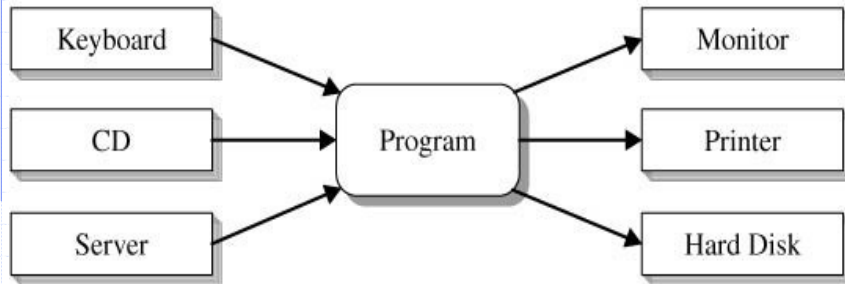


Simple Java I/O

Streams

- ◆ All modern I/O is stream-based
- ◆ A **stream** is a connection to a source of data or to a destination for data (sometimes both)
- ◆ An input stream may be associated with the keyboard
- ◆ An input stream or an output stream may be associated with a file
- ◆ Different streams have different characteristics:
 - A file has a definite length, and therefore an end
 - Keyboard input has no specific end

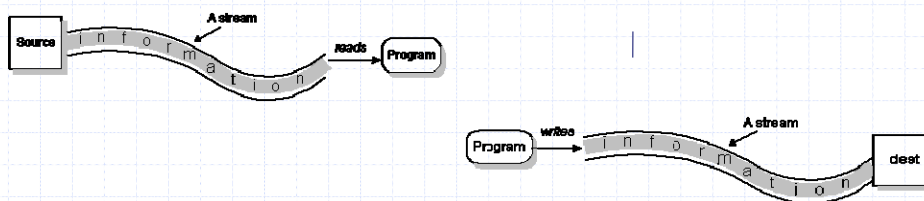
Sources and Destinations



Sources for input streams and destinations for output streams.

Types of Streams

- ◆ If data flows from a source into a program, it is called an *input* stream.
- ◆ If data flows from a program to a destination, it is called an *output* stream.
- ◆ The basic stream classes are defined in the package "java.io".



Streams

- ◆ I/O streams carry data.
 - Text streams have **character data** such as an HTML file or a Java source file.
 - Binary streams have **byte data** that may represent a graphic or executable code, such as a Java .class file.
- ◆ A stream carries data from a source to a destination in FIFO mode.

How to do I/O

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

- ◆ *Open* the stream
- ◆ *Use* the stream (read, write, or both)
- ◆ *Close* the stream

Opening a stream

- ◆ There is data external to your program that you want to get, or you want to put data somewhere outside your program
- ◆ When you open a stream, you are making a connection to that external place (then forgotten)

A `FileReader` is used to connect to a file that will be used for input:

```
FileReader fileReader =  
    new FileReader(fileName);
```

- ◆ The `fileName` specifies where the (external) file is to be found
- ◆ `fileName` not used again; use `fileReader`

Using a stream

- ◆ Some streams can be used only for input, others only for output, still others for both
- ◆ *Using* a stream means doing input from it or output to it
- ◆ manipulate the data as it comes in or goes out

```
int ch;  
ch = fileReader.read( );
```

- ◆ The `fileReader.read()` method reads one character and returns it *as an integer*, or -1 if there are no more characters to read
- ◆ The meaning of the integer depends on the file encoding (ASCII, Unicode, other)

Manipulating the input data

- ◆ Reading characters as integers is not usually what you want to do
- ◆ A `BufferedReader` will convert integers to characters; it can also read whole lines
- ◆ The constructor for `BufferedReader` takes a `FileReader` parameter:

```
BufferedReader bufferedReader =  
    new BufferedReader(fileReader);
```

Reading lines

```
String s;  
s = bufferedReader.readLine( );
```

- ◆ A `BufferedReader` will return null if there is nothing more to read

Closing

- ◆ A stream is an expensive resource
- ◆ There is a limit on the number of streams that you can have open at one time
- ◆ You should not have more than one stream open on the same file
- ◆ You must close a stream before you can open it again
- ◆ *Always close your streams!*

