Files & Text I/O

Khaled Ghosn

Credits: Slides are written with the help of Ms. Bader Serhan

Table Of Contents

Why do we need to store data in files?

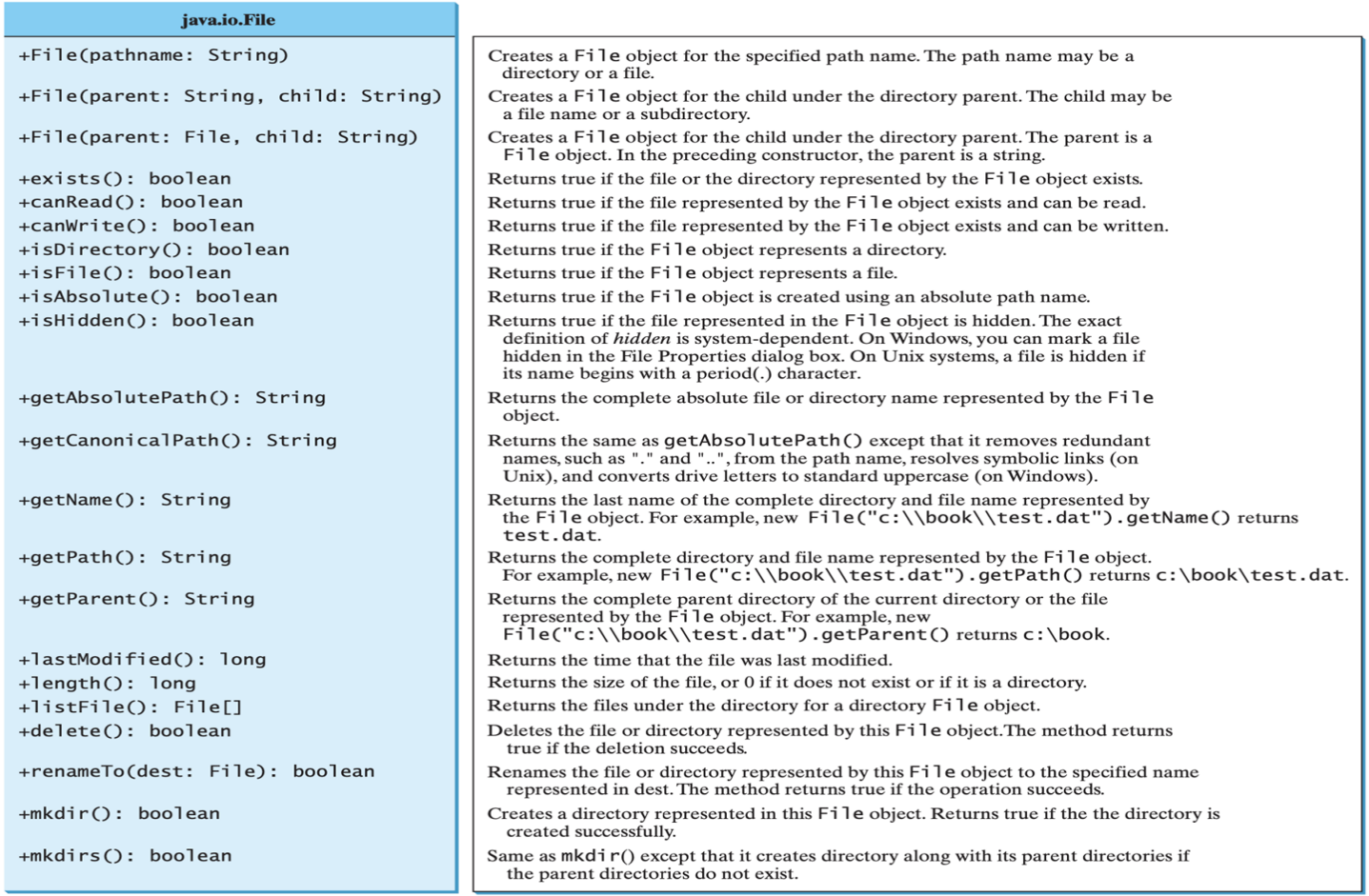
Data stored in the program are temporary; they are lost when the program terminates.

