CM1203 & CM1207 – Slides 8

Reading and writing files – part 2

Matt Williams

School of Computer Science
Cardiff University

CM1203 & CM1207 – Spring 2013

Recap: text and binary file formats

Java provides two ways to store data in files – *text* format and *binary* format.

Definition

In a *plain text* file data is represented as a sequence of characters, where each character is encoded as a particular sequence of bits. ASCII and Unicode are two common plain text *encodings*.

Definition

In a *binary* file data is represented as bytes. Each byte is composed of eight bits and can denote one of 256 values.

The `Reader` and `Writer` classes

- Previously, we used `Scanner( File )` to read text files. This is a convenience function provided by `Scanner`
- The package `java.io` provides many more classes for reading and writing text and binary files.
- The `Reader` and `Writer` classes and their subclasses are used to handle **text** files.
- The `Reader` and `Writer` classes are `abstract`.
- Recall:

```
public abstract SomeClass {
   abstract int someMethod() {}
}
```

- To perform text input/output we use one the subclasses of `Reader` or `Writer`

Reader and Writer

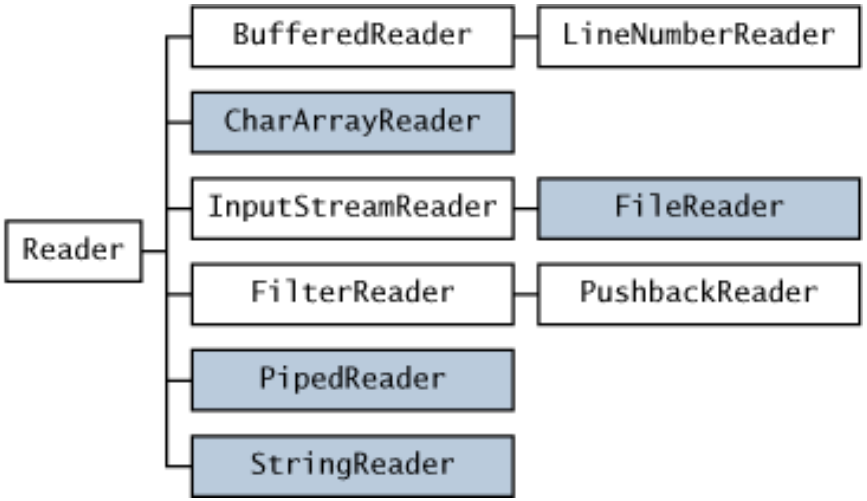
Example

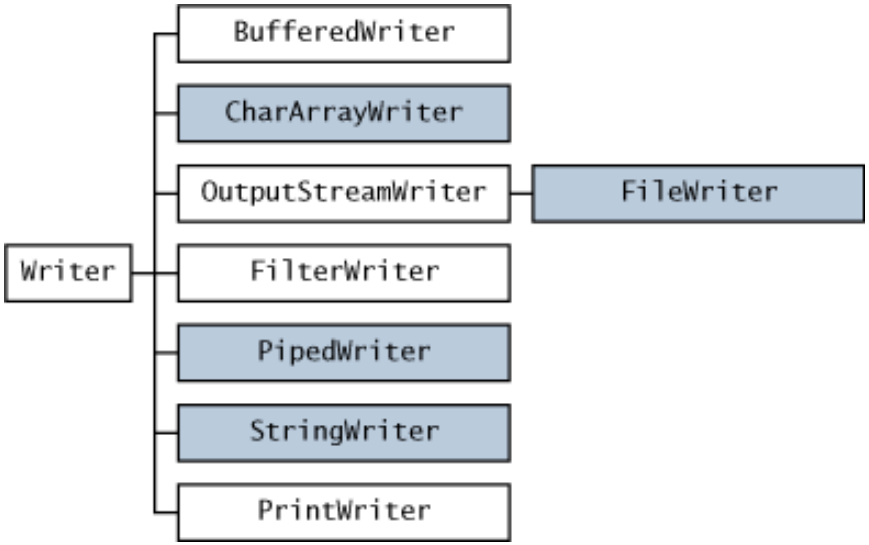
Reader

Example

Writer

Subclasses of `Reader` and `Writer`





Reading a file

Example

`FileReader`: constructs a `Reader` that can read from a file. Only provides low-level `read` methods.

`BufferedReader`: wraps around a `FileReader` and provides a high-level `readLine` method.

From: ReaderTest.java

```
import java.io.*;

public class ReaderTest {

    public static void main( String[] args )
        try {
            FileReader reader = new FileReader( "input.txt" );
            BufferedReader in = new BufferedReader( reader );

            String s;
            while ( (s = in.readLine()) != null )
                System.out.println( new StringBuffer(s).reverse() );
            }

            in.close();
        }
        catch ( FileNotFoundException e ) { // may be throw by new FileReader(...)
              System.out.println( e );
        }
        catch ( IOException e ) {  // may be thrown by readLine()
              System.out.println( e );
        }
    }
}
```

From WriterTest.java

```
import java.io.*;
import java.util.Date;

public class WriterTest {

    public static void main( String[] args ) {

        try {

            FileWriter writer = new FileWriter( "output.txt" );
            PrintWriter out = new PrintWriter( writer );

            out.println( "Hello world!" );
            out.print( new Date() );
            out.println();

            out.close();
        }
        catch ( Exception e ) {  // lazy exception handling!
            System.out.println( e );
        }
    }

}
```