Text files

- ◆ Text (.txt) files are the simplest kind of files
 - text files can be used by many different programs
- ◆ Formatted text files (such as .doc files) also contain binary formatting information
- ◆ Only programs that “know the secret code” can make sense of formatted text files
- ◆ Compilers, in general, work only with text

My LineReader class

```
class LineReader {
    BufferedReader bufferedReader;

    LineReader(String fileName) {...}

    String readLine( ) {...}

    void close( ) {...}
}
```

Basics of the LineReader constructor

- ◆ Create a FileReader for the named file:

```
FileReader fileReader =
    new FileReader(fileName);
```

- ◆ Use it as input to a BufferedReader:

```
BufferedReader bufferedReader =
    new BufferedReader(fileReader);
```

- ◆ Use the BufferedReader; but first, we need to catch possible Exceptions

The full LineReader constructor

```
LineReader(String fileName) {
    FileReader fileReader = null;
    try { fileReader = new FileReader(fileName); }
    catch (FileNotFoundException e) {
        System.err.println
            ("LineReader can't find input file:" + fileName);
        e.printStackTrace( );
    }
    bufferedReader = new BufferedReader(fileReader);
}
```

readLine

```
String readLine( ) {
    try {
        return bufferedReader.readLine( );
    }
    catch(IOException e) { e.printStackTrace( );}
    return null;
}
```

close

```
void close() {
    try {
        bufferedReader.close( );
    }
    catch(IOException e) { }
}
```


How did I figure that out?

- ◆ I wanted to read lines from a file
- ◆ I found a `readLine` method in the `BufferedReader` class
- ◆ The constructor for `BufferedReader` takes a `Reader` as an argument
- ◆ An `InputStreamReader` is a kind of `Reader`
- ◆ A `FileReader` is a kind of `InputStreamReader`

The `LineWriter` class

```
class LineWriter {
    PrintWriter printWriter;

    LineWriter(String fileName) {...}

    void writeLine(String line) {...}

    void close( ) {...}
}
```

The constructor for LineWriter

```
LineWriter(String fileName) {  
    try {  
        printWriter =  
            new PrintWriter(  
                new FileOutputStream(fileName), true);  
    }  
    catch(Exception e) {  
        System.err.println("LineWriter can't " +  
            "use output file: " + fileName);  
    }  
}
```

Flushing the buffer

- ◆ When you put information into a buffered output stream, it goes into a **buffer**
- ◆ The buffer may not be written out right away
- ◆ If your program crashes, you may not know how far it got before it crashed
- ◆ **Flushing** the buffer is forcing the information to be written out

PrintWriter

- ◆ Buffers are automatically flushed when the program ends normally
- ◆ Usually it is your responsibility to flush buffers if the program does not end normally
- ◆ PrintWriter can do the flushing for you

```
public PrintWriter(OutputStream out,  
                  boolean autoFlush)
```

writeLine

```
void writeLine(String line) {  
    printWriter.println(line);  
}
```

close

```
void close( ) {  
    printWriter.flush( );  
    try { printWriter.close( ); }  
    catch(Exception e) { }  
}
```

Class Reader

◆ **Constructor:** `protected Reader();`

◆ **Main Methods**

- `public int read() throws IOException:`
 - ◆ single character or -1 if at end
- `public int read(char[] cbuf) throws IOException:`
 - ◆ cbuf is the destination buffer, returns no. characters or -1 if EOS
 - ◆ reads characters from stream in cbuf until array full, error or EOS
- `public abstract int read(char[] cbuf, int off, int len) throws IOException:`
 - ◆ cbuf is the destination buffer, returns no. characters or -1 if EOS
 - ◆ reads characters from stream in cbuf until array full, error or EOS
 - ◆ off is the initial position, len the max no. characters to read
- `public abstract void close() throws IOException:`
 - ◆ Closes the stream

Class InputStream

◆ **Constructor:** `public InputStream();`

◆ Same as before, but to read byte and array of bytes:

- `public int read() throws IOException`
- `public int read(byte[] cbuf) throws IOException`
- `public abstract int read(byte[] cbuf, int off, int len) throws IOException`
- `public abstract void close() throws IOException`