To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device.

The file can then be transported and read later by other programs.

The File Class

- The **File** class contains the methods for obtaining the properties of a file / directory and for renaming and deleting a file / directory.
- However, the **File** class does not contain the methods for reading and writing file contents.
- The **File** class is used for creation of files and directories, file searching, file deletion, etc.
- The File object represents the actual file / directory on the disk.



java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\book\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\book.
+lastModified(): long	Returns the time that the file was last modified.
+length(): long	Returns the size of the file, or 0 if it does not exist or if it is a directory.
+listFile(): File[]	Returns the files under the directory for a directory File object.
+delete(): boolean	Deletes the file or directory represented by this File object.The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames the file or directory represented by this File object to the specified name represented in dest. The method returns true if the operation succeeds.
+mkdir(): boolean	Creates a directory represented in this File object. Returns true if the the directory is created successfully.
+mkdirs(): boolean	Same as mkdir() except that it creates directory along with its parent directories if the parent directories do not exist.

FIGURE 12.6 The `File` class can be used to obtain file and directory properties, to delete and rename files and directories, and to create directories.

The File Class

Constructors

Constructor	Description
File (String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File (String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File (File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File (URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

The File Class

Useful Methods

Method	Description
public boolean **isDirectory ( )**	Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
public boolean **isFile ( )**	Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise.

The File Class

Useful Methods

Method	Description
public String **getName ( )**	Returns the name of the file or directory denoted by this abstract pathname.
public String **getParent ( )**	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
public String **getPath ( )**	Converts this abstract pathname into a pathname string.
public boolean **exists ( )**	Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise.

The File Class

Useful Methods

Method	Description
public boolean **canRead ( )**	Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
public boolean **canWrite ( )**	Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
public boolean **isHidden ( )**	Returns true if the file represented in the File object is hidden. The exact definition of hidden is system-dependent.

The File Class

Useful Methods

Method	Description
public long **length ( )**	Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
public long **lastModified ( )**	Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds, or 0L if the file does not exist or if an I/O error occurs.
public String [ ] **list ( )**	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

The File Class

Useful Methods

Method	Description
public long **length ( )**	Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
public long **lastModified ( )**	Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds, or 0L if the file does not exist or if an I/O error occurs.
public String [ ] **list ( )**	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

The File Class

Useful Methods

Method	Description
public boolean **mkdir ( )**	Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise.
public boolean **mkdirs ( )**	Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
public int **compareTo (**File pathname**)**	Compares two abstract pathnames lexicographically. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.

The File Class

Useful Methods

Method	Description
public boolean **renameTo (**File dest**)**	Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise.
public boolean **createNewFile ( )** throws IOException	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.
public boolean **delete ( )**	Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.

File Naming

- The file name is a String.
- Every file is placed in a directory in the file system.
  - ➔ Absolute file name or full name, e.g. **C:\book\Welcome.java**
    - **C:\book** is the directory path for the file
    - Absolute file names are machine dependent
    - Absolute file names contain the file name with its complete path and drive letter
  - ➔ Relative file name, e.g. **Welcome.java**
    - Relative file names are in relation to the current working directory
    - So, the directory path is omitted for a relative file name

File Naming

- Note 1:
  - ✓ Constructing a File instance does not create a file on the machine.
  - ✓ You can create a File instance for any file name regardless whether it exists or not.
  - ✓ You can invoke the exists() method on a File instance to check whether the file exists.

- Note 2:
  - ✓ The directory separator for Windows is a backslash (\).
  - ✓ The backslash is a special character in Java and should be written as \\ in a string literal.

