Leggere file di testo

- Il modo più semplice di leggere testo: usare la classe `Scanner`
- Per leggere da un file su disco, si costruisca un `FileReader`
- Poi si usi il `FileReader` per costruire un oggetto `Scanner`

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
```

- Si usino i metodi di `Scanner` per leggere dati dal file

`next`, `nextLine`, `nextInt`, e `nextDouble`

Scrivere file di testo

- Per scrivere un file, si costruisca un oggetto `PrintWriter`
  ```
  PrintWriter out = new PrintWriter("output.txt");
  ```
- Se il file esiste, viene svuotato prima che i nuovi dati vi vengano scritti all'interno
- Se il file non esiste, viene creato un file vuoto
- Si usino `print` e `println` per scrivere nel `PrintWriter`:
  ```
  out.println(29.95);
  out.println(new Rectangle(5, 10, 15, 25));
  out.println("Hello, World!");
  ```
- Un file va chiuso quando si è terminata la sua elaborazione:
  ```
  out.close();
  ```

  Altrimenti non è garantito che tutto l'output venga scritto nel file

FileNotFoundException

- Quando il file di input o di output non esiste, può aver luogo una eccezione `FileNotFoundException`
- Per gestire l'eccezione, si etichetti il metodo main così:

```
public static void main(String[] args) throws
        FileNotFoundException
```

Un semplice programma

- Legge tutte le linee di un file e le manda ad un file di output precedute da numeri di linea
- Se il file di input fosse questo:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

- Il programma produrrebbe quanto segue:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

- Il programma può essere ad esempio impiegato per numerare le righe di un programma Java

ch11/fileio/LineNumberer.java

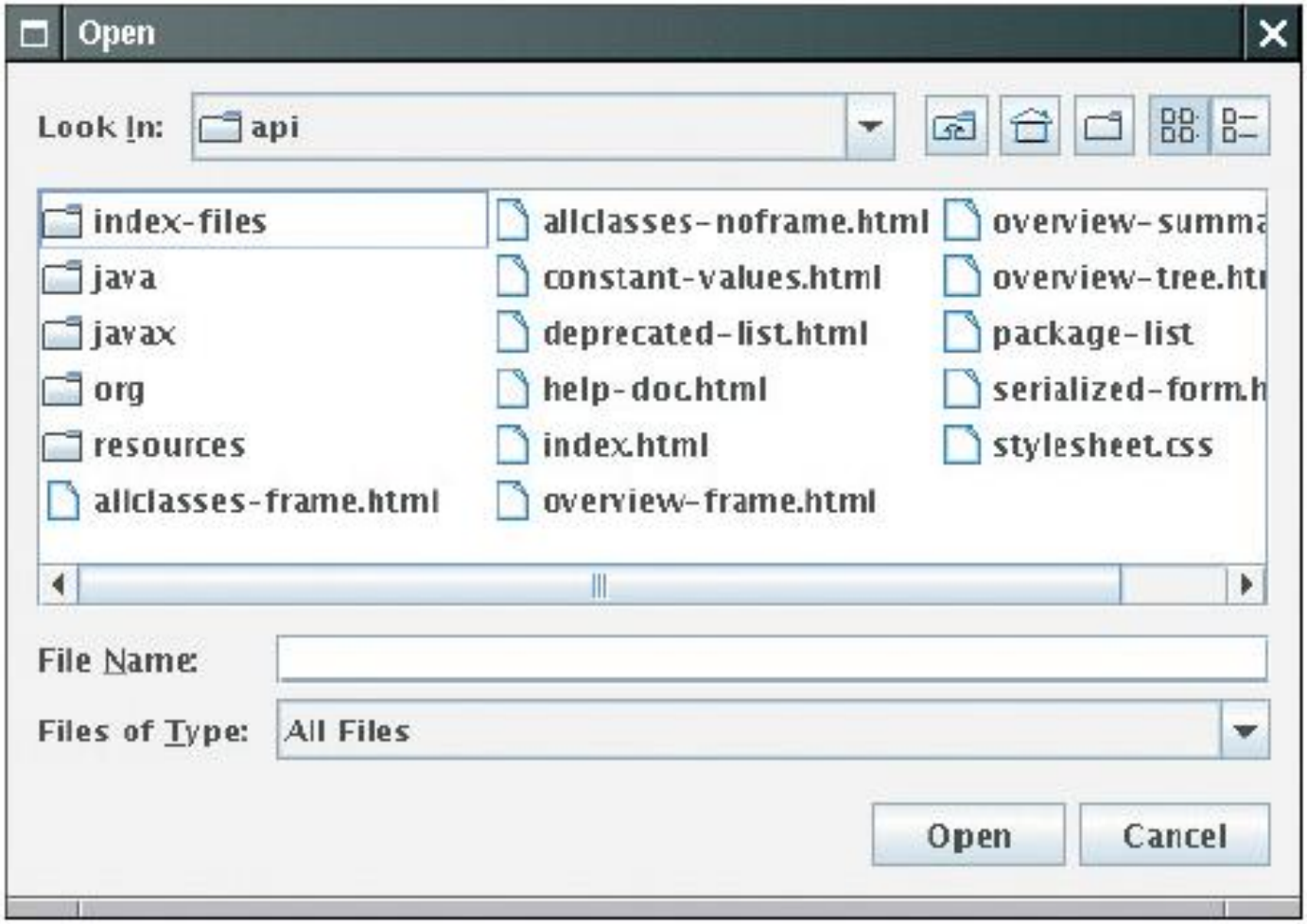
```
01: import java.io.FileReader;
02: import java.io.FileNotFoundException;
03: import java.io.PrintWriter;
04: import java.util.Scanner;
05:
06: public class LineNumberer
07: {
08:    public static void main(String[] args)
09:       throws FileNotFoundException
10:    {
11:       Scanner console = new Scanner(System.in);
12:       System.out.print("Input file: ");
13:       String inputFileName = console.next();
14:       System.out.print("Output file: ");
15:       String outputFileName = console.next();
16:
17:       FileReader reader = new FileReader(inputFileName);
18:       Scanner in = new Scanner(reader);
19:       PrintWriter out = new PrintWriter(outputFileName);
20:       int lineNumber = 1;
```

Continua

ch11/fileio/LineNumberer.java (continua)

