Fundamentos de los flujos de entrada/salida

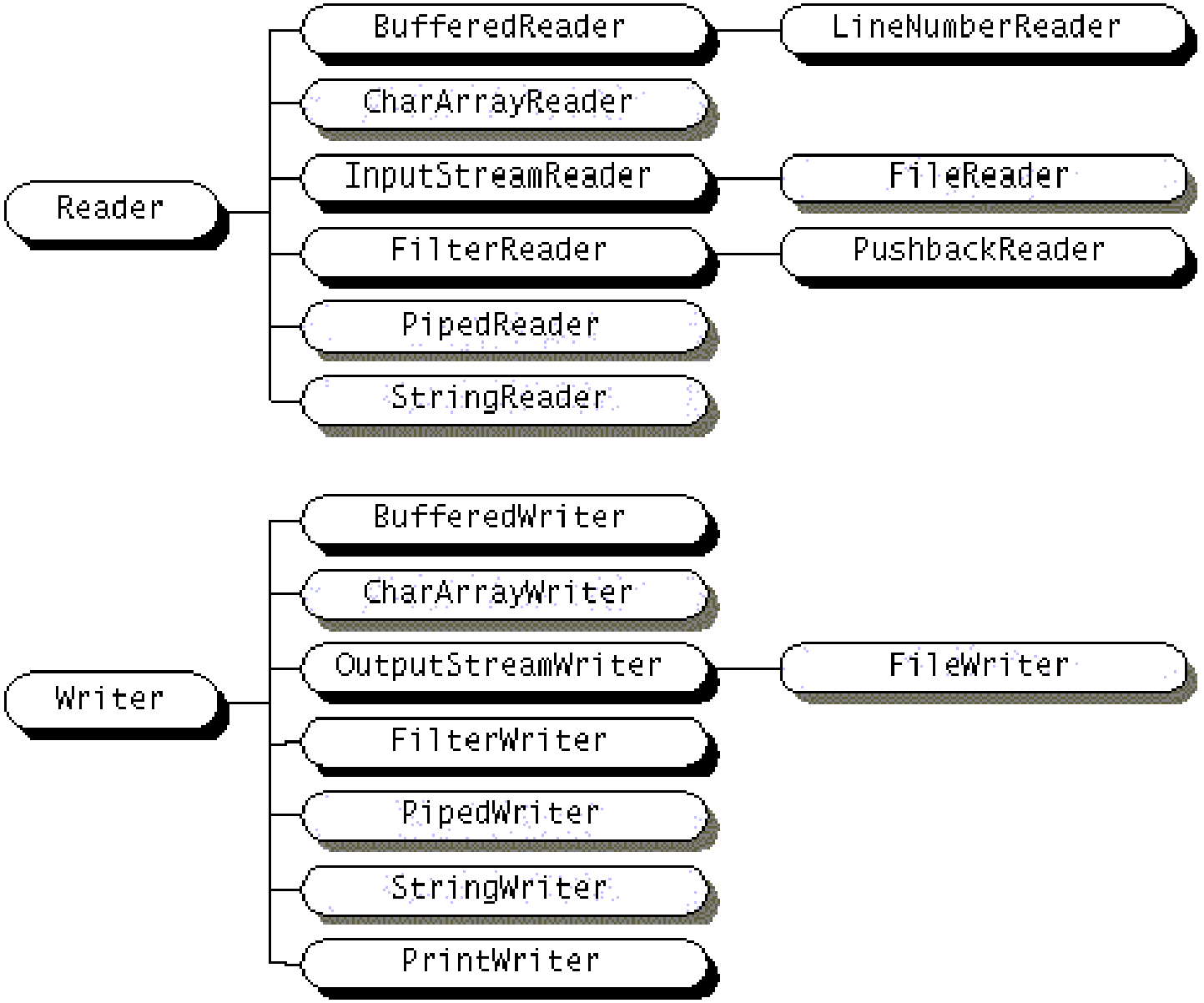
Un stream (corriente) de datos de un origen (source) a un consumidor (sink). Tipicamente su programa es el extremo de un stream, y otro nodo (por ejemplo un Archivo) es el otro extremo.

Los orígenes y consumidores son tambien llamados input streams y output streams, respectivamente. Se puede leer de un input stream pero no se puede escribir en el. Del mismo modo se puede escribir en un output stream pero no se puede leer de el.