Exceptions in Files

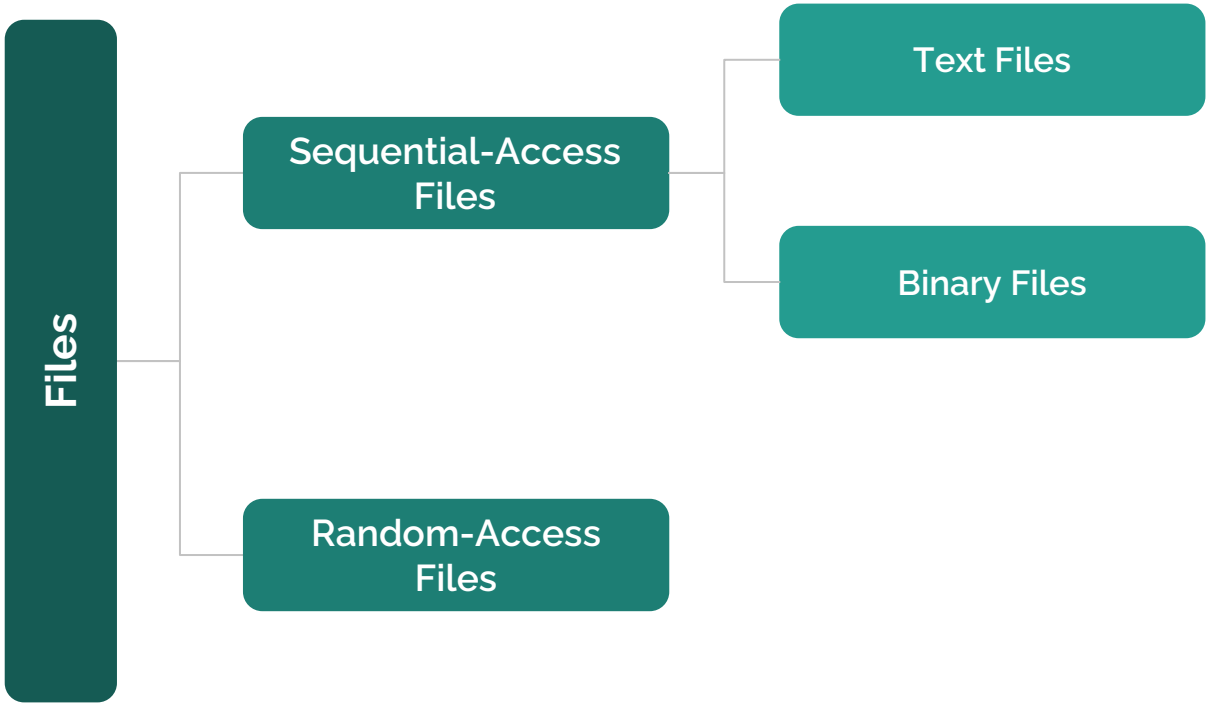
May happen because:

- File does not exist
- File exists but no rights to read/write
- File exists with needed rights but is taken by another app

➔ For the moment just throw **FileNotFoundException**

```
public static void main(String[] args)
                    throws FileNotFoundException {
        File file = new File("Top 10.txt");
```

Types of Files



Types of Files

A file can be:

- **Serial Access File**
  are files in which data is stored in physically adjacent locations, often in no particular logical order, with each new item of data being added to the end of the file
  - **Sequential files**
  - (**non-sequential** files ...)

- **Random Access File**
  a file to be read from and write to at random locations

Types of Files

- **Serial Access File**
  the internal structure of a serial file can be either:

  - Text File
  data stored in a text file are represented in human-readable form

  - Binary File
  data stored in a binary file are represented in binary form

Types of Files

- **Serial Access Files** are:
  - simple to handle
  - quite widely used in:
    - . small-scale applications
    - . temporary storage in larger-scale applications

Problems:

1- can’t go directly to a specific record ➔ not feasible

2- can’t add or modify records within an existing file

Sequential-Access vs. Random Access Files

	Sequential Access Files	Random Access Files
Definition	**Sequential Access** to a data file means that the computer system **reads** or **writes** information to the file **sequentially**, starting from the beginning of the file and proceeding step by step.	**Random Access** to a file means that the computer system can **read** or **write** information **anywhere** in the data file. This type of operation is called "**Direct Access**" because the computer system knows where the data is stored (using indexing) and hence goes directly and reads the data.
Advantages	• Faster than random access files (in some cases!!!) • Access information in the same order all the time	• Search through it and find the data you need more easily (e.g. using indexing)

Text Files

A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows or vi on UNIX is called a text file.