```
21:
22:         while (in.hasNextLine())
23:         {
24:            String line = in.nextLine();
25:            out.println("/* " + lineNumber + " */ " + line);
26:            lineNumber++;
27:         }
28:
29:         out.close();
30:      }
31: }
```

Finestre di dialogo file



A JFileChooser Dialog Box

Continua

Finestre di dialogo file (continua)

```
JFileChooser chooser = new JFileChooser();
FileReader reader = null;
if (chooser.showOpenDialog(null) ==
      JFileChooser.APPROVE_OPTION)
{
   File selectedFile = chooser.getSelectedFile();
   reader = new FileReader(selectedFile);
   . . .
}
```

Per una finestra di dialogo di tipo “Save” si usi invece il metodo *showSaveDialog*

Un’alternativa per leggere un file di testo

```
JFileChooser chooser = new JFileChooser();
if (chooser.showOpenDialog(null) ==
   JFileChooser.APPROVE_OPTION)
{
   File selectedFile = chooser.getSelectedFile();
   String s =Files.readString(selectedFile.toPath());
   . . .
}
```

La stringa s conterrà tutto il contenuto del file selezionato.

Lanciare eccezioni

- Si lanci un oggetto eccezione per segnalare una condizione eccezionale
- Esempio: `IllegalArgumentException`: valore di parametro illegale

```
IllegalArgumentException exception
      = new IllegalArgumentException("Amount exceeds
      balance");
throw exception;
```

- Non è necessario memorizzare l'oggetto eccezione in una variabile:

```
throw new IllegalArgumentException("Amount exceeds
      balance");
```

- Quando viene lanciata un'eccezione, il metodo termina immediatamente
  - *L'esecuzione continua con un gestore di eccezioni*

Esempio

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (amount > balance)
      {
         IllegalArgumentException exception
               = new IllegalArgumentException("Amount
               exceeds balance");
         throw exception;
      }
      balance = balance - amount;
   }
   . . .
}
```

