NFA035 – Exemple de sujet d'examen

juin 2013

Durée : 3h – Documents autorisés – Barème indicatif

Exercice 1 *Entrées/sorties 7 points*

Indication : pour cet exercice, nous ne pensons pas qu'il soit utile d'utiliser `StreamTokenizer`.
On veut écrire des méthodes pour afficher la table des matières d'un fichier texte au format décrit ci-après.

On suppose que le fichier lu est au format suivant :
- Les titres de niveau 3 (sous-sous-titres) sont sur une ligne séparée dont les trois premiers caractères sont `'***'` ;
- Les titres de niveau 2 (sous-titres) sont sur une ligne séparée dont les deux premiers caractères sont `'**'` (attention, une ligne qui commence par "***" commence aussi par deux "*", mais c'est un titre de niveau 3 et pas de niveau 2).
- Les titres de niveau 1 sont sur une ligne séparée dont le premier caractère est `'*'` (évidemment les titres de niveau 3 et 2 sont exclus) ;
- Les autres lignes sont du texte normal.

Par exemple le fichier ci-dessous à gauche a pour table des matière le texte ci-dessous à droite.

Contenu du fichier

```
* Programmation
  La programmation c'est bien.
  Ça ouvre l'esprit.
 ** Langages de programmation
  Les langages...
  *** Langage C
  Le langage C c'est bien car ça
  forme la jeunesse.
*** Langage Java
  Le langage Java...
** Compilation
 La compilation c'est mal connu.
* Système
Le système c'est bien aussi.
** Unix
Unix c'est bien
** Windows
Windows c'est moins bien
** Mac
Mac c'est pas mal
```

Table des matières

```
* Programmation
** Langages de programmation
*** Langage C
*** Langage Java
** Compilation
* Système
** Unix
** Windows
** Mac
```

Question 1.1 *2 points*

Écrivez la méthode `void tableDesMatieres(File f)` qui affiche les lignes qui commencent par "*" (y compris leur préfixes `'*'`, `'**'` et `'***'`) du fichier `f`.

Remarque : Il n'est pas permis d'utiliser la méthode `titres` ci-dessous.

Question 1.2 *4 points*

Écrivez la méthode `void titres(File f,int n)` qui affiche les lignes du fichier `f` qui sont des titres de niveau `n`.

Question 1.3 *1 points*

Question de cours : en utilisant la méthode `titres(File f,int n)` de l'exercice 1 (même si vous ne l'avez pas définie), écrivez la méthode `void titres(String s,int n)` qui affiche les lignes du fichier dont le nom est `s` qui sont des titres de niveau `n`.

Exercice 2 *collections (7 points)*

On souhaite modéliser le dossier de patients qui sont les clients d'une pharmacie. Chaque patient est représenté par son nom et sa liste de médicaments (chacun de type String) de son ordonnance, sans doublons. Les opérations sur le dossier des patients devront **garantir que les noms de patients sont différents, et qu'un médicament ne figure qu'une fois dans une ordonnance**. Une partie du code des classes Patient et `DossierPharmacie` vous est fournie : à vous de compléter le corps des méthodes signalées et de répondre aux questions posées.

Question 2.1 *classe Patient, 1 point*

Complétez le code pour la classe Patient donné plus bas.

```
private String nom;
private Set<String> ordonnance;

public Patient(String n){
  nom = n;
  ordonnance = new HashSet<String>();
}
public String getNom() { return nom;}

public boolean ordonnanceVide(){
      return ordonnance.isEmpty();
}
public void affiche(){
     Terminal.ecrireStringln(getNom());
     afficheOrdonnance();
}
/** Ajoute un medicament de nom m dans ordonnance   */
public void ajoutMedicament(String m) {
      ordonnance.add(m);
}

// A completer -> code methodes suivantes

/** Affiche l'ordonnance du patient       */
```

```
  public void afficheOrdonnance(){
      // A completer
  }
  /** Teste si ordonnance contient un medicament  m     */
  public boolean contientMedicament(String m) {
     // A completer
  }
}
```