E.g. java source files, they can be read by a text editor

Binary files cannot be read by text editors. They are designed to be read by programs.

E.g. java class files, they are read by the JVM

Binary Files

Text Files vs. Binary Files

- Although it is not technically precise and correct, you can envision a text file as consisting of a **sequence of characters** and a binary file as consisting of a **sequence of bits**.
- Characters in a text file are encoded using a character encoding scheme such as **ASCII or Unicode**.
  - For example, the decimal integer 199 is stored as a sequence of three characters 1, 9, 9 in a text file
  - The same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals hex C7
- The advantage of binary files is that they are **more efficient** to process than text files, because binary I/O does not require encoding/decoding.

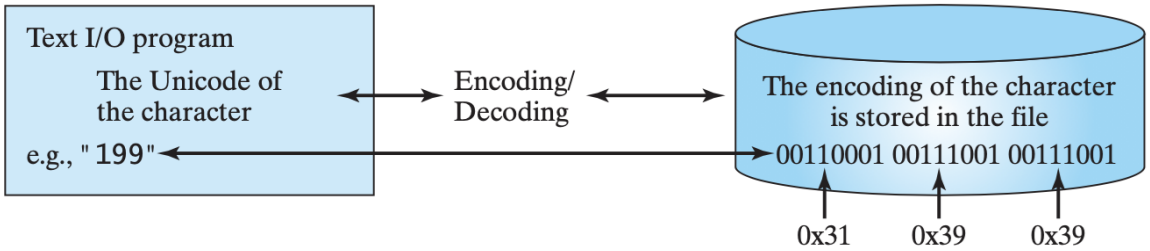
Text Files vs. Binary Files

- Computers do not differentiate between binary files and text files. All files are stored in binary format, and thus all files are essentially **binary files**.
- Text I/O is built upon binary I/O to provide a level of **abstraction** for character encoding and decoding.
- Encoding and decoding are automatically performed for text I/O. The JVM converts Unicode to a file-specific encoding when writing a character, and it converts a file-specific encoding to Unicode when reading a character.
- Binary files are independent of the encoding scheme on the host machine and thus are portable.

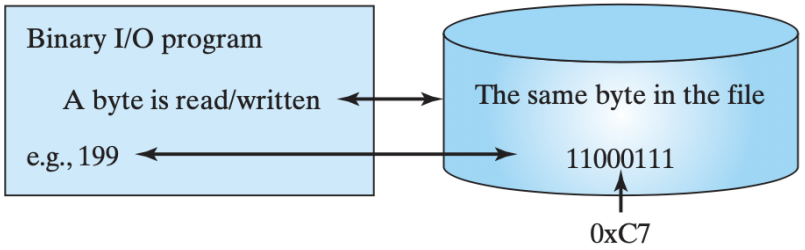
Text Files vs. Binary Files

- “199” → 1 + 9 + 9
- Default encoding for Windows text files is ASCII
- ASCII code for “1” is 49 → 0x31 in HEX

- To write “199”, 3 bytes are sent to the output using **text I/O**, whereas 1 byte is sent in **binary I/O**



(a)



