

# **Programming In Java**

**BOOK 4**

## **Streams and Threads**

## CONTENTS

<b>1. STREAMS</b>	<b>3</b>
1.1 Writing and Reading Java Primitive Types (the scalars)	3
1.1.1 Writing Scalars to a File	3
1.1.2 Reading Scalars from a File	3
1.2 Example 1 - Handling More Than One Exception	4
1.3 Reading and Writing Text	5
1.3.1 Reading Text	5
1.3.2 Writing Text	5
1.4 Example 2	6
1.5 Where to Handle (catch) the Exceptions	7
1.5.1 Catching The Exception in the Calling Method	7
1.5.2 Catching The Exception in the Method Where the Exception Occurs	8
1.6 Example 3	8
<b>2. THE PROPERTIES CLASS</b>	<b>9</b>
2.1 Example	9
<b>3. THE STREAMTOKENIZER CLASS</b>	<b>11</b>
<b>4. SERIALISATION AND PERSISTENT OBJECTS</b>	<b>12</b>
4.1 Serialization of Objects (Saving Objects)	12
4.2 Transient instance variables	13
4.3 A Persistent Object Example	13
<b>5. THREADS</b>	<b>15</b>
5.1 Creating Threads	15
5.1.1 Example 1 - Inheriting From the Thread Class	15
5.1.2 Example 2 - Implementing the Runnable Interface	16
5.2 Inheritance vs Runnable	17
5.2.1 Example - Thread vs Runnable	17
5.3 Shared Object - Synchronization	18
5.3.1 Example 4	19
5.3.2 What Happens if One Synchronised Method calls another Synchronised Method ?	20
5.3.3 What happens if one Synchronised Method Depends on Another Synchronised Method ?	20
5.4 The Readers and Writers Problem	21
5.4.1 Example 5 - Possible System lock	21
5.4.2 Example 6 - A Solution Using the wait() and notify() methods	23
5.4.3 Example 7 - Multiple Writers and Readers	25
5.5 TheYield() Method	27
5.5.1 Example 8	27
5.6 Thread Priorities	28
<b>6. SOME BITS AND PIECES</b>	<b>29</b>
6.1 Copying Objects	29
6.2 public class Vector extends Object implements Cloneable	29
6.2.1 Shallow Copy	31
6.2.2 Deep Copy	32
6.2.3 The Cloneable Interface	33
6.2.3.1 Example 1	33
6.2.3.2 Example 2	34
6.3 Broadcasting Mechanisms - Observable Class and Observer Interface	35

# 1. STREAMS

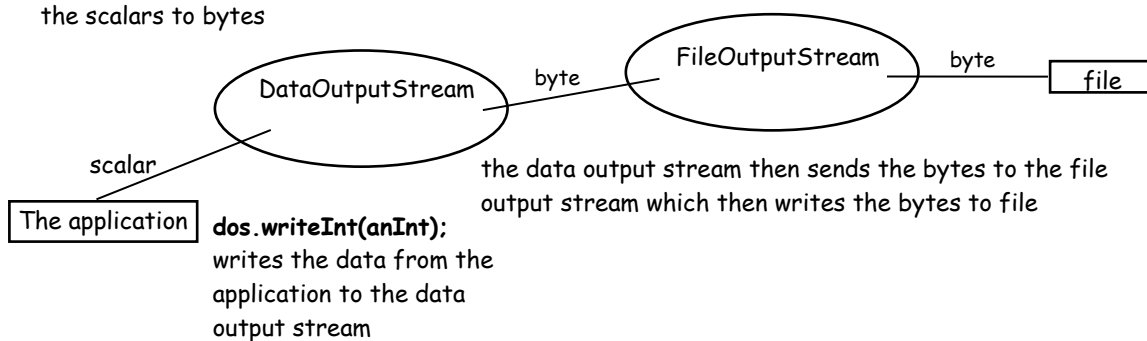
Java input and output is based on streams. A stream is a sequence of bytes moving from a *source* to a *destination*. If an application is writing to a stream then the *application is the source*. If it is reading from a stream then it *is the destination*.

## 1.1 Writing and Reading Java Primitive Types (the scalars)

### 1.1.1 Writing Scalars to a File

A `DataOutputStream` object converts the scalars to bytes

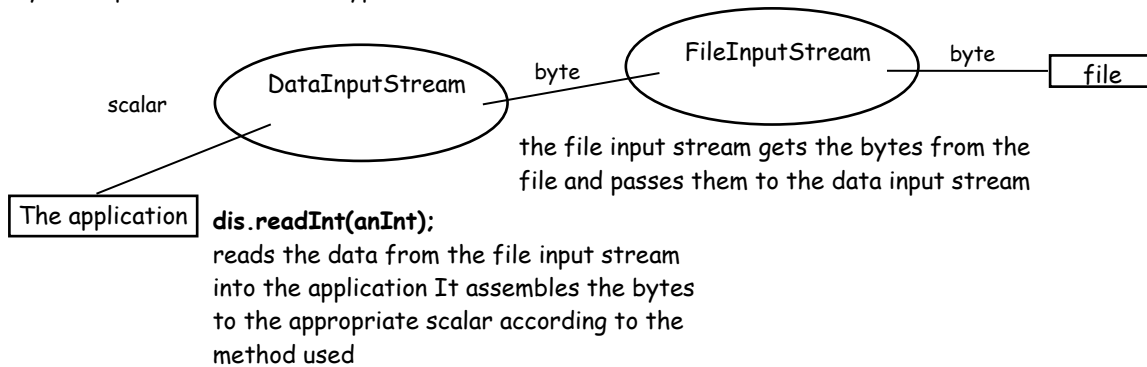
A `FileOutputStream` object is an output stream for writing bytes to a File.



### 1.1.2 Reading Scalars from a File

A `DataInputStream` object converts the bytes to primitive Java data types

A `FileInputStream` object is a byte stream for reading bytes from a File.



Some of the more useful applications of exceptions can be found in stream/file processing.

## 1.2 Example 1 - Handling More Than One Exception

To catch more than one exception the catch statement can be stacked up with each catch handling a different exception.

```
import java.io.*;
public class FileReader {
    private FileInputStream fis;
    private DataInputStream dis;
    private FileOutputStream fos;
    private DataOutputStream dos;
    public void writeData(String theFileName) {
        int anInt = 100;
        try {
            fos = new FileOutputStream(theFileName);
            dos = new DataOutputStream(fos);
            for (int i = 0; i < 5; ++i) {
                dos.writeInt(anInt);
                anInt += 100;
            }
            dos.close();
        }
        catch (IOException e) {
            System.out.println("Problem with writing...");
        }
    }
}
```

DataOutputStream and DataInputStream objects are used to write/read Java primitive types - scalars

### First create a FileOutPutStream object.

A FileOutputStream object is an output stream for writing data to a File.

Then create a DataOutputStream with the FileOutPutStream object as the constructor parameter, so that the dos knows which fos it is working with..

A DataOutputStream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a DataInputStream to read the data back in.

```
public void readData(String theFileName) {
    int anInt;
    try {
        fis = new FileInputStream(theFileName);
        dis = new DataInputStream(fis);
        anInt = dis.readInt();
        while (1 == 1) {
            System.out.println(anInt);
            anInt = dis.readInt();
        }
    }
    catch (FileNotFoundException e) {
        System.out.println("File not found...");
    }
    catch (EOFException e) {
        System.out.println("EOF reached...");
        try {
            dis.close();
        }
        catch (IOException ee) {}
    }
    catch (IOException e) {
        System.out.println("Problem with reading...");
    }
}
```

### First create a FileInPutStream object.

A FileInputStream object is an input stream for reading data from a File.

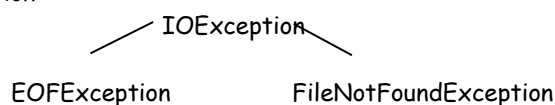
Then create a DataInputStream with the FileInPutStream object as the constructor parameter, so that the dis knows which fis it is working with.

A DataInputStream lets an application read primitive Java data types from an input stream in a portable way.

readInt() throws an EOFException when the end of file is reached.

We use this to exit the reading loop. We need the try/catch block since close() may throw an IOException.

When you stack a number of exceptions you have to be aware of any possible inheritance relationships. The order of the three exceptions is dictated by the fact that IOException is superclass to both EOFException and FileNotFoundException



```

public class FileReaderTest {
    public static void main(String arg[]) {
        FileReader f = new FileReader();
        f.writeData("IntFile.txt");
        f.readData("IntFile.txt");
        System.out.println("End of main()...");
    }
}

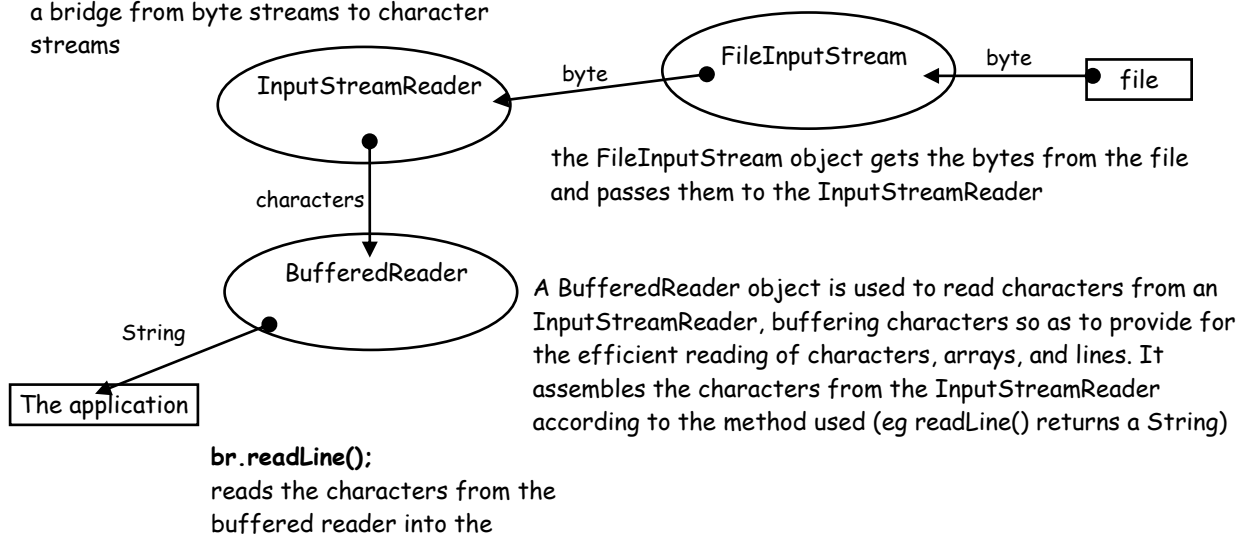
```