Question 2.2 *classe DossierPharmacie, 3 points*

L'ensemble de patients clients de la pharmacie est représenté par une table d'associations entre noms de patients (String) et objets Patient. Les noms des patients doivent être tous différents, sans distinction entre minuscules et majuscules. Afin d'éviter l'ajout multiple d'un nom identique aux majuscules/minuscules près, l'ajout d'une nouveau nom dans la table se fait après avoir convertit celui-ci en minuscules (méthode `nouveauPatient`). Complétez le code des méthodes signalées plus bas.

```
public class DossierPharmacie {
  private String nom;
  private HashMap<String,Patient> patients;

  public DossierPharmacie(String n){
     nom=n; patients = new HashMap<String,Patient>();
  }

  /** Ajoute un nouveau patient de nom et ordonnance donnés */
  public void nouveauPatient(String nom, String [] ord){
     Patient c = new Patient(nom);
     for (String n: ord){
        c.ajoutMedicament(n);
     }
     String key=nom.toLowerCase();
     patients.put(key, c);
  }

  // Compléter les methodes suivantes

  /** Ajoute un nouveau medicament sur un patient deja existant.
   * Renvoie false si le patient n'existe pas et
   * true si l'ajout a pu etre effectue                          */
   public boolean ajoutMedicament(String nom, String m){
        // Completer
   }
   /** Affiche nom + ordonnance du patient du nom donné          */
   public void affichePatient(String nom){
      // Completer
   }
   /** Affiche nom  pharmacie + tous les patients du dossier     */
   public void affiche(){
        // Completer
   }
}
```

Question 2.3 *(Patients ayant pris un médicament), 1,5 points*

Ajoutez dans la classe `DossierPharmacie` une méthode qui prend en paramètre le nom d'un médicament et retourne une collection contentant tous les patients de la pharmacie ayant pris ce médicament.

```
/** Retourne collection de patients qui prennent le medicament m    */
 public Collection<String> affichePatientAvecMedicament(String m){
       // Compléter
 }
```

Question 2.4 *Suppresion de patients, 1,5 points*

Ajoutez dans la classe `DossierPharmacie` une méthode permettant de supprimer du dossier des patients tous les patients dont l'ordonnance est vide (aucun médicamant).

```
 /** Enleve du dossier tous les les patients dont
  * l'ordonnance est vide (ne contient aucun medicament)    */
 public void enlevePatientOrdonnanceVide() {
   // Compléter
```

Exercice 3 *5 points*

Un bout d'interface pour un logiciel de gestion des inscriptions.

On suppose donnée la classe CoursCnam. Un object CoursCnam représente un cours donné au Cnam. Il a un code (codeUE, comme "NFA035") et une durée en heures (40 pour NFA035 par exemple).

La liste de tous les cours est disponible à travers une méthode statique : `creerlesCours()`.

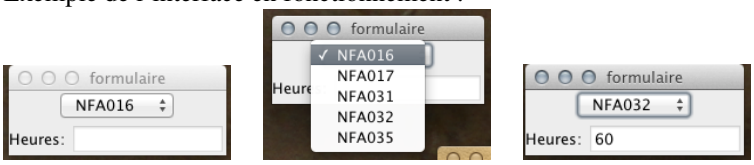
```
public class CoursCnam {
        private String codeUE;
        private int nombreHeures;

        /**
         * Méthode statique fabrique.
         * Retourne le tableau de tous les cours disponibles.
         * @return
         */
        public static CoursCnam[] creerLesCours() { ... }

        public CoursCnam(String codeUE, int nombreHeures) {...}

        public String getCodeUE() { return codeUE;}

        public int getNombreHeures() return nombreHeures;}

        public String toString() { return codeUE;}
}
```

Question 3.1 *5 points*

Complétez la classe suivante en ajoutant le code nécessaire pour que, quand on sélectionne une UE dans la JComboBox, le nombre d'heures correspondantes s'affiche dans le champ texte `heuresField`. *Vous pouvez écrire des*

méthodes et des classes supplémentaires si vous le désirez.