Fundamentos de los flujos de entrada/salida

Java soporta 2 tipos de datos en los streams, bytes (InputStream,OutputStream) o caracteres unicode (Reader, Writer). Mas específicamente la entrada de caracteres es implementada por subclases de Reader y la salida de caracteres es implementada por subclases de Writer. La entrada de bytes es implementada por subclases de InputStream y la salida de bytes es implementada por subclases de OutputStream.

Fundamentos de los flujos de entrada/salida

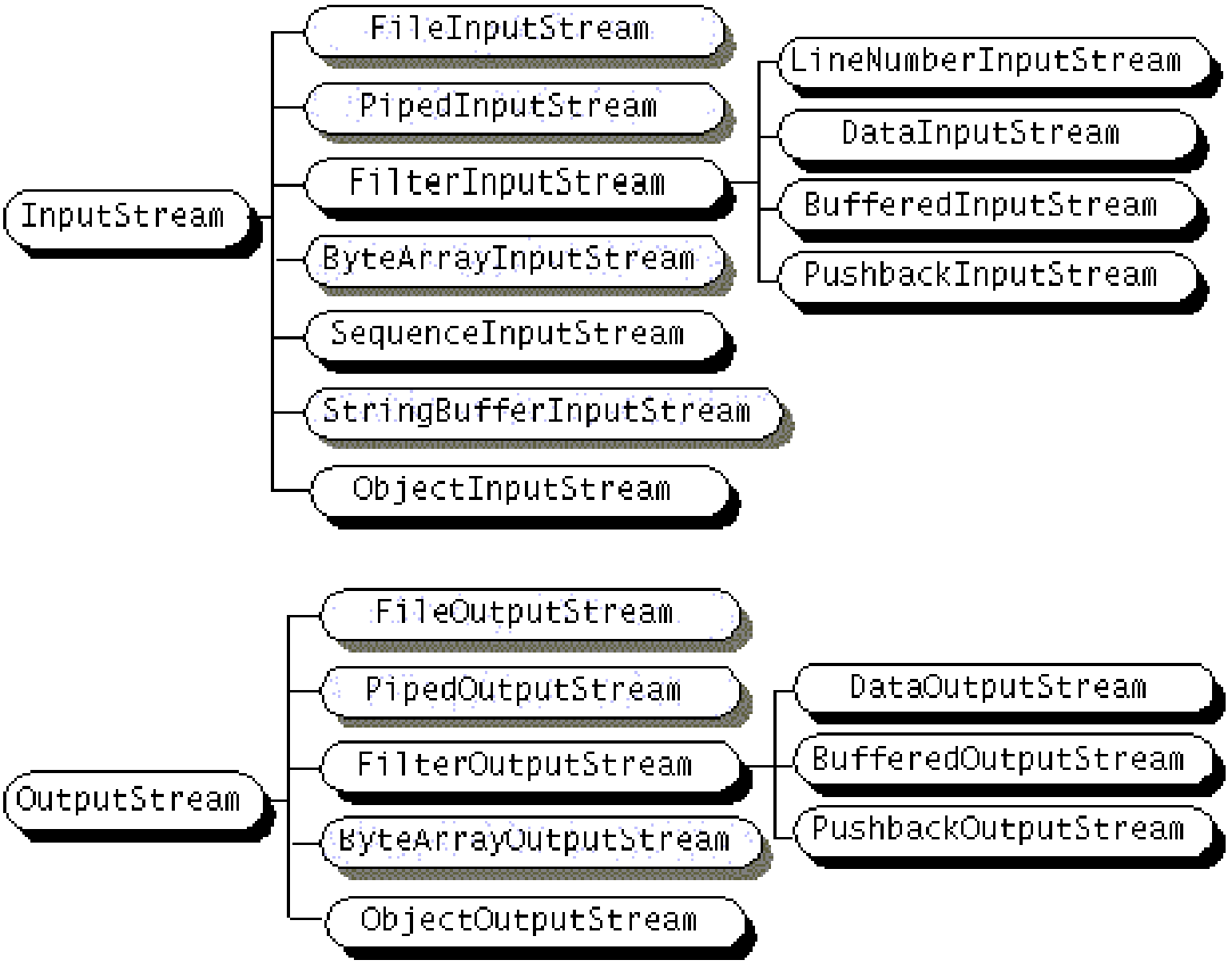


Fundamentos de los flujos de entrada/salida

Las clases InputStreamReader y OutputStreamReader son las versiones mas importantes de Reader y Writer, estas clases son utilizadas de interfaces entre streams de bytes y lectores y escritores de caracteres.
Las clases FileReader y FileWriter son streams nodos que son los equivalentes unicode de FileInputStream y FileOutputStream.
Las clases BufferedReader y BufferedWriter filtran los streams de caracteres para aumentar la eficiencia de las operaciones de entrada/salida.
Las clases StringReader y StringWriter, son clases nodo de caracteres para leer y escribir Strings de Java.
Las clases PipedReader y PipedWriter son utilizadas para la comunicación entre threads o procesos.

UNTDF
Universidad Nacional de Tierra del Fuego,
Antártida e Islas del Atlántico Sur

Fundamentos de los flujos de entrada/salida



Fundamentos de los flujos de entrada/salida

Las clases FileInputStream y FileOutputStream son streams nodos que utilizan archivos del disco
Las clases BufferedInputStream y BufferedOutputStream filtran el stream para incrementar la eficiencia de las operaciones de entrada/salida.
Las clases PipedInputStream y PipedOutputStream se utilizan para comunicar procesos o threads.
Las clases DataInputStream y DataOutputStream son llamadas stream filters y permiten leer y escribir tipos primitivos de java y algunos formatos especiales utilizando streams.
Las clases ObjectInputStream y ObjectOutputStream permiten leer o escribir objetos java a los streams.

Fundamentos de los flujos de entrada/salida

En java existen 3 tipos de nodos

- Archivos
- Memoria (arreglos o String)
- Pipes (un canal de un proceso o thread a otro, la salida de un pipe stream es conectada a una entrada pipe stream del otro thread o proceso)

Tipo	Stream de caracteres	Stream de bytes