## 1.3 Reading and Writing Text

### 1.3.1 Reading Text

A `InputStreamReader` object converts the bytes to characters. An `InputStreamReader` is a bridge from byte streams to character streams

A `FileInputStream` object is a byte input stream for reading bytes from a File.



```

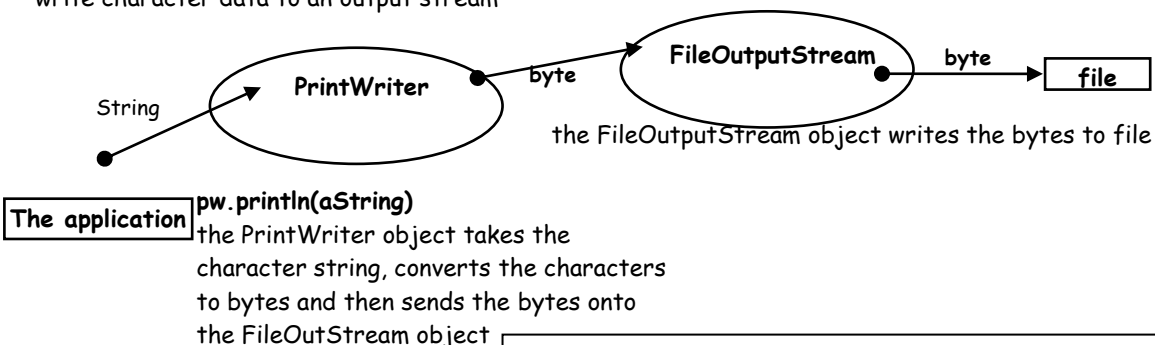
FileInputStream fis = new FileInputStream(theFileName);
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);

```

### 1.3.2 Writing Text

A `PrintWriter` object allows an application to write character data to an output stream

A `FileOutputStream` object is a byte output stream for writing bytes to a File.



```

FileOutputStream fos = new FileOutputStream(theFileName);
PrintWriter pw = new PrintWriter(fos);

```

The `FileOutputStream` object needs to know about the file and the `PrintWriter` object needs to know about the `FileOutputStream`. It is passed a reference to it.

## 1.4 Example 2

```
import java.io.*;
public class TextReaderWriter {
    private String[] theBuffer;
    private int bufferCount;

    public TextReaderWriter(int n) {
        theBuffer = new String[n];
        bufferCount = 0;
    }

    public void readFile(String theFileName) {
        FileInputStream fis;
        InputStreamReader isr;
        BufferedReader br;
        String aLine;
        try {
            fis = new FileInputStream(theFileName);
            isr = new InputStreamReader(fis);
            br = new BufferedReader(isr);
            while ((aLine = br.readLine()) != null) {
                theBuffer[bufferCount++] = aLine;
                System.out.println(aLine);
            }
            br.close();
            fis.close();
        }
        catch (FileNotFoundException e) {
            System.out.println("File not found...");
        }
        catch (IOException e) {
            System.out.println("Problem with reading...");
        }
    }

    public void writeFile(String theFileName) {
        FileOutputStream fos;
        PrintWriter pw;
        try {
            fos = new FileOutputStream(theFileName);
            pw = new PrintWriter(fos);
            for (int i = 0; i < bufferCount; ++i) {
                pw.println(theBuffer[i].toUpperCase());
                System.out.println(theBuffer[i].toUpperCase());
            }
            pw.close();
            fos.close();
        }
        catch (FileNotFoundException e) {
            System.out.println("File not found...");
        }
        catch (IOException e) {
            System.out.println("Problem with reading...");
        }
    }
}
```

The `FileInputStream` object (`fis`) is byte stream  
 An `InputStreamReader` is a bridge from byte streams to character streams: It reads bytes and translates them into characters according to a specified character encoding. The encoding that it uses may be specified by name, or the platform's default encoding may be accepted.

Make sure you get these in the right order, otherwise if you close the `fos` first the `pw` will have no where to buffer to as it closes and you will loose data.

```

public class TextReaderWriterTest {
    public static void main(String arg[]) {
        TextReaderWriter f = new TextReaderWriter(200);
        f.readFile("raven.txt");
        f.writeFile("newRaven.txt");
        System.out.println("End of main()...");
    }
}

```

## 1.5 Where to Handle (catch) the Exceptions

### 1.5.1 Catching The Exception in the Calling Method

This way the method announces that it may throw a `FileNotFoundException` or `IOException`. The thrown object is handled (caught) by the method that calls this method.

```

import java.io.*;
public class AccountClass {
    private String theFile;

    public AccountClass(String aFileName) throws FileNotFoundException, IOException {
        String accStr;
        FileInputStream fis;
        BufferedReader br;
        theFile = aFileName;
        fis = new FileInputStream(theFile);
        br = new BufferedReader(new InputStreamReader(fis));
        while ((accStr = br.readLine()) != null) {
            System.out.println(accStr);
        }
        br.close();
    }
}

import java.io.*;
public class ExceptionDemo {
    public static void main(String args[]) {
        try {
            AccountClass anAccount = new AccountClass("Accounts.txtt");
        }
        catch (FileNotFoundException e) {
            System.out.println("Account file not found...");
        }
        catch (IOException e) {
            System.out.println("Read error in account file...");
        }
        System.out.println("End of main()...");
    }
}

```

### 1.5.2 Catching The Exception in the Method Where the Exception Occurs

This way the method itself handles the `FileNotFoundException` or `IOException`. The previous approach is generally to be preferred since the way in which the error is handled is probably best left to the user of the method where the error occurs

```
import java.io.*;
public class AccountClass {
    private String theFile;

    public AccountClass(String aFileName) {
        String accStr;
        FileInputStream fis;
        BufferedReader br;
        theFile = aFileName;
        try {
            fis = new FileInputStream(theFile);
            br = new BufferedReader(new InputStreamReader(fis));
            while ((accStr = br.readLine()) != null) {
                System.out.println(accStr);
            }
            br.close();
        }
        catch (FileNotFoundException e) {
            System.out.println("Account file not found...");
        }
        catch (IOException e) {
            System.out.println("Read error in account file...");
        }
    }
}

import java.io.*;
public class ExceptionDemo {
    public static void main(String args[]) {
        AccountClass anAccount = new AccountClass("Accounts.txtt");
        System.out.println("End of main()...");
    }
}
```

## 1.6 Example 3

This example includes trapping a `NumberFormatException` when a line of text cannot be converted to an integer. The program is allowed to continue after reporting the exception. To do this a nested try/catch block is used.

```
import java.io.*;
public class FileReader {
    private FileInputStream fis;
    private BufferedReader br;

    public void getData(String theFileName) {
        int anInt;
        String tempStr;
        try {
            fis = new FileInputStream(theFileName);
            br = new BufferedReader(new InputStreamReader(fis));
            while ((tempStr = br.readLine()) != null) {
                try {
                    anInt = Integer.parseInt(tempStr);
                }
            }
        }
    }
}
```



```

        System.out.println(anInt);
    }
    catch (NumberFormatException e) {
        System.out.println("Bad integer format in file...");
    } // of inner try
} // of while
br.close();
} // of outer try
catch (FileNotFoundException e) {
    System.out.println("File not found...");
}
catch (IOException e) {
    System.out.println("Problem with reading...");
}
}
}

```

```

public class FileReaderTest {
    public static void main(String arg[]) {
        FileReader f = new FileReader();
        f.getData("IntFile.txt");
        System.out.println("End of main(...)");
    }
}

```

With a text file like  
this...  
100  
200  
300  
jeff

The output is...  
100  
200  
300  
Bad integer format in file...  
400  
500

## 2. THE PROPERTIES CLASS

The Properties class is a subclass of Hashtable.

**Each key and its corresponding value in the property list is a string.**

Instances of the Properties class can be read from or written to a stream.

The Properties class declares several new access methods over the Hashtable. The getProperty() method allows a property to be retrieved using a String object as the key. A second overloaded getProperty() method allows a value string to be used as the default in case the key is not contained in the Properties instance. The load() and save() methods are used to load a Properties object from an input stream and save it to an output stream.

The propertyNames() method provides an enumeration of all the property keys, and the list() method provides a convenient way to print a Properties object on a PrintStream object.

### 2.1 Example

```

import java.util.*;
import java.io.*;
public class SystemDetails {
    private Properties thePropertyTable;

    public SystemDetails() {
        thePropertyTable = new Properties();
        this.build();
    }

    private void build() {
        String[] keys = {"HP_LaserJet_Series_II",
                        "TRIO_DATAFAX",
                        "Sharp_JX-9500E",
                        "HP_LaserJet_5L"};

        String[] values = { "HPPCL,LPT1:,15,45",

```

Key strings cannot contain spaces... seems to cause problems when loading from file back to the Properties object.

```

        "DATAFAX,WINSERVE:,15,45",
        "HPPCL,LPT1:,15,45",
        "HPW,LPT1:,15,45");
    for (int i = 0; i < 4; ++i) thePropertyTable.put(keys[i],values[i]);
}

public void showFullDetails() {
    thePropertyTable.list(System.out);
}

public void showKeys() {
    String key;
    Enumeration e = thePropertyTable.propertyNames();
    while (e.hasMoreElements()) {
        key = (String)e.nextElement();
        System.out.println(key);
    }
}

public void showValues() {
    String key, value;
    Enumeration e = thePropertyTable.propertyNames();
    while (e.hasMoreElements()) {
        key = (String)e.nextElement();
        value = thePropertyTable.getProperty(key);
        System.out.println(value);
    }
}

public void writeToFile() {
    try {
        FileOutputStream fos = new FileOutputStream("SystemFile.txt");
        thePropertyTable.save(fos,"[System Details]");
        fos.close();
    }
    catch (IOException e) {}
}

public void readFromFile() {
    Properties temp = new Properties();
    try {
        System.out.println("Reading Properties object from file...");
        temp.load(new FileInputStream("SystemFile.txt"));
        temp.list(System.out);
    }
    catch (FileNotFoundException e) {}
    catch (IOException e) {}
}

public class SystemDetailsTest {
    public static void main(String arg[]) {
        SystemDetails theSystem = new SystemDetails();
        theSystem.showFullDetails();
        theSystem.showKeys();
        theSystem.showValues();
        theSystem.writeToFile();
        theSystem.readFromFile();
    }
}