Exemple de l'interface en fonctionnement :



Indications :

- on peut être averti quand l'utilisateur sélectionne une entrée dans une `JComboBox` en utilisant un `ActionListener`.
- Pour fixer le modèle d'une JComboBox, on utilise la méthode `setModel(ComboboxModel model)`
- La classe `DefaultComboboxModel` a les constructeurs suivants :
  - `DefaultComboBoxModel()` Construit un modèle vide ;
  - `DefaultComboBoxModel(Object[] items)` construit un modèle à partir d'une liste de valeurs.
  - la méthode `getSelectedItem()` de `JCombobox` retourne *l'objet* sélectionné par l'utilisateur.
- le code demandé n'est pas très long.

```
public class Exam2 {
        private JComboBox ueListe = new JComboBox();
        private JTextField heuresChamp = new JTextField(10);
        private JFrame frame = new JFrame("formulaire");

        public Exam2() {
                placeComposants();
                activeComposants();
        }

        private void activeComposants() {
          // À ÉCRIRE
        }


        /**
         *  Place les composants (NE PAS ÉCRIRE!)
         */
        private void placeComposants() { ... }

        public static void main(String[] args) {
          // Lance le programme :
          // À ÉCRIRE
        }
}
```

java.io

Class FileReader

java.lang.Object
  └ java.io.Reader
    └ java.io.InputStreamReader
      └ **java.io.FileReader**

All Implemented Interfaces:
Closeable, Readable

```
public class FileReader
extends InputStreamReader
```

Convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.

`FileReader` is meant for reading streams of characters. For reading streams of raw bytes, consider using a `FileInputStream`.

Since:
JDK1.1
See Also:
`InputStreamReader`, `FileInputStream`

Field Summary

Fields inherited from class java.io.Reader

`lock`

Constructor Summary

FileReader(File file)	Creates a new FileReader, given the File to read from.
FileReader(FileDescriptor fd)	Creates a new FileReader, given the FileDescriptor to read from.
FileReader(String fileName)	Creates a new FileReader, given the name of the file to read from.

Method Summary

Methods inherited from class java.io.InputStreamReader

`close`, `getEncoding`, `read`, `read`, `ready`

Methods inherited from class java.io.Reader

`mark`, `markSupported`, `read`, `read`, `reset`, `skip`

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Detail

FileReader


```
public FileReader(String fileName)
          throws FileNotFoundException
```

Creates a new FileReader, given the name of the file to read from.

Parameters:
`fileName` - the name of the file to read from
Throws:
`FileNotFoundException` - if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

FileReader

```
public FileReader(File file)
          throws FileNotFoundException
```

Creates a new FileReader, given the File to read from.

Parameters:
`file` - the File to read from
Throws:
`FileNotFoundException` - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

FileReader

```
public FileReader(FileDescriptor fd)
```

Creates a new FileReader, given the FileDescriptor to read from.

Parameters:
`fd` - the FileDescriptor to read from

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Java™ Platform Standard Ed. 6

java.io

Class Reader

java.lang.Object
  └ java.io.Reader

All Implemented Interfaces:
Closeable, Readable

Direct Known Subclasses:
BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader

```
public abstract class Reader
extends Object
implements Readable, Closeable
```

Abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

Since:
JDK1.1
See Also:
BufferedReader, LineNumberReader, CharArrayReader, InputStreamReader, FileReader, FilterReader, PushbackReader, PipedReader, StringReader, Writer

Field Summary

protected Object	**lock** The object used to synchronize operations on this stream.

Constructor Summary

protected	**Reader**() Creates a new character-stream reader whose critical sections will synchronize on the reader itself.
protected	**Reader**(Object lock) Creates a new character-stream reader whose critical sections will synchronize on the given object.

Method Summary

abstract void	**close**() Closes the stream and releases any system resources associated with it.
void	**mark**(int readAheadLimit) Marks the present position in the stream.
boolean	**markSupported**() Tells whether this stream supports the mark() operation.
int	**read**() Reads a single character.
int	**read**(char[] cbuf) Reads characters into an array.
abstract int	**read**(char[] cbuf, int off, int len) Reads characters into a portion of an array.
int	**read**(CharBuffer target) Attempts to read characters into the specified character buffer.
boolean	**ready**() Tells whether this stream is ready to be read.
void	**reset**() Resets the stream.
long	**skip**(long n) Skips characters.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

lock

```
protected Object lock
```

The object used to synchronize operations on this stream. For efficiency, a character-stream object may use an object other than itself to protect critical sections. A subclass should therefore use the object in this field rather than `this` or a synchronized method.