Saving the PhoneBook

savePhoneBook method

From PhoneBook-Text → AddEntry.java

```
public static void savePhoneBook( String filename, PhoneBook pb ) throws Exception {
    FileWriter writer = new FileWriter( filename );  // no need for File!
    PrintWriter out = new PrintWriter( writer );

    for( PhoneBookEntry pbe : pb.getEntries() ) {
        String name = pbe.getName();
        String number = pbe.getNumber();
        String line = name + "," + number;
        out.println( line );
    }

    out.close();
}
```

Saving the PhoneBook
main method

From PhoneBook-Text → AddEntry.java

```
// Load an existing phone book...
String filename = args[0];
String name = args[1];
String number = args[2];

PhoneBook pb = loadPhoneBook( filename );

// Print the current phone book
System.out.println( "Phone book from file..." );
System.out.println( pb );

// Add the entry
pb.add( name, number );

// Print updated phone book
System.out.println( "Updated phone book..." );
System.out.println( pb );

savePhoneBook( filename, pb );
```

Reading and writing binary files

Use subclasses of `InputStream` and `OutputStream` classes:

```
Date d = new Date();
ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream( "test.dat" ) );
out.writeObject( d );
out.close();

ObjectInputStream in = new ObjectInputStream( new FileInputStream( "test.dat" ) );
Date sc = (Date)in.readObject();
in.close();
```

Note the object read from file must be *cast* to the correct type

Reading and writing binary files

Use subclasses of `InputStream` and `OutputStream` classes:

```
Date d = new Date();
ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream( "test.dat" ) );
out.writeObject( d );
out.close();

ObjectInputStream in = new ObjectInputStream( new FileInputStream( "test.dat" ) );
Date sc = (Date)in.readObject();
in.close();
```

Note the object read from file must be *cast* to the correct type

Binary output of our classes

Simply add `implements Serializable` to class definition

Example

```
public class PhoneBook implements Serializable
```

From: SomeClass.java

```
import java.util.Date;
import java.io.*;

public class SomeClass implements Serializable {

    int i;
    String s;
    Date d;

    public SomeClass( int j, String t ) {
        i = j;
        s = t;
        d = new Date();
    }

    public String toString() {
        return Integer.toString(i) + s + d.toString();
    }

}
```

From: `WriteTest.java`

```
import java.io.*;

public class WriteTest {

    public static void main( String[] args ) {

        SomeClass sc = new SomeClass( 15, "bob" );
        System.out.println( sc );

        try {
            ObjectOutputStream out = new ObjectOutputStream
                ( new FileOutputStream( "test.dat" ) );
            out.writeObject(sc);
            out.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

From: `ReadTest.java`

```
import java.io.*;
import java.util.Date;

public class ReadTest {

    public static void main( String[] args ) {

        try {
            ObjectInputStream in = new ObjectInputStream
                ( new FileInputStream( "test.dat" ) );
            SomeClass sc = (SomeClass)in.readObject();
            System.out.println(sc);
            in.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Phone book application

From `PhoneBookBinary` → `PhoneBookApp.java`

```
import java.io.*;
public class PhoneBookApp {
    public static void main( String[] args ) {
        PhoneBookApp pba = new PhoneBookApp(args);  }

    private PhoneBook pb;

    private PhoneBookApp( String[] args ) {
        switch (args[1].charAt(1)) {
            case 'n': createNewBook(args[0]);
                      break;
            case 'p': readPhoneBook(args[0]);
                      System.out.println(pb);
                      break;
            case 'a': readPhoneBook( args[0] );
                      pb.add( args[2], args[3] );
                      writePhoneBook( args[0] );
                      break;
            case 'f': readPhoneBook( args[0] );
                      System.out.println( pb.numberFor(args[2]) );
                      break;
            default:  break;
        }
    }
}
```

Phone book application

From `PhoneBookBinary` → `PhoneBookApp.java`

```
private void createNewBook(String filename) {
    pb = new PhoneBook();
    writePhoneBook(filename);
}

private void writePhoneBook( String filename ) {
    try {
        ObjectOutputStream out = new ObjectOutputStream
            ( new FileOutputStream( filename ) );
        out.writeObject(pb);
        out.close();
    }
    catch (IOException e) {
        System.out.println(e);
        System.out.println("Phone book not saved");
        System.exit(0);
    }
}
```

Phone book application

From `PhoneBookBinary` → `PhoneBookApp.java`

```
private void readPhoneBook( String filename ) {
    try {
        ObjectInputStream in = new ObjectInputStream
            ( new FileInputStream( filename ) );
        pb = (PhoneBook)in.readObject();
        in.close();
    }
    catch (Exception e) {
        System.out.println(e);
        System.out.println("Phone book could not be opened");
        System.exit(0);
    }
}
```

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Advantages/disadvantages of file formats

Binary

- Compact files
- Easier to code
- Includes some error checking
- Awkward if classes change

Text

- Easy for user to read
- More portable
- More complex to code
- Error checking needs to be coded

Summary

- Input and output to files in binary and text format
- Exception handling