Class Writer

◆ **Constructor:** `protected Writer();`

◆ **Main Methods:**

- `public void write(int c) throws IOException`
- `public void write(char[] cbuf) throws IOException`
- `public abstract void write(char[] cbuf, int off, int len) throws IOException`
 - ◆ `cbuf`: array of characters, `off` initial position, `len` max no. to write
- `public abstract void flush() throws IOException`
 - ◆ buffer `cbuf` is immediately emptied
- `public abstract void close() throws IOException`
 - ◆ closes the stream

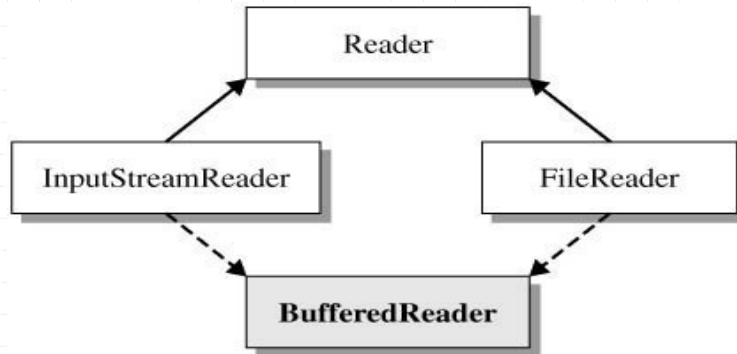
Class OutputStream

◆ **Constructor:** `public OutputStream();`

◆ **Main Methods:**

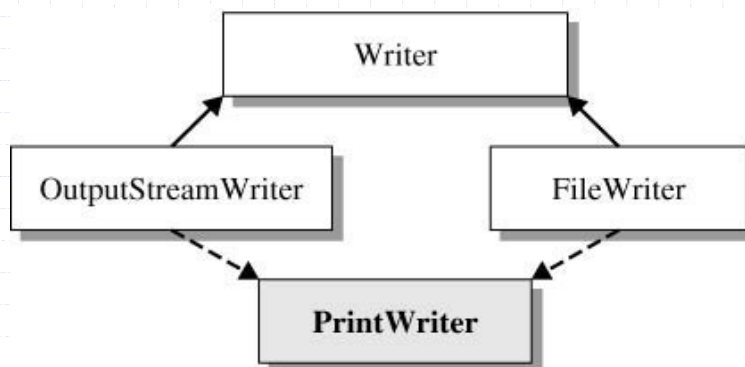
- `public void write(byte[] b) throws IOException`
 - ◆ writes `b.length()` bytes of `b` on the stream
- `public void write(byte[] b, int off, int len) throws IOException`
 - ◆ `b` array of bytes, `off` the initial position, `len` no. bytes to write
- `public abstract void write(int b) throws IOException`
- `public abstract void flush() throws IOException`
 - ◆ buffer is immediately emptied
- `public abstract void close() throws IOException`
 - ◆ closes the stream

The Reader Hierarchy



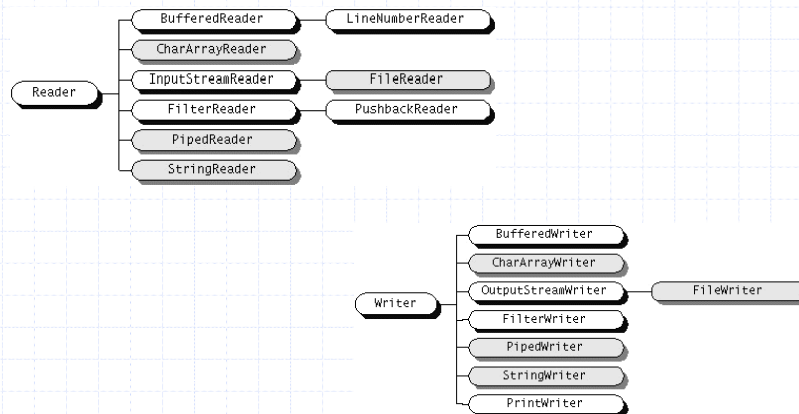
The abstract Reader class with subclasses for buffered input from a file or the keyboard.