Constructor Detail

Reader

```
protected Reader()
```

Creates a new character-stream reader whose critical sections will synchronize on the reader itself.

Reader

```
protected Reader(Object lock)
```

Creates a new character-stream reader whose critical sections will synchronize on the given object.

Parameters:
`lock` - The Object to synchronize on.

Method Detail

read

```
public int read(CharBuffer target)
         throws IOException
```

Attempts to read characters into the specified character buffer. The buffer is used as a repository of characters as-is: the only changes made are the results of a put operation. No flipping or rewinding of the buffer is performed.

Specified by:
read in interface Readable
Parameters:
`target` - the buffer to read characters into
Returns:
The number of characters added to the buffer, or -1 if this source of characters is at its end
Throws:
IOException - if an I/O error occurs
NullPointerException - if target is null
`ReadOnlyBufferException` - if target is a read only buffer
Since:
1.5

read

```
public int read()
         throws IOException
```

Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.

Subclasses that intend to support efficient single-character input should override this method.

Returns:
The character read, as an integer in the range 0 to 65535 (`0x00-0xffff`), or -1 if the end of the stream has been reached

Throws:
`IOException` - If an I/O error occurs

---

read

```
public int read(char[] cbuf)
         throws IOException
```

Reads characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.

Parameters:
`cbuf` - Destination buffer

Returns:
The number of characters read, or -1 if the end of the stream has been reached

Throws:
`IOException` - If an I/O error occurs

---

read

```
public abstract int read(char[] cbuf,
                         int off,
                         int len)
                  throws IOException
```

Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.

Parameters:
`cbuf` - Destination buffer
`off` - Offset at which to start storing characters
`len` - Maximum number of characters to read

Returns:
The number of characters read, or -1 if the end of the stream has been reached

Throws:
`IOException` - If an I/O error occurs

---

skip

```
public long skip(long n)
          throws IOException
```

Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached.

Parameters:
`n` - The number of characters to skip

Returns:
The number of characters actually skipped

Throws:
`IllegalArgumentException` - If `n` is negative.
`IOException` - If an I/O error occurs

---

ready

```
public boolean ready()
             throws IOException
```

Tells whether this stream is ready to be read.

Returns:
True if the next read() is guaranteed not to block for input, false otherwise. Note that returning false does not guarantee that the next read will block.

Throws:
`IOException` - If an I/O error occurs

---

markSupported

```
public boolean markSupported()
```

Tells whether this stream supports the mark() operation. The default implementation always returns false. Subclasses should override this method.

Returns:
true if and only if this stream supports the mark operation.

---

mark

```
public void mark(int readAheadLimit)
          throws IOException
```

Marks the present position in the stream. Subsequent calls to reset() will attempt to reposition the stream to this point. Not all character-input streams support the mark() operation.

Parameters:
`readAheadLimit` - Limit on the number of characters that may be read while still preserving the mark. After reading this many characters, attempting to reset the stream may fail.

Throws:
`IOException` - If the stream does not support mark(), or if some other I/O error occurs

---

reset

```
public void reset()
          throws IOException
```

Resets the stream. If the stream has been marked, then attempt to reposition it at the mark. If the stream has not been marked, then attempt to reset it in some way appropriate to the particular stream, for example by repositioning it to its starting point. Not all character-input streams support the reset() operation, and some support reset() without supporting mark().

Throws:
`IOException` - If the stream has not been marked, or if the mark has been invalidated, or if the stream does not support reset(), or if some other I/O error occurs

---

close

```
public abstract void close()
                  throws IOException
```

Closes the stream and releases any system resources associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect.

Specified by:
`close` in interface `Closeable`

Throws:
`IOException` - If an I/O error occurs

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Java™ Platform Standard Ed. 6

java.io

Class File

java.lang.Object
  └ **java.io.File**

All Implemented Interfaces:
Serializable, Comparable<File>

```
public class File
extends Object
implements Serializable, Comparable<File>
```

An abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

1. An optional system-dependent *prefix* string, such as a disk-drive specifier, `"/"` for the UNIX root directory, or `"\\\\"` for a Microsoft Windows UNC pathname, and
2. A sequence of zero or more string *names*.