Gerarchia di classi di eccezioni

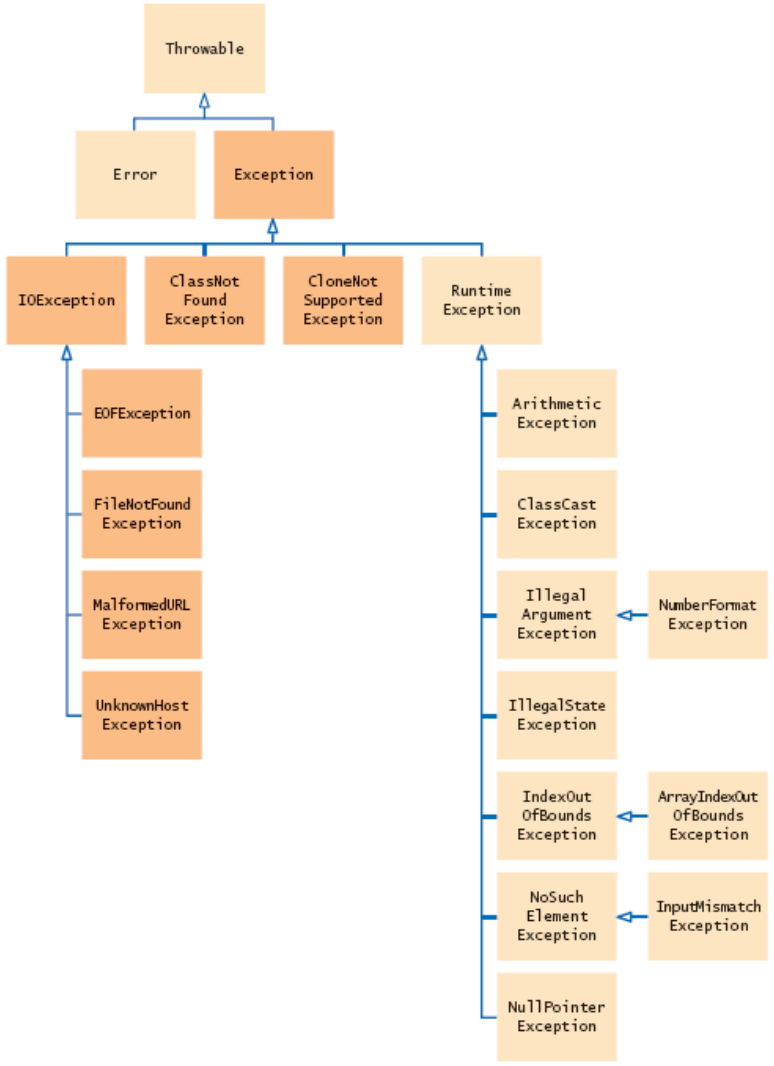


Figure 1 The Hierarchy of Exception Classes

Sintassi 11.1 Lanciare un'eccezione

```
throw exceptionObject;
```

Esempio:

```
throw new IllegalArgumentException();
```

Scopo:

Lanciare un'eccezione e trasferire il controllo ad un gestore per questo tipo di eccezione.

Eccezioni verificate e non verificate

- Due tipi di eccezioni:
  - *Checked (verificate, controllate):*
    - Il compilatore controlla che non vengano ignorate dal programmatore
    - Dovute a circostanza esterne che il programmatore non può prevenire
    - La maggior parte di esse ha luogo con l'input o l'output
    - Ad esempio, `IOException`
  - *Unchecked (non verificate, non controllate):*
    - Estendono la classe `RuntimeException` o `Error`
    - Sono dovute a difetti del programmatore
    - Esempi di eccezioni runtime (in esecuzione):
      `NumberFormatException`
      `IllegalArgumentException`
      `NullPointerException`
    - Esempi di errore:
      `OutOfMemoryError`

Eccezioni verificate e non verificate

- Le categorie non sono perfette:
  - *`Scanner.nextInt` lancia la non verificata `InputMismatchException`*
  - *Il programmatore non può impedire agli utenti di inserire un input non corretto*
  - *Questa scelta rende la classe di facile uso per programmatori alle prime armi*
- Si ha a che fare con eccezioni verificate principalmente programmando con file e flussi
- Per esempio, si usi uno `Scanner` per leggere un file

```
String filename = . . .;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
```

- Ma, il costruttore di `FileReader` può lanciare una `FileNotFoundException`

Eccezioni verificate e non verificate

Due scelte:

1. *Gestire l'eccezione*
2. *Dire al compilatore che si desidera che il metodo termini quando l'eccezione ha luogo*

- Si usi lo specificatore `throws` per consentire al metodo di lanciare una eccezione verificata

```
public void read(String filename) throws
      FileNotFoundException
{
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   . . .
}
```

- Per eccezioni multiple:

```
public void read(String filename)
      throws IOException, ClassNotFoundException
```

Continua

Eccezioni verificate e non verificate (continua)

- Si tenga in mente la gerarchia di ereditarietà:
  Se un metodo può lanciare una `IOException` e una `FileNotFoundException`, si usi soltanto la IOException

- Meglio dichiarare un'eccezione che gestirla male

Sintassi 11.2 Specifica di un'eccezione

```
accessSpecifier returnType methodName(parameterType
      parameterName, . . .)
   throws ExceptionClass, ExceptionClass, . . .
```

Esempio:

```
public void read(BufferedReader in)
   throws IOException
```

Scopo:

Indicare le eccezioni verificate che il metodo può lanciare.

Catturare eccezioni

- Un gestore di eccezioni viene predisposto con l'istruzione `try/catch`
- Il blocco `try` contiene le istruzioni che possono causare un'eccezione
- La clausola `catch` contiene il gestore per un certo tipo di eccezione

Continua

Catturare eccezioni (continua)

- Esempio:

```
try
{
   String filename = . . .;
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader); String input =
         in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

Catturare eccezioni

- Le istruzioni nel blocco `try` sono eseguite
- Se non ha luogo nessuna eccezione, la clausola `catch` viene ignorata
- Se un'eccezione di un tipo indicato ha luogo, l'esecuzione salta alla clausola `catch`
- Se ha luogo un'eccezione di un altro tipo, viene lanciata fino alla sua cattura da un altro blocco `try`
- `catch (IOException exception) block`
  - *`exception` contiene il riferimento all'oggetto eccezione lanciato*
  - *`catch` può analizzare l'oggetto per trovare maggiore dettagli*
  - *`exception.printStackTrace()`: produce la catena di chiamate di metodo che hanno condotto all'eccezione*

Sintassi 11.3 Blocco Try generale

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
. . .
```

Continua

Sintassi 11.3 Blocco Try generale (continua)

Esempio:

```
try
{
   System.out.println("How old are you?");
   int age = in.nextInt();
   System.out.println("Next year, you'll be " + (age
         + 1));
}
catch (InputMismatchException exception)
{
   exception.printStackTrace();
}
```

Continua

Sintassi 11.3 Blocco Try generale (continua)

Scopo:

Eseguire uno o più istruzioni che possono generare eccezioni. Se un'eccezione ha luogo e corrisponde ad una di quelle indicate nella clausola `catch`, viene eseguita la prima che corrisponda. Se non ha luogo nessuna eccezione, o viene lanciata un'eccezione che non corrisponde a nessuna di quelle indicate in una clausola `catch`, la clausola `catch` viene ignorata.

La clausola `finally`

- Le eccezioni fanno terminare il metodo corrente
- Pericoloso: può far saltare del codice essenziale
- Esempio:

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close(); // May never get here
```

- L'istruzione `reader.close()` va eseguita in ogni caso
- Si usa la clausola `finally` per il codice che deve essere eseguito non importa cosa accada

La clausola `finally`

```
FileReader reader = new FileReader(filename);
try
{
   Scanner in = new Scanner(reader);
   readData(in);
}
finally
{
   reader.close();
  // if an exception occurs, finally clause is also
 // executed before exception is passed to its handler
}
```

La clausola `finally`

- Eseguita quando si esce dal blocco `try` in uno qualunque dei seguenti tre modi:
  - *Dopo l'ultima istruzione del blocco `try`*
  - *Dopo l'ultima istruzione della clausola `catch`, se questo blocco `try` ha lanciato un'eccezione*
  - *Quando è stata lanciata un'eccezione nel blocco `try` e non è stata catturata*

Sintassi 11.4 La clausola `finally`

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

Continua

Sintassi 11.4 La clausola `finally`

Esempio:

```
FileReader reader = new FileReader(filename);
try
{
   readData(reader);
}
finally
{
   reader.close();
}
```

Scopo:

Garantire che le istruzioni nella clausola `finally` sono eseguite sia che le istruzioni nel blocco `try` lancino un'eccezione sia che non la lancino.

Progettare i propri tipi di eccezione

- Si possono progettare i propri tipi di eccezione – sottoclassi di `Exception` o `RuntimeException`
-
```
if (amount > balance)
{
   throw new InsufficientFundsException(
      "withdrawal of " + amount + " exceeds balance of "
            + balance);
}
```
- La si renda un'eccezione non verificata – il programmatore avrebbe potuto evitarla chiamando prima `getBalance`
- Si estenda `RuntimeException` o una delle sue sottoclassi
- Si forniscano due costruttori
  1. *Un costruttore di default*
  2. *Un costruttore che accetta una stringa di messaggio che descriva il motivo dell'eccezione*

Progettare i propri tipi di eccezione

```
public class InsufficientFundsException
      extends RuntimeException
{
   public InsufficientFundsException() {}

   public InsufficientFundsException(String message)
   {
      super(message);
   }
}
```

Un esempio completo

- Il programma
  - *Chiede all'utente il nome di un file*
  - *Il file dovrebbe contenere valori*
  - *La prima riga contiene il numero totale di valori*
  - *Le restanti righe contengono i dati*
  - *Esempio di file di input:*

    3
    1.45
    -2.1
    0.05

Un esempio completo

- Cosa può non funzionare?
  - *Il file potrebbe non esistere*
  - *Il file potrebbe avere i dati in formato errato*
- Chi può individuare gli errori?
  - *Il costruttore di* `FileReader` *lancerà un'eccezione quando il file non esiste*
  - *I metodi che trattano l'input devono lanciare un'eccezione se trovano un errore nel formato dei dati*
- Quali eccezioni possono essere lanciate?
  - `FileNotFoundException` *può essere lanciata dal costruttore di* `FileReader`
  - `IOException` *può essere lanciata dal metodo close di* `FileReader`
  - `BadDataException`*, un'eccezione verificata personalizzata*

Continua

Un esempio completo (continua)

- Chi può rimediare agli errori individuati dalle eccezioni?
  - *Soltanto il metodo main del programma* `DataSetTester` *interagisce con l'utente*
  - *Cattura eccezioni*
  - *Visualizza messaggi di errori appropriati*
  - *Dà all'utente un'altra possibilità di inserire il file corretto*

ch11/data/DataAnalyzer.java

```
01: import java.io.FileNotFoundException;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:    This program reads a file containing numbers and analyzes its contents.
07:    If the file doesn't exist or contains strings that are not numbers, an
08:    error message is displayed.
09: */
10: public class DataAnalyzer
11: {
12:    public static void main(String[] args)
13:    {
14:       Scanner in = new Scanner(System.in);
15:       DataSetReader reader = new DataSetReader();
16:
17:       boolean done = false;
18:       while (!done)
19:       {
20:          try
21:          {
22:             System.out.println("Please enter the file name: ");
23:             String filename = in.next();
```

Continua

ch11/data/DataAnalyzer.java (cont.)

```
24:
25:             double[] data = reader.readFile(filename);
26:             double sum = 0;
27:             for (double d : data) sum = sum + d;
28:             System.out.println("The sum is " + sum);
29:             done = true;
30:          }
31:          catch (FileNotFoundException exception)
32:          {
33:             System.out.println("File not found.");
34:          }
35:          catch (BadDataException exception)
36:          {
37:             System.out.println("Bad data: " + exception.getMessage());
38:          }
39:          catch (IOException exception)
40:          {
41:             exception.printStackTrace();
42:          }
43:       }
44:    }
45: }
```

Il metodo `readFile` della classe `DataSetReader`

- Costruisce un oggetto `Scanner`
- Chiama il metodo `readData`
- Senza alcun collegamento con alcuna eccezione
- Se c'è un problema con un file di input, semplicemente passa l'eccezione al chiamante

Continua

Il metodo `readFile` della classe `DataSetReader` (cont.)

