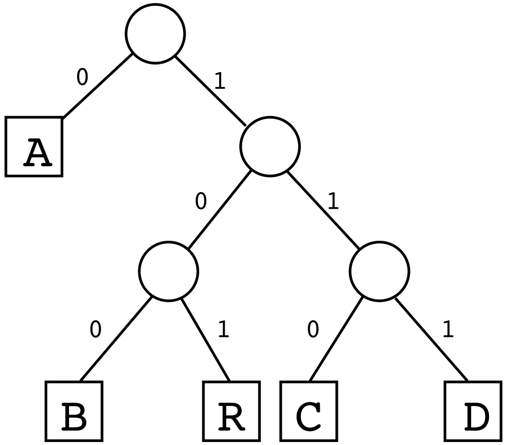
`1100011110101110110001111` (25 bits)

The string can be decoded unambiguously.

However, this prefix code is *not optimal*.
An optimal prefix code can be determined by **Huffman's algorithm**.

Binary tries

The code is represented by a tree, a so-called **trie** (pronounced *try*).



The characters are stored in the leaves.
A left branch corresponds to 0.
A right branch corresponds to 1.

Code: `A` = `0`, `B` = `100`, `C` = `110`, `D` = `111`, `R` = `101`.

The string `ABRACADABRA` is encoded as
`01001010110011101001010` (23 bits)

Huffman's algorithm

(D. A. Huffman, 1952)

Count frequency of occurrence for the characters in the string.
(or use a pre-defined frequency table).

Character	Frequency
`A`	5
`B`	2
`C`	1
`D`	1
`R`	2

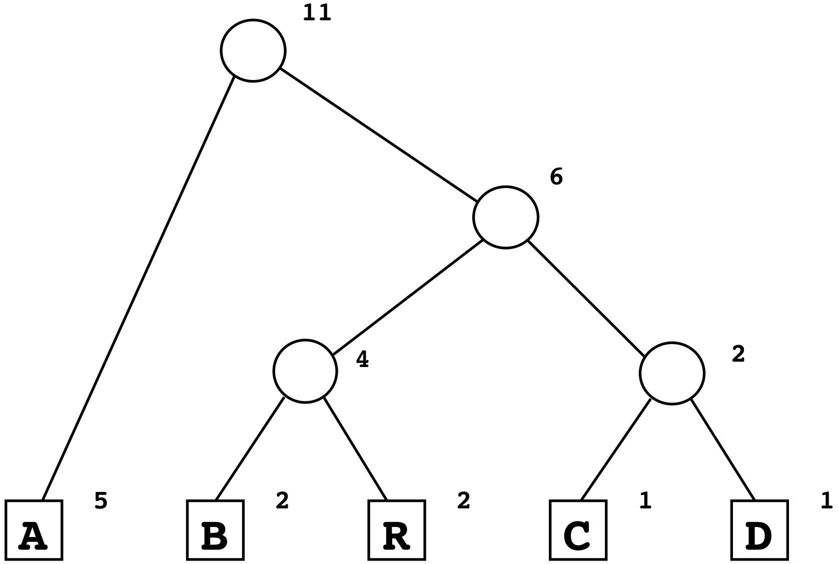
Build a trie by successively combining the two smallest frequencies.

Huffman's algorithm (1952)

David Huffman

Start with a single node tree for each character.
As long as there is more than one tree in the forest:
combine the two "cheapest" trees into one tree
by adding a new node as root.

Greedy algorithm



The tree is optimal (i.e., it minimizes $\sum \text{depth}_i * \text{frequency}_i$) – but it need not be unique.

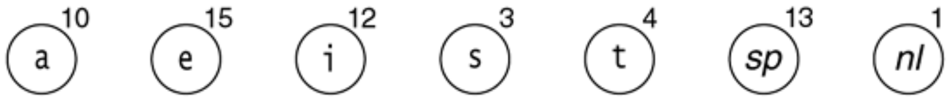


figure 12.6

Initial stage of Huffman's algorithm

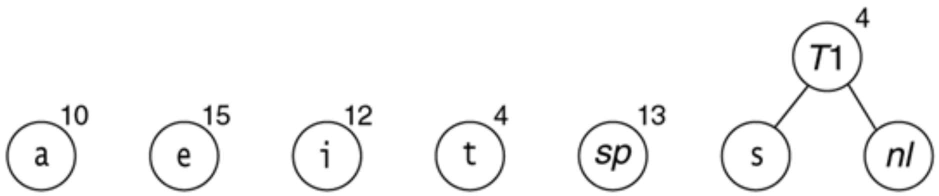


figure 12.7

Huffman's algorithm after the first merge

figure 12.8

Huffman's algorithm after the second merge

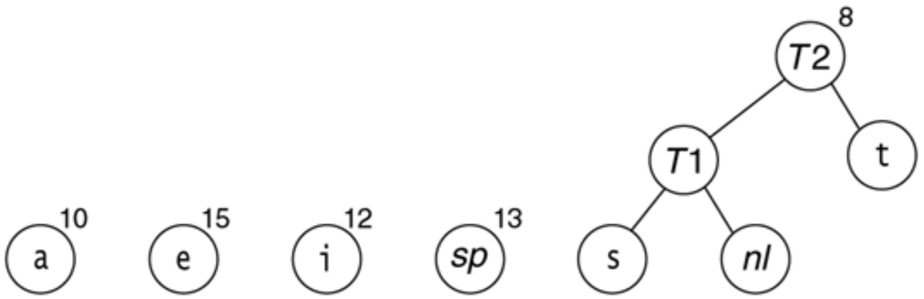


figure 12.9

Huffman's algorithm after the third merge

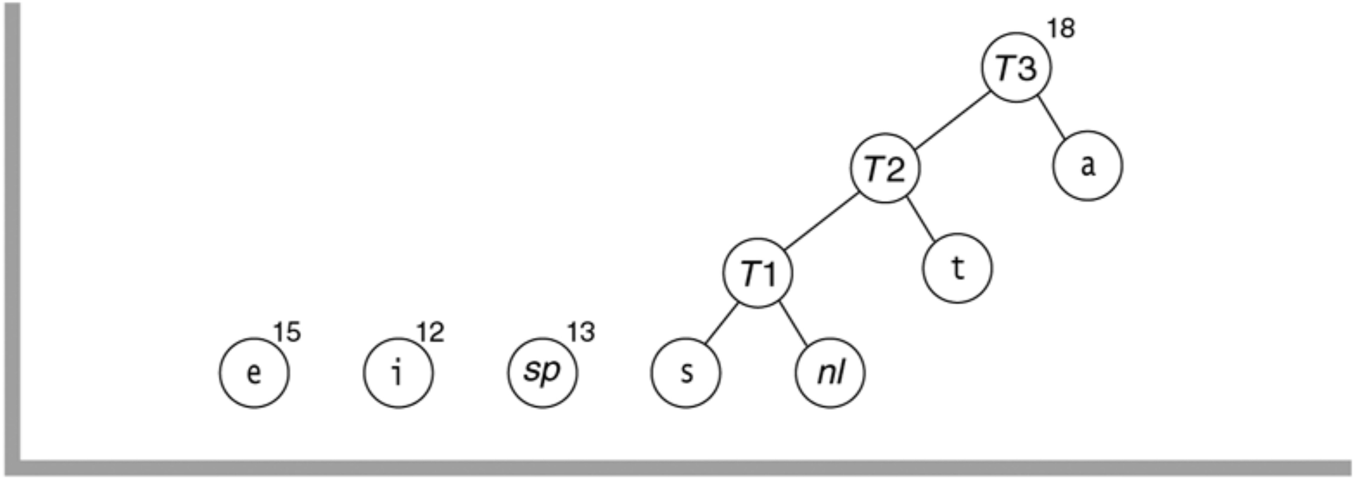


figure 12.10

Huffman's algorithm after the fourth merge

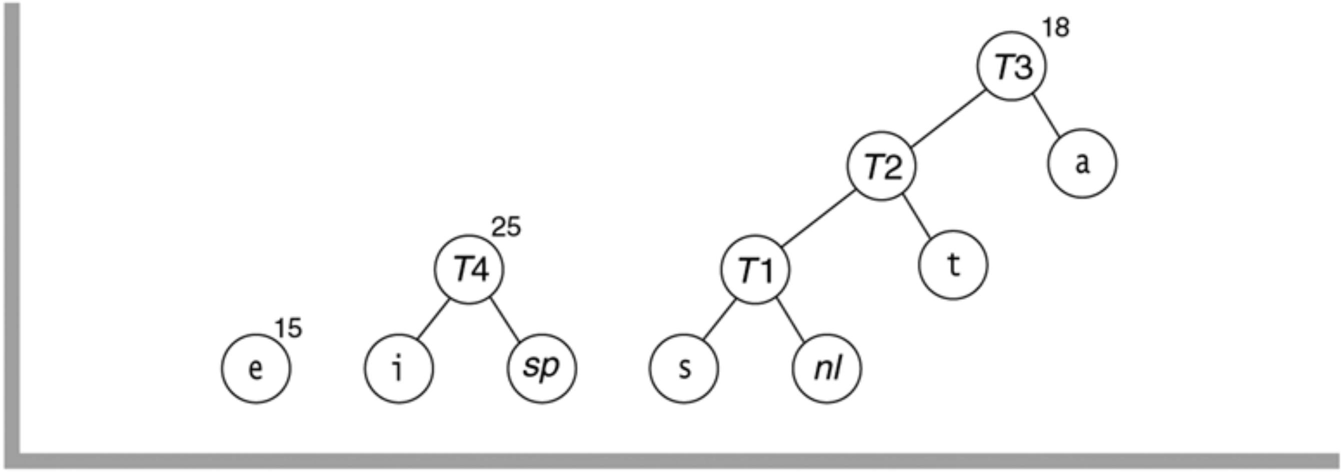


figure 12.11

Huffman's algorithm after the fifth merge

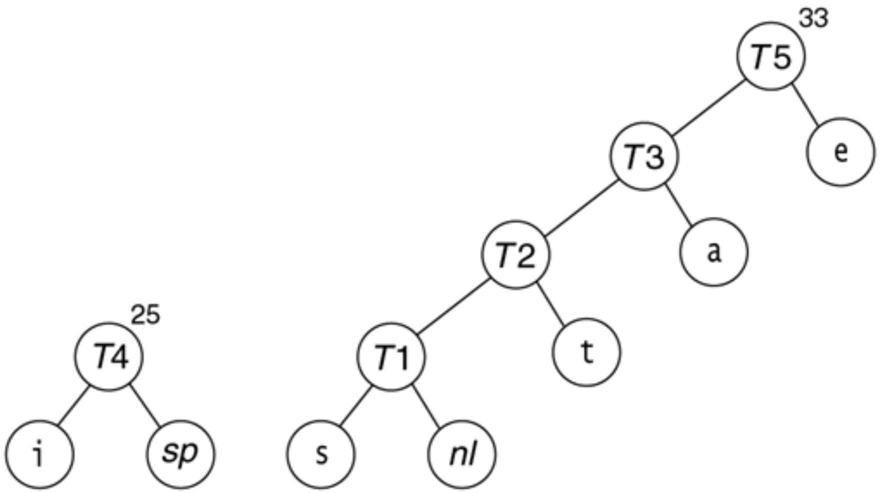
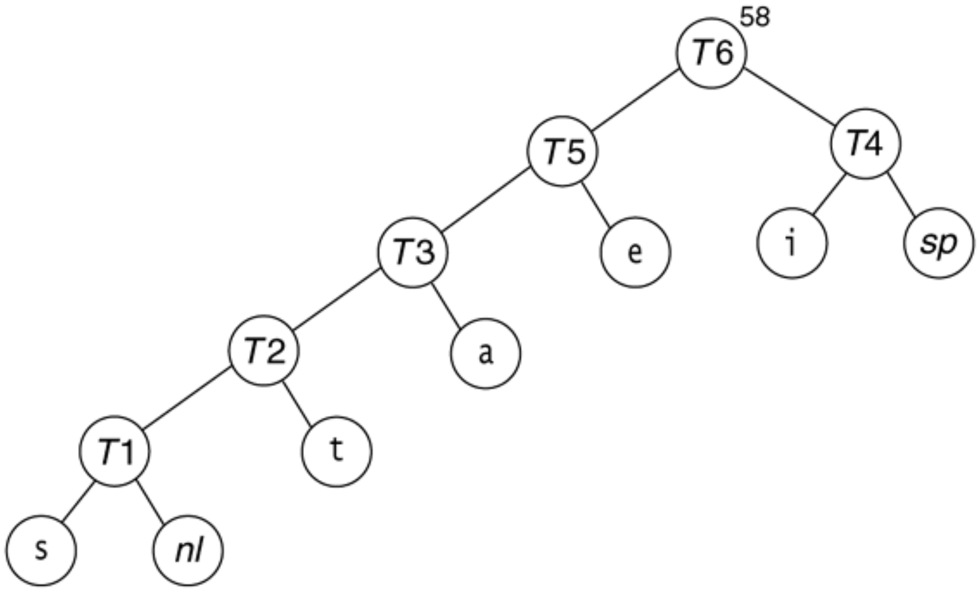


figure 12.12

Huffman's algorithm after the final merge



Implementation of Huffman’s algorithm

Representation of the tree:

```
class HuffmanTree {
    HuffmanTree(Node root) {
        this.root = root;
    }

    Node root;
}
```

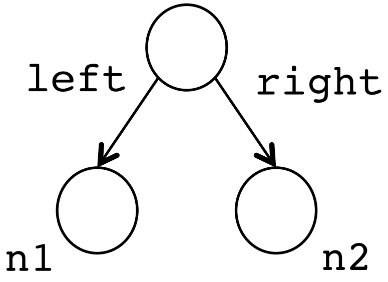
```
class Node {...}

class Character extends Node {...}
class Node implements Comparable<Node> {
    Node(int w) { weight = w; }

    Node(int w, Node l, Node r) {
        weight = w; left = l; right = r;
    }

    public int compareTo(Node n) {
        return weight - n.weight;
    }

    int weight;
    Node left, right;
}
```

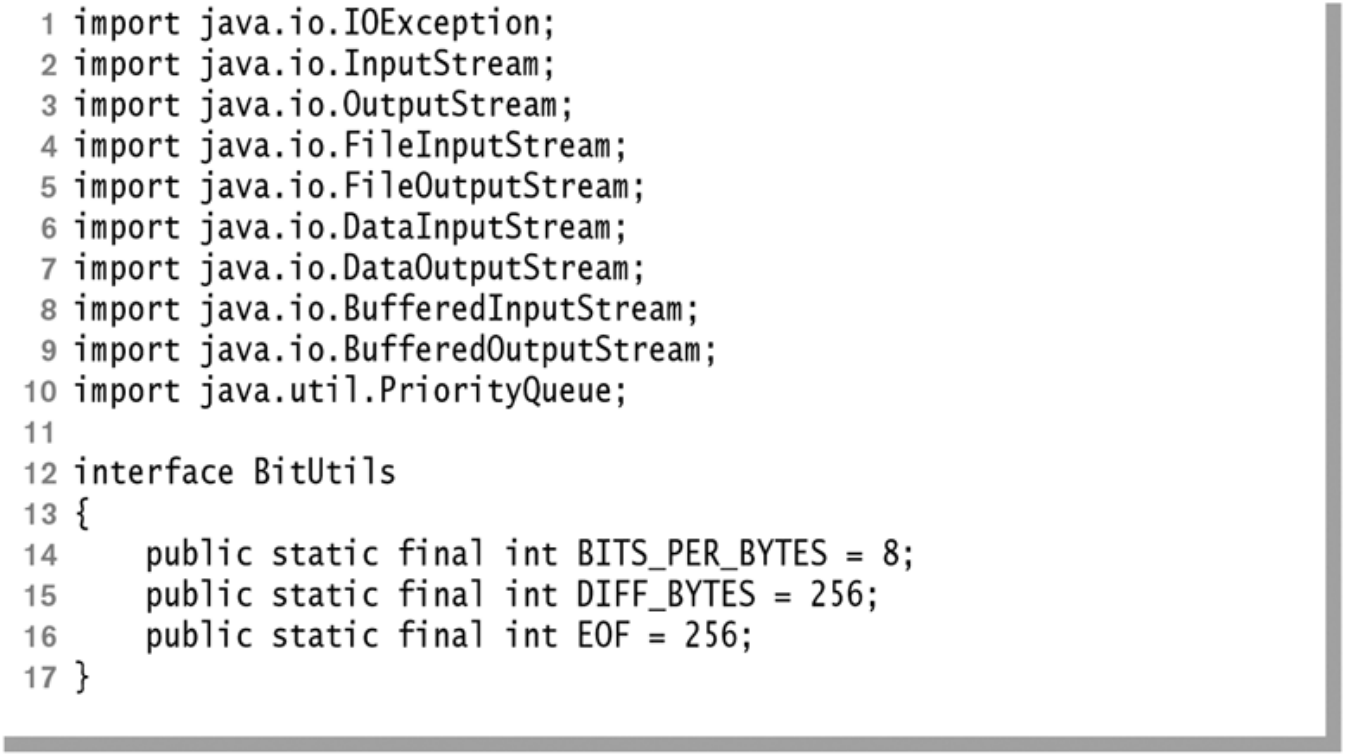
`weight` contains the sum of the frequencies of the leaves in the tree that has this node as root.

`Character` objects are leaves of the tree

```
class Character extends Node {
    Character(char c, int w) {
        super(w);
        character = c;
    }

    char character;
}
```

```
HuffmanTree buildHuffmanTree(List<Character> list) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    for (Character c : list)
       pq.add(c);
    while (pq.size() > 1) {
        Node n1 = pq.remove();
        Node n2 = pq.remove();
        pq.add(new Node(n1.weight + n2.weight, n1, n2));
    }
    return new HuffmanTree(pq.remove());
}
```

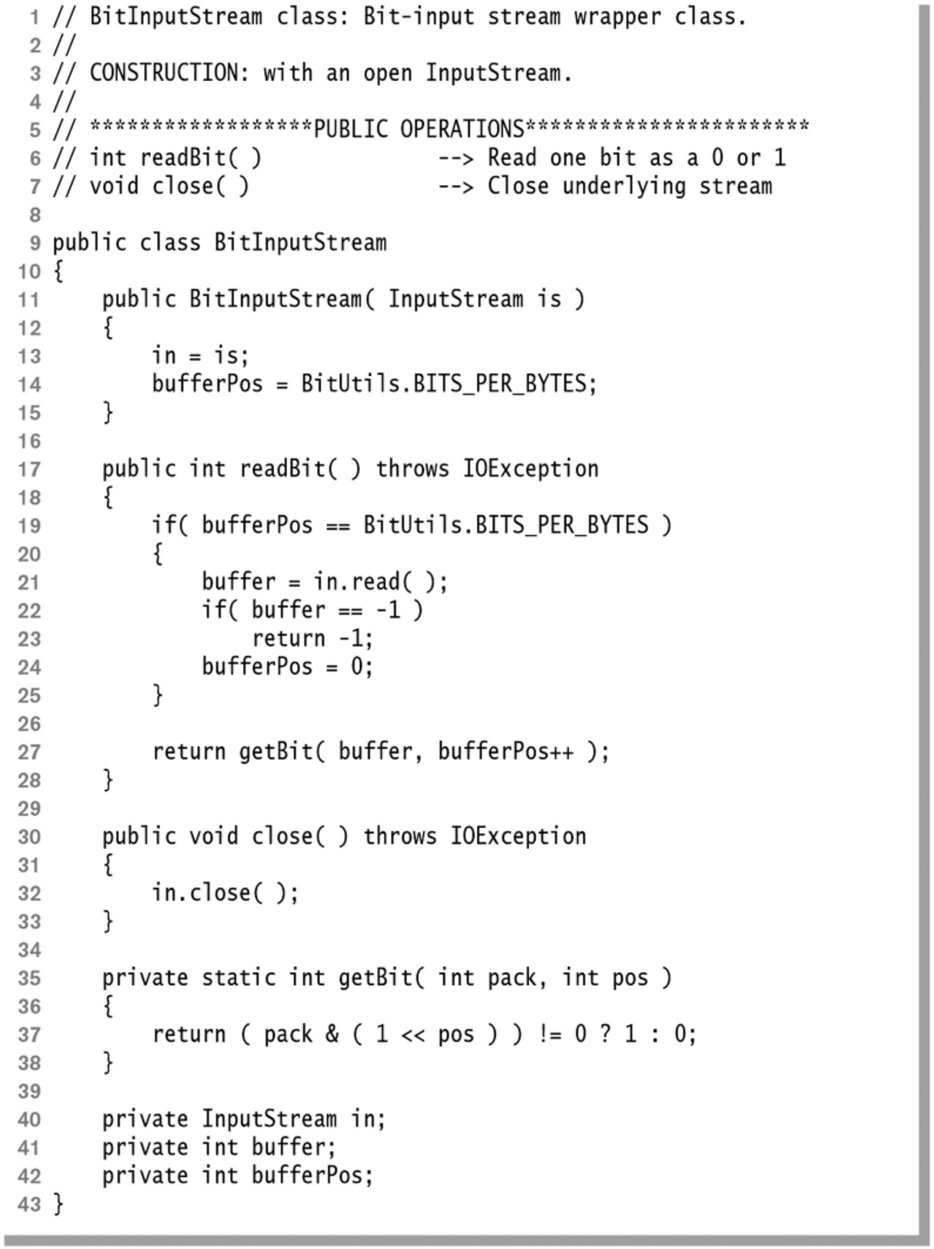




```java
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.util.PriorityQueue;

interface BitUtils
{
    public static final int BITS_PER_BYTES = 8;
    public static final int DIFF_BYTES = 256;
    public static final int EOF = 256;
}
```

figure 12.13

The `import` directives and some constants used in the main compression program algorithms