```

list the Properties object on the

Create an enumeration of the keys

FileInputStream() may raise a FileNotFoundException

load() may raise an IOException

The created file, SystemFile.txt, looks like this.

```
#[System Details]
#Sat Jan 16 16:48:17 GMT 1999
TRIO_DATAFAX=DATAFAX,WINSERVE:,15,45
Sharp_JX-9500E=HPPCL,LPT1:,15,45
HP_LaserJet_Series_II=HPPCL,LPT1:,15,45
HP_LaserJet_5L=HPW,LPT1:,15,45
```

### 3. THE STREAMTOKENIZER CLASS

A class to turn an input stream into a stream of tokens. There are a number of methods that define the lexical syntax of tokens. This class performs lexical analysis of a specified input stream and breaks the input up into tokens. It can be extremely useful when writing simple parsers. The method `nextToken()` returns the next token in the stream--this is either one of the constants defined by the class (which represent end-of-file, end-of-line, a parsed floating-point number, and a parsed word) or a character value. `pushBack()` "pushes" the token back onto the stream, so that it is returned by the next call to `nextToken()`. The public variables `sval` and `nval` contain the string and numeric values (if applicable) of the most recently read token. They are applicable when the returned token is `TT_WORD` and `TT_NUMBER`. `lineno()` returns the current line number.

The remaining methods allow you to specify how tokens are recognized. `wordChars()` specifies a range of characters that should be treated as parts of words. `whitespaceChars()` specifies a range of characters that serve to delimit tokens. `ordinaryChars()` and `ordinaryChar()` specify characters that are never part of tokens and should be returned as-is. `resetSyntax()` makes all characters "ordinary." `eollsSignificant()` specifies whether end-of-line is significant. If so, the `TT_EOL` constant is returned for end-of-lines. Otherwise they are treated as whitespace.

A token is a character or sequence of characters, (a `String`). Tokens are separated by white spaces. The method `nextToken()` returns the next token and appends the token to either the public instance variables (`String`)`sval` or (`double`)`nval`. So for example with the tokens `abcd efg`, 2 strings are produced and with `123` a double is produced. If the token starts with a character then the token is treated as a string. The real problem is when the first character is a digit...

```
import java.io.*;
public class Tokenizer {
    public static void main (String[] args) {
        InputStream is ;
        StreamTokenizer s ;
        try {
            is = new FileInputStream("Minute3.txt");
            s = new StreamTokenizer(is);
            s.wordChars('/', '/');
            int theTType;
            theTType = s.nextToken();
            while (theTType != StreamTokenizer.TT_EOF) {
                if (theTType == StreamTokenizer.TT_WORD)
                    System.out.println(s.sval);
                if (theTType == StreamTokenizer.TT_NUMBER)
                    System.out.println(s.nval);
                theTType = s.nextToken();
            }
        }
        catch (FileNotFoundException f) { System.out.println("File not found"); }
        catch (IOException i) { System.out.println("Problem with IO"); }
    }
}
```

MEETING : Sales  
 DATE : 22/1/95  
 Sales meeting held on 22/1/95  
 For the first 2 meetings product  
 abc123 was discussed  
 along with product 123abc.

MEETING  
 Sales  
 DATE  
 22  
 /1/95  
 Sales  
 meeting  
 held  
 on  
 22  
 /1/95  
 For  
 the

first  
 2  
 meetings  
 product  
 abc123  
 was  
 discussed  
 along  
 with  
 product  
 123  
 abc.

## 4. SERIALISATION AND PERSISTENT OBJECTS

### 4.1 Serialization of Objects (Saving Objects)

Most meaningful Java applications need to provide a way to save the objects they create and to restore the objects at some later date. The capability for an object to exist from one run of the application to another is known as *persistence*. Serialization is the key to implementing persistence. Serialization provides the capability to write an object to a stream and to read the object back at a later time. Serialization allows you to store objects in files, to communicate them access networks and to use them in distributed applications.

When an object is written to a stream, information about its class must be stored along with the object. Without class information there is no way to reconstruct an object that is read from a stream. In addition to class information all objects that are referenced by that object must also be stored. If the referenced objects are not stored the references of the stored object are of course meaningless.

For an object to be made persistent its class must be serialized. Serializability of a class is enabled by the class implementing the `java.io.Serializable` interface. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class `Serializable` in this case. The error will be detected at runtime.

During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;
```

The `writeObject` method is responsible for writing the state of the object for its particular class so that the corresponding `readObject` method can restore it. The default mechanism for saving the Object's fields can be invoked by calling `out.defaultWriteObject`. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the `ObjectOutputStream` using the `writeObject` method or by using the methods for primitive data types supported by `DataOutput`.

The `readObject` method is responsible for reading from the stream and restoring the object. It may call `in.defaultReadObject` to invoke the default mechanism for restoring the object's non-static and non-transient fields. The `defaultReadObject` method uses information in the stream to assign the fields of the object saved in the stream with the correspondingly named fields in the current object. This handles the case when the class has evolved to add new fields. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the `ObjectOutputStream` using the `writeObject` method or by using the methods for primitive data types supported by `DataOutput`.

## 4.2 Transient instance variables

A transient instance variable is one whose value is not saved when the object is serialized. In the example the `Address` instance variable in the `Person` class is made transient. When the `Person` object is read from the disc this instance variable is set to null.

## 4.3 A Persistent Object Example

All classes that form part of the serialised object must implement the `Serializable` interface. `Person` objects will be the persistent objects.

```
import java.io.*;
public class Address implements Serializable {
    private String theCity;
    private String thePostCode;

    public Address(String aCity,String aPostCode) {
        theCity = aCity;
        thePostCode = aPostCode;
    }
    public String toString() {
        return theCity + ":" + thePostCode;
    }
}

import java.io.*;
public class Person implements Serializable {
    private String theName;
    private Address theAddress;
    private transient long ticketNumber;

    public Person(String aName, Address anAddress) {
        this.setName(aName);
        this.setAddress(anAddress);
        this.setTicketNumber();
    }
    public void setName(String aName) {
        theName = aName;
    }
    public void setAddress(Address anAddress) {
        theAddress = anAddress;
    }
    public void setTicketNumber() {
        ticketNumber = Math.round(Math.random()*1000);
    }
    public String toString() {
        return theName + ":" + theAddress + ":" + ticketNumber;
    }
}
```

```

import java.io.*;
public class WritePerson {
    public static void main(String args[]) {
        Address newAddress = new Address("Manchester","M3 2KJ");
        Person me = new Person("Jeff",newAddress);
        System.out.println("Before write... " + me.toString());

        // Save the Person object...
        try {
            FileOutputStream fos = new FileOutputStream("Person.tst");
            ObjectOutputStream theObjectStream = new ObjectOutputStream(fos);
            theObjectStream.writeObject(me);
            theObjectStream.close();
            fos.close();
        }
        catch (IOException e) {
            System.out.println("\nIOException on attempt to open : " + e.getMessage());
        }
    }
}

import java.io.*;
public class ReadPerson {
    public static void main(String args[]) {
        try {
            FileInputStream fis = new FileInputStream("Person.tst");
            ObjectInputStream theObjectStream = new ObjectInputStream(fis);
            Person thePerson = (Person)theObjectStream.readObject();
            theObjectStream.close();
            fis.close();
            System.out.println("After read... " + thePerson.toString());
            // will have to set the ticketNumber variable...
            thePerson.setTicketNumber();
            System.out.println("After reset ticketNumber... " + thePerson.toString());
        }
        catch (ClassNotFoundException e) {
            System.out.println("Class not found... : " + e.getMessage());
        }
        catch (NotSerializableException e) {
            System.out.println("Not serializable error... : " + e.getMessage());
        }
        catch (FileNotFoundException e) {
            System.out.println("File error... can't find the Person file : " + e.getMessage());
        }
        catch (IOException e) {
            System.out.println("File read error... : " + e.getMessage());
        }
    }
}

```

When the Person object is read from disc, all the classes referenced by the de-serialised object need to be available. If they are not then a `ClassNotFoundException` will be raised. Try removing the Address class after writing and before reading and see the result.

Serialisation is at the heart of Java's distributed computing strategy. Objects are transported around a distributed system using Java's Stream classes.

## 5. THREADS

A Thread is a single sequential flow of control within a process. This simply means that while executing within a program, each thread has a beginning, a sequence, a point of execution occurring at any time during runtime of the thread and of course, an ending. Thread objects are the basis for multi-threaded programming. Multi-threaded programming allows a single program to conduct concurrently running threads that perform different tasks.

A Thread terminates when its `run()` method is completed, or when the `stop()` method is called.

To execute a thread we call the thread's `start()` method. This puts the thread into the runnable state and informs the host operating system that the thread can be run when its turn comes up in the scheduler. When this happens the thread's `run()` method will be invoked. The thread will enter a queue of processes waiting for processor time.

The strategy used to determine which thread should execute at a given time is known as scheduling. Java's approach to scheduling is referred to as pre-emptive scheduling. When a thread of higher priority becomes runnable it preempts threads of lower priority and is immediately executed. The highest priority thread continues to run until it is blocked, has its priority lowered or an even higher priority thread becomes runnable.

Java provides two approaches to creating threads,

- you create a subclass of **Thread** and override the **run()** method to provide the entry point for the thread's execution.
- you create a class implemented from the **Runnable** Interface and then use this class to create Thread instances.

### 5.1 Creating Threads

#### 5.1.1 Example 1 - Inheriting From the Thread Class