File	FileReader FileWriter	FileInputStream FileOutputStream
Array en memoria	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
String en memoria	StringReader StringWriter	
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream

Fundamentos de los flujos de entrada/salida

El siguiente ejemplo muestra la copia de un archivo

```
1 import java.io.*;
2 public class TestBufferedStreams
3 {
4 public static void main(String[] args)
5 {
6   try
7    {
8      FileReader input=new FileReader(args[0]);
9      BufferedReader bufInput=new BufferedReader(input);
10
11    FileWriter output=new FileWriter(args[1]);
12    BufferedWriter bufOutput=new BufferedWriter(output);
13
14     String line;
```

Fundamentos de los flujos de entrada/salida

```
15    while ((line=bufInput.readLine())!=null)
16    { bufOutput.write(line,0,line.length());
17      bufOutput.newLine();
18    }
19
20    bufOutput.close();
21    bufInput.close();
22
23  } catch (IOException e) {e.printStackTrace();}
24
25 }
26}
```

Fundamentos de los flujos de entrada/salida

Un programa raramente utiliza un solo objeto stream. En su lugar, encadena una serie de flujos para procesar los datos. En el ejemplo anterior en la linea 9 se empaqueta un FileReader con un BufferedReader, para obtener la funcionalidad de BufferedReader sobre el stream que estamos leyendo, BufferedReader nos provee un método readLine (linea 15) que nos permite leer una linea completa del archivo.

Serialización

guardar un objeto en algún tipo de almacenamiento permanente es llamado persistencia.

La serialización es un mecanismo para guardar objetos como una secuencia de bytes y luego cuando se necesite reconstruir la secuencia de bytes de nuevo en una copia del objeto. Para que un objeto sea serializable debe implementar la interface java.io.Serializable. La interface Serializable no tiene métodos y solo sirve para marcar la clase indicando que la clase puede ser considerada para serialización. Cuando se serializa un Objeto solo los campos del objeto son preservados, los metodos y constructores no son parte del stream serializado. Cuando un campo es una referencia a un objeto, este también es serializado si es serializable, sino lanza una NotSerializableException.

Los atributos de clase no son serializados, tampoco los campos marcados con la palabra clave transient.