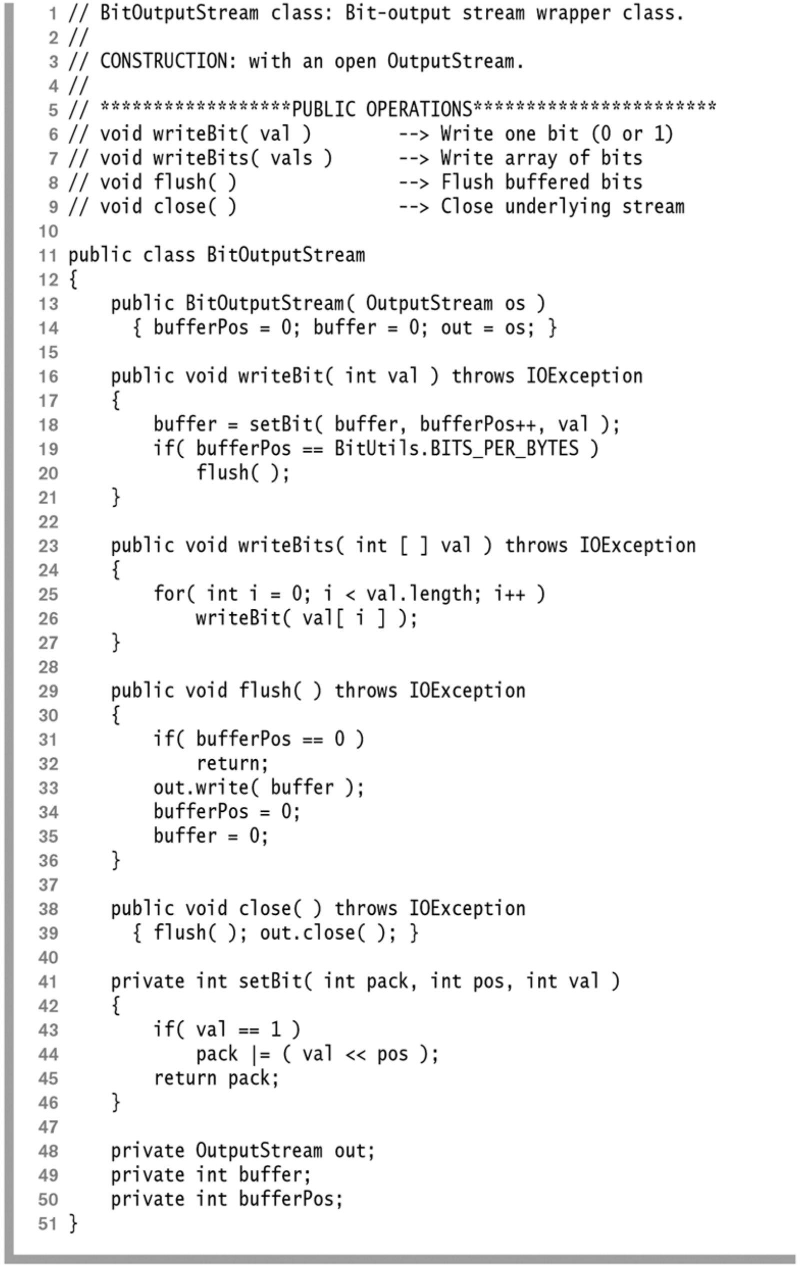
```java
// BitInputStream class: Bit-input stream wrapper class.
//
// CONSTRUCTION: with an open InputStream.
//
// ******************PUBLIC OPERATIONS***********************
// int readBit( )              --> Read one bit as a 0 or 1
// void close( )               --> Close underlying stream

public class BitInputStream
{
    public BitInputStream( InputStream is )
    {
        in = is;
        bufferPos = BitUtils.BITS_PER_BYTES;
    }

    public int readBit( ) throws IOException
    {
        if( bufferPos == BitUtils.BITS_PER_BYTES )
        {
            buffer = in.read( );
            if( buffer == -1 )
                return -1;
            bufferPos = 0;
        }

        return getBit( buffer, bufferPos++ );
    }

    public void close( ) throws IOException
    {
        in.close( );
    }

    private static int getBit( int pack, int pos )
    {
        return ( pack & ( 1 << pos ) ) != 0 ? 1 : 0;
    }

    private InputStream in;
    private int buffer;
    private int bufferPos;
}
```

figure 12.14

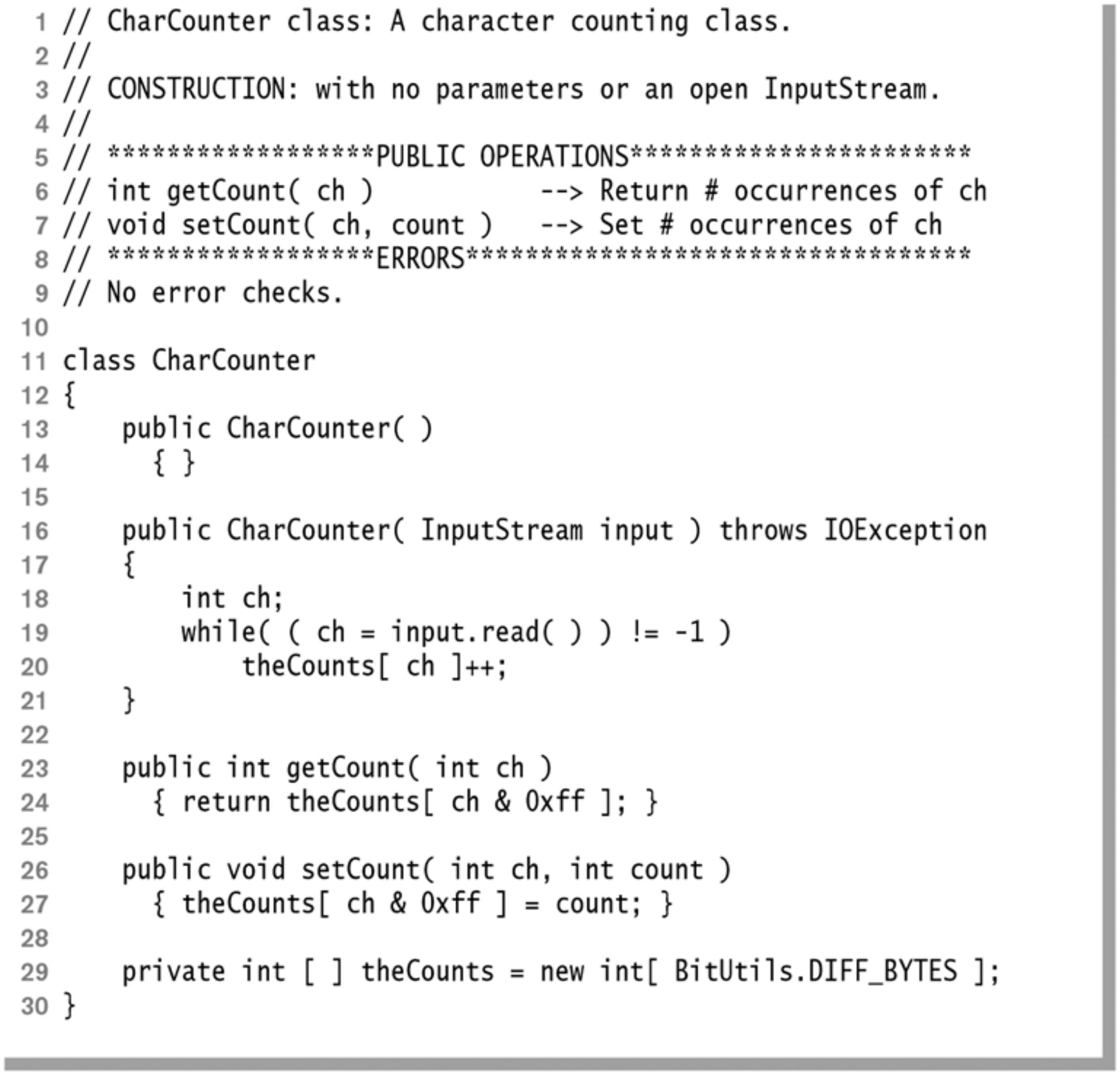
The `BitInputStream` class

figure 12.15

The BitOutputStream class



```
// BitOutputStream class: Bit-output stream wrapper class.
//
// CONSTRUCTION: with an open OutputStream.
//
// ******************PUBLIC OPERATIONS***********************
// void writeBit( val )       --> Write one bit (0 or 1)
// void writeBits( vals )     --> Write array of bits
// void flush( )              --> Flush buffered bits
// void close( )              --> Close underlying stream

public class BitOutputStream
{
    public BitOutputStream( OutputStream os )
      { bufferPos = 0; buffer = 0; out = os; }

    public void writeBit( int val ) throws IOException
    {
        buffer = setBit( buffer, bufferPos++, val );
        if( bufferPos == BitUtils.BITS_PER_BYTES )
            flush( );
    }

    public void writeBits( int [ ] val ) throws IOException
    {
        for( int i = 0; i < val.length; i++ )
            writeBit( val[ i ] );
    }

    public void flush( ) throws IOException
    {
        if( bufferPos == 0 )
            return;
        out.write( buffer );
        bufferPos = 0;
        buffer = 0;
    }

    public void close( ) throws IOException
      { flush( ); out.close( ); }

    private int setBit( int pack, int pos, int val )
    {
        if( val == 1 )
            pack |= ( val << pos );
        return pack;
    }

    private OutputStream out;
    private int buffer;
    private int bufferPos;
}
```



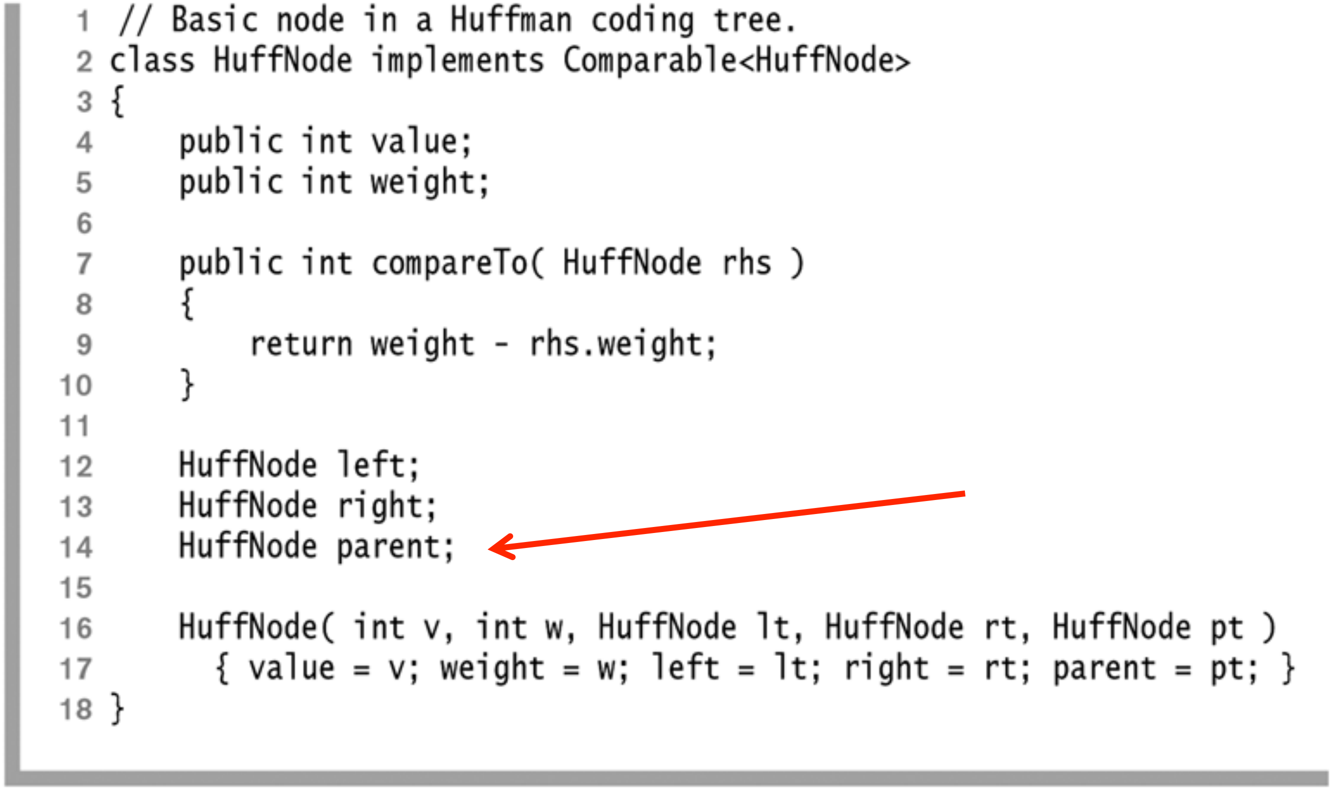
```java
// CharCounter class: A character counting class.
//
// CONSTRUCTION: with no parameters or an open InputStream.
//
// ******************PUBLIC OPERATIONS***********************
// int getCount( ch )           --> Return # occurrences of ch
// void setCount( ch, count )   --> Set # occurrences of ch
// ******************ERRORS**********************************
// No error checks.

class CharCounter
{
    public CharCounter( )
      { }

    public CharCounter( InputStream input ) throws IOException
    {
        int ch;
        while( ( ch = input.read( ) ) != -1 )
            theCounts[ ch ]++;
    }

    public int getCount( int ch )
      { return theCounts[ ch & 0xff ]; }

    public void setCount( int ch, int count )
      { theCounts[ ch & 0xff ] = count; }

    private int [ ] theCounts = new int[ BitUtils.DIFF_BYTES ];
}
```

figure 12.16

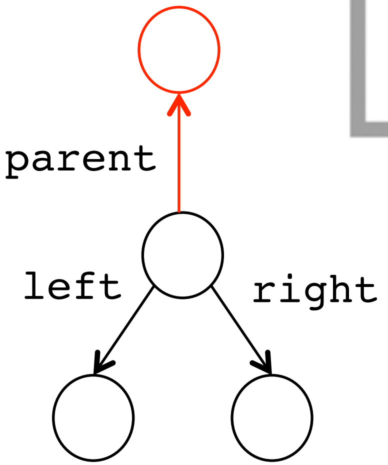
The CharCounter class

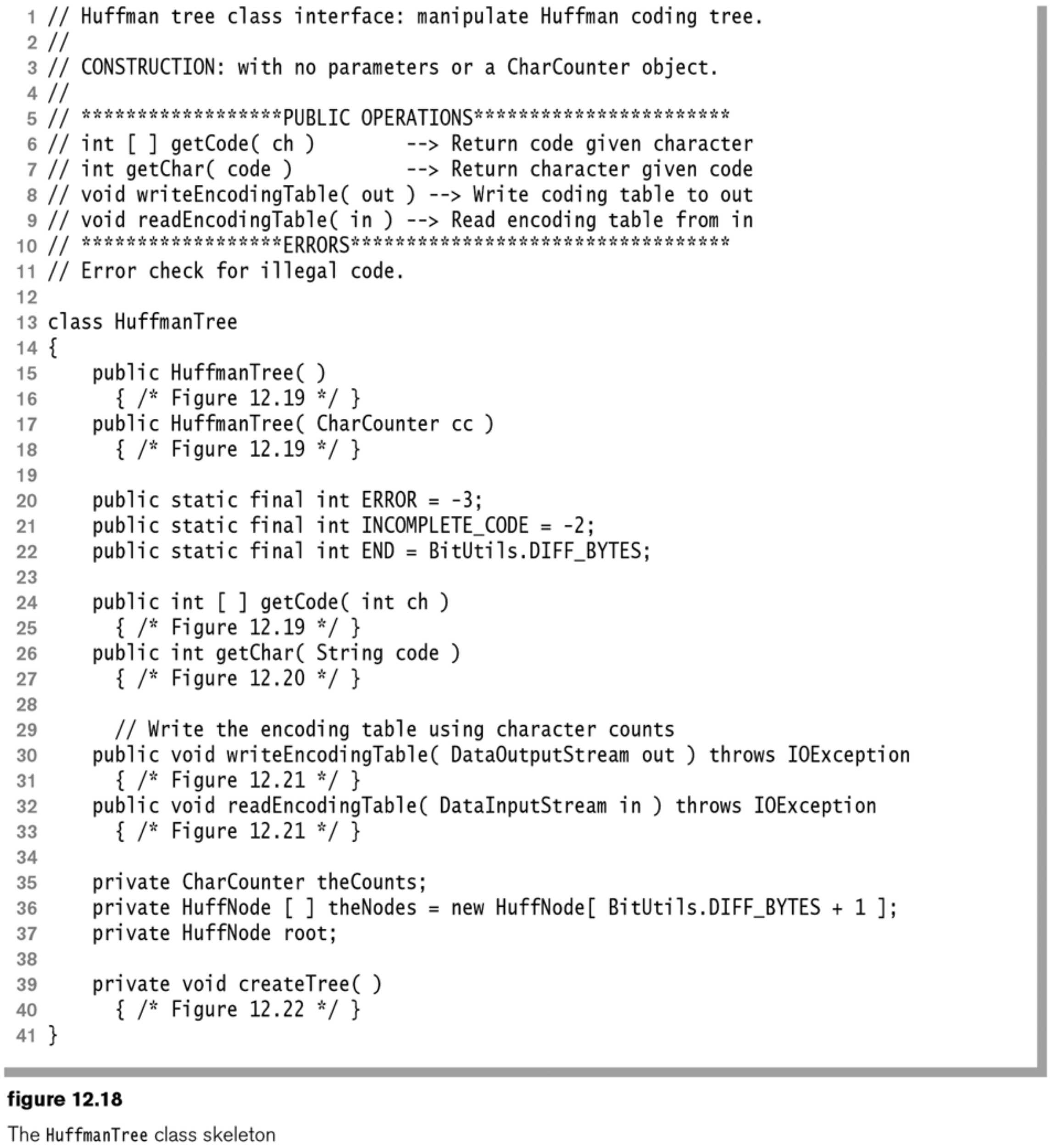
figure 12.17

Node declaration for the Huffman coding tree



```
// Basic node in a Huffman coding tree.
class HuffNode implements Comparable<HuffNode>
{
    public int value;
    public int weight;

    public int compareTo( HuffNode rhs )
    {
        return weight - rhs.weight;
    }

    HuffNode left;
    HuffNode right;
    HuffNode parent;

    HuffNode( int v, int w, HuffNode lt, HuffNode rt, HuffNode pt )
      { value = v; weight = w; left = lt; right = rt; parent = pt; }
}
```





```java
// Huffman tree class interface: manipulate Huffman coding tree.
//
// CONSTRUCTION: with no parameters or a CharCounter object.
//
// ******************PUBLIC OPERATIONS***********************
// int [ ] getCode( ch )        --> Return code given character
// int getChar( code )          --> Return character given code
// void writeEncodingTable( out ) --> Write coding table to out
// void readEncodingTable( in ) --> Read encoding table from in
// ******************ERRORS**********************************
// Error check for illegal code.

class HuffmanTree
{
    public HuffmanTree( )
      { /* Figure 12.19 */ }
    public HuffmanTree( CharCounter cc )
      { /* Figure 12.19 */ }

    public static final int ERROR = -3;
    public static final int INCOMPLETE_CODE = -2;
    public static final int END = BitUtils.DIFF_BYTES;

    public int [ ] getCode( int ch )
      { /* Figure 12.19 */ }
    public int getChar( String code )
      { /* Figure 12.20 */ }

      // Write the encoding table using character counts
    public void writeEncodingTable( DataOutputStream out ) throws IOException
      { /* Figure 12.21 */ }
    public void readEncodingTable( DataInputStream in ) throws IOException
      { /* Figure 12.21 */ }

    private CharCounter theCounts;
    private HuffNode [ ] theNodes = new HuffNode[ BitUtils.DIFF_BYTES + 1 ];
    private HuffNode root;

    private void createTree( )
      { /* Figure 12.22 */ }
}
```