```
public class Counter extends Thread {
    private int delay;

    public Counter(String theName, int aDelay) {
        super(theName);
        delay = aDelay;
    }
    public void run() {
        for (int i = 0; i < 50; ++i) {
            System.out.println(this.getName() + " count = " + i);
            try {
                Thread.sleep(delay);
            }
            catch (InterruptedException e) {}
        }
    }
}

public class ThreadTest {
    public static void main(String argv[]) {
        Counter t1 = new Counter("Thread_1",500);
        Counter t2 = new Counter("Thread_2",1250);
        t1.start();
        t2.start();
        while (1==1){};
    }
}
```

Output on one run looked like the following, read down columns left to right for the results.

Thread_1 count = 0	Thread_1 count = 4	Thread_2 count = 30	Thread_1 count = 28
Thread_2 count = 0	Thread_1 count = 5	Thread_2 count = 31	Thread_1 count = 29
Thread_2 count = 1	Thread_1 count = 6	Thread_2 count = 32	Thread_1 count = 30
Thread_2 count = 2	Thread_1 count = 7	Thread_2 count = 33	Thread_1 count = 31
Thread_2 count = 3	Thread_1 count = 8	Thread_2 count = 34	Thread_1 count = 32
Thread_2 count = 4	Thread_1 count = 9	Thread_2 count = 35	Thread_1 count = 33
Thread_2 count = 5	Thread_1 count = 10	Thread_2 count = 36	Thread_1 count = 34
Thread_2 count = 6	Thread_1 count = 11	Thread_2 count = 37	Thread_1 count = 35
Thread_2 count = 7	Thread_1 count = 12	Thread_2 count = 38	Thread_1 count = 36
Thread_2 count = 8	Thread_1 count = 13	Thread_2 count = 39	Thread_1 count = 37
Thread_2 count = 9	Thread_1 count = 14	Thread_2 count = 40	Thread_1 count = 38
Thread_2 count = 10	Thread_1 count = 15	Thread_2 count = 41	Thread_1 count = 39
Thread_2 count = 11	Thread_1 count = 16	Thread_2 count = 42	Thread_1 count = 40
Thread_2 count = 12	Thread_1 count = 17	Thread_2 count = 43	Thread_1 count = 41
Thread_2 count = 13	Thread_1 count = 18	Thread_2 count = 44	Thread_1 count = 42
Thread_2 count = 14	Thread_1 count = 19	Thread_2 count = 45	Thread_1 count = 43
Thread_2 count = 15	Thread_1 count = 20	Thread_2 count = 46	Thread_1 count = 44
Thread_2 count = 16	Thread_1 count = 22	Thread_2 count = 47	Thread_1 count = 45
Thread_2 count = 17	Thread_2 count = 23	Thread_2 count = 48	Thread_1 count = 46
Thread_2 count = 18	Thread_2 count = 24	Thread_2 count = 49	Thread_1 count = 47
Thread_2 count = 19	Thread_2 count = 25	Thread_1 count = 21	Thread_1 count = 48
Thread_2 count = 20	Thread_2 count = 26	Thread_1 count = 22	Thread_1 count = 49
Thread_2 count = 21	Thread_2 count = 27	Thread_1 count = 23	
Thread_1 count = 1	Thread_2 count = 28	Thread_1 count = 24	
Thread_1 count = 2	Thread_2 count = 29	Thread_1 count = 25	
Thread_1 count = 3		Thread_1 count = 26	
		Thread_1 count = 27	

### 5.1.2 Example 2 - Implementing the Runnable Interface

**public class Counter implements Runnable {**

private String theName;

private int delay;

public Counter(String aName, int aDelay) {

theName = aName;

delay = aDelay;

}

public void run() {

for (int i = 0; i < 50; ++i) {

System.out.println(theName + " count = " + i);

try {

Thread.sleep(delay);

}

catch (InterruptedException e) {}

}

}

**public class ThreadTest {**

public static void main(String argv[]) {

Thread t1 = new Thread(new Counter("Thread\_1",500));

Thread t2 = new Thread(new Counter("Thread\_2",1250));

t1.start();

t2.start();

}

}

The Runnable interface contains just one method run() which must be defined in the class that implements the Runnable interface.

The Thread class has a constructor that takes a Runnable object as its argument and creates a Thread object from it.

The run() method of the t1 and t2 threads is the run() method defined in the Counter class which implements Runnable and hence Counter objects are Runnable objects.

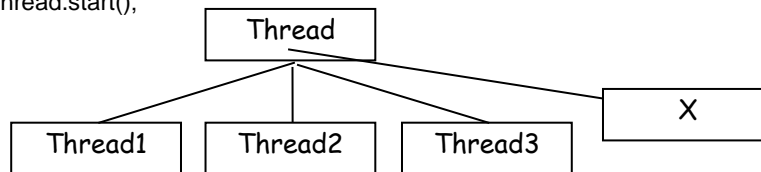
Output is similar to the previous output



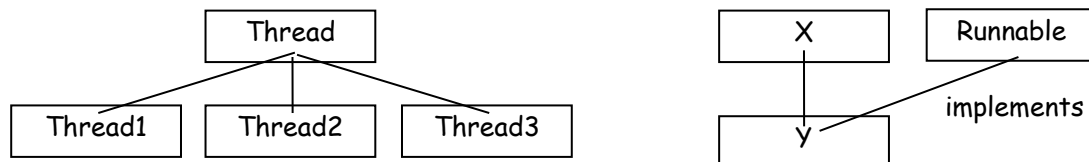
## 5.2 Inheritance vs Runnable

Creating Threads using the Runnable interface is a little more complex than inheriting from the Thread class. However in many situations the Runnable approach is more convenient since the inheritance approach requires your Thread class to be located within the Thread hierarchy.

So what's the benefit of using a Runnable interface to create a thread ? One benefit may be in making an existing class behave as a thread. Suppose you have an existing class X which you would like to become part of the previous thread system. One possible solution is of course as follows, `theThread.start();`



This solution means that you will have to amend the source code for class X. If you don't want to change (you'll have to add a run method at least), or do not have access to, the source code of X, the following diagram shows how this might be achieved.



The derived class Y can now be implemented as a thread class. (we can't do.... class Y extends Thread, Y already extends X).

### 5.2.1 Example - Thread vs Runnable

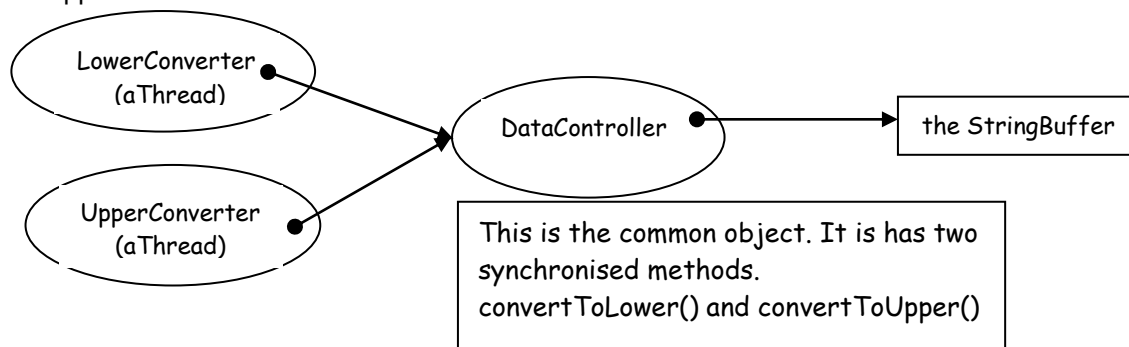
Comparison of the two approaches is demonstrated with the following...

Inheritance from Thread	Implementing the Runnable interface
<pre> public class Server3 {     public static void main(String args[]) {         SimpleServer ss = new SimpleServer();         ss.start();     } } </pre>	<pre> public class Server3 {     public static void main(String args[]) {         SimpleServer ss = new SimpleServer();     } } </pre>
<pre> public class SimpleServer extends Thread {     public void run() {         ConnectionHandler ch = new ConnectionHandler();         ch.start();     } } </pre>	<pre> public class SimpleServer implements Runnable {     private Thread aThread;     public SimpleServer() {         // create thread from this instance of SimpleServer         aThread = new Thread(this);         // Effectively start SimpleServer as a thread...         aThread.start();     }     public void run() {         ConnectionHandler ch = new ConnectionHandler();     } } </pre>
<pre> public class ConnectionHandler extends Thread { } </pre>	<pre> public class ConnectionHandler implements Runnable {     private Thread theThread;     public ConnectionHandler() {         // create thread from this instance of         // ConnectionHandler         theThread = new Thread(this);         // effectively start the ConnectionHandler as a thread...         theThread.start();     }     public void run() {     } } </pre>

You could of course, if circumstances demand, use a 'mix' of the two approaches.

### 5.3 Shared Object – Synchronization

There are many situations where multiple threads must share access to common objects. Often the thread objects have to synchronise their actions on the common object so that they can work together in an ordered manner. There are times when you might want to co-ordinate access to database records where you have one thread which is responsible for updating a record and another whose job it is to read a record. Here you would need to ensure that the read thread did not attempt to access the record whilst the write thread was updating the record. Java enables you to co-ordinate the actions of multiple threads using synchronised methods. In this example we demonstrate 2 threads which share access to a common object (a StringBuffer object - managed by a DataController object). Whilst one thread is processing the StringBuffer the other thread must be denied access to the common object, otherwise the StringBuffer may become a mix of lower and upper case.



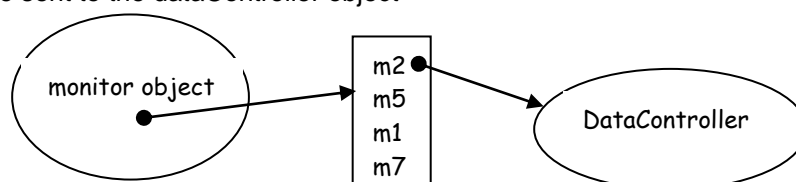
Both objects (LowerConverter and UpperConverter) access the common object with the view of performing different operations on the common object's data. In this case by setting the characters of the common objects StringBuffer to either lower case or upper case. If the conversion operation is not synchronised, then either thread can interrupt the other's access to the operation and the outcome of the conversion is then unpredictable. Variations on this shared object problem are often known as the producer/consumer problem.

Any object, which has synchronised methods, is associated with a monitor. The role of the monitor is to control the way in which synchronised methods of the object are allowed access to the object. When a synchronised method is invoked for an object it is said to acquire the monitor for that object. If the monitor is not available, because it has been acquired by another method, the method has to wait until the monitor becomes available before it can proceed. The monitor is automatically released when a method completes, or when a wait() method is invoked from within the currently executing method.

In the example the common object is a DataController object. The synchronised methods convertToLower() and convertToUpper() each attempt to acquire the monitor of the DataController object. Because these two methods are synchronised the execution of convertToLower() cannot interrupt the execution of convertToUpper, and vice-versa. Both methods run to completion when started. Only one synchronised method can be invoked on an object at a given point in time. A method that is declared as synchronised cannot be executed by more than one process at any time. Once a thread has gained access to a synchronized method, it will finish the method before any other thread is allowed access to that or any other synchronized method for that object.

It should be noted that synchronisation does not guarantee exclusive access to an object's data since a method that is not declared as synchronised may execute independently. The lock is on the synchronised method not the data item.

The monitor for the DataController object is an object that maintains a queue of synchronised methods waiting to be sent to the dataController object



m2 currently has the monitor and is being executed by the DataController object.  
 m2 has been sent to the DataController by Thread\_1  
 m5 has been sent to the DataController by Thread\_2, before m2 has completed, m5 is synchronised with m2 and therefore has to wait.  
 etc  
 Each time a synchronised method is sent to the DataController object it is placed in the queue of waiting methods.

#### 5.3.1 Example 4

**public class DataController {**

private StringBuffer theStringBuf;

public DataController(StringBuffer theData) {  
 theStringBuf = theData;  
}

public synchronized void convertToLower() {  
 // public void convertToLower() {  
 char ch;  
 try {  
 for (int i = 0; i < theStringBuf.length(); ++i) {  
 ch = theStringBuf.charAt(i);  
 if ((64 < ch) && (ch < 91)) {  
 ch += 32;  
 theStringBuf.setCharAt(i,ch);  
 }  
 // force a delay... simulate some longer processing  
 Thread.sleep(100);  
 }  
 }  
 catch (InterruptedException e) {}  
 System.out.println("From Lower : string = " + theStringBuf);  
}

Note the alternative signatures of the two conversion methods.

public synchronized void convertToUpper() {  
 // public void convertToUpper() {  
 char ch;  
 try {  
 for (int i = 0; i < theStringBuf.length(); ++i) {  
 ch = theStringBuf.charAt(i);  
 if ((96 < ch) && (ch < 123)) {  
 ch -= 32;  
 theStringBuf.setCharAt(i,ch);  
 }  
 // force a delay... simulate some longer processing  
 Thread.sleep(150);  
 }  
 }  
 catch (InterruptedException e) {}  
 System.out.println("From Upper : string = " + theStringBuf);  
}

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread does not loose ownership of any monitors.

<pre> public class LowerConverter implements Runnable {     private DataController theController;     public LowerConverter(DataController aController) {         theController = aController;     }      public void run() {         for (int i = 0; i &lt; 10; ++i)             theController.convertToLower();     } } </pre>	<pre> public class UpperConverter implements Runnable {     private DataController theController;     public UpperConverter(DataController aController) {         theController = aController;     }      public void run() {         for (int i = 0; i &lt; 10; ++i)             theController.convertToUpper();     } } </pre>
--	--

### public class SynchExample {