Escribiendo un Objeto a un ObjectStream

Leer y escribir un objeto a un stream es un proceso simple. El siguiente ejemplo muestra como escribir y leer un objeto de un stream.

```
import java.io.*;
import java.util.Date;
public class SerializeDate
{
   SerializeDate()
   {
      Date d=new Date();
      try {
       FileOutputStream f=new FileOutputStream("date.ser");
       ObjectOutputStream s=new ObjectOutputStream(f);
       s.writeObject(d);
       s.close();
    } catch(IOException e){e.printStackTrace();}
   }
   public static void main(String[] args)
   {
      new SerializeDate();
   }
  }
```

Leyendo un objeto de un ObjectStream

leer un objeto es tan simple como escribirlo, pero con un pequeño detalle, el método readObject() retorna un objeto del stream como un Object y debemos castearlo al nombre de clase apropiado.

```
import java.io.*;
import java.util.Date;
public class DeSerializeDate
{
    DeSerializeDate()
    {
        Date d=null;
        try
        { FileInputStream f=new FileInputStream("date.ser");
          ObjectInputStream s=new ObjectInputStream(f);
          d=(Date)s.readObject();
          s.close();
        }catch (Exception e){e.printStackTrace();}
        System.out.println("Objeto Date de date.ser: "+d);
    }
    public static void main(String[] args)
    {
        new DeSerializeDate();
    }
}
```

Fundamentos de los flujos de entrada/salida

XMLEncoder/XMLDecoder

La clase XMLEncoder es una alternativa complementaria a ObjectOutputStream y puede ser usada para generar representaciones textuales de un JavaBean del mismo modo que ObjectOutputStream puede ser usada para crear una representacion binaria de objetos serializables. A pesar de la similaridad de sus APIs, la clase XMLEncoder esta exclusivamente diseñada para el proposito de archivar grafos de JavaBeans como representaciones textuales de sus propiedades publicas. El ObjectOutputStream continua siendo recomendado para la comunicación entre procesos y serializacion de proposito general.

La clase XMLDecoder es usada para leer documentos XML creados usando XMLEncoder y es usada de la misma forma que ObjectInputStream.

JavaBeans

Los JavaBeans son componentes reusables de software de Java. En la practica,son clases escritas en lenguaje de programacion java que cumplen con una convención particular. Son usados para encapsular varios objetos en un objeto simple (el bean), de forma que pueden ser usados como un unico objeto bean en lugar de multiples objetos individuales.

Las convenciones requeridas son:

- Debe tener un constructor sin argumentos.
- Sus propiedades deben ser accesibles mediante métodos get y set que siguen una convención de nomenclatura estándar.
- Debe ser serializable

Fundamentos de los flujos de entrada/salida

Ejemplo:

```
public class PersonaBean implements java.io.Serializable {
  private String nombre;
  private int edad;

 public PersonaBean() {/*Constructor sin argumentos*/ }

//metodos accesores
  public void setNombre(String n) {this.nombre = n;}
  public void setEdad(int e) { this.edad = e;}
  public String getNombre() { return (this.nombre); }
  public int getEdad() { return (this.edad); }
}
```

Escribiendo un JavaBean a un XMLEncoder

```
import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.io.*;
public class Codifica
{
    public Codifica()
    {
      PersonaBean p=new PersonaBean();
     p.setNombre("Juan");
     p.setEdad(26);
Thread.currentThread().setContextClassLoader(getClass().getClassLoader());
// ^ linea necesaria en BlueJ
     try{
        XMLEncoder encoder = new XMLEncoder(new
FileOutputStream("PersonaXML.xml"));
        encoder.writeObject(p);
        encoder.close();
    }catch (IOException e){e.printStackTrace();}
    }

    public static void main(String args[])
    {     Codifica c=new Codifica();    }
}
```

Leyendo un JavaBean de un XMLDecoder

```
import java.beans.*;
import java.io.*;
public class Decodifica
{
  public Decodifica()
  { try{

Thread.currentThread().setContextClassLoader(getClass().getClassLoader(
));//linea necesaria en BlueJ
      XMLDecoder decoder = new XMLDecoder(new
FileInputStream("PersonaXML.xml"));
       PersonaBean p=(PersonaBean)decoder.readObject();
       System.out.println(p.getNombre()+" "+p.getEdad());
       decoder.close();
   }catch (IOException e) {e.printStackTrace();}
  }
  public static void main(String args[])
  { Decodifica d=new Decodifica();
  }
}
```