The first name in an abstract pathname may be a directory name or, in the case of Microsoft Windows UNC pathnames, a hostname. Each subsequent name in an abstract pathname denotes a directory; the last name may denote either a directory or a file. The *empty* abstract pathname has no prefix and an empty name sequence.

The conversion of a pathname string to or from an abstract pathname is inherently system-dependent. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default *separator character*. The default name-separator character is defined by the system property `file.separator`, and is made available in the public static fields `separator` and `separatorChar` of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying system.

A pathname, whether abstract or in string form, may be either *absolute* or *relative*. An absolute pathname is complete in that no other information is required in order to locate the file that it denotes. A relative pathname, in contrast, must be interpreted in terms of information taken from some other pathname. By default the classes in the `java.io` package always resolve relative pathnames against the current user directory. This directory is named by the system property `user.dir`, and is typically the directory in which the Java virtual machine was invoked.

The *parent* of an abstract pathname may be obtained by invoking the `getParent()` method of this class and consists of the pathname's prefix and each name in the pathname's name sequence except for the last. Each directory's absolute pathname is an ancestor of any `File` object with an absolute abstract pathname which begins with the directory's absolute pathname. For example, the directory denoted by the abstract pathname `"/usr"` is an ancestor of the directory denoted by the pathname `"/usr/local/bin"`.

The prefix concept is used to handle root directories on UNIX platforms, and drive specifiers, root directories and UNC pathnames on Microsoft Windows platforms, as follows:

- For UNIX platforms, the prefix of an absolute pathname is always `"/"`. Relative pathnames have no prefix. The abstract pathname denoting the root directory has the prefix `"/"` and an empty name sequence.
- For Microsoft Windows platforms, the prefix of a pathname that contains a drive specifier consists of the drive letter followed by `":"` and possibly followed by `"\\"` if the pathname is absolute. The prefix of a UNC pathname is `"\\\\"`; the hostname and the share name are the first two names in the name sequence. A relative pathname that does not specify a drive has no prefix.

Instances of this class may or may not denote an actual file-system object such as a file or a directory. If it does denote such an object then that object resides in a *partition*. A partition is an operating system-specific portion of storage for a file system. A single storage device (e.g. a physical disk-drive, flash memory, CD-ROM) may contain multiple partitions. The object, if any, will reside on the partition named by some ancestor of the absolute form of this pathname.

A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as *access permissions*. The file system may have multiple sets of access permissions on a single object. For example, one set may apply to the object's *owner*, and another may apply to all other users. The access permissions on an object may cause some methods in this class to

fail.

Instances of the `File` class are immutable; that is, once created, the abstract pathname represented by a `File` object will never change.

Since:
JDK1.0
See Also:
Serialized Form

Field Summary

static String	**pathSeparator** The system-dependent path-separator character, represented as a string for convenience.
static char	**pathSeparatorChar** The system-dependent path-separator character.
static String	**separator** The system-dependent default name-separator character, represented as a string for convenience.
static char	**separatorChar** The system-dependent default name-separator character.

Constructor Summary

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

Method Summary

boolean	**canExecute**() Tests whether the application can execute the file denoted by this abstract pathname.
boolean	**canRead**() Tests whether the application can read the file denoted by this abstract pathname.
boolean	**canWrite**() Tests whether the application can modify the file denoted by this abstract pathname.
int	**compareTo**(File pathname) Compares two abstract pathnames lexicographically.
boolean	**createNewFile**() Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
static File	**createTempFile**(String prefix, String suffix) Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static File	**createTempFile**(String prefix, String suffix, File directory) Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
boolean	**delete**() Deletes the file or directory denoted by this abstract pathname.
void	**deleteOnExit**() Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
boolean	**equals**(Object obj) Tests this abstract pathname for equality with the given object.
boolean	**exists**() Tests whether the file or directory denoted by this abstract pathname exists.
File	**getAbsoluteFile**() Returns the absolute form of this abstract pathname.
String	**getAbsolutePath**() Returns the absolute pathname string of this abstract pathname.