```

    public static void main(String args[]) {
        StringBuffer theData = new StringBuffer("java programming");
        DataController theController = new DataController(theData);
        Thread lowerThread = new Thread(new LowerConverter(theController));
        Thread upperThread = new Thread(new UpperConverter(theController));
        lowerThread.start(); upperThread.start();
    }
}

```

Without synchronisation the target string becomes a mix of upper and lower case. This is because the `accessBuffer()` method gets interrupted by a change of thread...

From Upper : string = Java progRAMMING From Upper : string = JAVA PROGRAMMING From Lower : string = jAVA PROGRAMMING From Upper : string = JAVA PrograMMING From Upper : string = JAVA Programming From Lower : string = java PROGRAMMING From Upper : string = JAVA PROGRAMmiNG From Upper : string = JAVA PROGRAMming From Lower : string = java PROGRAMMING From Upper : string = JAVA PROGRAMMING	From Upper : string = JAVA PROGRAMMING From Lower : string = java proGRAMMING From Upper : string = JAVA PROGRAMMING From Lower : string = jAVA PROGRAMMING From Upper : string = JAva progRAMMING From Upper : string = JAVA programming From Lower : string = java programming From Lower : string = java programming From Lower : string = java programming From Lower : string = java programming
--	--

Output when the method (`accessBuffer()`) was synchronized. This is the desired result.

From Lower : string = java programming From Upper : string = JAVA PROGRAMMING From Upper : string = JAVA PROGRAMMING From Lower : string = java programming From Upper : string = JAVA PROGRAMMING From Upper : string = JAVA PROGRAMMING From Lower : string = java programming From Lower : string = java programming From Upper : string = JAVA PROGRAMMING From Upper : string = JAVA PROGRAMMING From Lower : string = java programming	From Lower : string = java programming From Upper : string = JAVA PROGRAMMING From Upper : string = JAVA PROGRAMMING From Lower : string = java programming From Lower : string = java programming From Upper : string = JAVA PROGRAMMING From Upper : string = JAVA PROGRAMMING From Lower : string = java programming From Lower : string = java programming
--	--

### 5.3.2 What Happens if One Synchronised Method calls another Synchronised Method ?

If a synchronised method calls another synchronised method the outer method passes the monitor to the inner method, When the inner method completes it passes the monitor back to the outer method. This way the original outer method effectively maintains control of the monitor.

### 5.3.3 What happens if one Synchronised Method Depends on Another Synchronised Method ?

A problem can arise when a situation arises where the currently executing synchronised method waits on the action of another synchronised method. This is looked at in the next section.

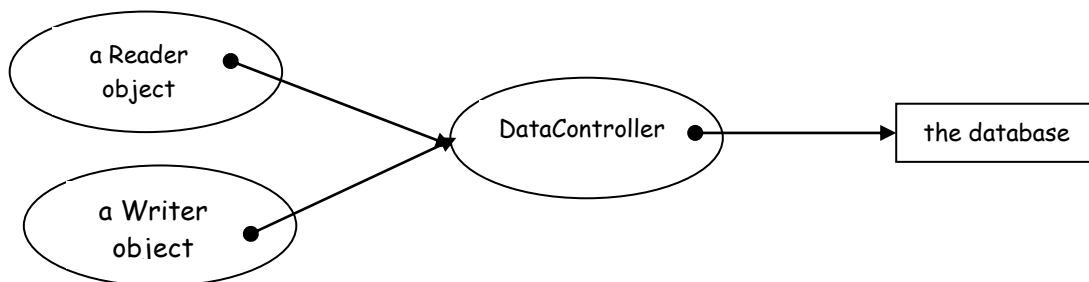
## 5.4 The Readers and Writers Problem

The features of the problem are,

- several concurrent process wish to access a common data structure
- some wish to read the data structure; some wish to write to the data structure
- shared read accesses to the data structure are required, but exclusive access is required by the writers

The classical application is the airline reservation system where many enquiries on a shared database are allowed, but only one travel agent/booking office at a time can be allowed to change the database and reserve a seat.

One way in which the readers/writers problem is approached is to delegate an object to have control of the database so that other objects (readers and writers) wishing to access the database, send messages to the controller object.



In this next section we look at a variation of the readers/writers problem where the writers and readers work with a particular kind of data structure, that of a character buffer where the writers place items (characters) and the readers remove items (characters).

At first it might seem no more than a synchronisation exercise that could easily be handled by the previous synchronisation mechanism. All we have to do is to get the writing and reading operations synchronised and all will be well. Indeed this is essentially the case... except with a twist in the tale if we are not careful.

We will take a very simple case first of all to demonstrate the major problem involved.

### 5.4.1 Example 5 - Possible System lock

```

import java.util.*;
public class DataController {
    private Vector theBuffer;

    public DataController() {
        theBuffer = new Vector();
    }

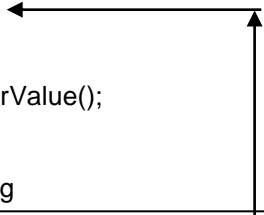
    public synchronized void write(int n) {
        try {
            Random rnd = new Random();
            char c;
            for (int i = 0; i < n; ++i) {
                c = (char)(65 + Math.abs(rnd.nextInt()) % 26);
                System.out.println("Writing : " + c);
                theBuffer.addElement(new Character(c));
                Thread.sleep(200);
            }
        }
        catch (InterruptedException e) {}
    }
}
  
```

```

public synchronized void read(int n) {
    try {
        for (int i = 0; i < n; ++i) {
            while (this.hasNoData()) {
                System.out.println("Reader is waiting...");
                Thread.sleep(100);
            }
            char c = ((Character)theBuffer.firstElement()).charValue();
            System.out.println("Reading : " + c);
            theBuffer.removeElementAt(0);
            // force a delay... simulate some longer processing
            Thread.sleep(200);
        }
    }
    catch (InterruptedException e) {}
}

public boolean hasNoData() {
    return theBuffer.isEmpty();
}
}

```



This is where the problem lies... while the buffer remains empty the read() method goes into a loop, hoping that while it goes to sleep something will appear in the buffer. The write() method is the only way this can happen, but the read() and write() methods are synchronised methods so whilst the read() has the DataController object's monitor the write() method cannot execute. Hence the whole system will hang. If the buffer happens to be empty when the read() method has the monitor, the system will lock.

#### **public class Writer implements Runnable {**

```

    private DataController theController;

    public Writer(DataController aController) {
        theController = aController;
    }

    public void run() {
        theController.write(7);
        theController.write(4);
        theController.write(6);
    }
}

```

#### **public class Reader implements Runnable {**

```

    private DataController theController;

    public Reader(DataController aController) {
        theController = aController;
    }

    public void run() {
        theController.read(4);
        theController.read(4);
        theController.read(4);
        theController.read(4);
    }
}

```

#### **public class SynchExample {**

```

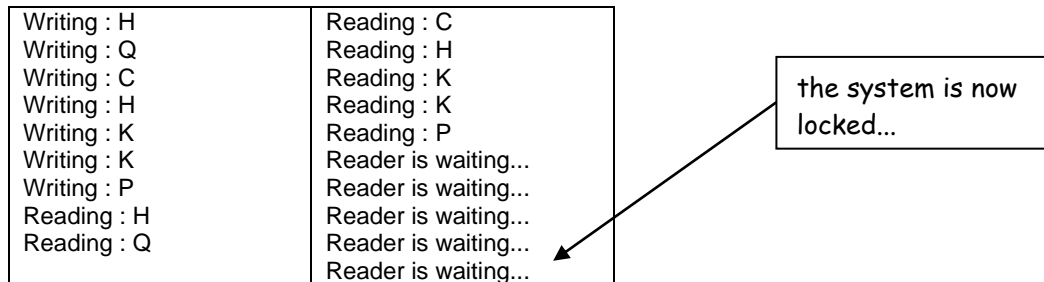
    public static void main(String args[]) {
        DataController theController = new DataController();
        Thread theWriter = new Thread(new Writer(theController));
        Thread theReader = new Thread(new Reader(theController));
        theWriter.setPriority(3);
    }
}

```