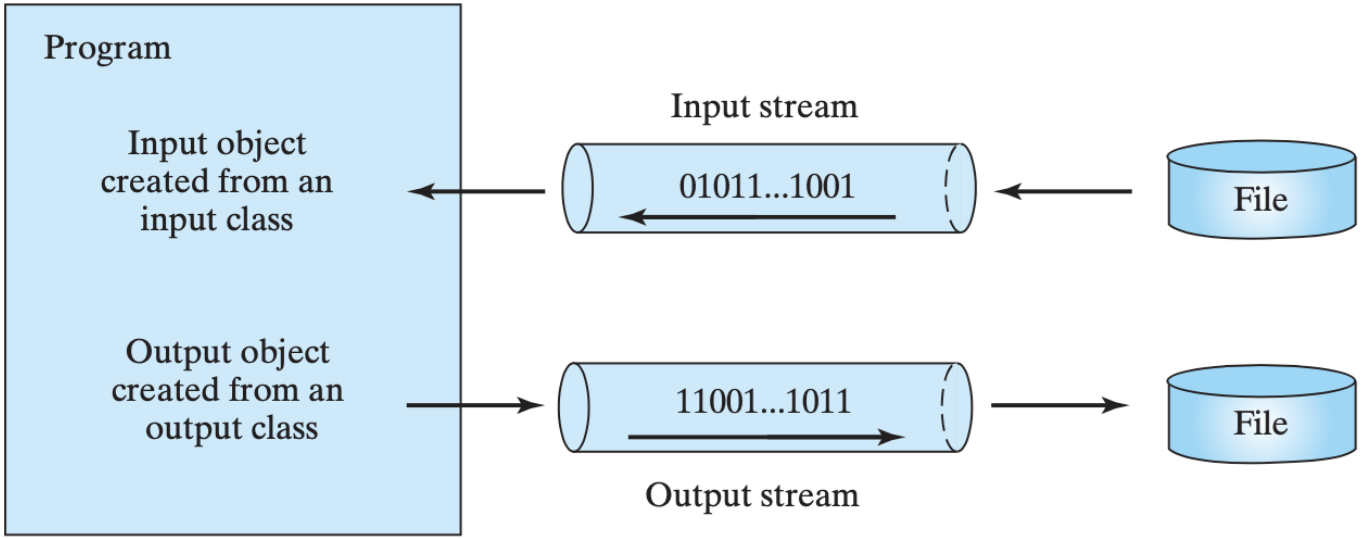
(b)

File Input & Output

- A **File** object encapsulates the properties of a file or a path, but it does **not** contain the methods for creating a file or for writing/reading data to/from a file (referred to as data input and output, or I/O for short).

- In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file.
  → input class contains methods to read data
  → output class contains methods to write data

File Input & Output



How is I/O Handled in Java?

Java offers many classes for performing file input and output. These classes can be categorized as:

I. **Text I/O** classes:
   - **PrintWriter**
     - PrintWriter class can be used to create a file and write data to a text file
   - **Scanner**
     - Scanner class will read the contents of the text file which exists already.
     - Scanner class breaks the input of the file into tokens. As a result, we get the output as the tokens into various types using methods such as nextLine ( ), hasnextLine ( ) etc

II. **Binary I/O** classes:
   - Using **I/O Streams**

Writing data into a Text file using PrintWriter

java.io.PrintWriter	
`+PrintWriter(file: File)`	Creates a `PrintWriter` object for the specified file object.
`+PrintWriter(filename: String)`	Creates a `PrintWriter` object for the specified file-name string.
`+print(s: String): void`	Writes a string to the file.
`+print(c: char): void`	Writes a character to the file.
`+print(cArray: char[]): void`	Writes an array of characters to the file.
`+print(i: int): void`	Writes an `int` value to the file.
`+print(l: long): void`	Writes a `long` value to the file.
`+print(f: float): void`	Writes a `float` value to the file.
`+print(d: double): void`	Writes a `double` value to the file.
`+print(b: boolean): void`	Writes a `boolean` value to the file.
Also contains the overloaded `println` methods.	A `println` method acts like a `print` method; additionally, it prints a line separator. The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix.
Also contains the overloaded `printf` methods.	The `printf` method was introduced in §4.6, "Formatting Console Output."

The **PrintWriter** class contains the methods for writing data to a text file.

Reading data from a Text file using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that scans tokens from the specified file.
+Scanner(source: String)	Creates a Scanner that scans tokens from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

The **Scanner** class contains the methods for scanning data.

Closing Resources Automatically

Programmers often forget to close the file.

JDK 7 provides the followings new **try-with-resources** syntax that automatically closes the files.

```
try (declare and create resources) {
Use the resource to process the file;
}
```