figure 12.18

The `HuffmanTree` class skeleton

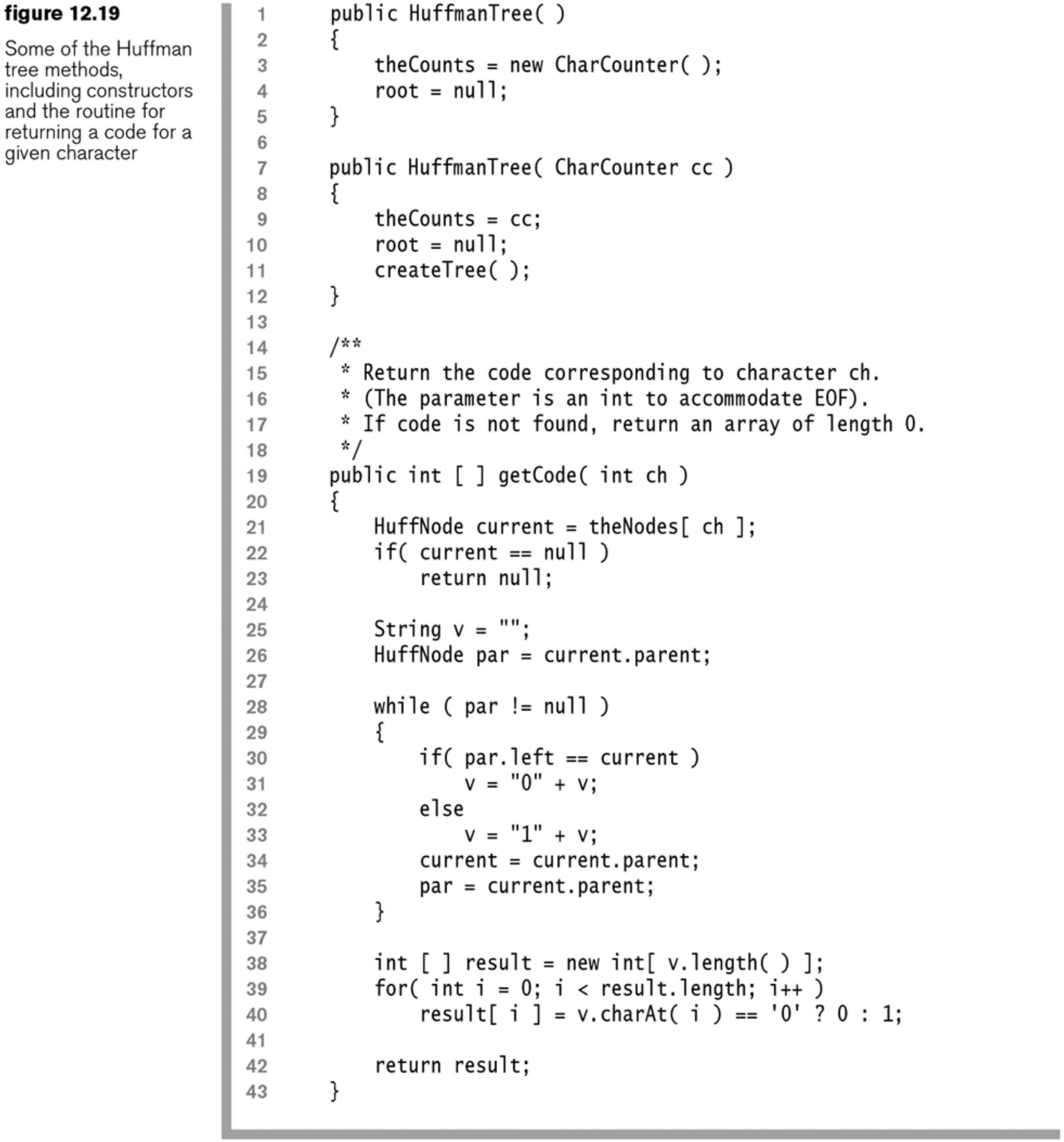
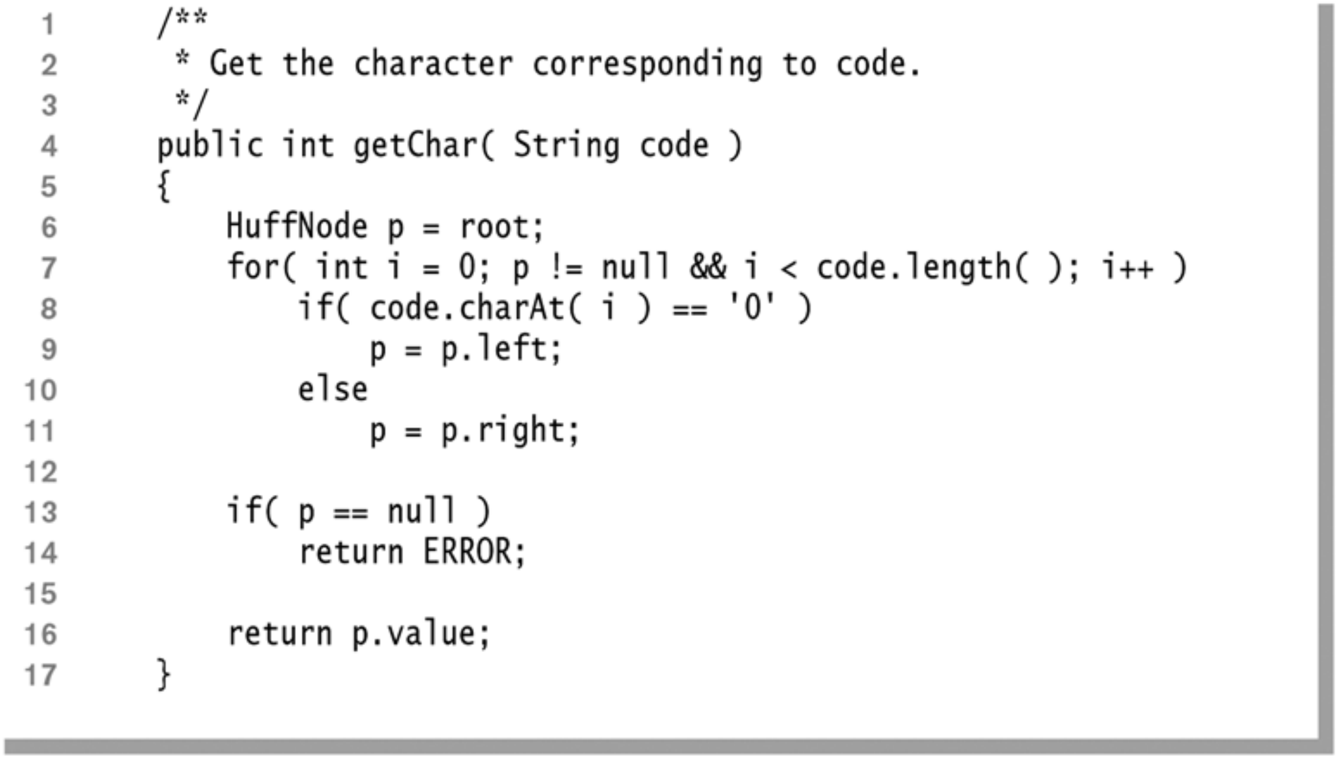


figure 12.19

Some of the Huffman tree methods, including constructors and the routine for returning a code for a given character

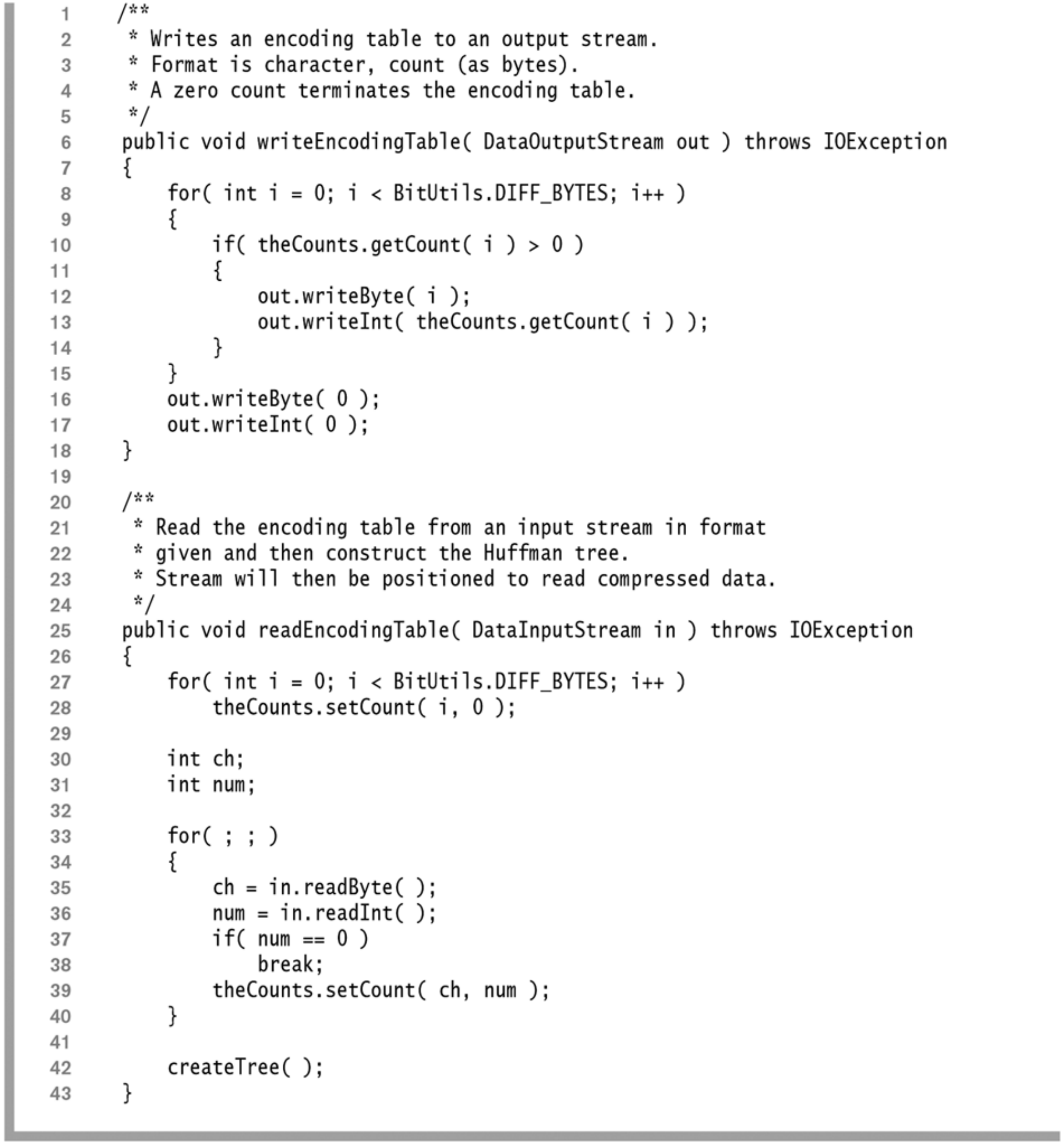
```java
public HuffmanTree( )
{
    theCounts = new CharCounter( );
    root = null;
}

public HuffmanTree( CharCounter cc )
{
    theCounts = cc;
    root = null;
    createTree( );
}

/**
 * Return the code corresponding to character ch.
 * (The parameter is an int to accommodate EOF).
 * If code is not found, return an array of length 0.
 */
public int [ ] getCode( int ch )
{
    HuffNode current = theNodes[ ch ];
    if( current == null )
        return null;

    String v = "";
    HuffNode par = current.parent;

    while ( par != null )
    {
        if( par.left == current )
            v = "0" + v;
        else
            v = "1" + v;
        current = current.parent;
        par = current.parent;
    }

    int [ ] result = new int[ v.length( ) ];
    for( int i = 0; i < result.length; i++ )
        result[ i ] = v.charAt( i ) == '0' ? 0 : 1;

    return result;
}
```



```java
/**
 * Get the character corresponding to code.
 */
public int getChar( String code )
{
    HuffNode p = root;
    for( int i = 0; p != null && i < code.length( ); i++ )
        if( code.charAt( i ) == '0' )
            p = p.left;
        else
            p = p.right;

    if( p == null )
        return ERROR;

    return p.value;
}
```

figure 12.20

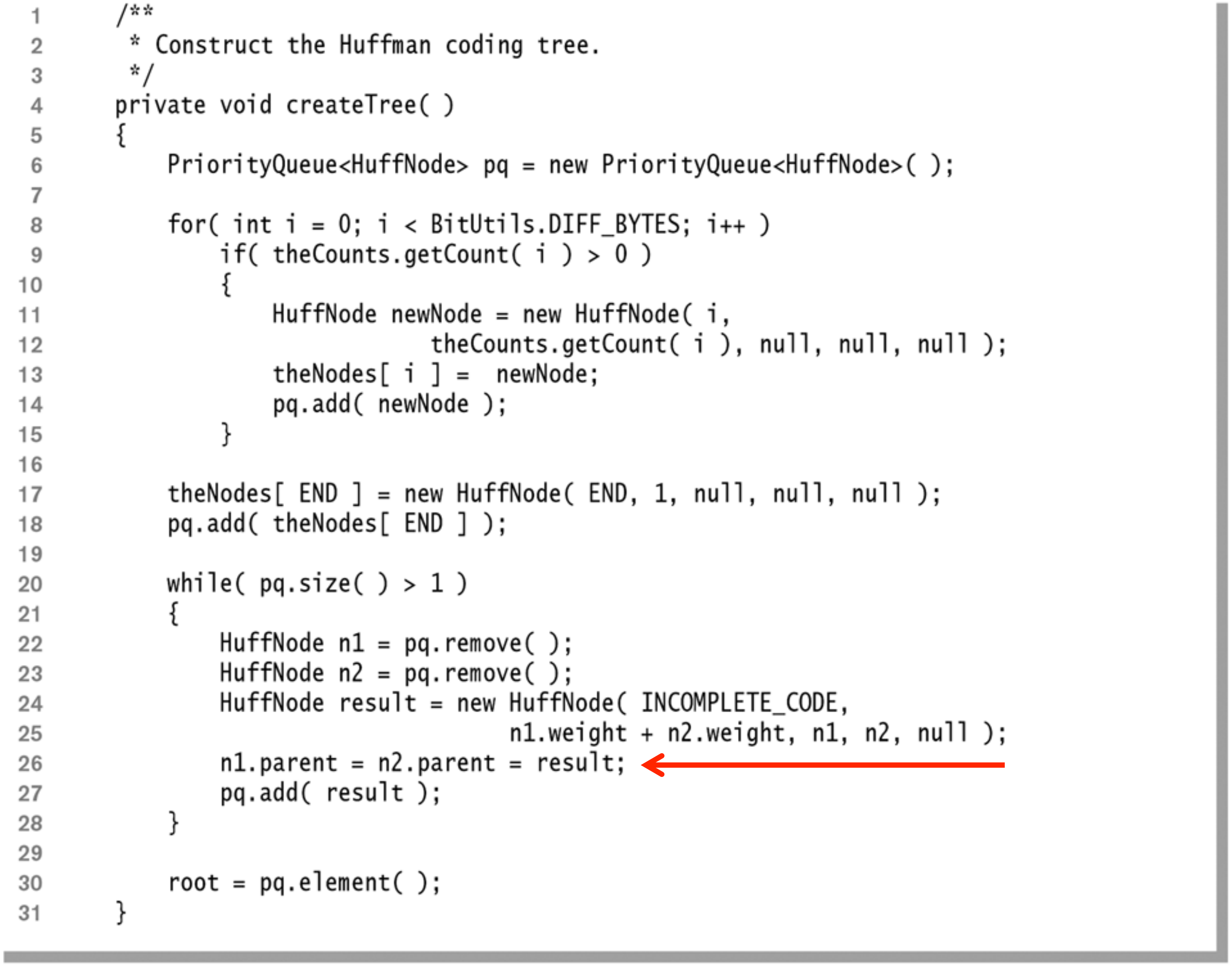
A routine for decoding (generating a character, given the code)



```java
/**
 * Writes an encoding table to an output stream.
 * Format is character, count (as bytes).
 * A zero count terminates the encoding table.
 */
public void writeEncodingTable( DataOutputStream out ) throws IOException
{
    for( int i = 0; i < BitUtils.DIFF_BYTES; i++ )
    {
        if( theCounts.getCount( i ) > 0 )
        {
            out.writeByte( i );
            out.writeInt( theCounts.getCount( i ) );
        }
    }
    out.writeByte( 0 );
    out.writeInt( 0 );
}

/**
 * Read the encoding table from an input stream in format
 * given and then construct the Huffman tree.
 * Stream will then be positioned to read compressed data.
 */
public void readEncodingTable( DataInputStream in ) throws IOException
{
    for( int i = 0; i < BitUtils.DIFF_BYTES; i++ )
        theCounts.setCount( i, 0 );

    int ch;
    int num;

    for( ; ; )
    {
        ch = in.readByte( );
        num = in.readInt( );
        if( num == 0 )
            break;
        theCounts.setCount( ch, num );
    }

    createTree( );
}
```

figure 12.21

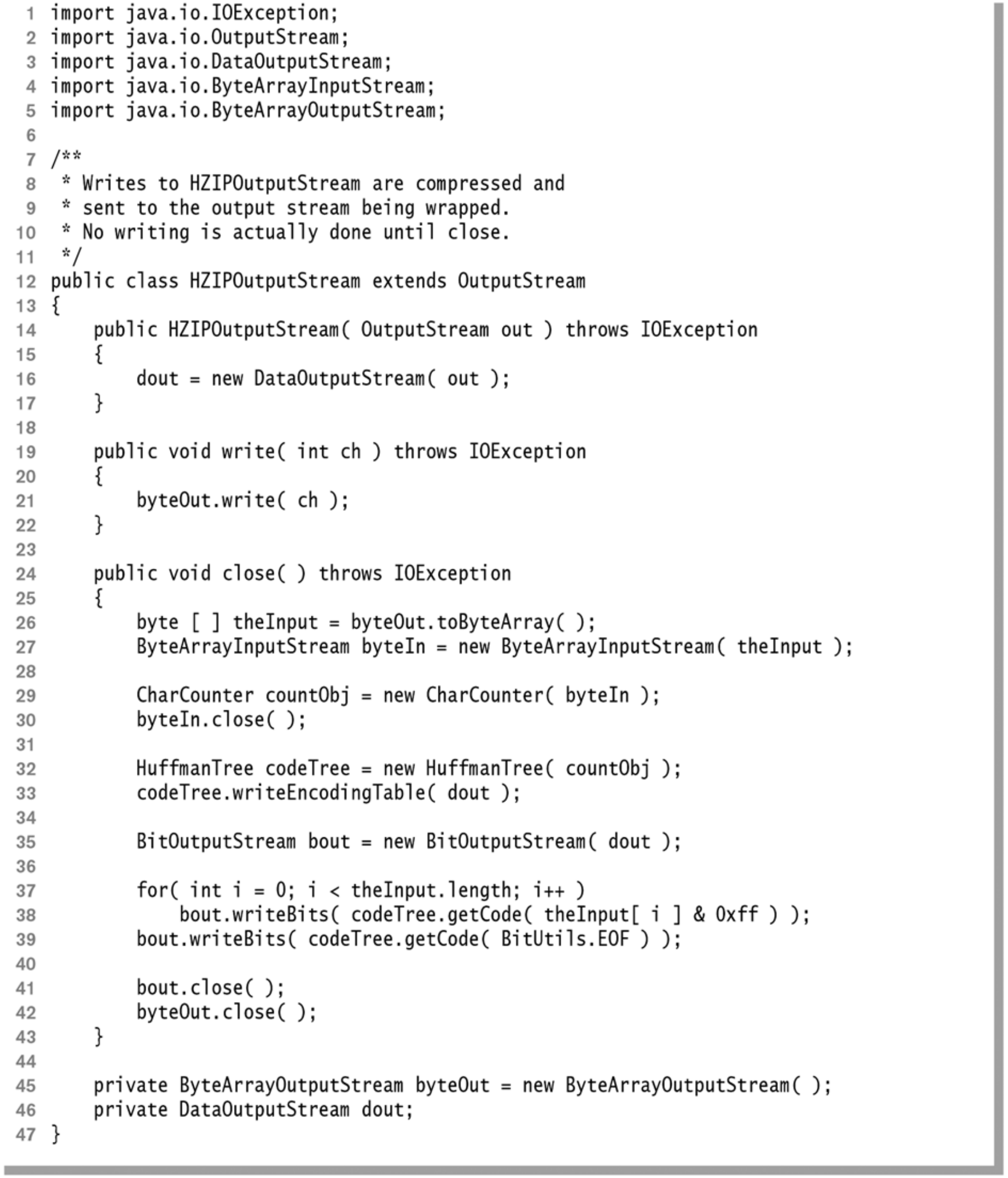
Routines for reading and writing encoding tables



```
/**
 * Construct the Huffman coding tree.
 */
private void createTree( )
{
    PriorityQueue<HuffNode> pq = new PriorityQueue<HuffNode>( );

    for( int i = 0; i < BitUtils.DIFF_BYTES; i++ )
        if( theCounts.getCount( i ) > 0 )
        {
            HuffNode newNode = new HuffNode( i,
                        theCounts.getCount( i ), null, null, null );
            theNodes[ i ] =  newNode;
            pq.add( newNode );
        }

    theNodes[ END ] = new HuffNode( END, 1, null, null, null );
    pq.add( theNodes[ END ] );

    while( pq.size( ) > 1 )
    {
        HuffNode n1 = pq.remove( );
        HuffNode n2 = pq.remove( );
        HuffNode result = new HuffNode( INCOMPLETE_CODE,
                              n1.weight + n2.weight, n1, n2, null );
        n1.parent = n2.parent = result;
        pq.add( result );
    }

    root = pq.element( );
}
```

figure 12.22

A routine for constructing the Huffman coding tree



```java
import java.io.IOException;
import java.io.OutputStream;
import java.io.DataOutputStream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

/**
 * Writes to HZIPOutputStream are compressed and
 * sent to the output stream being wrapped.
 * No writing is actually done until close.
 */
public class HZIPOutputStream extends OutputStream
{
    public HZIPOutputStream( OutputStream out ) throws IOException
    {
        dout = new DataOutputStream( out );
    }

    public void write( int ch ) throws IOException
    {
        byteOut.write( ch );
    }

    public void close( ) throws IOException
    {
        byte [ ] theInput = byteOut.toByteArray( );
        ByteArrayInputStream byteIn = new ByteArrayInputStream( theInput );

        CharCounter countObj = new CharCounter( byteIn );
        byteIn.close( );

        HuffmanTree codeTree = new HuffmanTree( countObj );
        codeTree.writeEncodingTable( dout );

        BitOutputStream bout = new BitOutputStream( dout );

        for( int i = 0; i < theInput.length; i++ )
            bout.writeBits( codeTree.getCode( theInput[ i ] & 0xff ) );
        bout.writeBits( codeTree.getCode( BitUtils.EOF ) );

        bout.close( );
        byteOut.close( );
    }

    private ByteArrayOutputStream byteOut = new ByteArrayOutputStream( );
    private DataOutputStream dout;
}
```