```

    theReader.setPriority(4);
    theWriter.start(); theReader.start();
}
}

```



#### 5.4.2 Example 6 - A Solution Using the wait() and notify() methods

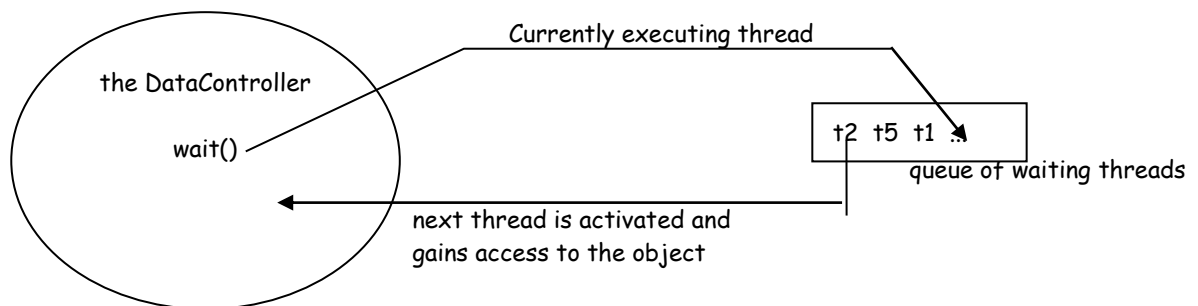
The problem of locking can be solved with the use of the wait() and notify() methods.

The wait() and notify() methods are implemented in the Object class so that they are available to threads that are not subclasses of Thread. That is, they are threads that have been created from the Runnable interface. The wait() method causes a thread to deactivate and join a queue of other waiting threads. The notify() method is used to notify waiting threads that their wait is over.

An object that is being accessed by a number of threads has a queue associated with it. This queue will hold the thread objects that wish to gain access to the target object.

When an object executes the wait() method, the thread that currently has access to the object is deactivated and placed in the queue of waiting threads. This allows any other thread to gain access to the object.

When an object executes the notify() method, any thread currently queueing for access to the object can be activated and executed.



```

import java.util.*;
public class DataController {
    private Vector theBuffer;

    public DataController() {
        theBuffer = new Vector();
    }

    public synchronized void write(int n) {
        try {
            Random rnd = new Random();
            char c;
            for (int i = 0; i < n; ++i) {
                c = (char)(65 + Math.abs(rnd.nextInt()) % 26);
                System.out.println("Writing : " + c);
                theBuffer.addElement(new Character(c));
                Thread.sleep(200);
            }
            this.notify();
        }
        catch (InterruptedException e) {}
    }

    public synchronized void read(int n) {
        try {
            for (int i = 0; i < n; ++i) {
                while (this.hasNoData()) {
                    System.out.println("Reader is waiting...");
                    this.wait();
                }
                char c = ((Character)theBuffer.firstElement()).charValue();
                System.out.println("Reading : " + c);
                theBuffer.removeElementAt(0);
                // force a delay... simulate some longer processing
                Thread.sleep(200);
                this.notify();
            }
        }
        catch (InterruptedException e) {}
    }

    public boolean hasNoData() {
        return theBuffer.isEmpty();
    }
}

```

the write() method is now completed and hence the dataController object notifies the queue of waiting threads that it is now available and so another thread can be activated

the DataController object executes the wait() method... the currently active reader thread is deactivated and placed on the queue of waiting threads. Another thread can now gain access to the DataController object.

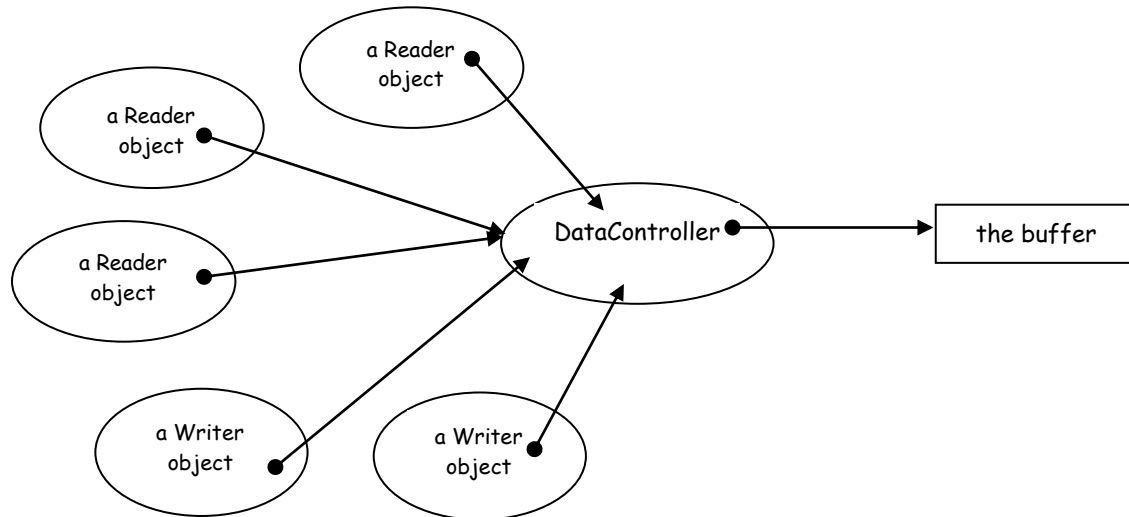
the read() method is now completed and hence the DataController object notifies the queue of waiting threads that it is now available and so another thread can be activated



### 5.4.3 Example 7 - Multiple Writers and Readers

In this example we have reorganised the code somewhat and added a number of writers and readers. The main differences from the previous code are,

- there are now 2 writer objects sending messages to the DataController object; an upper case writer and a lower case writer
- there are now 3 reader objects
- some of the writing and reading functionality has been taken out of the DataController and moved to the writer and reader objects



```

import java.util.*;
public class DataController {
    private Vector theBuffer;

    public DataController() {
        theBuffer = new Vector();
    }

    public synchronized void write(char c) {
        try {
            theBuffer.addElement(new Character(c));
            Thread.sleep(200);
            this.notify();
        }
        catch (InterruptedException e) {}
    }

    public synchronized char read() {
        char c = ' ';
        try {
            while (this.hasNoData()) {
                System.out.println("Reader is waiting...");
                this.wait();
            }
            c = ((Character)theBuffer.firstElement()).charValue();
            theBuffer.removeElementAt(0);
            // force a delay... simulate some longer processing
            Thread.sleep(200);
            this.notify();
        }
        catch (InterruptedException e) {}
        return c;
    }
}
  
```

```

        public boolean hasNoData() {
            return theBuffer.isEmpty();
        }
    }

import java.util.*;
public class Writer implements Runnable {
    private DataController theController;
    private String id;
    private int startValue;

    public Writer(DataController aController, char kase) {
        theController = aController;
        if (kase == 'l') {
            startValue = 97;
            id = "lower";
        }
        else {
            startValue = 65;
            id = "upper";
        }
    }

    public void run() {
        this.writeToController(7);
        this.writeToController(4);
        this.writeToController(6);
    }

    public void writeToController(int n) {
        try {
            Random rnd = new Random();
            char c;
            for (int i = 0; i < n; ++i) {
                c = (char)(startValue + Math.abs(rnd.nextInt()) % 26);
                System.out.println("Writing from " + id + " : " + c);
                theController.write(c);
                Thread.sleep(200);
            }
        }
        catch (InterruptedException e) {}
    }
}

```

```

public class Reader implements Runnable {
    private DataController theController;
    private String id;

    public Reader(DataController aController, String id) {
        theController = aController;
        this.id = id;
    }

    public void run() {
        this.readFromController(4); this.readFromController(4);
        this.readFromController(4); this.readFromController(4);
    }

    public void readFromController(int n) {

```

```

    try {
        for (int i = 0; i < n; ++i) {
            char c = theController.read();
            System.out.println("Reading by " + id + " : " + c);
            // force a delay... simulate some longer processing
            Thread.sleep(200);
        }
    }
    catch (InterruptedException e) {}
}
}

```

```

public class SynchExample {
    public static void main(String args[]) {
        DataController theController = new DataController();

        Thread theLowerCaseWriter = new Thread(new Writer(theController, 'l'));
        Thread theUpperCaseWriter = new Thread(new Writer(theController, 'u'));

        Thread bilboReader = new Thread(new Reader(theController, "Bilbo"));
        Thread frodoReader = new Thread(new Reader(theController, "frodo"));
        Thread gandalfReader = new Thread(new Reader(theController, "gandalf"));

        theLowerCaseWriter.start();
        theUpperCaseWriter.start();
        bilboReader.start();
        frodoReader.start();
        gandalfReader.start();
    }
}

```

## 5.5 TheYield() Method

### 5.5.1 Example 8

The `yield()` method causes the currently executing Thread object to yield. If there are other runnable Threads they will be scheduled next. Note that you do not have control over the order of thread execution. This is entirely up to the operating system and the Java Virtual Machine.

<pre> public class Driver {     public static void main(String args[]) {         Thread1 t1 = new Thread1(10);         Thread2 t2 = new Thread2(15);         Thread3 t3 = new Thread3(20);         t1.start(); t2.start(); t3.start();     } } </pre>	<pre> public class Thread1 extends Thread {     private int maxVal;      public Thread1(int n) {         super();         maxVal = n;     }      public void run() {         for (int i = 0; i &lt; maxVal; ++i) {             System.out.println("From " + this.getName() + " : " + i);             if (i % 2 == 0) {                 System.out.println("\t" + this.getName() + " yielding...");                 this.yield();             }         }         System.out.println("&gt;&gt;&gt; " + this.getName() + " COMPLETED...");     } } </pre>
---	---