Writer Inheritance Hierarchy



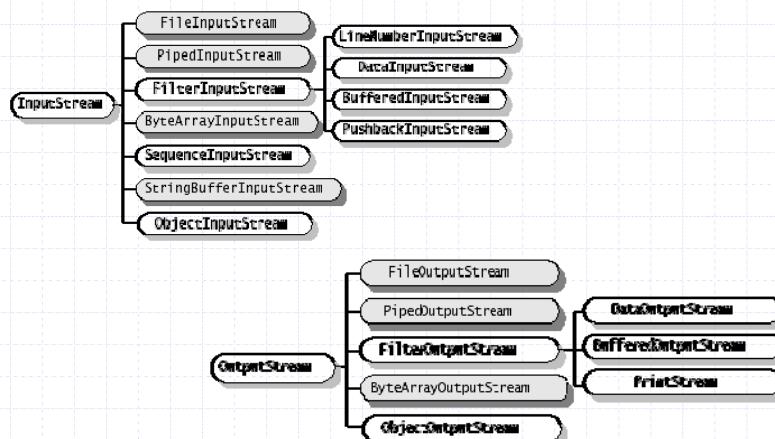
Writer inheritance hierarchy with physical character stream classes and filter class PrintWriter.

Stream of characters



- ◆ grey: only read/write
- ◆ white: other processing as well

Stream of bytes



- ◆ grey: only read/write
- ◆ white: other processing as well

FILES

- ◆ In Java they are represented by the class FILE
- ◆ abstract representation of files and directory
- ◆ Methods to manipulate files/directory, but not to read/write
- ◆ To read/write from/on file, need to associate a stream to the file
- ◆ next:
 - FileInputStream/FileOutputStream
 - Useful Methods of the class FILE

FileInputStream/FileOutputStream

- ◆ Subclasses of InputStream and OutputStream
- ◆ Open stream of byte from/to file
- ◆ same methods as InputStream and OutputStream
- ◆ Constructors:
 - `public FileInputStream(File file) throws FileNotFoundException`
 - `public FileInputStream(String name) throws FileNotFoundException`
 - `public FileOutputStream(File file) throws FileNotFoundException`
 - `public FileOutputStream(String name) throws FileNotFoundException`
 - `public FileOutputStream(String name, boolean append) throws FileNotFoundException`

FileReader/FileWriter

- ◆ Subclasses of Reader and Writer
- ◆ Open stream of characters from/to file
- ◆ same methods as Reader and Writer
- ◆ Costruttori:
 - `public FileReader(File file) throws FileNotFoundException`
 - `public FileReader(String name) throws FileNotFoundException`
 - `public FileWriter(File file) throws FileNotFoundException`
 - `public FileWriter(String name) throws FileNotFoundException`
 - `public FileWriter(String name, boolean append) throws FileNotFoundException`

Class FILE

- ◆ Abstract representation of file and directory
- ◆ Methods to manipulate file and directory (create, destroy etc.)
- ◆ Does not contain methods to read/write
- ◆ Constructors:
 - `public File(String pathname) throws NullPointerException`
 - `public File(File parent, String child) throws NullPointerException`
 - `public File(String parent, String child) throws NullPointerException`
- ◆ If the file does not exist, it is effectively created when a stream to it is open

Class FILE

◆ Main methods:

- `public boolean delete() throws SecurityException`
- `public boolean exists() throws SecurityException`
- `public String getAbsolutePath()`
- `public String getName()`
- `public boolean isDirectory()`
- `public boolean isFile()`
- `public long length()`
- `public String[] list()`
- `public File[] listFiles()`
- `public static File[] listRoots()`
- `public boolean mkdir() throws SecurityException`
- `public boolean mkdirs() throws SecurityException`

◆ SecurityException is a subclass of RuntimeException

The Scanner Class

◆ A Scanner object partitions text from an input stream into tokens by means of its "next" methods.

◆ Declare a Scanner object as follows:

```
Scanner keyIn = new Scanner(System.in);
Scanner fileIn = new Scanner(
    new FileReader("demo.dat"));
```

Scanner Methods



```
Shortcut to cmd.exe
C:\> 17 deposit 450.75 false A
```

- ◆ `String line = sc.nextLine(); // whole line`
- ◆ `int i = sc.nextInt(); // i = 17`
- ◆ `String str = sc.next(); // str = "deposit"`
- ◆ `double x = sc.nextDouble(); // x = 450.75`
- ◆ `boolean b = sc.nextBoolean(); // b = false`
- ◆ `char ch = sc.next().charAt(0); // ch = 'A'`