figure 12.23

The `HZIPOutputStream` class

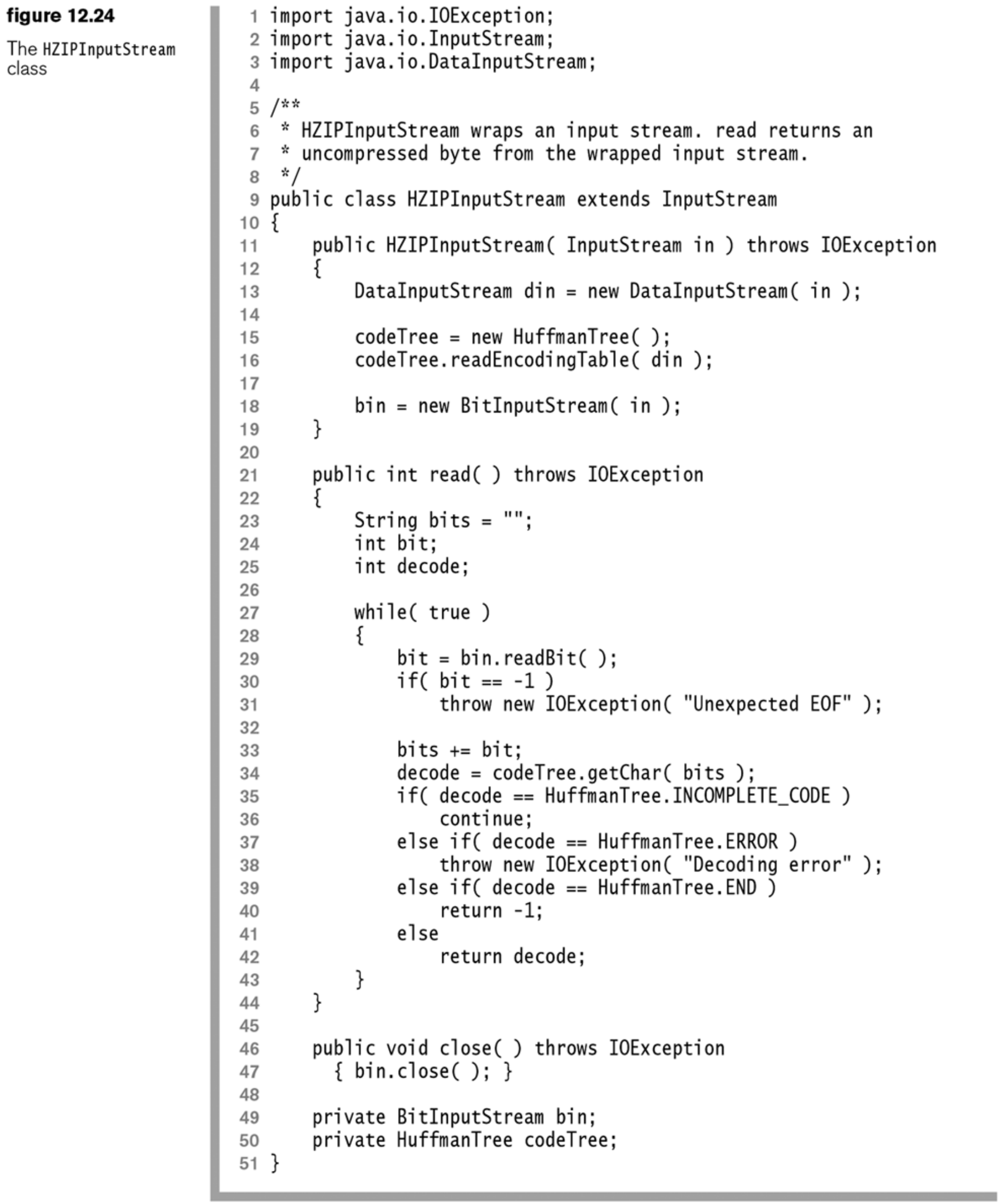
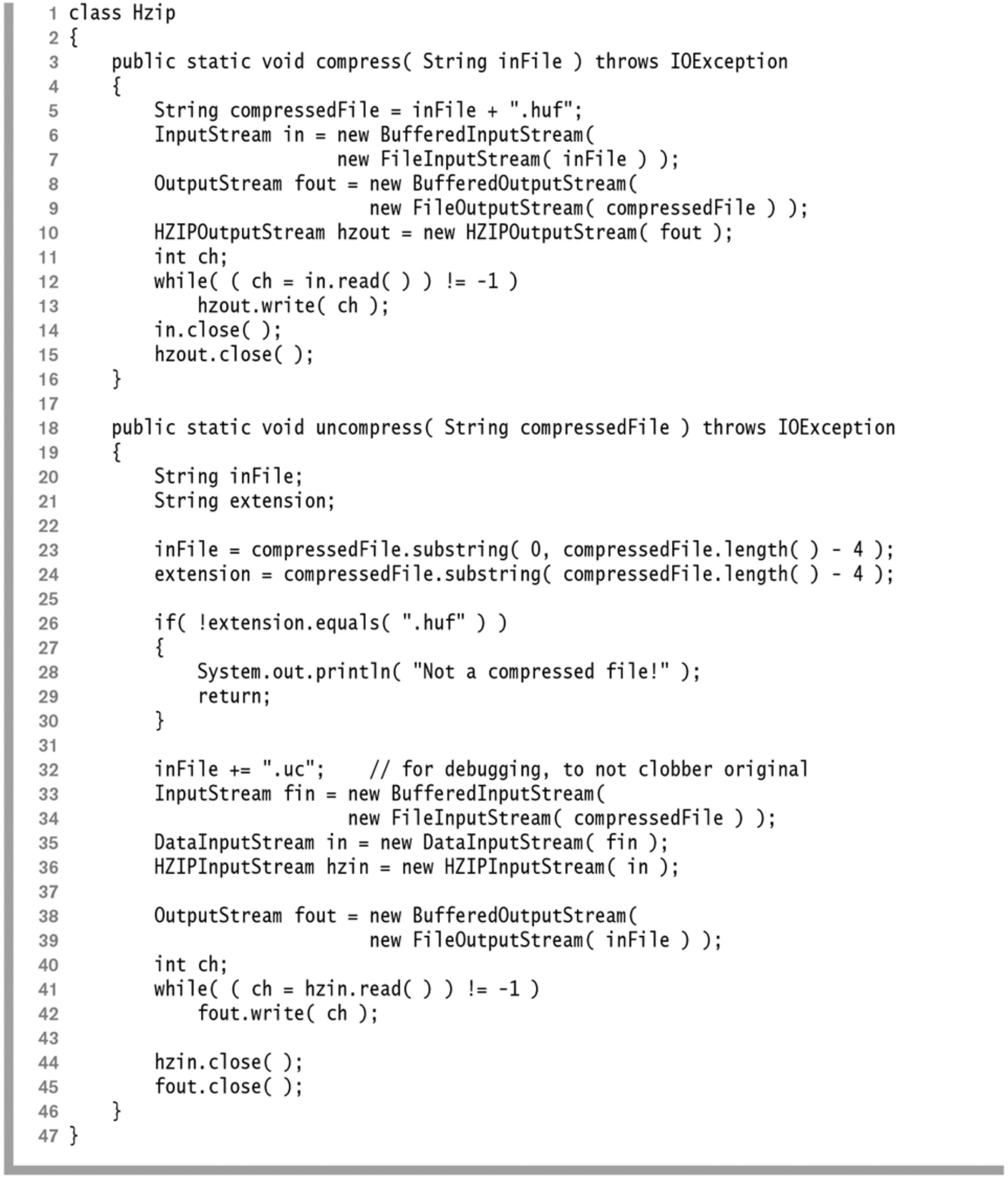


figure 12.24

The `HZIPInputStream` class

```java
import java.io.IOException;
import java.io.InputStream;
import java.io.DataInputStream;

/**
 * HZIPInputStream wraps an input stream. read returns an
 * uncompressed byte from the wrapped input stream.
 */
public class HZIPInputStream extends InputStream
{
    public HZIPInputStream( InputStream in ) throws IOException
    {
        DataInputStream din = new DataInputStream( in );

        codeTree = new HuffmanTree( );
        codeTree.readEncodingTable( din );

        bin = new BitInputStream( in );
    }

    public int read( ) throws IOException
    {
        String bits = "";
        int bit;
        int decode;

        while( true )
        {
            bit = bin.readBit( );
            if( bit == -1 )
                throw new IOException( "Unexpected EOF" );

            bits += bit;
            decode = codeTree.getChar( bits );
            if( decode == HuffmanTree.INCOMPLETE_CODE )
                continue;
            else if( decode == HuffmanTree.ERROR )
                throw new IOException( "Decoding error" );
            else if( decode == HuffmanTree.END )
                return -1;
            else
                return decode;
        }
    }

    public void close( ) throws IOException
      { bin.close( ); }

    private BitInputStream bin;
    private HuffmanTree codeTree;
}
```



```
class Hzip
{
    public static void compress( String inFile ) throws IOException
    {
        String compressedFile = inFile + ".huf";
        InputStream in = new BufferedInputStream(
                          new FileInputStream( inFile ) );
        OutputStream fout = new BufferedOutputStream(
                             new FileOutputStream( compressedFile ) );
        HZIPOutputStream hzout = new HZIPOutputStream( fout );
        int ch;
        while( ( ch = in.read( ) ) != -1 )
            hzout.write( ch );
        in.close( );
        hzout.close( );
    }

    public static void uncompress( String compressedFile ) throws IOException
    {
        String inFile;
        String extension;

        inFile = compressedFile.substring( 0, compressedFile.length( ) - 4 );
        extension = compressedFile.substring( compressedFile.length( ) - 4 );

        if( !extension.equals( ".huf" ) )
        {
            System.out.println( "Not a compressed file!" );
            return;
        }

        inFile += ".uc";    // for debugging, to not clobber original
        InputStream fin = new BufferedInputStream(
                           new FileInputStream( compressedFile ) );
        DataInputStream in = new DataInputStream( fin );
        HZIPInputStream hzin = new HZIPInputStream( in );

        OutputStream fout = new BufferedOutputStream(
                             new FileOutputStream( inFile ) );
        int ch;
        while( ( ch = hzin.read( ) ) != -1 )
            fout.write( ch );

        hzin.close( );
        fout.close( );
    }
}
```

figure 12.25

A simple `main` for file compression and uncompression

Problems for Huffman’s algorithm

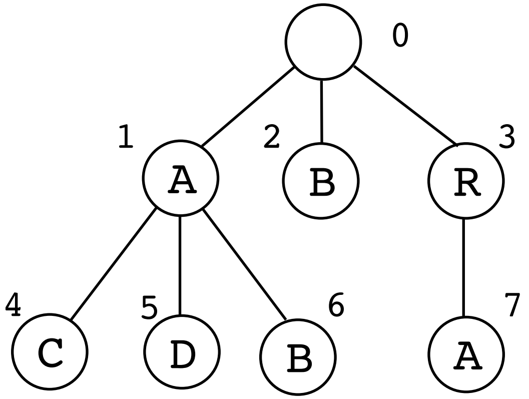
- The encoding table must be transmitted
- Two parses of the file (frequency counting + encoding)
- Typically 25% space reduction, but not optimal

LZW compression

(Lempel, Ziv and Welch, 1977)

Successively builds a dictionary in form of a trie.

Example: `ABRACADABRA`



Encoding: `ABR1C1D1B3A`

A cross-reference generator

Development of a program that scans a Java source file, sorts the identifiers, and outputs the identifiers, along with the line numbers on which they occur.

Identifiers that occur inside comments and string constants should not be included.

Example

input:

```
/* Trivial application that displays a string */1
public class TrivialApplication {                2
    public static void main(String[] args) {     3
        System.out.println("Hello World!");      4
    }                                            5
}                                                6
```

output:

```
String: 3
System: 4
TrivialApplication: 2
args: 3
class: 2
main: 3
out: 4
println: 4
public: 2, 3
static: 3
void: 3
```

Data structures and algorithm

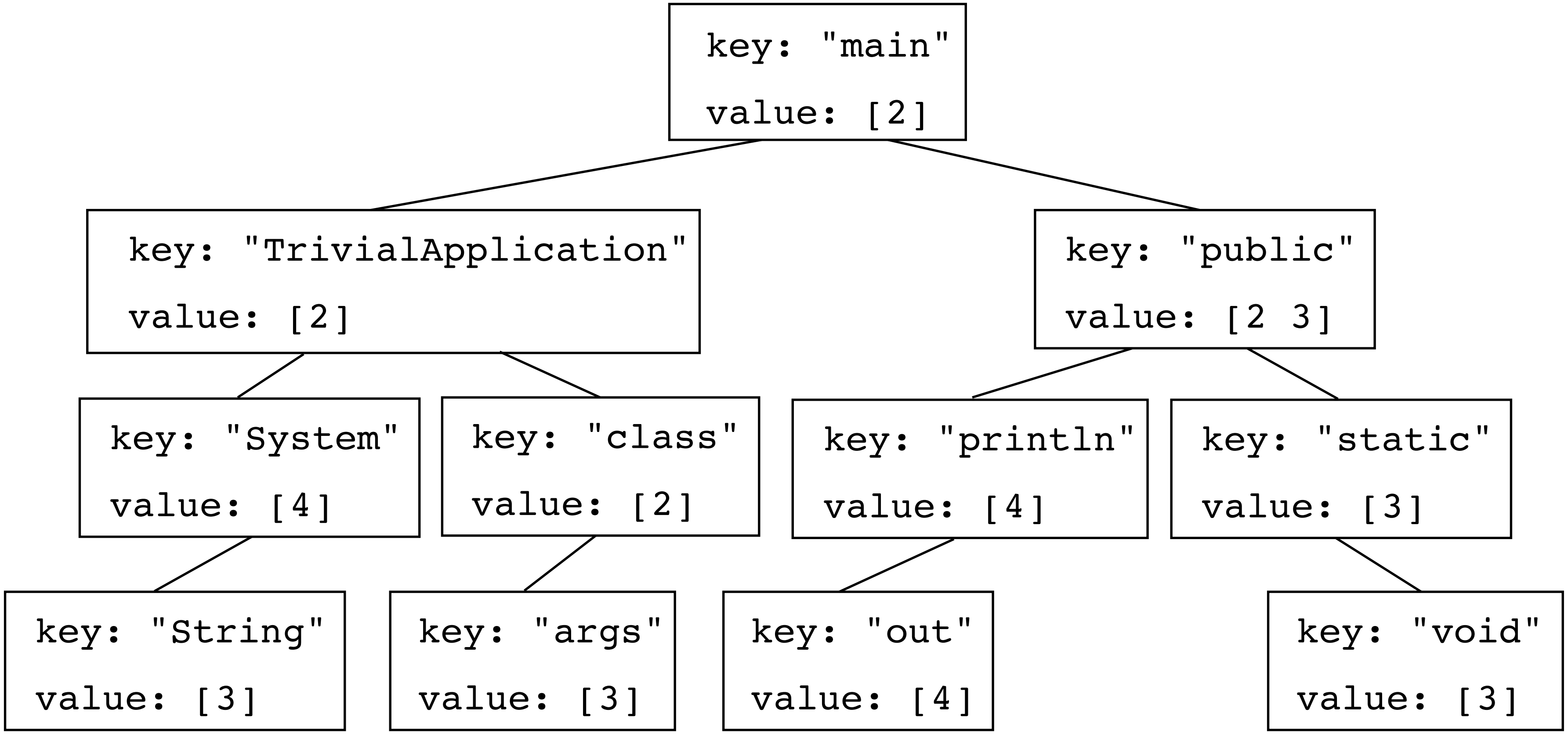
Build a **binary search tree** of all found identifiers.
Each node contains an identifier and a **list** of the lines on which it occurs.
Finally, print the nodes of the tree in sorted order.

```
Map<String,List<Integer>> theIdentifiers =
    new TreeMap<>();
```

Building the map

```
public void generateCrossReference() {
    Map<String,List<Integer>> theIdentifiers =
        new TreeMap<>();
    String id;
    while ((id = tok.getNextID()) != null) {
        List<Integer> lines = theIdentifiers.get(id);
        if (lines == null) {
            lines = new ArrayList<Integer>();
            theIdentifiers.put(id, lines);
        }
        lines.add(tok.getLineNumber());
    }
    // ... print the cross-references  ...
}
```

Example of a binary search tree



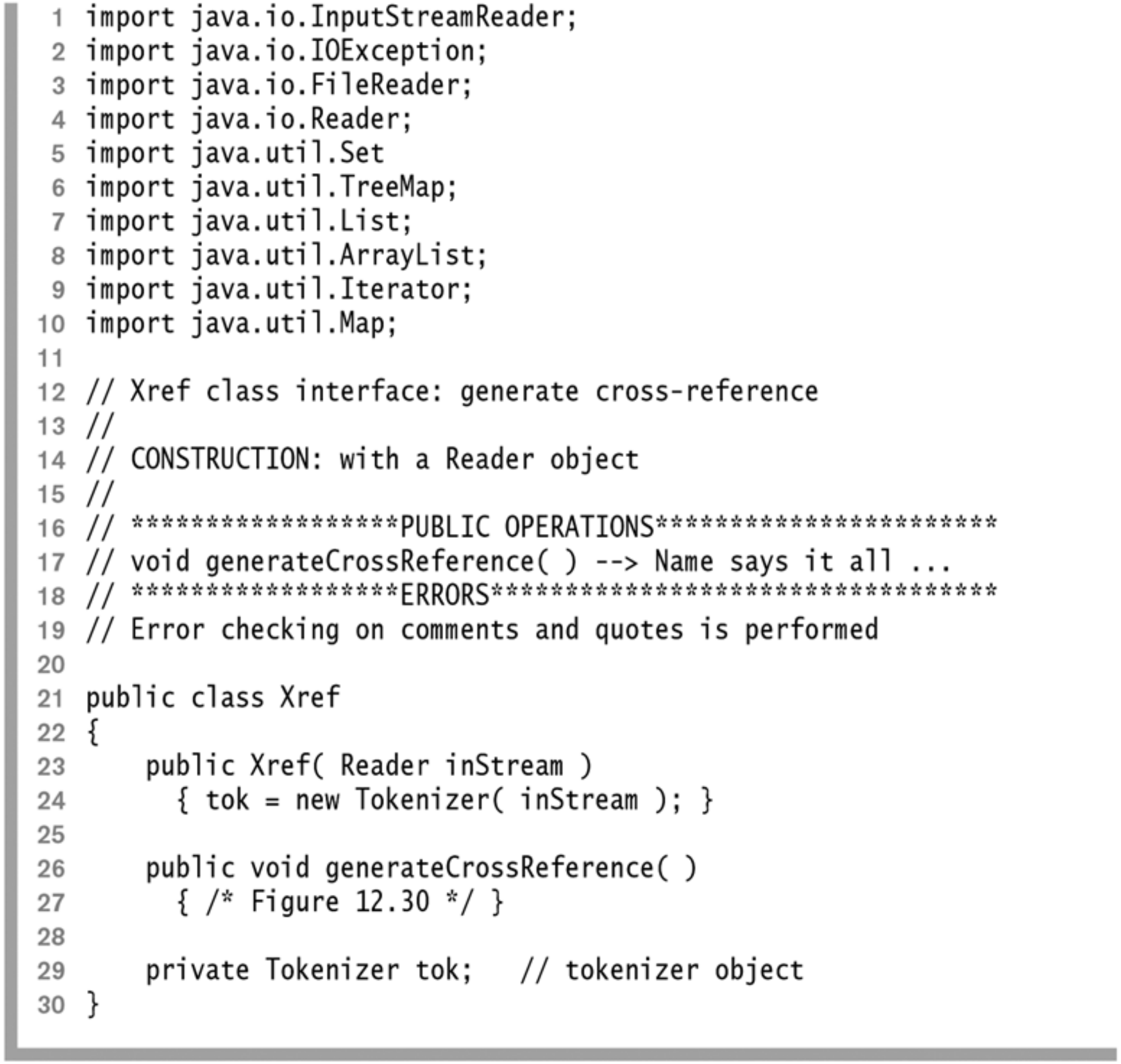


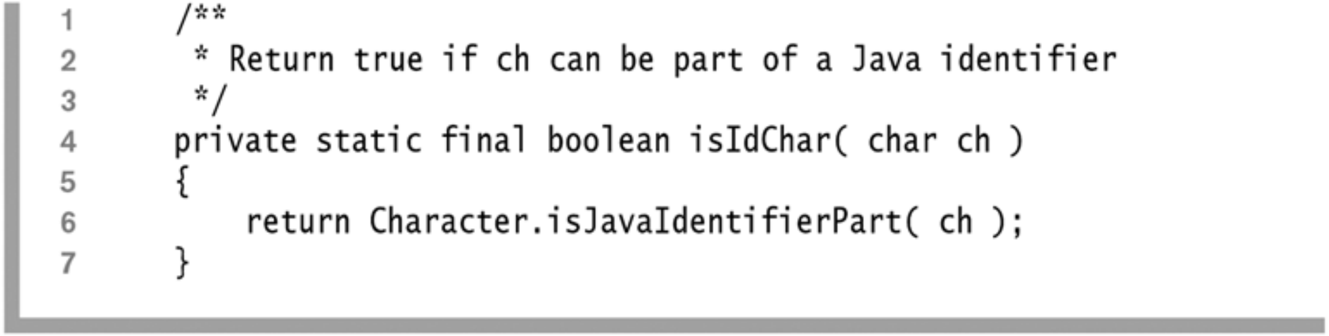
figure 12.26

The Xref class skeleton

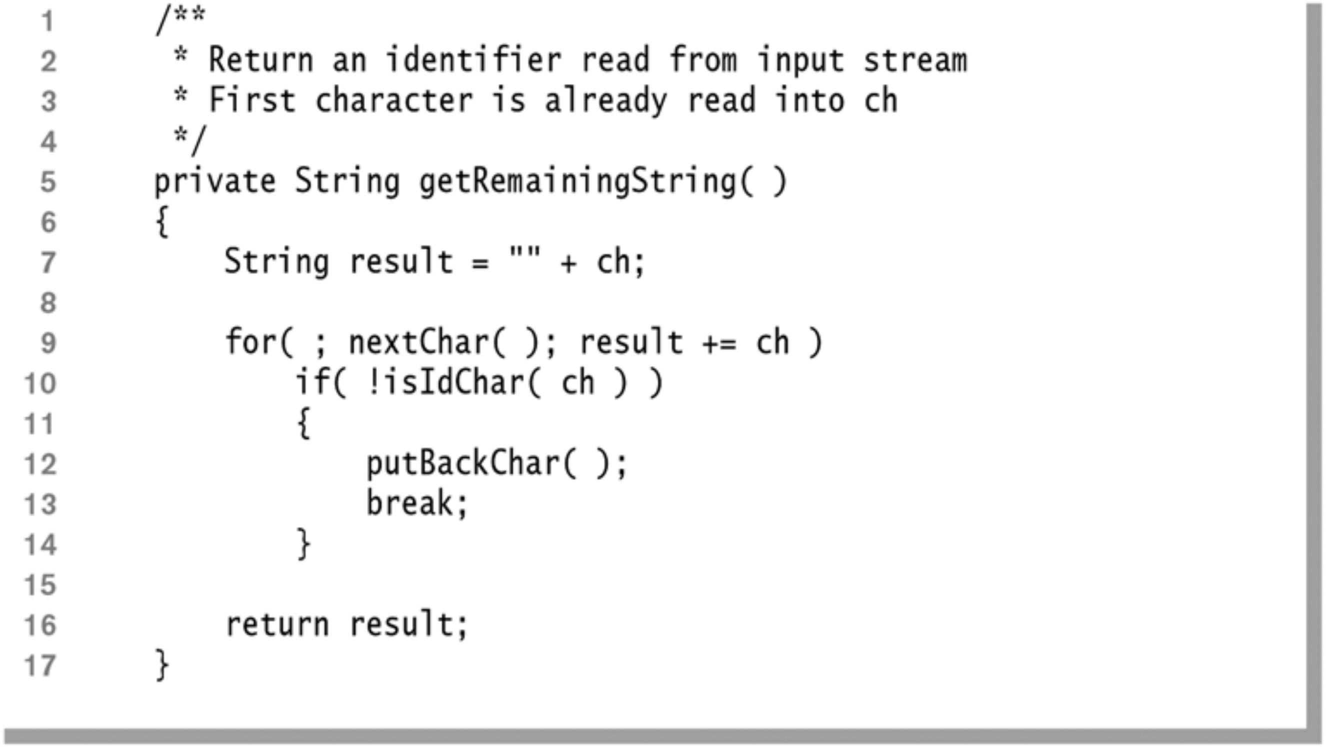
```java
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.FileReader;
import java.io.Reader;
import java.util.Set
import java.util.TreeMap;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;

// Xref class interface: generate cross-reference
//
// CONSTRUCTION: with a Reader object
//
// ******************PUBLIC OPERATIONS***********************
// void generateCrossReference( ) --> Name says it all ...
// ******************ERRORS**********************************
// Error checking on comments and quotes is performed

public class Xref
{
    public Xref( Reader inStream )
      { tok = new Tokenizer( inStream ); }

    public void generateCrossReference( )
      { /* Figure 12.30 */ }

    private Tokenizer tok;   // tokenizer object
}
```