```

public class Thread2 extends Thread {
    private int maxValue;

    public Thread2(int n) {
        super();
        maxValue = n;
    }

    public void run() {
        for (int i = 0; i < maxValue; ++i) {
            System.out.println("From " + this.getName() + " : " + i);
            if (i % 3 == 0) {
                System.out.println("\t" + this.getName() + " yielding...");
                this.yield();
            }
        }
        System.out.println(">>> " + this.getName() + " COMPLETED...");
    }
}

```

```

import java.util.*;
public class Thread3 extends Stack implements Runnable {
    private int maxValue;
    private Thread t;

    public Thread3(int n) {
        super();
        maxValue = n;
        t = new Thread(this);
    }

    public void start() {
        t.start();
    }

    public void run() {
        for (int i = 0; i < maxValue; ++i) {
            System.out.println("From " + t.getName() + " : " + i);
            if (i % 4 == 0) {
                System.out.println("\t" + t.getName() + " yielding...");
                t.yield();
            }
        }
        System.out.println(">>> " + t.getName() + "
COMPLETED...");
    }
}

```

Not how the Runnable interface is used with Thread\_3. There is no significance (or indeed use...) of the Stack class, it is just to demonstrate how Runnable can be used.

Output from one run was as follows... read down left column then down right column for the results.

From Thread-1 : 0	From Thread-3 : 4	From Thread-2 : 9	Thread-1 yielding...
From Thread-2 : 0	Thread-3 yielding...	Thread-2 yielding...	From Thread-3 : 13
Thread-2 yielding...	From Thread-2 : 4	From Thread-1 : 5	From Thread-3 : 14
Thread-1 yielding...	From Thread-2 : 5	From Thread-1 : 6	From Thread-3 : 15
From Thread-3 : 0	From Thread-2 : 6	Thread-1 yielding...	From Thread-3 : 16
Thread-3 yielding...	Thread-2 yielding...	From Thread-3 : 9	Thread-3 yielding...
From Thread-2 : 1	From Thread-1 : 3	From Thread-3 : 10	From Thread-2 : 13
From Thread-2 : 2	From Thread-1 : 4	From Thread-3 : 11	From Thread-2 : 14
From Thread-2 : 3	Thread-1 yielding...	From Thread-3 : 12	>>> Thread-2 COMPLETED...
Thread-2 yielding...	From Thread-3 : 5	Thread-3 yielding...	From Thread-1 : 9
From Thread-1 : 1	From Thread-3 : 6	From Thread-2 : 10	>>> Thread-1 COMPLETED...
From Thread-1 : 2	From Thread-3 : 7	From Thread-2 : 11	From Thread-3 : 17
Thread-1 yielding...	From Thread-3 : 8	From Thread-2 : 12	From Thread-3 : 18
From Thread-3 : 1	Thread-3 yielding...	Thread-2 yielding...	From Thread-3 : 19
From Thread-3 : 2	From Thread-2 : 7	From Thread-1 : 7	>>> Thread-3 COMPLETED...
From Thread-3 : 3	From Thread-2 : 8	From Thread-1 : 8	

## 5.6 Thread Priorities

A thread's priority is an integer value between MIN\_PRIORITY (=1) and MAX\_PRIORITY (=10). These constants are defined in the Thread class. A thread's priority is set when it is created and is set to the same priority as the thread that creates it. The default priority is NORM\_PRIORITY (=5). The priority of a thread can be changed with setPriority(n) where 1 <= n <= 10.

## 6. SOME BITS AND PIECES

### 6.1 Copying Objects

There's more to copying objects than you might at first think.

You should realise by now that if you do

```
Book aBook = new Book("Tolkein", "09-08-07", "The Hobbit", 1234);
```

```
Book myBook = aBook;
```

You have not created a new Book object. Only one instance (object) of the Book class exists. You have simply got 2 references (pointers) to it. The same Book object can be referred to as *aBook* or *myBook*. Both variables point to the same Book object.

If you want to 'copy' the Book object in the sense that you have a 'new' (ie different) Book object with the same state as the original Book object you will have to create a new Book object whose instance variables have the same values as the original.

### 6.2 public class Vector extends Object implements Cloneable

A class implements the Cloneable interface to indicate to the clone method in class Object that it is legal for that method to make a field-for-field copy of instances of that class. Attempts to clone instances that do not implement the Cloneable interface result in the exception CloneNotSupportedException being thrown. The Vector class implements the Cloneable interface hence it is possible to make copies of a Vector object using the clone() method from the Object class.

We use the following classes to demonstrate object copying.

```
public class Book {
    private String author;
    private String isbn;
    private String title;
    private int price;

    public Book(String anAuthor, String anISBN, String aTitle, int aPrice) {
        this.setAuthor(anAuthor);
        this.setISBN(anISBN);
        this.setTitle(aTitle);
        this.setPrice(aPrice);
    }

    public void setAuthor(String anAuthor) { author = anAuthor;}
    public String getAuthor() { return author;}
    public void setISBN(String anISBN) { isbn = anISBN; }
    public String getISBN() { return isbn;}
    public void setTitle(String aTitle) { title = aTitle; }
    public String getTitle() { return title;}
    public void setPrice(int aPrice) { price = aPrice; }
    public int getPrice() { return price; }
    public String toString() {
        return "Author : " + author + " ISBN : " + isbn + " Title : " + title + " Price : " + price;
    }
}
```

```

import java.util.*;
public class Catalogue {
    private Vector theBookList;

    public Catalogue() {
        theBookList = new Vector();
    }

    public void addBook(Book aBook) {
        theBookList.addElement(aBook);
    }

    public Book getBook(String isbn) {
        Book theBook = null;
        Enumeration e = theBookList.elements();
        while (e.hasMoreElements()) {
            theBook = (Book)e.nextElement();
            if (isbn.equals(theBook.getISBN())) return theBook;
        }
        return theBook;
    }

    public Catalogue copy() {
        // See later for the details of this method...
    }

    public String toString() {
        Book aBook;
        String tempStr = "";
        Enumeration e = theBookList.elements();
        while (e.hasMoreElements()) {
            aBook = (Book)e.nextElement();
            tempStr += aBook + "\n";
        }
        return tempStr;
    }
}

public class BookTest {
    public static void main(String args[]) {
        Catalogue theCatalogue = new Catalogue();
        Book aBook = new Book("Tolkein", "09-08-07", "The Hobbit", 1234);
        theCatalogue.addBook(aBook);
        aBook = new Book("Gray", "11-22-33", "The Return", 1000);
        theCatalogue.addBook(aBook);
        aBook = new Book("Jameson", "00-01-02", "The New Land", 2000);
        theCatalogue.addBook(aBook);
        System.out.println("theCatalogue\n" + theCatalogue);

        Catalogue newCatalogue = theCatalogue.copy();
        System.out.println("newCatalogue\n" + newCatalogue);

        aBook = theCatalogue.getBook("11-22-33");
        aBook.setPrice(3456);

        System.out.println("theCatalogue\n" + theCatalogue);
        System.out.println("newCatalogue\n" + newCatalogue);
    }
}

```

### 6.2.1 Shallow Copy

The lines of interest in the BookTest class main method are...

```
Catalogue newCatalogue = theCatalogue.copy();
aBook = theCatalogue.getBook("11-22-33");
aBook.setPrice(3456);
```

the price of this book in  
theCatalogue is changed

which causes the following method to be executed, our first version of a copy() method for the Catalogue class.

```
public Catalogue copy() {
    Catalogue theCopy = new Catalogue();
    Enumeration e = theBookList.elements();
    while (e.hasMoreElements()) theCopy.addBook((Book)e.nextElement());
    return theCopy;
}
```

Here's the output from a run of BookTest,

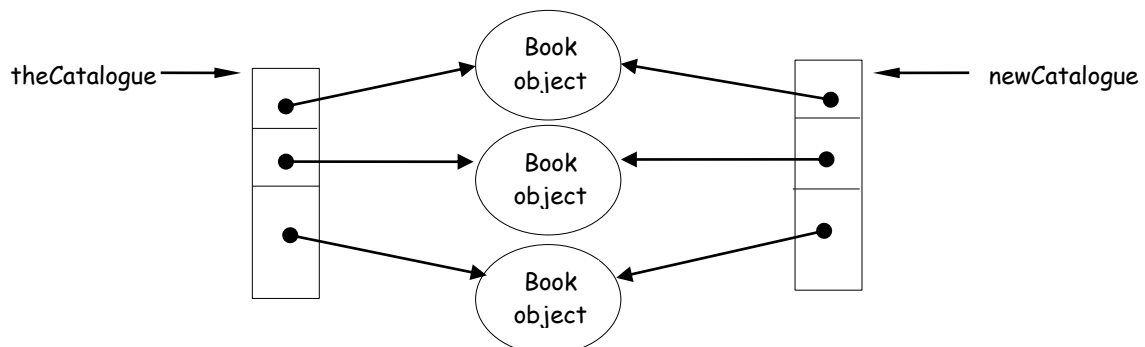
```
C:\dump>java BookTest
theCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

```
newCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

```
theCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 3456
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

```
newCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 3456
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

A new Catalogue is created and all the elements of the instance variable (theBookList, a Vector) from the original Catalogue (theCatalogue) are copied to the corresponding instance variable of the newCatalogue. Since these elements are pointers the result (as a picture) is as follows,



Although we have created a new Catalogue object (newCatalogue) we have not created any new Book objects and hence both Catalogue objects reference the same 'set' of Book objects. This is why the change of price of the 'Gray' book in theCatalogue object also effected the change of price of the 'Gray'book in the newCatalogue object..

In a case like this we say a **shallow copy** has been created.

### 6.2.2 Deep Copy

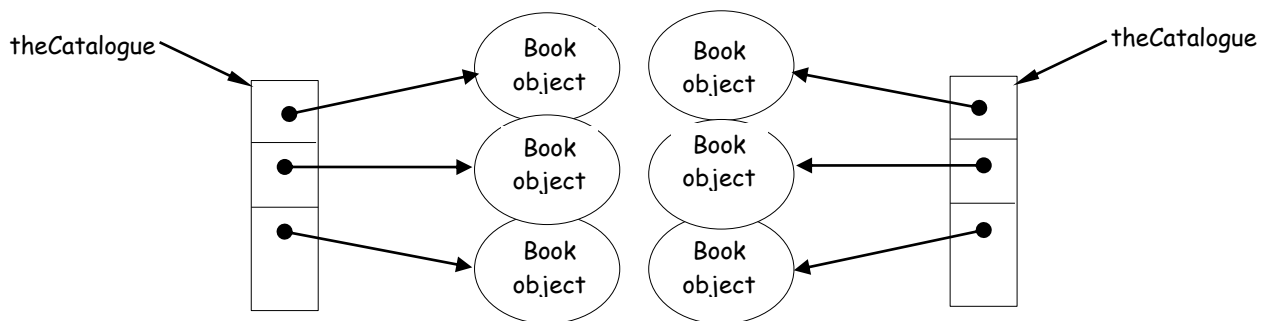
In order to create a deep copy we have a bit more work to do. This time we have to create new Book objects as well.

Again the lines of interest in the BookTest class main method are...