```
public double[] readFile(String filename)
      throws IOException, BadDataException
      // FileNotFoundException is an IOException
{
   FileReader reader = new FileReader(filename);
   try
   {
      Scanner in = new Scanner(reader);
      readData(in);
   }
   finally
   {
      reader.close();
   }
   return data;
}
```

Il metodo `readData` della classe `DataSetReader`

- Legge il numero di valori
- Costruisce un array
- Chiama `readValue` per ogni valore

```
private void readData(Scanner in) throws BadDataException
{
   if (!in.hasNextInt())
   throw new BadDataException("Length expected");
   int numberOfValues = in.nextInt();
   data = new double[numberOfValues];

   for (int i = 0; i < numberOfValues; i++)
      readValue(in, i);

   if (in.hasNext())
   throw new BadDataException("End of file expected");
}
```

Continued

Il metodo `readData` della classe `DataSetReader` (cont.)

- Controlla due potenziali errori
  - *Il file potrebbe non iniziare con un intero*
  - *Il file potrebbe avere ulteriori dati dopo aver letto tutti i valori*
- Non fa alcun tentativo di catturare alcuna eccezione

Il metodo `readValue` della classe `DataSetReader`

```
private void readValue(Scanner in, int i) throws
      BadDataException
{
   if (!in.hasNextDouble())
      throw new BadDataException("Data value expected");
   data[i] = in.nextDouble();
}
```

Scenario

1. `DataSetTester.main` chiama `DataSetReader.readFile`
2. `readFile` chiama `readData`
3. `readData` chiama `readValue`
4. `readValue` non trova il valore atteso e lancia `BadDataException`
5. `readValue` non ha alcun gestore di eccezione e termina
6. `readData` non ha alcun gestore di eccezione e termina
7. `readFile` non ha alcune gestore di eccezione e termina dopo aver eseguito la clausola `finally`
8. `DataSetTester.main` ha un gestore per `BadDataException`; il gestore visualizza un messaggio, e l'utente ha un'altra occasione di inserire il nome del file

ch11/data/DataSetReader.java

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:    Reads a data set from a file. The file must have the format
07:    numberOfValues
08:    value1
09:    value2
10:    . . .
11: */
12: public class DataSetReader
13: {
14:    /**
15:       Reads a data set.
16:       @param filename the name of the file holding the data
17:       @return the data in the file
18:    */
19:    public double[] readFile(String filename)
20:          throws IOException, BadDataException
21:    {
22:       FileReader reader = new FileReader(filename);
```

Continua

ch11/data/DataSetReader.java (cont.)

```
23:         try
24:         {
25:            Scanner in = new Scanner(reader);
26:            readData(in);
27:         }
28:         finally
29:         {
30:            reader.close();
31:         }
32:         return data;
33:      }
34:
35:      /**
36:         Reads all data.
37:         @param in the scanner that scans the data
38:      */
39:      private void readData(Scanner in) throws BadDataException
40:      {
41:         if (!in.hasNextInt())
42:            throw new BadDataException("Length expected");
43:         int numberOfValues = in.nextInt();
44:         data = new double[numberOfValues];
```

Continua

ch11/data/DataSetReader.java (cont.)

```
45:
46:          for (int i = 0; i < numberOfValues; i++)
47:             readValue(in, i);
48:
49:          if (in.hasNext())
50:             throw new BadDataException("End of file expected");
51:       }
52:
53:       /**
54:          Reads one data value.
55:          @param in the scanner that scans the data
56:          @param i the position of the value to read
57:       */
58:       private void readValue(Scanner in, int i) throws BadDataException
59:       {
60:          if (!in.hasNextDouble())
61:             throw new BadDataException("Data value expected");
62:          data[i] = in.nextDouble();
63:       }
64:
65:       private double[] data;
66: }
```

Serializzazione

La serializzazione consente di salvare e leggere oggetti.
Questi oggetti devono implementare l'interfaccia **Serializable.**

Lettura

```
FileInputStream fis = new FileInputStream("oggetto.sav");
ObjectInputStream ois = new ObjectInputStream(fis);
Object daCaricare = ois.readObject();
ois.close();
```

Salvataggio

```
FileOutputStream fos = new FileOutputStream("oggetto.sav");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(daSalvare);
oos.close();
```

Gli ArrayList implementano l'interfaccia Serializable per default.