figure 12.27

A routine for testing whether a character could be part of an identifier



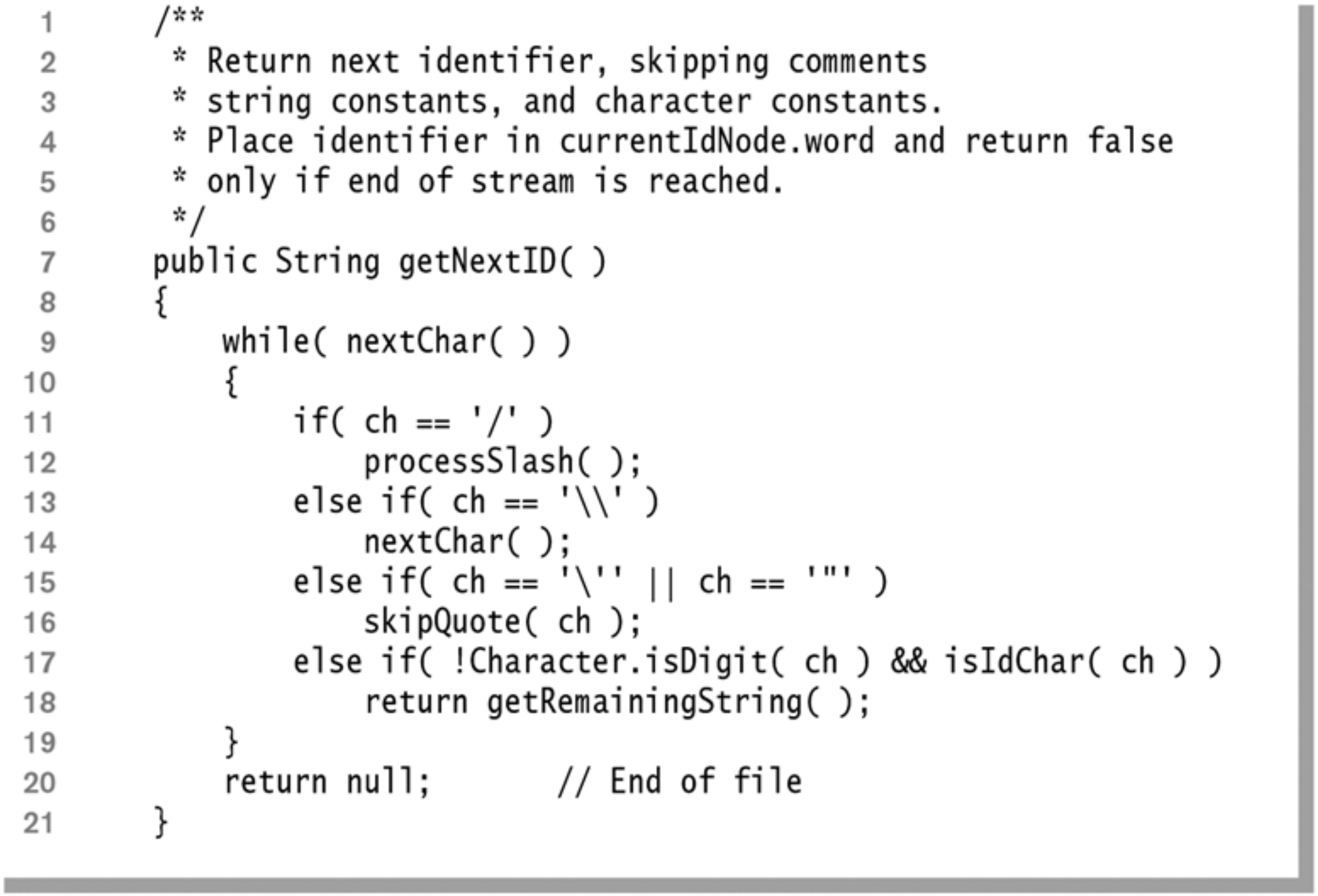
```
/**
 * Return true if ch can be part of a Java identifier
 */
private static final boolean isIdChar( char ch )
{
    return Character.isJavaIdentifierPart( ch );
}
```



```
/**
 * Return an identifier read from input stream
 * First character is already read into ch
 */
private String getRemainingString( )
{
    String result = "" + ch;

    for( ; nextChar( ); result += ch )
        if( !isIdChar( ch ) )
        {
            putBackChar( );
            break;
        }

    return result;
}
```

figure 12.28

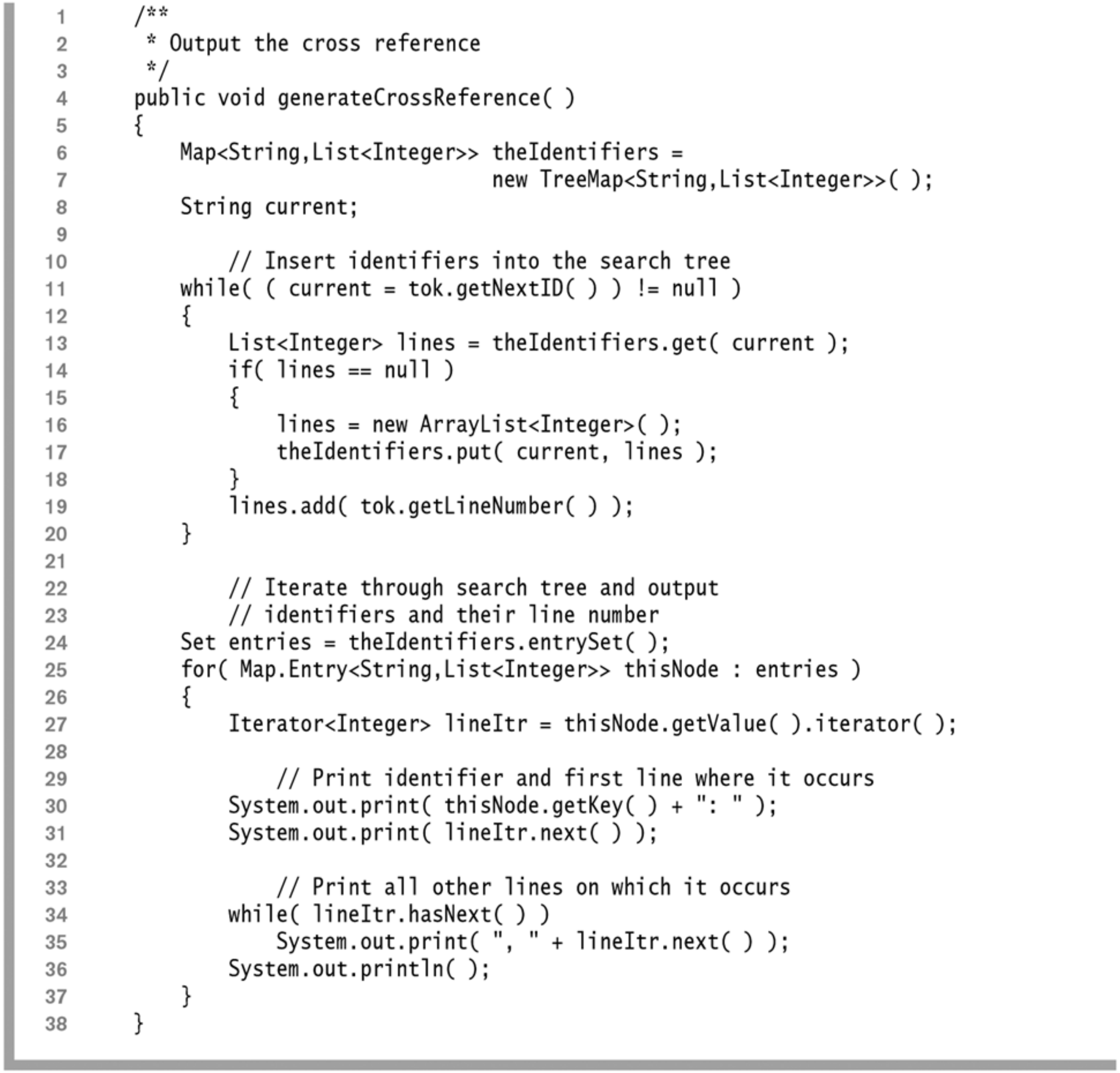
A routine for returning a String from input



```
/**
 * Return next identifier, skipping comments
 * string constants, and character constants.
 * Place identifier in currentIdNode.word and return false
 * only if end of stream is reached.
 */
public String getNextID( )
{
    while( nextChar( ) )
    {
        if( ch == '/' )
            processSlash( );
        else if( ch == '\\' )
            nextChar( );
        else if( ch == '\'' || ch == '"' )
            skipQuote( ch );
        else if( !Character.isDigit( ch ) && isIdChar( ch ) )
            return getRemainingString( );
    }
    return null;      // End of file
}
```

figure 12.29

A routine for returning the next identifier

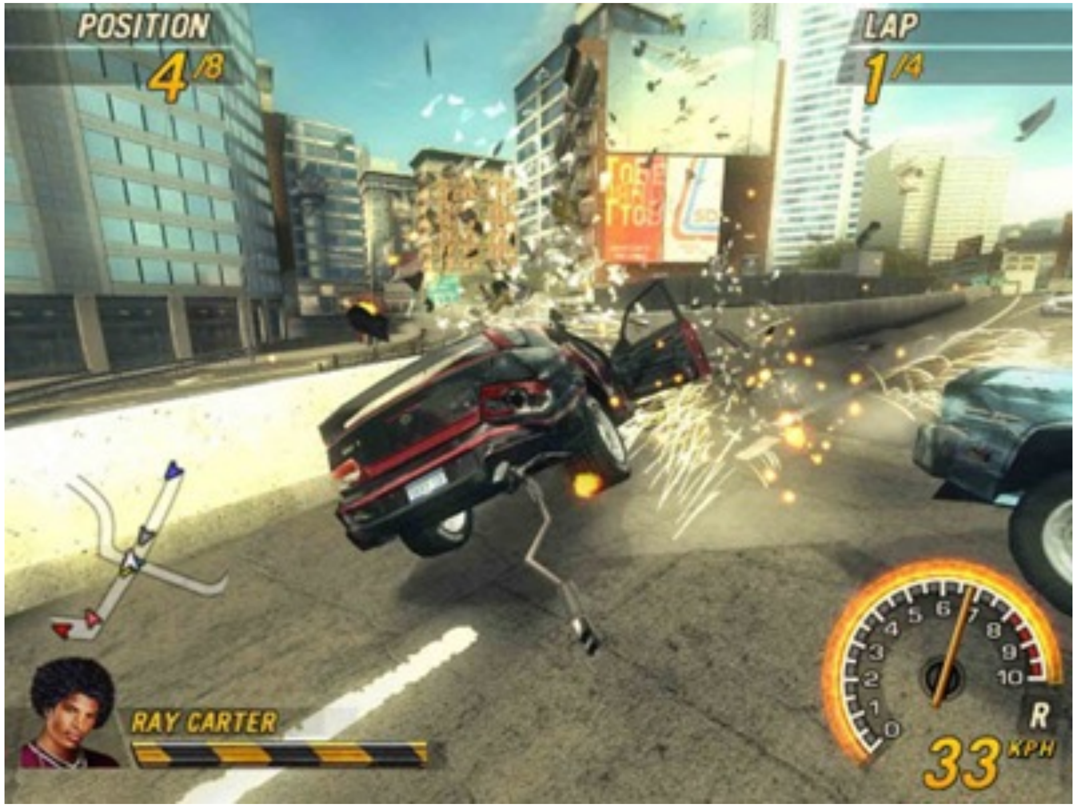


```
/**
 * Output the cross reference
 */
public void generateCrossReference( )
{
    Map<String,List<Integer>> theIdentifiers =
                              new TreeMap<String,List<Integer>>( );
    String current;

        // Insert identifiers into the search tree
    while( ( current = tok.getNextID( ) ) != null )
    {
        List<Integer> lines = theIdentifiers.get( current );
        if( lines == null )
        {
            lines = new ArrayList<Integer>( );
            theIdentifiers.put( current, lines );
        }
        lines.add( tok.getLineNumber( ) );
    }

        // Iterate through search tree and output
        // identifiers and their line number
    Set entries = theIdentifiers.entrySet( );
    for( Map.Entry<String,List<Integer>> thisNode : entries )
    {
        Iterator<Integer> lineItr = thisNode.getValue( ).iterator( );

            // Print identifier and first line where it occurs
        System.out.print( thisNode.getKey( ) + ": " );
        System.out.print( lineItr.next( ) );

            // Print all other lines on which it occurs
        while( lineItr.hasNext( ) )
            System.out.print( ", " + lineItr.next( ) );
        System.out.println( );
    }
}
```

figure 12.30

The main cross-reference algorithm

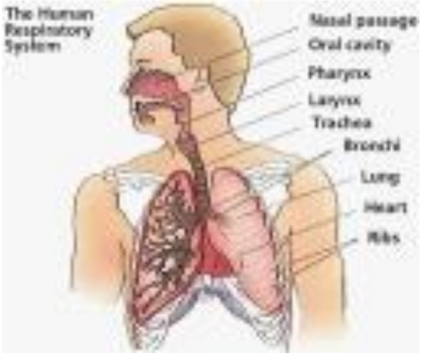
Simulation



What is simulation?

Experiments with
models
on a computer

Models and systems



Model

Representation of a system

System

A chosen extract of reality

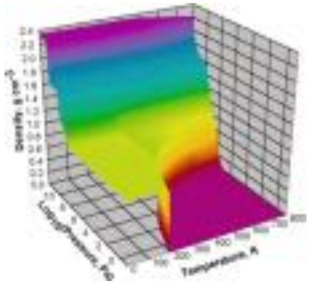
Classification of models

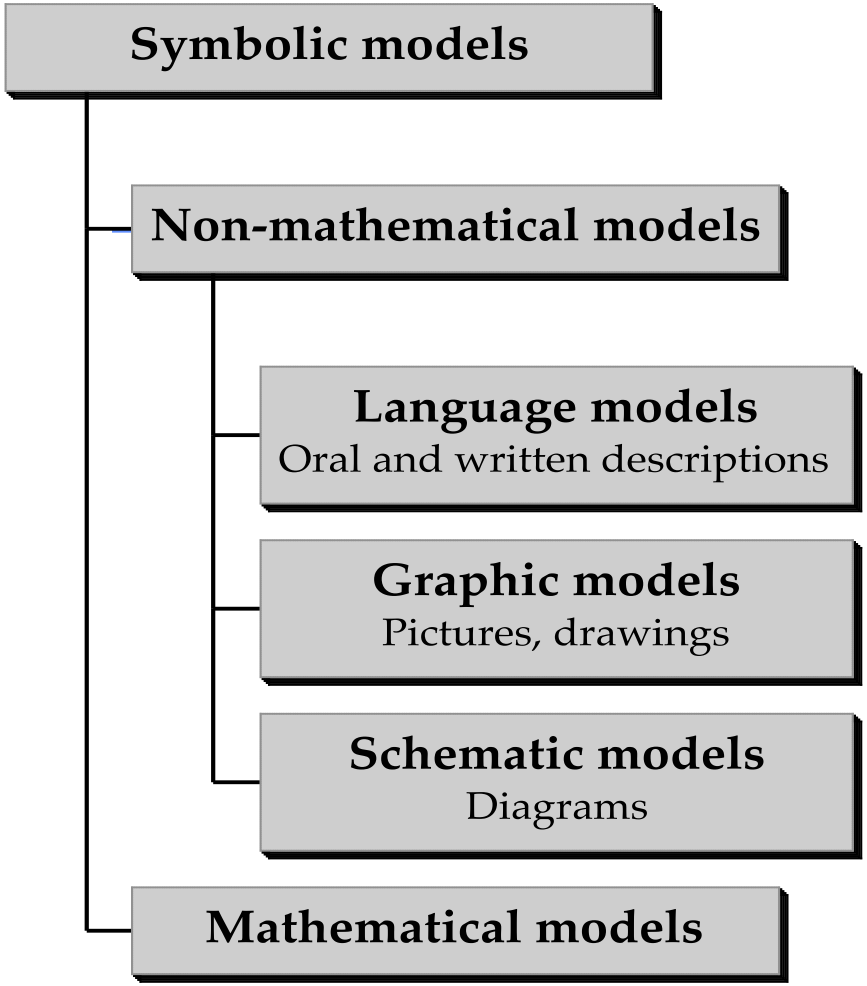
- Mental

  (e.g., a person’s perception of an object, a “world view”)

- Physical
  (e.g., a model railway, a wax figure, a globe)

- Symbolic
  (e.g., $H_2 + 0 \Rightarrow$ water, $F = ma$)





Symbolic models

Non-mathematical models

Language models
Oral and written descriptions

Graphic models
Pictures, drawings

Schematic models
Diagrams

Mathematical models

Mathematical models (1)

- **Static**

  Representation of a system in a given **fixed state**

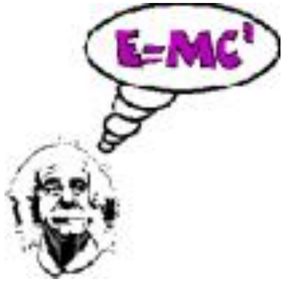
- **Dynamic**

  Representation of a system’s behavior over time

Mathematical models (2)

- **Analytica**l

  Relevant questions about the model can be answered by mathematical reasoning (they have a *closed form solution*)



- **Non-analytical**

  Relevant questions about the model are mathematically *unmanageable* (holds for most real-world models)

Simulation

a possible narrowing

Simulation is experimentation with **dynamic**, **non-analytical** models on a computer

Application examples

- **Biology**
  an ecosystem (e.g., the life in a lake), cell growth, the human circulatory system, vegetation)

- **Physics**
  nuclear processes, mechanical movement (e.g., solar systems, launching of rockets)

- **Chemistry**
  chemical reactions, chemical process plants

- **Geography**
  urban development, growth of a population

- **Computer science**
  computers, networks, video games, robotics

- **Management science**
  organizational decision making

Modeling is purposive

Models can neither be false or true.
They can be more or less appropriate in relation to their purpose.

A *good* model is a model that serves its *purpose*.

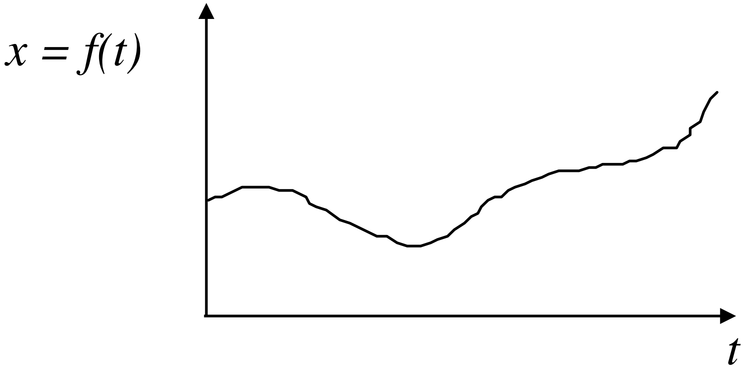
The first step of a modeling process is a clarification of what the model is to be used for.

Abstraction and *aggregation* are used for obtaining *manageable* models.

Abstraction: Ignorance from irrelevant properties
Aggregation: Grouping several things together and considering them as a whole

Dynamic model types

- **Continuous**
  The state of the model is described by variables that vary continuously (without jumps).