Scanner Class API

class SCANNER	java.util
Constructors	
	Scanner ((InputStream source) Creates a Scanner object that produces values read from the specified input stream (Typically standard input System.in that denotes the keyboard)
	Scanner (Readable source) Creates a Scanner object that produces values read from the specified input stream (Typically a FileReader that denotes a file)

Scanner Class API (2)

Methods	
void	close() Close the scanner.
boolean	hasNext() Returns true if the scanner has another token in the input stream
boolean	hasNextBoolean() Returns true if the next token in the input stream can be interpreted as a boolean value
boolean	hasNextDouble() Returns true if the next token in the input stream can be interpreted as a double value
boolean	hasNextInt() Returns true if the next token in the input stream can be interpreted as an int value

Scanner Class API (end)

String	next() Finds and returns the next complete token in the input stream as a String.
boolean	nextBoolean() Scans the next token in the input stream into a boolean value and returns that value.
double	nextDouble() Scans the next token in the input stream into a double value and returns that value.
int	nextInt () Scans the next token in the input stream into an int value and returns that value.

Testing for Scanner Tokens

```
// loop reads tokens in the line
while (sc.hasNext())
{
    token = sc.next();
    System.out.println("In loop next token = " + token);
}
```

Output:

```
In loop next token = 17
In loop next token = deposit
In loop next token = 450.75
In loop next token = false
In loop next token = A
```

File Input using Scanner

A sports team creates the file "attendance.dat" to store attendance data for home games during the season. The example uses the Scanner object `dataIn` and a loop to read the sequence of attendance values from the file and determine the total attendance. The condition `hasNextInt()` returns false when all of the data has been read from the file.

```
// create an Scanner object attached to the file "attendance.dat"
Scanner dataIn = new Scanner(new FileReader("attendance.dat"));
int gameAtt, totalAtt = 0;

// the loop reads the integer game attendance until end-of-file
while(dataIn.hasNextInt()) // loop iteration condition
{
    gameAtt = dataIn.nextInt(); // input next game attendance
    totalAtt += gameAtt; // add to the total
}
```

Program 2.1

```
import java.util.Scanner;
import java.io.*;
import java.text.DecimalFormat;

public class Program2_1 {
    public static void main(String[] args)
    {
        final double SALESTAX = 0.05;

        // input streams for the keyboard and a file
        Scanner fileIn = null;
        // input variables and pricing information
        String product;
        int quantity;
        double unitPrice, quantityPrice, tax,
            totalPrice;
        char taxStatus;
```

Program 2.1 (2)

```
        // create formatted strings for
        // aligning output
        DecimalFormat fmtA = new DecimalFormat("#"),
            fmtB = new DecimalFormat("$#.00");

        // open the file; catch exception
        // if file not found
        // use regular expression as delimiter
        try
        {
            fileIn =
                new Scanner(new FileReader("food.dat"));
            fileIn.useDelimiter("[\\t\\n\\r]+");
        }
```

Program 2.1 (3)

```
catch (IOException ioe)
{
    System.err.println("Cannot open file " +
        "'food.dat'");
    System.exit(1);
}

// header for listing output
System.out.println("Product" +
    align("Quantity", 16) +
    align("Price", 10) +
    align("Total", 12));
```

Program 2.1 (4)

```
// read to end of file; break
// when no more tokens
while(fileIn.hasNext())
{
    // input product/purchase input fields
    product = fileIn.next();
    quantity = fileIn.nextInt();
    unitPrice = fileIn.nextDouble();
    taxStatus = fileIn.next().charAt(0);

    // calculations and output
    quantityPrice = unitPrice * quantity;
    tax = (taxStatus == 'Y') ?
        quantityPrice *
        SALESTAX : 0.0;
    totalPrice = quantityPrice + tax;
```

Program 2.1 (end)

```
        System.out.println(product +
            align("", 15-product.length()) +
            align(fmtA.format(quantity), 6) +
            align(fmtB.format(unitPrice), 13) +
            align(fmtB.format(totalPrice),12) +
            ((taxStatus == 'Y') ? " *" : ""));
    }
}
// aligns string right justified in a field of width n
public static String align(String str, int n) {
    String alignStr = "";
    for (int i = 1; i < n - str.length(); i++)
        alignStr += " ";
    alignStr += str;
    return alignStr;
}
}
```