Archivos y entrada/salida de archivos

El objeto File provee utilidades para manejar archivos y obtener informacion acerca de ellos. En java un directorio es solo otro archivo. Se puede crear un objeto File que represente un directorio y utilizarlo para identificar otros archivos.

```
File myFile;
myFile=new File(“myfile.txt”);

myFile=new File(“MyDocs”,”myfile.txt”);

File myDir=new File(“MyDocs”);
myFile=new File(myDir,”myfile.txt”);
```

El objeto File no permite acceder al contenido del archivo, pero se puede utilizar el objeto file como argumento del constructor de un FileReader o un FileWriter.

Entrada/Salida a Archivo

java soporta la lectura de dos formas:

- Utiliza FileReader para leer caracteres
- Utiliza BufferedReader para utilizar el metodo readLine()

java soporta la salida de dos formas:

- Utiliza FileWriter para escribir caracteres
- Utiliza un PrintWriter para utilizar los metodos print y println.

Leyendo de un archivo

El siguiente ejemplo muestra como leer un archivo de texto y mostrarlo en la salida estandar.

```
1  import java.io.*;
2  public class ReadFile{
3   public static void main(String[] args)
4  {//crea el archivo
5   File file=new File(args[0]);
6   try{
7     //crea un BufferedReader
8     //para leer cada linea del archivo
9     BufferedReader in=new BufferedReader(new
FileReader(file));
10     String s;
```

Fundamentos de los flujos de entrada/salida

```
11
12      while((s=in.readLine())!=null)
13      { System.out.println("Read: "+s);
14      }
15      in.close();
16     } catch (FileNotFoundException e1)
{System.err.println("Archivo no encontrado "+file);}
17     catch (IOException e2) {e2.printStackTrace();}
18 }
19 }
```

la linea 5 crea un objeto File basado en el nombre de archivo pasado por parametros. La linea 9 crea un BufferedReader que empaqueta un FileReader. Este codigo lanza una FileNotFoundException si el archivo no existe.

Escribiendo a un archivo

El siguiente ejemplo lee lineas desde el teclado y las escribe en un archivo.

```
1  import java.io.*;
2  public class WriteFile{
3  public static void main(String[] args)
4   {
5     File file=new File(args[0]);
6     try{
7       InputStream Reader isr=new
InputStreamReader(System.in);
8       BufferedReader in=new BufferedReader(isr);
9       PrintWriter out=new PrintWriter(new FileWriter(file));
10       String s;
11      System.out.println(“ingrese el archivo de texto”);
12       System.out.println(“ctrl-d en Unix, ctrl-z en Windows
para finalizar”);
```

Fundamentos de los flujos de entrada/salida

```
13      while ((s=in.readLine())!=null)
14      { out.println(s);
15      }
16      in.close();
17      out.close();
18    }catch (IOException e) {e.printStackTrace();}
19  }
20 }
```

La linea 5 crea un objeto File basado en el primer argumento de la linea de comando. La linea 7 crea un stream para lectura de caracteres desde un stream binario (System.in). En linea 8 se crea un BufferedReader para la entrada estandar. La linea 9 crea un printWriter que decora un FileWriter creado en linea 5. De la linea 13 a la 15 se lee de la entrada y se escribe al archivo, una linea a la vez. Las lineas 16 y 17 cierran el flujo de entrada y salida.

La clase Scanner

Esta clase que se encuentra disponible desde Java 1.5, nos permite leer datos de una forma más sencilla que el clásico InputStream con un BufferedReader.

Para utilizarla tan solo tenemos que crearnos un objeto de tipo Scanner (importando previamente el paquete java.util.Scanner) e indicándole a este que lea de la consola con System.in.

```
Scanner pruebaScanner = new Scanner(System.in);
```

Una vez hecho esto podemos utilizar la función .next() , para leer por consola hasta que encuentre un retorno de carro y salto de linea. El valor lo guardaremos en un String.

```
String texto = pruebaScanner.next();
```

Con esto podríamos estar leyendo por teclado hasta que se introduzca una palabra determinada por ejemplo “fin”, utilizando un bucle while:

```
while (!texto.equals("fin")) {
        texto = pruebaScanner.next();
         System.out.println(texto);
    }
```