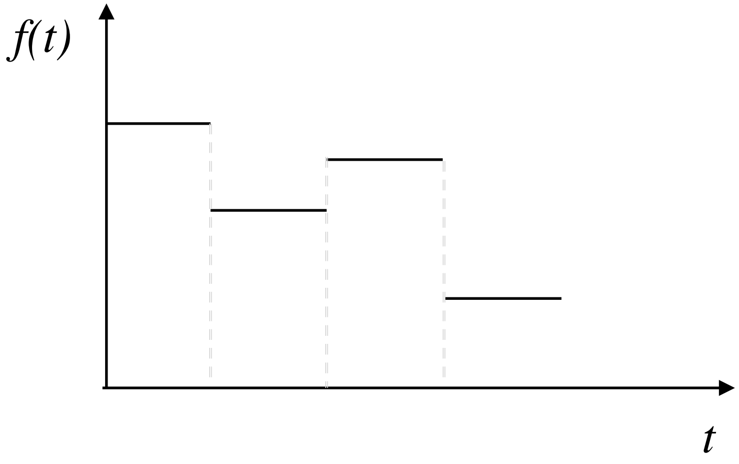
The model is usually expressed as ordinary differential equations and/or difference equations.

$$\boxed{\frac{dx}{dt} = g(x,t)}$$

$$\boxed{x_{next} = x_{now} + g(x_{now}, t)\Delta t}$$

- **Discrete**

The state of the model is described by variables that vary in jumps (caused by *events*).

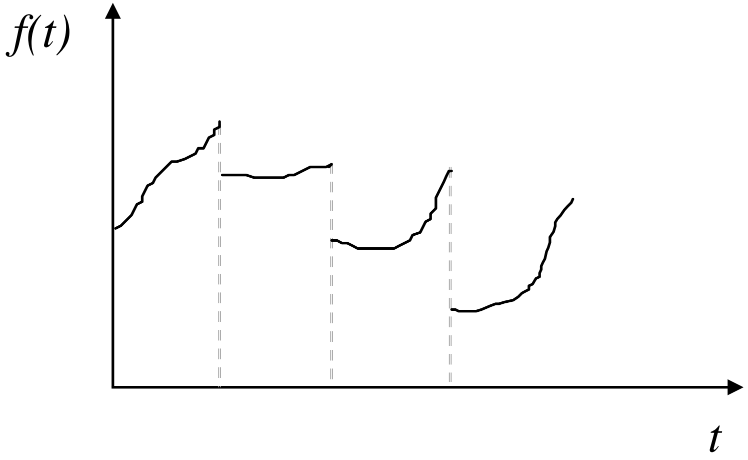


Example:

A queue system (customers in a bank, patients in a health centre).

- **Combined continuous and discrete**

  The state may be described by variables that vary continuously and are changed in jumps.



Examples:

Refrigerator (the heat exchange with the surroundings is continuous, whereas the thermostat causes discrete events)

Elevator (the movement between floors is continuous, whereas start and stop of the elevator are discrete events).

Reasons for using simulation

- The system does not exist
- Experiments with the real system are too expensive, too time-consuming, or too dangerous
- Experiments with the real system are practically impossible (e.g., the sun system)

Purpose of simulation

(1) **Decision making**

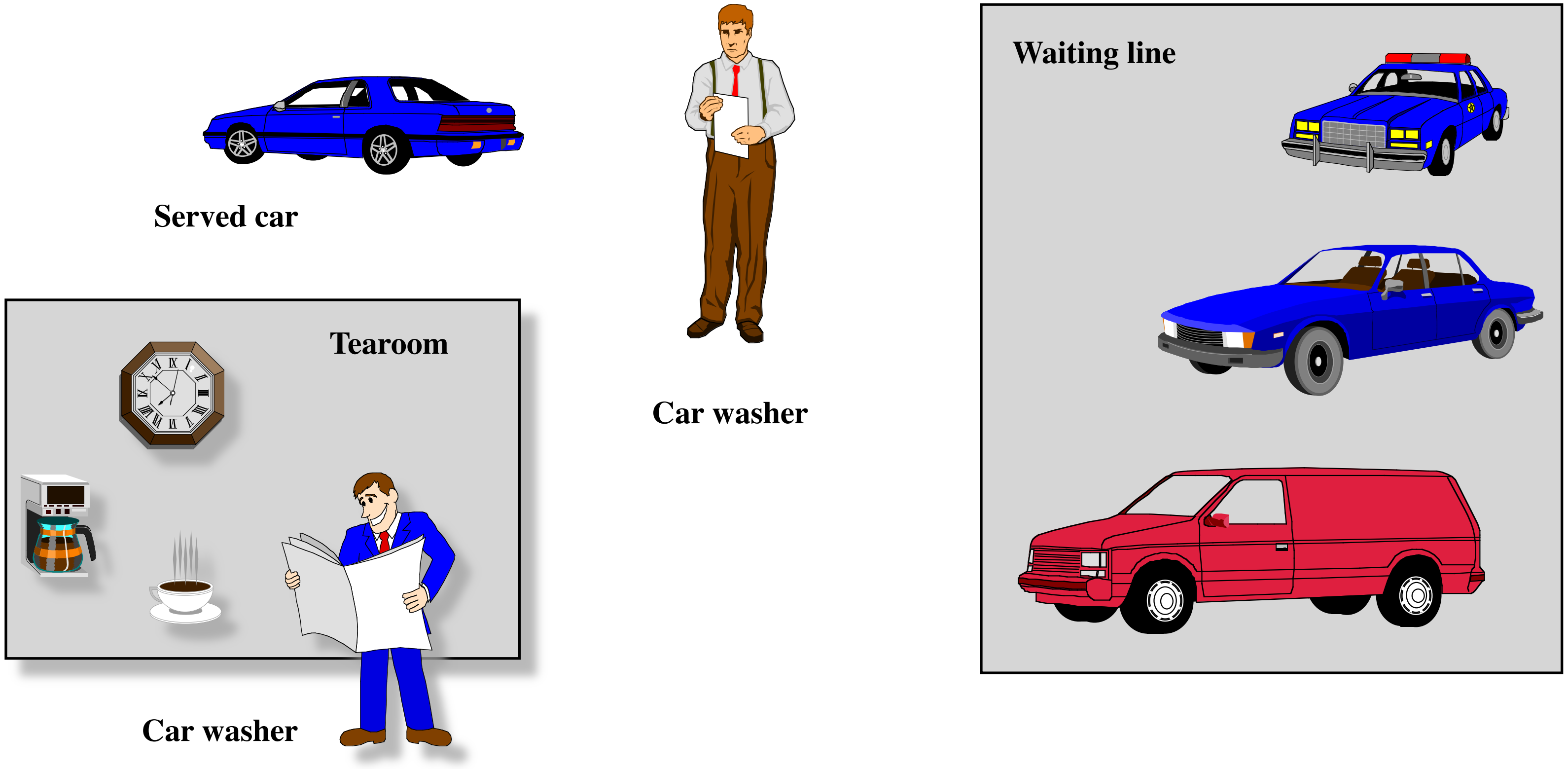
(2) **Insight**

Difficulties of simulation

- May be very expensive, in machine as well as man resources
- Validation is difficult
- Collection of data, and analysis and interpretation of results usually implies good knowledge of statistics

Carwash simulation

Simulation of a carwash



Served car

Car washer

Waiting line

Tearoom

Car washer

System description

(1) The average time between car arrivals has been estimated at 11 minutes.

(2) When a car arrives, it goes straight into the car wash if this is idle; otherwise, it must wait in a queue.

(3) As long as cars are waiting, the car wash is in continuous operation serving on a first-come basis.

(4) Each service takes exactly 10 minutes.

(5) The car washer starts his day in a tearoom and returns there each time he has no work to do.

(6) The carwash is open 8 hours per day.

(7) All cars that have arrived before the carwash closes down are washed.

Purpose of the simulation

(determines the model)

The purpose is to evaluate how much waiting time is reduced by engaging one more car washer.

Model type

A **discrete event** model

Simulation paradigms

(1) **Event-based** ⟸
(E.g., “A car arrives”, “A wash is finished”)

(2) **Activity-based**
(E.g., “A car is being washed”)

(3) **Process-based** ⟸
(E.g., “A car”, “A car washer”)

Identification of events

(1) A car arrives (`CarArrival`)

(2) A wash is started (`StartCarWashing`)

(3) A wash is finished (`StopCarWashing`)

The package `simulation.event`

```
public abstract class Event {
   protected abstract void actions();
   public final void schedule(double evTime);
   public final void cancel();
   public final static double time();
   public final static void runSimulation(double period);
   public final static void stopSimulation();
}
```

Events and their associated actions are defined in subclasses of class `Event`.

```
import simulation.event.*;
import simset.*;
import random.*;
```

```
public class CarWashSimulation extends Simulation {
    int noOfCarWashers, noOfCustomers;
    double openPeriod = 8 * 60, throughTime;
    Head tearoom = new Head(), waitingLine = new Head();
    Random random = new Random(7913);

    CarWashSimulation(int n) { noOfCarWashers = n; ... }

    class CarWasher extends Link {}
    class Car extends Link { double entryTime = time(); }

    class CarArrival extends Event {...}
    class StartCarWashing extends Event {...}
    class StopCarWashing extends Event {...}

    public static void main(String args[]) {
        new CarWashSimulation(2);
    }
}
```

The constructor in `CarWashSimulation`

```
CarWashSimulation(int n) {
    noOfCarWashers = n;
    for (int i = 1; i <= noOfCarWashers; i++)
        new CarWasher().into(tearoom);
    new CarArrival().schedule(0);
    runSimulation(openPeriod + 1000000);
    report();
}
```

CarArrival

```
class CarArrival extends Event {
    public void actions() {
        if (time() <= openPeriod) {
            new Car().into(waitingLine);
            if (!tearoom.empty())
                new StartCarWashing().schedule(time());
            new CarArrival().schedule(time() +
                                      random.negexp(1 / 11.0));
        }
    }
}
```

StartCarWashing

```
class StartCarWashing extends Event {
    public void actions() {
        CarWasher theCarWasher = (CarWasher) tearoom.first();
        theCarWasher.out();
        Car theCar = (Car) waitingLine.first();
        theCar.out();
        new StopCarWashing(theCarWasher, theCar).
            schedule(time() + 10);
    }
}
```

StopCarWashing

```
class StopCarWashing extends Event {
    CarWasher theCarWasher;
    Car theCar;

    StopCarWashing(CarWasher cw, Car c) {
        theCarWasher = cw; theCar = c;
    }

    public void actions() {
        theCarWasher.into(tearoom);
        if (!waitingLine.empty())
            new StartCarWashing().schedule(time());
        noOfCustomers++;
        throughTime += time() - theCar.entryTime;
    }
}
```

The method `report`

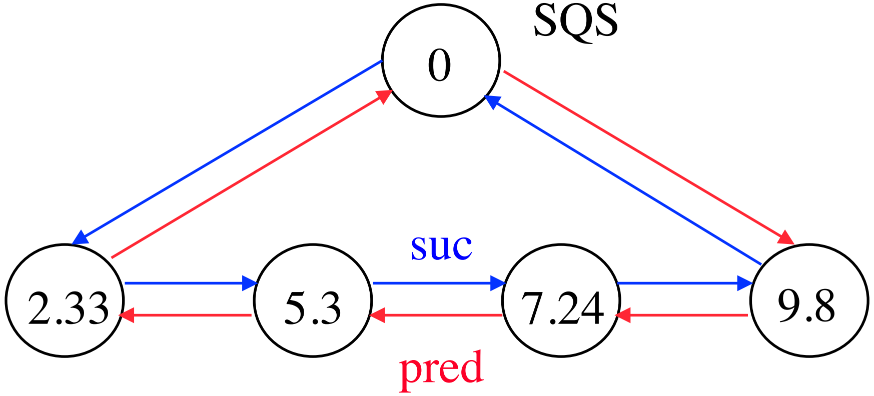
```
void report() {
    System.out.println(noOfCarWashers +
                       " car washer simulation");
    System.out.println("No.of cars through the system = " +
                       noOfCustomers);
    System.out.printf("Av.elapsed time = %1.2f\n",
                      throughTime / noOfCustomers);
}
```

Experimental results

```
1 car washer simulation
No.of cars through the system = 43
Av.elapsed time = 29.50

2 car washer simulation
No.of cars through the system = 43
Av.elapsed time = 12.46

3 car washer simulation
No.of cars through the system = 43
Av.elapsed time = 10.51
```

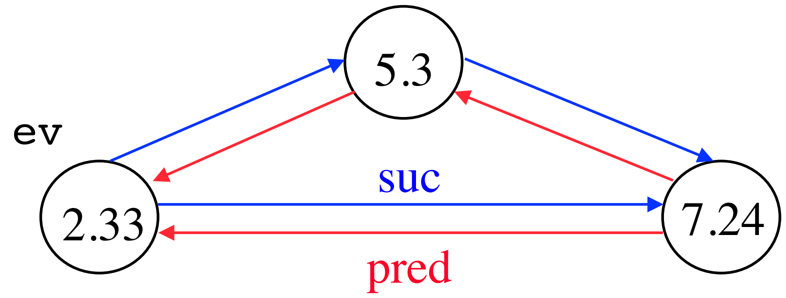
Implementation of the package `simulation.event`

Scheduled events are kept in a circular two-way list, SQS, sorted in increasing order of their associated event times.

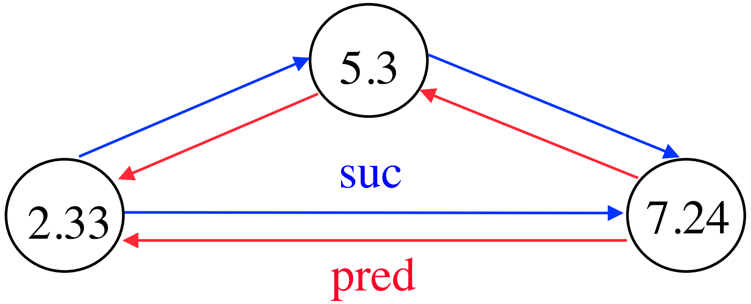


```
public abstract class Event {
    protected abstract void actions();
    ...
    private final static Event SQS = new Event() {
        { pred = suc = this; }
        protected void actions() {}
    };

    private static double time = 0;

    private double eventTime;
    private Event pred, suc;
}
```

```
public void schedule(final double evTime) {
    if (evTime < time)
        throw new RuntimeException
              ("attempt to schedule event in the past");
    cancel();
    eventTime = evTime;
    Event ev = SQS.pred;
    while (ev.eventTime > eventTime)
        ev = ev.pred;
    pred = ev;
    suc = ev.suc;
    ev.suc = suc.pred = this;
}
```



```
public void cancel() {
    if (suc != null) {
        suc.pred = pred;
        pred.suc = suc;
        suc = pred = null;
    }
}
```

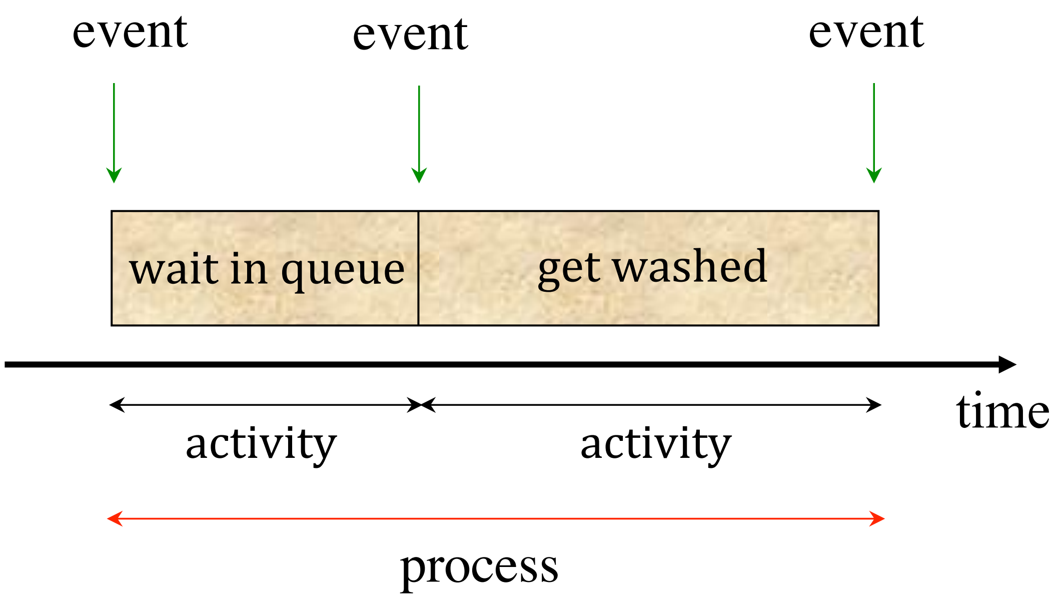


```
public static void runSimulation(double period) {
    time = 0;
    while (SQS.suc != SQS) {
        Event ev = SQS.suc;
        time = ev.eventTime;
        if (time > period)
            break;
        ev.cancel();
        ev.actions();
    }
    stopSimulation();
}
```

```
public static void stopSimulation() {
    while (SQS.suc != SQS)
        SQS.suc.cancel();
}
```

Process-based simulation

A **process** is a system component that executes a sequence of activities in simulated time.



Identification of processes

(1) `Car`

(2) `CarWasher`

(3) `CarGenerator`

The package `javaSimulation`

```
public abstract class Process extends Link {
   protected abstract void actions();

   public static double time();
   public static void activate(Process p);
   public static void hold(double t);
   public static void passivate();
   public static void wait(Head q);
}
```

Processes and their associated actions are defined in subclasses of class `Process`.

```
import javaSimulation.*;
import javaSimulation.Process;
```

```
public class CarWashSimulation extends Process {
    int noOfCarWashers, noOfCustomers;
    double openPeriod = 8 * 60, throughTime;
    Head tearoom = new Head(), waitingLine = new Head();
    Random random = new Random(7913);

    CarWashSimulation(int n) { noOfCarWashers = n; }

    public void actions() {...}

    class Car extends Process {...}
    class CarWasher extends Process {...}
    class CarGenerator extends Process {...}

    public static void main(String args[]) {
        activate(new CarWashSimulation(2));
    }
}
```

The actions of the main process

```
public void actions() {
    for (int i = 1; i <= noOfCarWashers; i++)
        new CarWasher().into(tearoom);
    activate(new CarGenerator());
    hold(openPeriod + 1000000);
    report();
}
```

Class CarGenerator

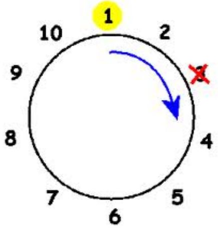
```
class CarGenerator extends Process {
    public void actions() {
        while (time() <= openPeriod) {
            activate(new Car());
            hold(random.negexp(1 / 11.0));
        }
    }
}
```

Class Car

```
class Car extends Process {
    public void actions() {
        double entryTime = time();
        into(waitingLine);
        if (!tearoom.empty())
            activate((CarWasher) tearoom.first());
        passivate();
        noOfCustomers++;
        throughTime += time() - entryTime;
    }
}
```

Class CarWasher

```
class CarWasher extends Process {
    public void actions() {
        while (true) {
            out();
            while (!waitingLine.empty()) {
                Car served = (Car) waitingLine.first();
                served.out();
                hold(10);
                activate(served);
            }
            wait(tearoom);
        }
    }
}
```

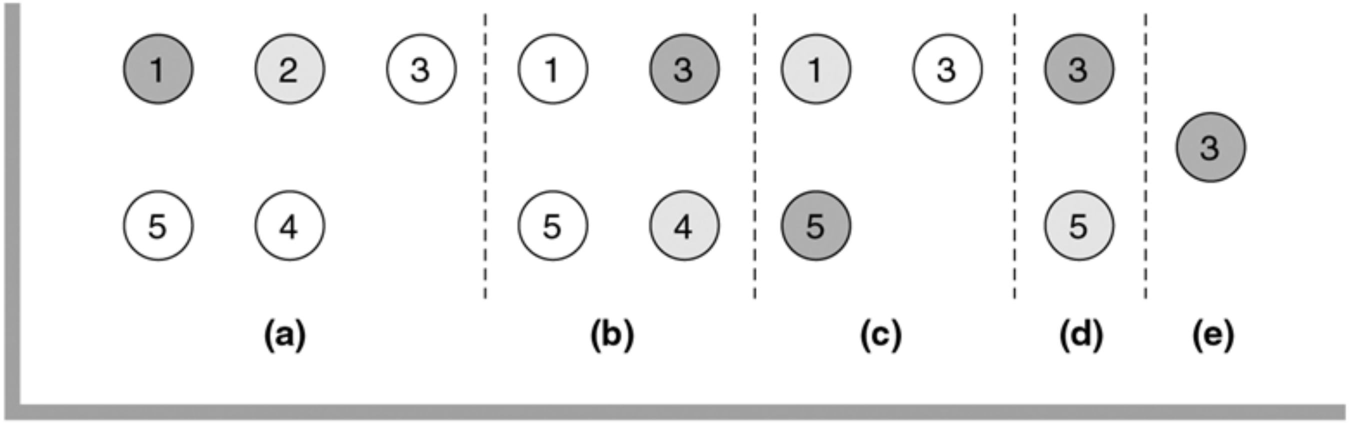


The Josephus problem

N people are sitting in a circle waiting to be eliminated. Starting at person 1, a hot potato is passed; after *M* passes, the person holding the potato is eliminated. The game continues with the person who was sitting after the eliminated person picking up the potato. This continues until only the last person remains.

figure 13.1

The Josephus problem: At each step, the darkest circle represents the initial holder and the lightly shaded circle represents the player who receives the hot potato (and is eliminated). Passes are made clockwise.