File	**getCanonicalFile**() Returns the canonical form of this abstract pathname.
String	**getCanonicalPath**() Returns the canonical pathname string of this abstract pathname.
long	**getFreeSpace**() Returns the number of unallocated bytes in the partition named by this abstract path name.
String	**getName**() Returns the name of the file or directory denoted by this abstract pathname.
String	**getParent**() Returns the pathname string of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.
File	**getParentFile**() Returns the abstract pathname of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.
String	**getPath**() Converts this abstract pathname into a pathname string.
long	**getTotalSpace**() Returns the size of the partition named by this abstract pathname.
long	**getUsableSpace**() Returns the number of bytes available to this virtual machine on the partition named by this abstract pathname.
int	**hashCode**() Computes a hash code for this abstract pathname.
boolean	**isAbsolute**() Tests whether this abstract pathname is absolute.
boolean	**isDirectory**() Tests whether the file denoted by this abstract pathname is a directory.
boolean	**isFile**() Tests whether the file denoted by this abstract pathname is a normal file.
boolean	**isHidden**() Tests whether the file named by this abstract pathname is a hidden file.
long	**lastModified**() Returns the time that the file denoted by this abstract pathname was last modified.
long	**length**() Returns the length of the file denoted by this abstract pathname.
String[]	**list**() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String[]	**list**(FilenameFilter filter) Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
File[]	**listFiles**(FileFilter filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**(FilenameFilter filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
static File[]	**listRoots**() List the available filesystem roots.
boolean	**mkdir**() Creates the directory named by this abstract pathname.
boolean	**mkdirs**() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
boolean	**renameTo**(File dest) Renames the file denoted by this abstract pathname.
boolean	**setExecutable**(boolean executable) A convenience method to set the owner's execute permission for this abstract pathname.
boolean	**setExecutable**(boolean executable, boolean ownerOnly) Sets the owner's or everybody's execute permission for this abstract pathname.

boolean	**setLastModified**(long time) Sets the last-modified time of the file or directory named by this abstract pathname.
boolean	**setReadable**(boolean readable) A convenience method to set the owner's read permission for this abstract pathname.
boolean	**setReadable**(boolean readable, boolean ownerOnly) Sets the owner's or everybody's read permission for this abstract pathname.
boolean	**setReadOnly**() Marks the file or directory named by this abstract pathname so that only read operations are allowed.
boolean	**setWritable**(boolean writable) A convenience method to set the owner's write permission for this abstract pathname.
boolean	**setWritable**(boolean writable, boolean ownerOnly) Sets the owner's or everybody's write permission for this abstract pathname.
String	**toString**() Returns the pathname string of this abstract pathname.
URI	**toURI**() Constructs a `file:` URI that represents this abstract pathname.
URL	**toURL**() **Deprecated.** *This method does not automatically escape characters that are illegal in URLs. It is recommended that new code convert an abstract pathname into a URL by first converting it into a URI, via the `toURI` method, and then converting the URI into a URL via the `URI.toURL` method.*

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

separatorChar

```
public static final char separatorChar
```

The system-dependent default name-separator character. This field is initialized to contain the first character of the value of the system property `file.separator`. On UNIX systems the value of this field is `'/'`; on Microsoft Windows systems it is `'\\'`.

See Also:
System.getProperty(java.lang.String)

---

separator

```
public static final String separator
```

The system-dependent default name-separator character, represented as a string for convenience. This string contains a single character, namely separatorChar.

---

pathSeparatorChar

```
public static final char pathSeparatorChar
```

The system-dependent path-separator character. This field is initialized to contain the first character of the value of the system property `path.separator`. This character is used to separate filenames in a sequence of files given as a *path list*. On UNIX systems, this character is `':'`; on Microsoft Windows systems it is `';'`.

See Also:
System.getProperty(java.lang.String)

---

pathSeparator

```
public static final String pathSeparator
```

The system-dependent path-separator character, represented as a string for convenience. This string contains a single character, namely pathSeparatorChar.