Program 2.1 Run

```
Input file ('food.dat')
Soda      3 2.69 Y
Eggs      2 2.89 N
Bread     3 2.49 N
Grapefruit 8 0.45 N
Batteries 10 1.15 Y
Bakery    1 14.75 N
```

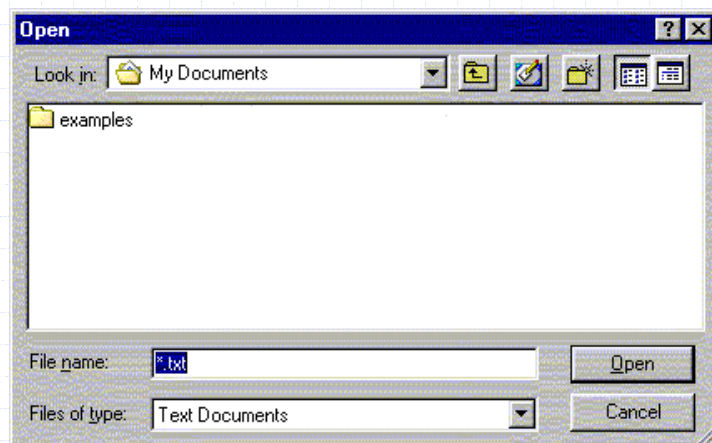
Run:

Product	Quantity	Price	Total
Fruit Punch	4	\$2.69	\$11.30 *
Eggs	2	\$2.89	\$5.78
Rye Bread	3	\$2.49	\$7.47
Grapefruit	8	\$.45	\$3.60
AA Batteries	10	\$1.15	\$12.08 *
Ice Cream	1	\$3.75	\$3.75

About FileDialogs

- ◆ The FileDialog class displays a window from which the user can select a file
- ◆ The FileDialog window is **modal**--the application cannot continue until it is closed
- ◆ Only applications, not applets, can use a FileDialog; only applications can access files
- ◆ Every FileDialog window is associated with a Frame

Typical FileDialog window



FileDialog constructors

- ◆ `FileDialog(Frame f)`
 - Creates a FileDialog attached to Frame `f`
- ◆ `FileDialog(Frame f, String title)`
 - Creates a FileDialog attached to Frame `f`, with the given title
- ◆ `FileDialog(Frame f, String title, int type)`
 - Creates a FileDialog attached to Frame `f`, with the given title; the type can be either `FileDialog.LOAD` or `FileDialog.SAVE`

Useful FileDialog methods I

- ◆ `String getDirectory()`
 - Returns the selected directory
- ◆ `String getFile()`
 - Returns the name of the currently selected file, or null if no file is selected
- ◆ `int getMode()`
 - Returns either `FileDialog.LOAD` or `FileDialog.SAVE`, depending on what the dialog is being used for

Useful FileDialog methods II

- ◆ `void setDirectory(String directory)`
 - Changes the current directory to `directory`
- ◆ `void setFile(String fileName)`
 - Changes the current file to `fileName`
- ◆ `void setMode(int mode)`
 - Sets the mode to either `FileDialog.LOAD` or `FileDialog.SAVE`

Serialization

- ◆ You can also read and write *objects* to files
- ◆ Object I/O goes by the awkward name of **serialization**
- ◆ Serialization in other languages can be *very* difficult, because objects may contain references to other objects
- ◆ Java makes serialization (almost) easy

Conditions for serializability

- ◆ If an object is to be serialized:
 - The class must be declared as public
 - The class must implement Serializable
 - The class must have a no-argument constructor
 - All fields of the class must be serializable: either primitive types or serializable objects

Implementing the Serializable interface

- ◆ To “implement” an interface means to define all the methods declared by that interface, but...
- ◆ The Serializable interface does not define any methods!
 - Question: What possible use is there for an interface that does not declare any methods?
 - Answer: Serializable is used as flag to tell Java it needs to do extra work with this class

Writing objects to a file

```
ObjectOutputStream objectOut =  
    new ObjectOutputStream(  
        new BufferedOutputStream(  
            new FileOutputStream(fileName)));  
  
objectOut.writeObject(serializableObject);  
  
objectOut.close( );
```

Reading objects from a file

```
ObjectInputStream objectIn =  
    new ObjectInputStream(  
        new BufferedInputStream(  
            new FileInputStream(fileName)));  
  
myObject = (itsType)objectIn.readObject( );  
  
objectIn.close( );
```