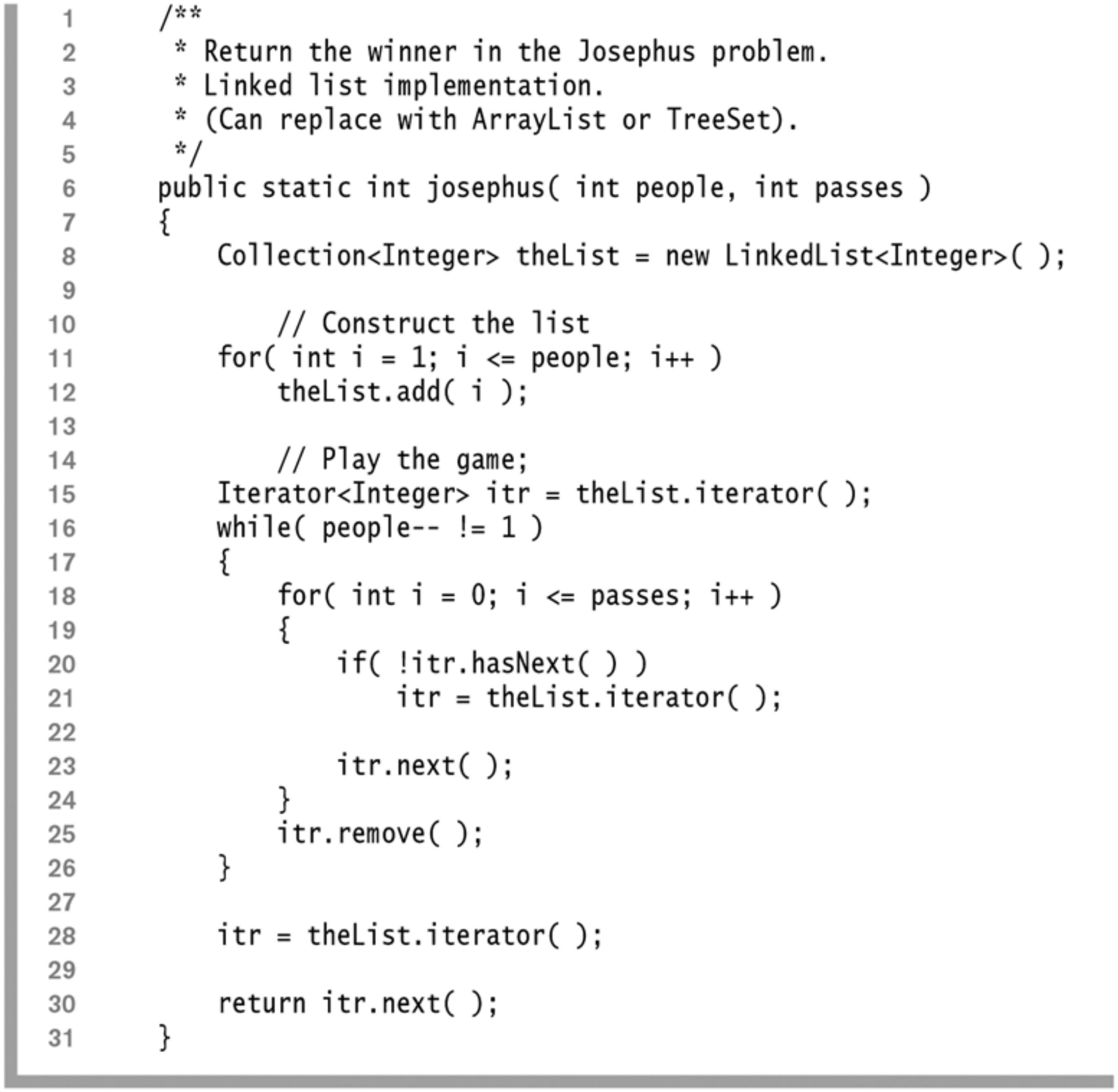
$N = 5, M = 1$

An $O(NM)$ solution

figure 13.2

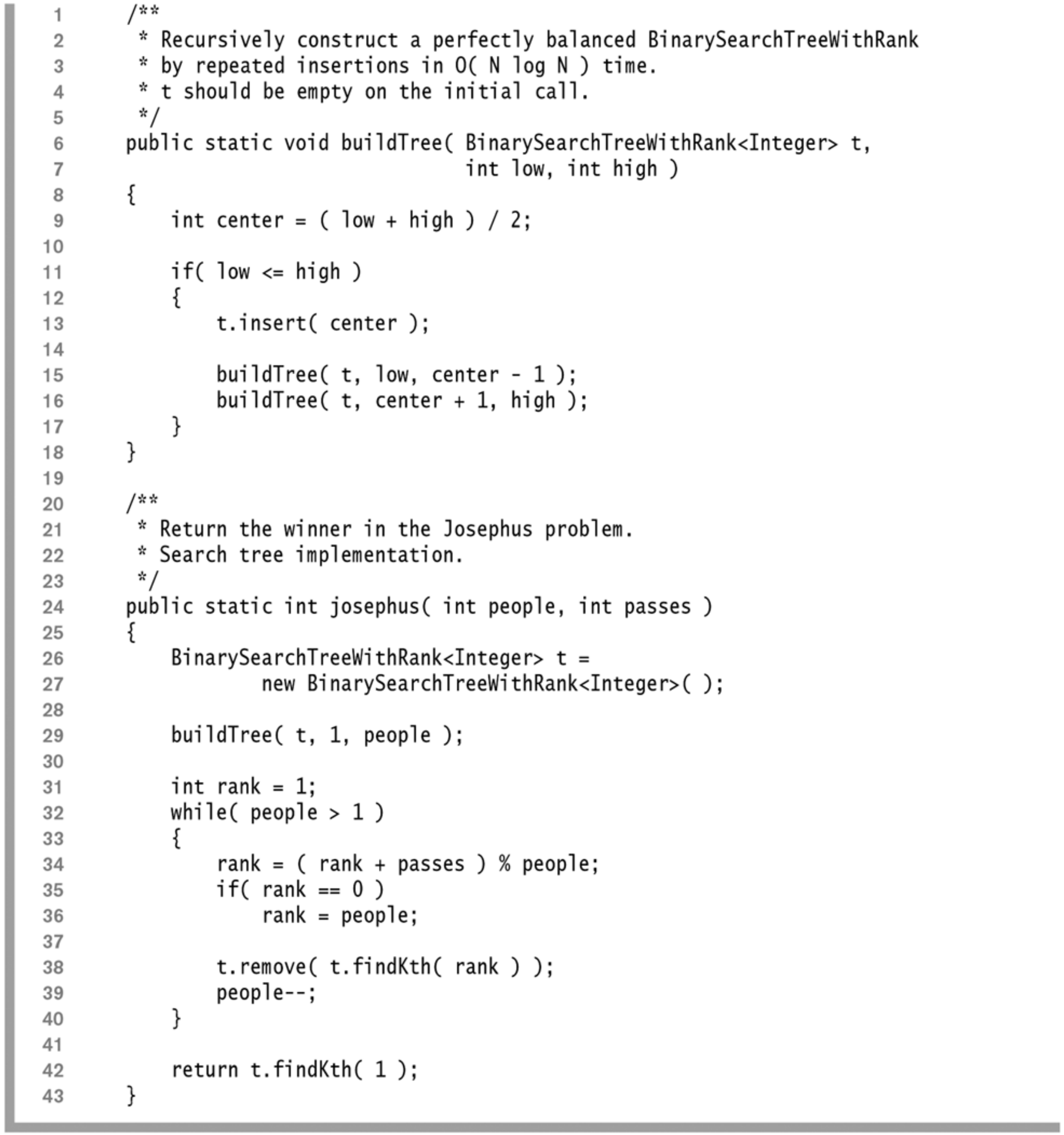
Linked list implementation of the Josephus problem

`people`: N
`passes`: M



```java
/**
 * Return the winner in the Josephus problem.
 * Linked list implementation.
 * (Can replace with ArrayList or TreeSet).
 */
public static int josephus( int people, int passes )
{
    Collection<Integer> theList = new LinkedList<Integer>( );

        // Construct the list
    for( int i = 1; i <= people; i++ )
        theList.add( i );

        // Play the game;
    Iterator<Integer> itr = theList.iterator( );
    while( people-- != 1 )
    {
        for( int i = 0; i <= passes; i++ )
        {
            if( !itr.hasNext( ) )
                itr = theList.iterator( );

            itr.next( );
        }
        itr.remove( );
    }

    itr = theList.iterator( );

    return itr.next( );
}
```

An *O*(*N* log*N*) solution



```
/**
 * Recursively construct a perfectly balanced BinarySearchTreeWithRank
 * by repeated insertions in O( N log N ) time.
 * t should be empty on the initial call.
 */
public static void buildTree( BinarySearchTreeWithRank<Integer> t,
                              int low, int high )
{
    int center = ( low + high ) / 2;

    if( low <= high )
    {
        t.insert( center );

        buildTree( t, low, center - 1 );
        buildTree( t, center + 1, high );
    }
}

/**
 * Return the winner in the Josephus problem.
 * Search tree implementation.
 */
public static int josephus( int people, int passes )
{
    BinarySearchTreeWithRank<Integer> t =
            new BinarySearchTreeWithRank<Integer>( );

    buildTree( t, 1, people );

    int rank = 1;
    while( people > 1 )
    {
        rank = ( rank + passes ) % people;
        if( rank == 0 )
            rank = people;

        t.remove( t.findKth( rank ) );
        people--;
    }

    return t.findKth( 1 );
}
```

figure 13.3

An $O(N \log N)$ solution of the Josephus problem

A call bank simulation

A call bank consists of a large number of operators who handle phone calls. An operator is reached by dialing one phone number.

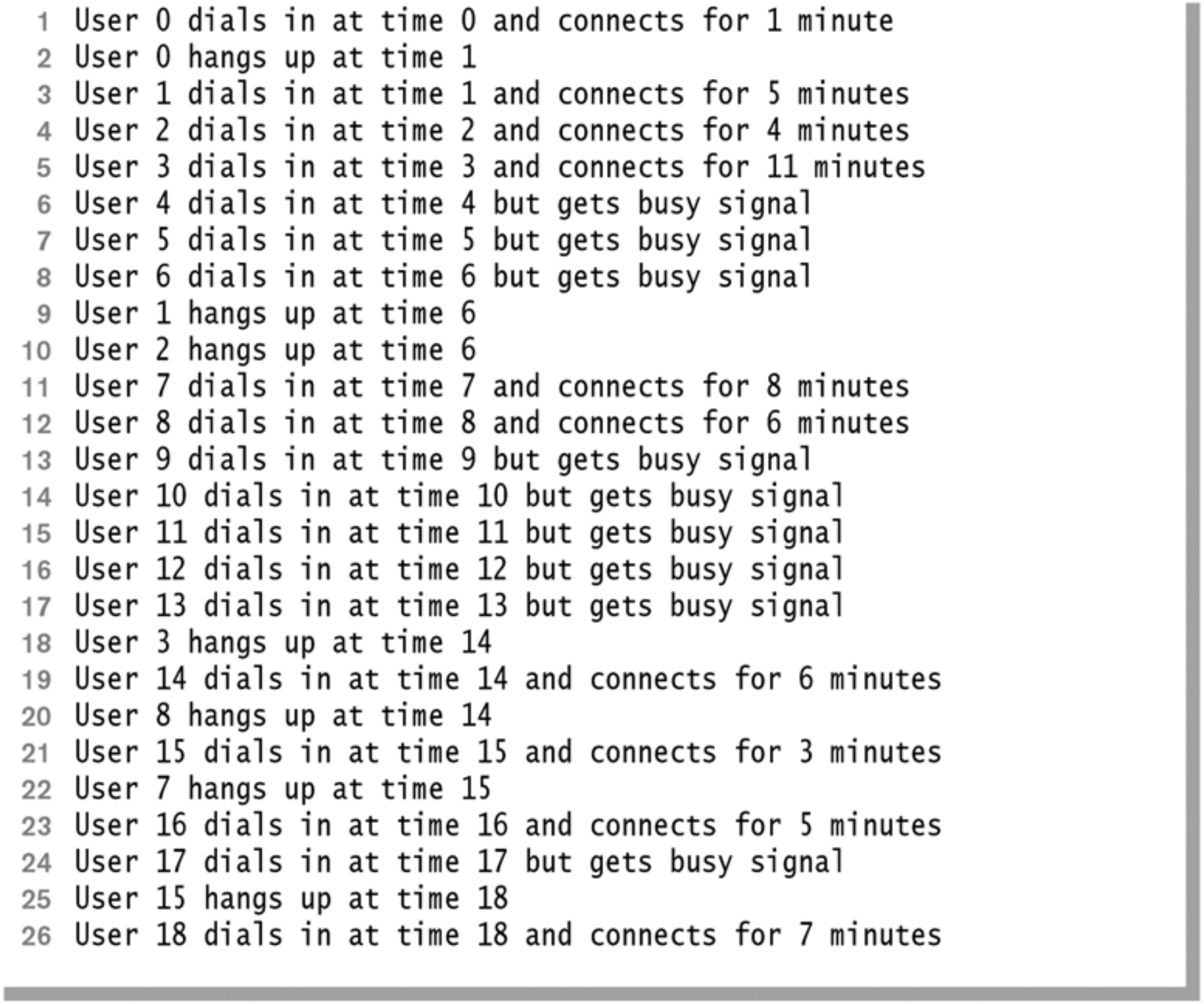
If any of the operators are available, the user is connected to one of them.

If all operators are already taking a phone, the phone will give a busy signal, and the user will hang up.

Simulate the service provided by the pool of operators. The variables are

- The number of operators in the bank
- The probability distribution that governs dial-in attempts
- The probability distribution that governs connect time
- The length of time the simulation is to be run

Sample output



```
User 0 dials in at time 0 and connects for 1 minute
User 0 hangs up at time 1
User 1 dials in at time 1 and connects for 5 minutes
User 2 dials in at time 2 and connects for 4 minutes
User 3 dials in at time 3 and connects for 11 minutes
User 4 dials in at time 4 but gets busy signal
User 5 dials in at time 5 but gets busy signal
User 6 dials in at time 6 but gets busy signal
User 1 hangs up at time 6
User 2 hangs up at time 6
User 7 dials in at time 7 and connects for 8 minutes
User 8 dials in at time 8 and connects for 6 minutes
User 9 dials in at time 9 but gets busy signal
User 10 dials in at time 10 but gets busy signal
User 11 dials in at time 11 but gets busy signal
User 12 dials in at time 12 but gets busy signal
User 13 dials in at time 13 but gets busy signal
User 3 hangs up at time 14
User 14 dials in at time 14 and connects for 6 minutes
User 8 hangs up at time 14
User 15 dials in at time 15 and connects for 3 minutes
User 7 hangs up at time 15
User 16 dials in at time 16 and connects for 5 minutes
User 17 dials in at time 17 but gets busy signal
User 15 hangs up at time 18
User 18 dials in at time 18 and connects for 7 minutes
```

figure 13.4

Sample output for the modem bank simulation involving three modems: A dial-in is attempted every minute; the average connect time is 5 minutes; and the simulation is run for 18 minutes

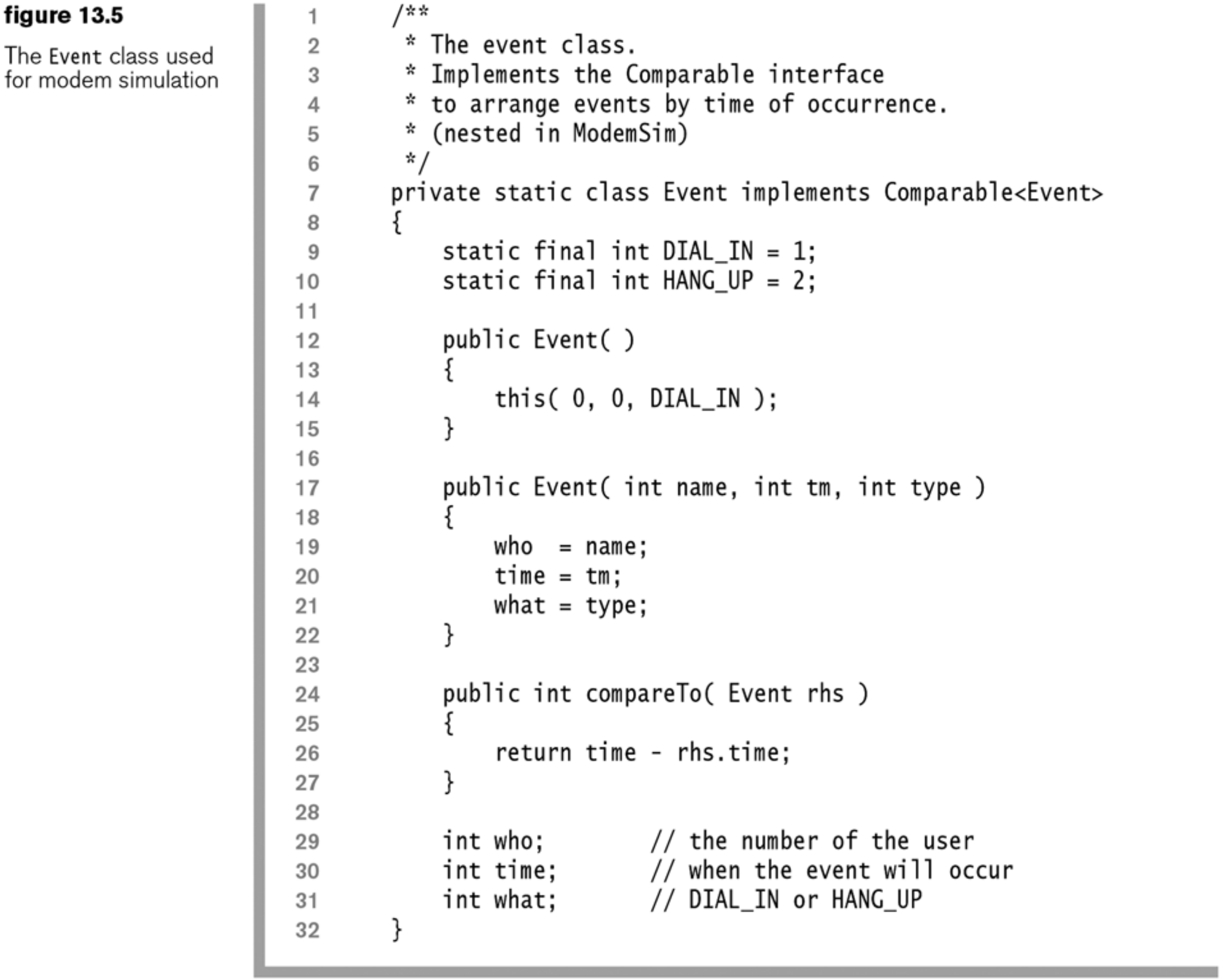
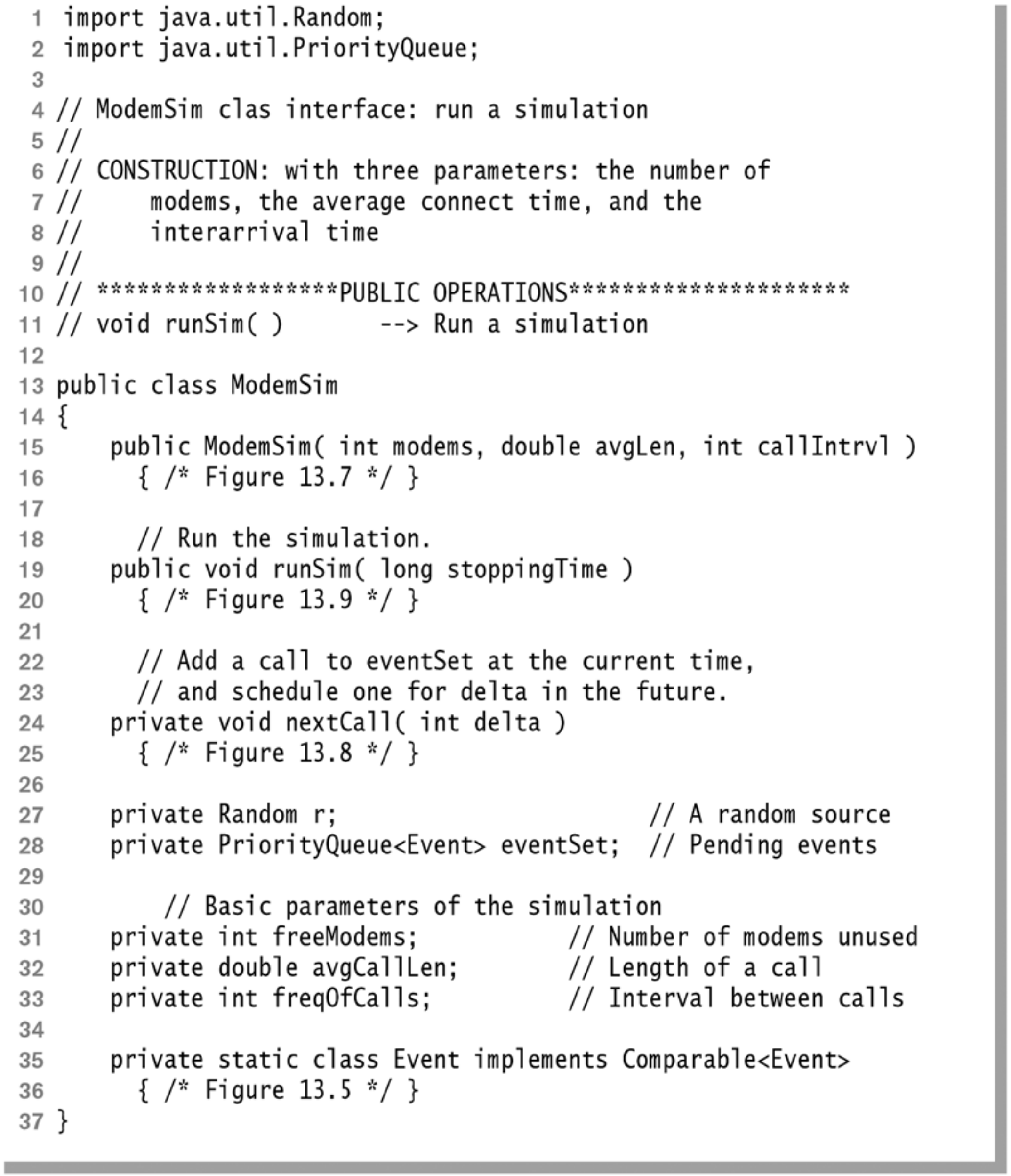


figure 13.5

The Event class used for modem simulation

```
/**
 * The event class.
 * Implements the Comparable interface
 * to arrange events by time of occurrence.
 * (nested in ModemSim)
 */
private static class Event implements Comparable<Event>
{
    static final int DIAL_IN = 1;
    static final int HANG_UP = 2;

    public Event( )
    {
        this( 0, 0, DIAL_IN );
    }

    public Event( int name, int tm, int type )
    {
        who  = name;
        time = tm;
        what = type;
    }

    public int compareTo( Event rhs )
    {
        return time - rhs.time;
    }

    int who;        // the number of the user
    int time;       // when the event will occur
    int what;       // DIAL_IN or HANG_UP
}
```