```
Catalogue newCatalogue = theCatalogue.copy();
```

and our new version of copy is,

```
public Catalogue copy() {
    Book tempBook, bookCopy;
    Catalogue theCopy = new Catalogue();
    Enumeration e = theBookList.elements();
    while (e.hasMoreElements()) {
        tempBook = (Book)e.nextElement();
        bookCopy = new Book(tempBook.getAuthor(), tempBook.getISBN(), tempBook.getTitle(),
tempBook.getPrice());
        theCopy.addBook(bookCopy);
    }
    return theCopy;
}
```



We now have a true (?) copy of the original object. This is called a **deep copy**. Here's the output from a run of BookTest with the new version of copy(),

```
C:\dump>java BookTest
theCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000

newCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000

theCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 3456
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000

newCatalogue
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```



### 6.2.3 The Cloneable Interface

A class implements the Cloneable interface to indicate to the clone method in class Object that it is legal for that method to make a field-for-field copy of instances of that class. Attempts to clone instances that do not implement the Cloneable interface result in the exception CloneNotSupportedException being thrown.

So this means that if you want to be able to make copies of your objects your class should implement the Cloneable interface. Here's how you might do it...

#### 6.2.3.1 Example 1

```
public class A implements Cloneable {
    private String theName;
    private int age;

    public A(String aName, int anAge) {
        this.setName(aName);
        age = anAge;
    }

    public void setName(String aName) { theName = aName; }

    public String getName() { return theName; }

    public int getAge() { return age; }

    public Object clone() {
        try { return (A)super.clone(); }
        catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError();
        }
    }

    public String toString() {
        return "Name = " + this.getName() + " Age = " + this.getAge();
    }
}

public class Atest {
    public static void main(String args[]) {
        A p = new A("Jeff",21);
        A q;
        q = (A)p.clone();
        System.out.println("For p :: " + p);
        System.out.println("For q :: " + q);

        p.setName("Pat");
        System.out.println("For p :: " + p);
        System.out.println("For q :: " + q);
    }
}
```

### 6.2.3.2 Example 2

If we remove the copy() method from the Catalogue class and make the Catalogue class implement the Cloneable interface we can then provide a clone() method for the class, as follows,

#### public class Catalogue implements Cloneable

and implement the clone() method as,

```
public Object clone() {
    try { return (Catalogue)super.clone(); }
    catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}
```

Here's the output from a run of BookTest, with the line  
Catalogue newCatalogue = theCatalogue.copy();

replaced by the line,

```
Catalogue newCatalogue = (Catalogue)theCatalogue.clone();
```

```
C:\dump>java BookTest
```

```
theCatalogue
```

```
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
```

```
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
```

```
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

```
newCatalogue
```

```
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
```

```
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
```

```
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

```
theCatalogue
```

```
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
```

```
Author : Gray ISBN : 11-22-33 Title : The Return Price : 3456
```

```
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

```
newCatalogue
```

```
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
```

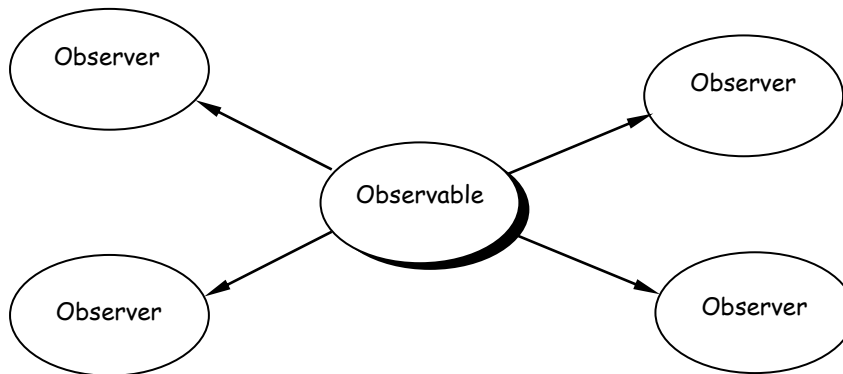
```
Author : Gray ISBN : 11-22-33 Title : The Return Price : 3456
```

```
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
```

So what kind of 'copy' does the clone() method make ???

### 6.3 Broadcasting Mechanisms - Observable Class and Observer Interface

Using the Observable class and Observer interface it is possible to create situations where changes in one object (the Observable object) can be broadcast to other interested objects (the Observer objects).



From the Java API...

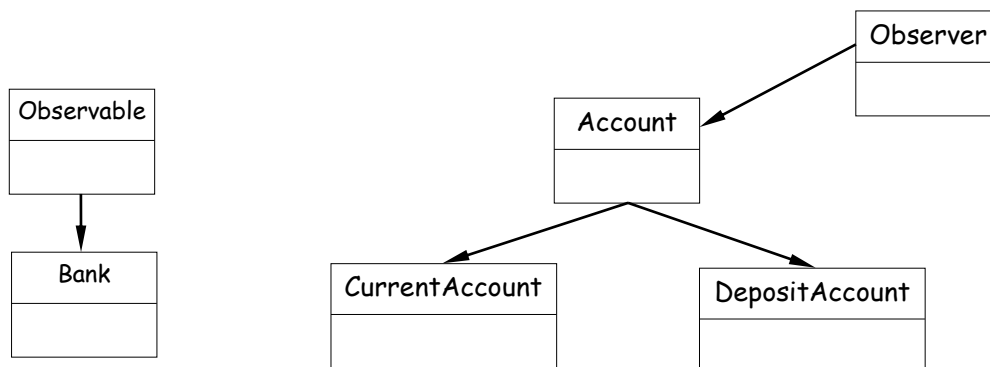
public class Observable extends Object

This class represents an observable object. It can be subclassed to represent an object that the application wants to have observed. An observable object can have one or more observers. After an observable instance changes, an application calling the Observable's `notifyObservers()` method causes all of its observers to be notified of the change by a call to their `update()` method.

A class can implement the Observer interface when it wants to be informed of changes in observable objects. The class whose instances are to be observers must implement the Observer interface and provide an implementation of the `update()` method. This method is called whenever the observed object is changed. An application calls an observable object's `notifyObservers()` method to have all the object's observers notified of the change.

In our example that follows, the application is the test class.

Some formatting on the double output would help, but is ignored in order to keep the demo as simple as possible.



```

import java.util.*;
public class Bank extends Observable {
    private double theBaseRate;

    public Bank() {
        this.setBaseRate(0);
    }
  }
  
```

```

public void setBaseRate(double theRate) {
    theBaseRate = theRate;
    this.setChanged();
    System.out.println("From the Bank object... the baseRate is changed to : " + this.getBaseRate());
}
public double getBaseRate() {
    return theBaseRate;
}
}

```

**Important : note this... must flag the change in the Observable object.**

```

import java.util.*;
public class Account implements Observer {
    private String theHolderName;
    private double currentRate;
    private double theRateFactor;

    public Account(String aName) {
        this.setHolderName(aName);
        this.setCurrentRate(0.0);
    }

    public void setHolderName(String aName) {
        theHolderName = aName;
    }

    public String getHolderName() {
        return theHolderName;
    }

    public void setCurrentRate(double baseRate) {
        currentRate = (1 + theRateFactor) * baseRate;
    }

    public double getCurrentRate() {
        return currentRate;
    }

    public void setRateFactor(double anAmount) {
        theRateFactor = anAmount;
    }

    public void update(Observable o, Object arg) {
        double newBaseRate = ((Double)arg).doubleValue();
        this.setCurrentRate(newBaseRate);
    }
}

public class CurrentAccount extends Account {
    public CurrentAccount(String aName, double anAmount) {
        super(aName);
        this.setRateFactor(anAmount);
    }

    public String toString() {
        String tempStr = "";
        tempStr = "(Current Account) :: Name = " + this.getHolderName() + " : Rate = " +
this.getCurrentRate();
        return tempStr;
    }
}

```

```

public class DepositAccount extends Account {
    public DepositAccount(String aName, double anAmount) {
        super(aName);
        this.setRateFactor(anAmount);
    }

    public String toString() {
        String tempStr = "";
        tempStr = "(Deposit Account) :: Name = " + this.getHolderName() + " : Rate = " +
this.getCurrentRate();
        return tempStr;
    }
}

```

```

public class TestBank {    // 'the application'
    public static void main(String args[]) {
        Bank theBank = new Bank();
        CurrentAccount jeffsCurrentAcc = new CurrentAccount("jeff",0.005);
        CurrentAccount patsCurrentAcc = new CurrentAccount("pat",0.005);
        theBank.addObserver(jeffsCurrentAcc);
        theBank.addObserver(patsCurrentAcc);

        DepositAccount jeffsDepositAcc = new DepositAccount("jeff",0.010);
        DepositAccount patsDepositAcc = new DepositAccount("pat",0.010);
        theBank.addObserver(jeffsDepositAcc);
        theBank.addObserver(patsDepositAcc);

        theBank.setBaseRate(5.00);
        if (theBank.hasChanged()) {
            System.out.println("Bank has changed...");
            theBank.notifyObservers(new Double(theBank.getBaseRate()));
        }

        System.out.println(jeffsCurrentAcc);
        System.out.println(patsCurrentAcc);

        System.out.println(jeffsDepositAcc);
        System.out.println(patsDepositAcc);
    }
}

```

output from a run was...

```

C:\dump>java TestBank
From the Bank object... the baseRate is changed to : 0.0
From the Bank object... the baseRate is changed to : 5.0
Bank has changed...
(Current Account) :: Name = jeff : Rate = 5.024999999999995
(Current Account) :: Name = pat : Rate = 5.024999999999995
(Deposit Account) :: Name = jeff : Rate = 5.05
(Deposit Account) :: Name = pat : Rate = 5.05

```