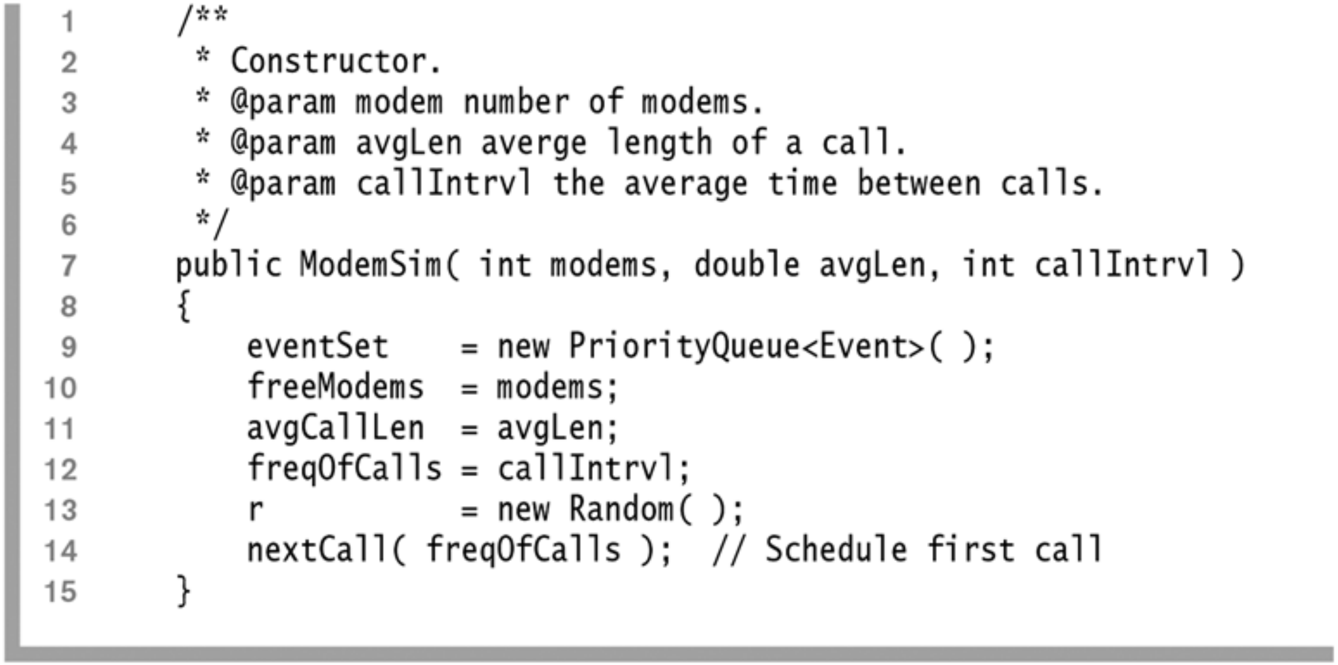
```java
import java.util.Random;
import java.util.PriorityQueue;

// ModemSim clas interface: run a simulation
//
// CONSTRUCTION: with three parameters: the number of
//     modems, the average connect time, and the
//     interarrival time
//
// ******************PUBLIC OPERATIONS*********************
// void runSim( )       --> Run a simulation

public class ModemSim
{
    public ModemSim( int modems, double avgLen, int callIntrvl )
      { /* Figure 13.7 */ }

      // Run the simulation.
    public void runSim( long stoppingTime )
      { /* Figure 13.9 */ }

      // Add a call to eventSet at the current time,
      // and schedule one for delta in the future.
    private void nextCall( int delta )
      { /* Figure 13.8 */ }

    private Random r;                       // A random source
    private PriorityQueue<Event> eventSet;  // Pending events

        // Basic parameters of the simulation
    private int freeModems;           // Number of modems unused
    private double avgCallLen;        // Length of a call
    private int freqOfCalls;          // Interval between calls

    private static class Event implements Comparable<Event>
      { /* Figure 13.5 */ }
}
```

figure 13.6

The `ModemSim` class skeleton

figure 13.7

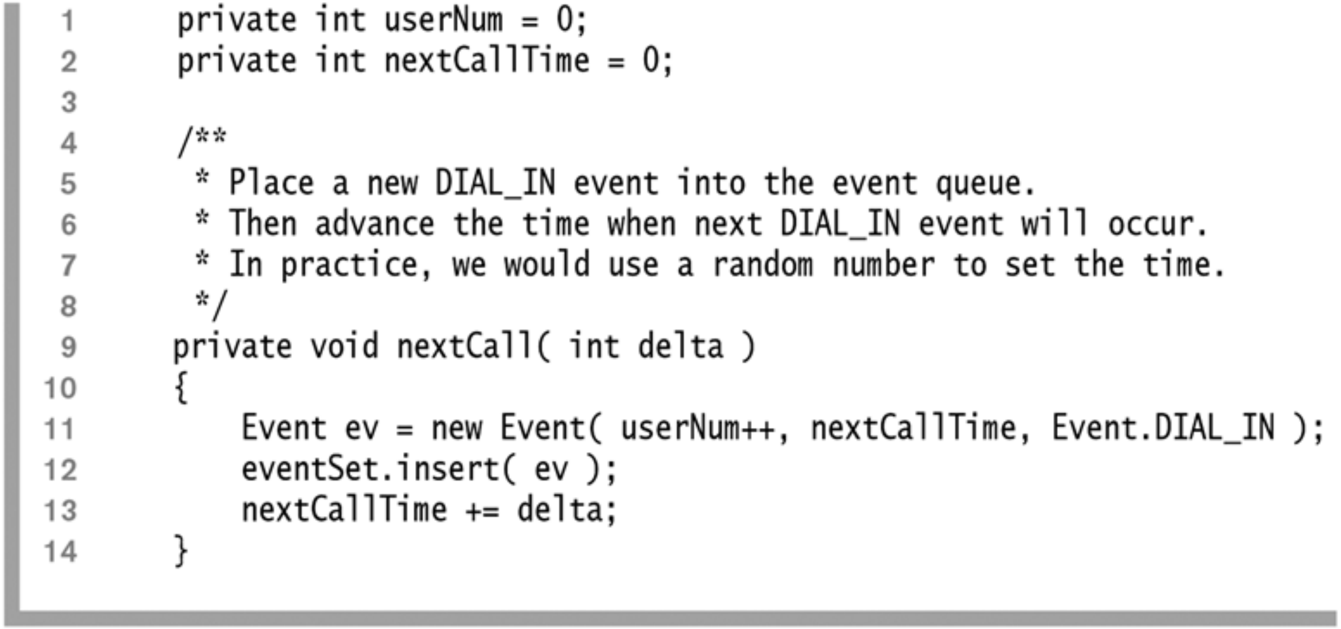
The `ModemSim` constructor



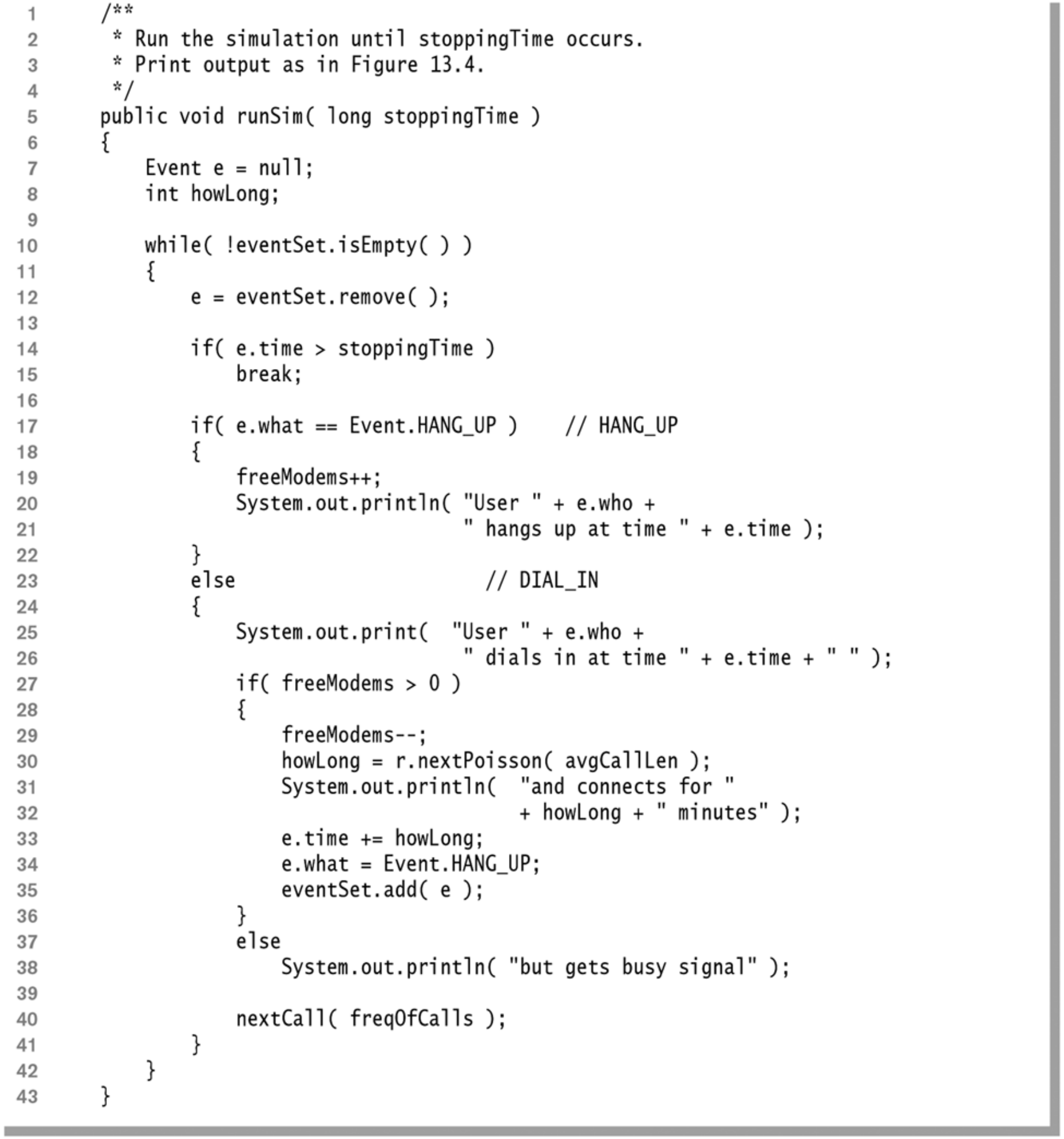
```
/**
 * Constructor.
 * @param modem number of modems.
 * @param avgLen averge length of a call.
 * @param callIntrvl the average time between calls.
 */
public ModemSim( int modems, double avgLen, int callIntrvl )
{
    eventSet    = new PriorityQueue<Event>( );
    freeModems  = modems;
    avgCallLen  = avgLen;
    freqOfCalls = callIntrvl;
    r           = new Random( );
    nextCall( freqOfCalls );  // Schedule first call
}
```

figure 13.8

The nextCall method places a new DIAL_IN event in the event queue and advances the time when the next DIAL_IN event will occur



```
    private int userNum = 0;
    private int nextCallTime = 0;

    /**
     * Place a new DIAL_IN event into the event queue.
     * Then advance the time when next DIAL_IN event will occur.
     * In practice, we would use a random number to set the time.
     */
    private void nextCall( int delta )
    {
        Event ev = new Event( userNum++, nextCallTime, Event.DIAL_IN );
        eventSet.insert( ev );
        nextCallTime += delta;
    }
```



```java
/**
 * Run the simulation until stoppingTime occurs.
 * Print output as in Figure 13.4.
 */
public void runSim( long stoppingTime )
{
    Event e = null;
    int howLong;

    while( !eventSet.isEmpty( ) )
    {
        e = eventSet.remove( );

        if( e.time > stoppingTime )
            break;

        if( e.what == Event.HANG_UP )    // HANG_UP
        {
            freeModems++;
            System.out.println( "User " + e.who +
                                " hangs up at time " + e.time );
        }
        else                      // DIAL_IN
        {
            System.out.print(  "User " + e.who +
                                " dials in at time " + e.time + " " );
            if( freeModems > 0 )
            {
                freeModems--;
                howLong = r.nextPoisson( avgCallLen );
                System.out.println(  "and connects for "
                                     + howLong + " minutes" );
                e.time += howLong;
                e.what = Event.HANG_UP;
                eventSet.add( e );
            }
            else
                System.out.println( "but gets busy signal" );

            nextCall( freqOfCalls );
        }
    }
}
```

figure 13.9

The basic simulation routine

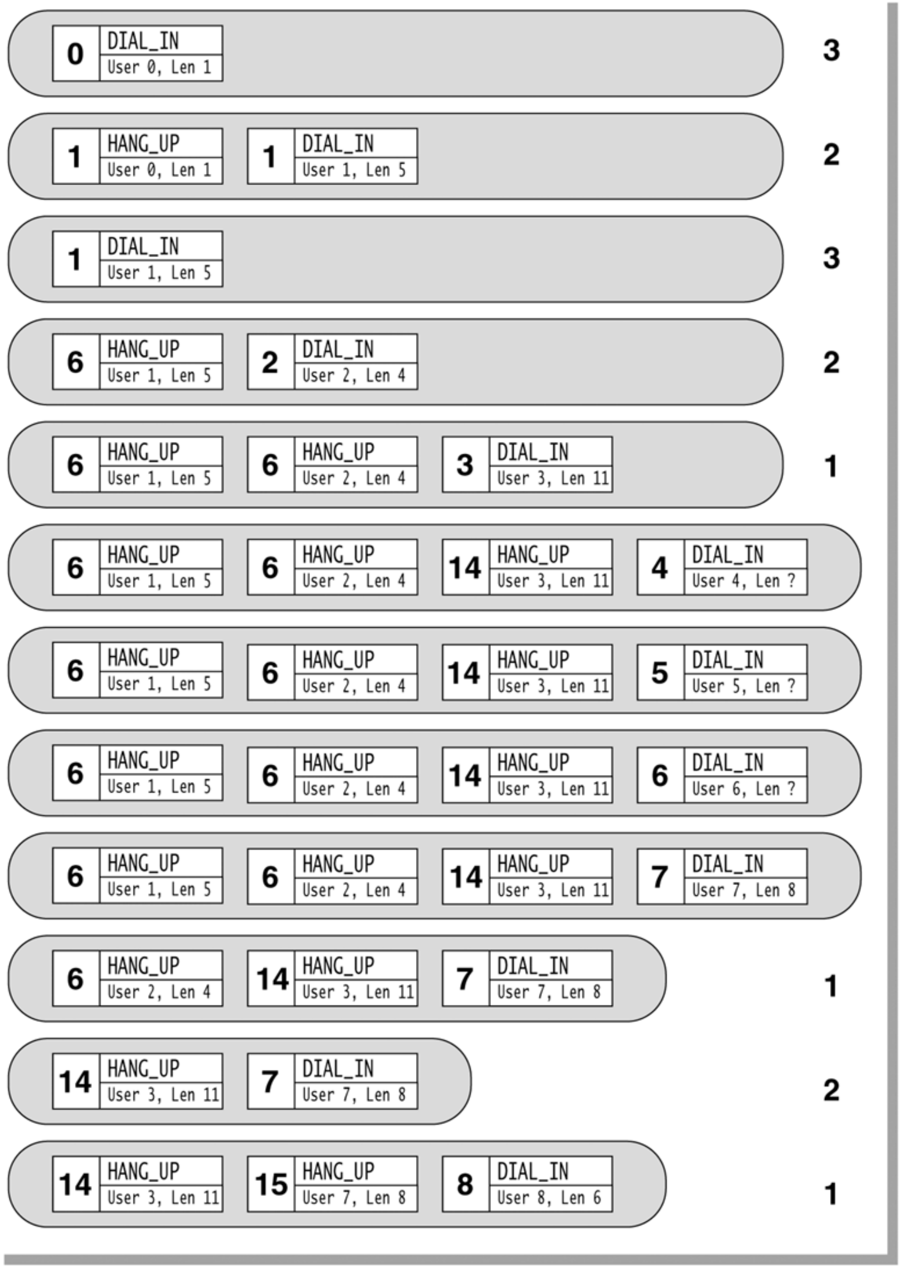


figure 13.10

The priority queue for modem bank simulation after each step

The time at which each event occurs is shown in boldface.

The number of free operators (if any) are shown to the right of the priority queue.

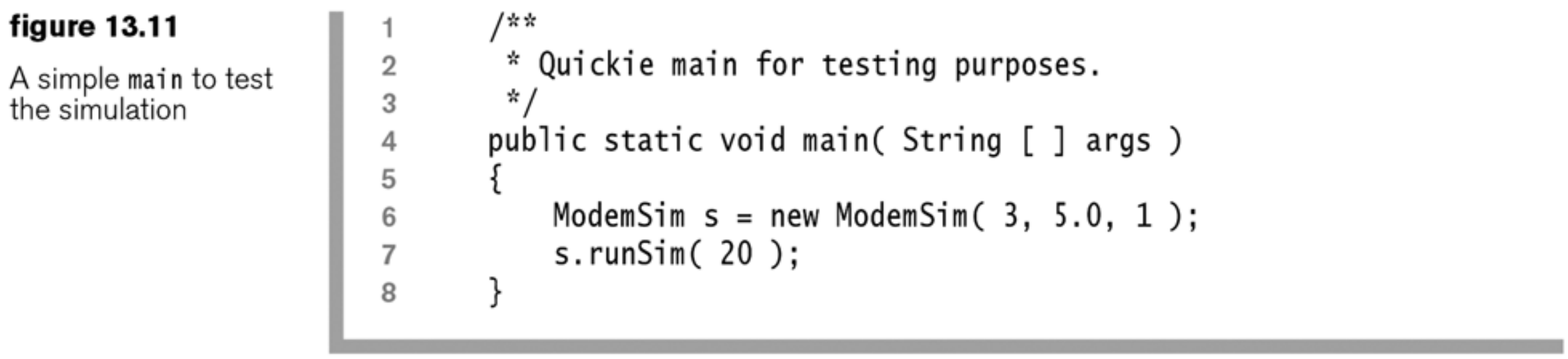


figure 13.11

A simple `main` to test the simulation

```java
/**
 * Quickie main for testing purposes.
 */
public static void main( String [ ] args )
{
    ModemSim s = new ModemSim( 3, 5.0, 1 );
    s.runSim( 20 );
}
```

Using simulation.event

```
public class CallSim extends Simulation {
    public CallSim(int operators, double avgLen,
                   int callIntrvl) {
        availableOperators = operators;
        avgCallLen = avgLen;
        freqOfCalls = callIntrvl;
    }

    class DialIn extends Event { ... }
    class HangUp extends Event { ... }

    public static void main(String[] args) {
        new CallSim(3, 5.0, 1);
        new DialIn(0).schedule(0.0);
        runSimulation(20);
    }

    int availableOperators, freqOfCalls;
    double avgCallLen;
    Random r = new Random();
}
```

```
class DialIn extends Event {
    DialIn(int who) { this.who = who; }

    @Override public void actions() {
        System.out.print("User " + who +
                         " dials in at time " + time() + " ");
        if (availableOperators > 0) {
            availableOperators--;
            int howLong = r.poisson(avgCallLen);
            System.out.println("and connects for " +
                               howLong + " minutes");
            new HangUp(who).schedule(time() + howLong);
        } else
            System.out.println("but gets busy signal");
        new DialIn(who + 1).schedule(time() + freqOfCalls);
    }

    int who;
}
```

```
class HangUp extends Event {
    HangUp(int who) { this.who = who; }

    @Override public void actions() {
        availableOperators++;
        System.out.println("User " + who +
                           " hangs up at time " + time());
    }

    int who;
}
```

Using `javaSimulation`

```
public class CallSim extends Process {
    public CallSim(int operators, double avgLen,
                   int callIntrvl, int stopTime) {
        availableOperators = operators; avgCallLen = avgLen;
        freqOfCalls = callIntrvl; simTime = stopTime;
    }

    @Override public void actions() {
        activate(new User(0));
        hold(simTime);
    }

    class User extends Process { ... }

    public static void main(String[] args) {
        activate(new CallSim(3, 5.0, 1, 20));
    }

    int availableOperators, freqOfCalls, simTime;
    double avgCallLen;
    Random r = new Random();
}
```

```
class User extends Process {
    User(int who) { this.who = who; }

    @Override public void actions() {
        activate(new User(who + 1), delay, freqOfCalls);
        System.out.print("User " + who +
                         " dials in at time " + time() + " ");
        if (availableOperators > 0) {
            availableOperators--;
            int howLong = r.poisson(avgCallLen);
            System.out.println("and connects for " +
                               howLong + " minutes");
            hold(howLong);
            availableOperators++;
            System.out.println("User " + who +
                               " hangs up at time " + time());
        } else
            System.out.println("but gets busy signal");
    }

    int who;
}
```