Manipulation de Fichiers

Table des matières

1. Introduction

Il peut parfois être pratique ou même indispensable de pouvoir lire et écrire dans les fichiers. La mémoire vive ou RAM est une mémoire temporaire ; les données disparaissent lors de l'extinction de l'ordinateur. Heureusement, on peut lire et écrire dans des fichiers. Ces fichiers seront écrits sur le disque dur de l'ordinateur : l'avantage est donc qu'ils restent là, même si le programme ou l'ordinateur s'arrête.

2. Langage C

2.1. Ouvrir et fermer un fichier

Pour lire et écrire dans des fichiers, nous allons nous servir de fonctions situées dans les bibliothèques stdio.h et stdlib.h.

Voici ce qu'il faut faire à chaque fois dans l'ordre quand on veut ouvrir un fichier, que ce soit pour le lire ou pour y écrire :

1. On appelle la fonction d'ouverture de fichier fopen qui nous renvoie un pointeur sur le fichier.
2. On vérifie si l'ouverture a réussi (c'est-à-dire si le fichier existait) en testant la valeur du pointeur qu'on a reçu. Si le pointeur vaut NULL, c'est que l'ouverture du fichier n'a pas fonctionné, dans ce cas on ne peut pas continuer (il faut afficher un message d'erreur).
3. Si l'ouverture a fonctionné (si le pointeur est différent de NULL donc), alors on peut lire et écrire dans le fichier.
4. Une fois qu'on a terminé de travailler sur le fichier, il faut penser à le « fermer » avec la fonction fclose.

Exemple :

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
	FILE* fichier = NULL;	// déclaration et initialisation du pointeur

	// Ouverture du fichier
	fichier = fopen("test.txt", "r+");

	if ( fichier == NULL )
	{
		// On affiche un message d'erreur si on veut
		printf("Impossible d'ouvrir le fichier test.txt");
		return -1;
	}
	else
	{
		// On peut lire et écrire dans le fichier
	}

	fclose(fichier); // On ferme le fichier qui a été ouvert
```

```
    return 0;
}
```

Voyons le prototype de la fonction fopen :

```
FILE* fopen(const char* nomDuFichier, const char* modeOuverture);
```

Cette fonction attend deux paramètres :

- le nom du fichier à ouvrir ;
- le mode d'ouverture du fichier, c'est-à-dire une indication qui mentionne ce que vous voulez faire : seulement écrire dans le fichier, seulement le lire, ou les deux à la fois.

Cette fonction renvoie un pointeur sur FILE. C'est un pointeur sur une structure de type FILE. Cette structure est définie dans stdio.h.

Voici les modes d'ouverture possibles :

- "r" : lecture seule. Vous pourrez lire le contenu du fichier, mais pas y écrire. Le fichier doit avoir été créé au préalable.
- "w" : écriture seule. Vous pourrez écrire dans le fichier, mais pas lire son contenu. Si le fichier n'existe pas, il sera créé.
- "a" : mode d'ajout. Vous écrirez dans le fichier, en partant de la fin du fichier. Vous ajouterez donc du texte à la fin du fichier. Si le fichier n'existe pas, il sera créé.
- "r+" : lecture et écriture. Vous pourrez lire et écrire dans le fichier. Le fichier doit avoir été créé au préalable.
- "w+" : lecture et écriture, avec suppression du contenu au préalable. Le fichier est donc d'abord vidé de son contenu, vous pouvez y écrire, et le lire ensuite. Si le fichier n'existe pas, il sera créé.
- "a+" : ajout en lecture / écriture à la fin. Vous écrivez et lisez du texte à partir de la fin du fichier. Si le fichier n'existe pas, il sera créé.

Pour chaque mode, si vous ajoutez un "b" après le premier caractère ("rb", "wb", "ab", "rb+", "wb+", "ab+"), alors le fichier est ouvert en mode binaire (le mode texte est fait pour stocker du texte, tandis que le mode binaire permet de stocker des données autres que de type char).

Le fichier est situé dans le même dossier que l'exécutable à moins que le nom de chemin soit spécifié :

```
fichier = fopen("dossier/test.txt", "r+");    // chemin relatif
fichier = fopen("/home/user/dossier/test.txt", "r+");      // chemin absolu
```

Remarque : les chemins absolus ne fonctionnent que sur un OS précis. Ce n'est pas une solution portable. Sous M$ Windows il aurait fallut écrire :

```
fichier = fopen("C:\\Documents\\user\\dossier\\test.txt", "r+");
```

Après l'ouverture du fichier, le pointeur fichier devrait contenir l'adresse de la structure de type FILE qui sert de descripteur de fichier. Celui-ci a été chargé en mémoire par la fonction fopen().

À partir de là, deux possibilités :

- soit l'ouverture a réussi, et vous pouvez continuer (c'est-à-dire commencer à lire et écrire dans le fichier) ;

- soit l'ouverture a échoué parce que le fichier n'existait pas ou était utilisé par un autre programme. Dans ce cas, vous devez arrêter de travailler sur le fichier.

Juste après l'ouverture du fichier, il faut impérativement vérifier si le pointeur vaut NULL (l'ouverture a échoué) ou s'il vaut autre chose que NULL (l'ouverture a réussi).

Une fois que vous aurez fini de travailler avec le fichier, il faudra le « fermer ». On utilise pour cela la fonction fclose qui a pour rôle de libérer la mémoire, c'est-à-dire supprimer le fichier chargé dans la mémoire vive.

Son prototype est : `int fclose(FILE* pointeurSurFichier);`

Cette fonction prend un paramètre : le pointeur sur le fichier.

Elle renvoie un int qui indique si elle a réussi à fermer le fichier. Cet int vaut :

- 0 : si la fermeture a réussi
- EOF : si la fermeture a échoué. EOF est un define situé dans stdio.h qui correspond à un nombre spécial, utilisé pour dire soit qu'il y a eu une erreur, soit que nous sommes arrivés à la fin du fichier.

Si vous oubliez de libérer la mémoire, votre programme risque à la fin de prendre énormément de mémoire qu'il n'utilise plus.

2.2. Différentes méthodes d'écriture

Il existe plusieurs fonctions capables d'écrire dans un fichier. Ce sera à vous de choisir celle qui est la plus adaptée à votre cas :

- fputc : écrit un caractère dans le fichier (UN SEUL caractère à la fois) ;
- fputs : écrit une chaîne dans le fichier ;
- fprintf : écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.

Voici le prototype de fputc : `int fputc(int caractere, FILE* pointeurSurFichier);`

Elle prend deux paramètres :

- Le caractère à écrire
- Le pointeur sur le fichier dans lequel écrire.

La fonction retourne EOF si l'écriture a échoué, sinon il a une autre valeur.

Exemple : le code suivant écrit la lettre 'A' dans test.txt.

```
int main(int argc, char *argv[])
{
	FILE* fichier = NULL;

	fichier = fopen("test.txt", "w");

	if (fichier != NULL)
	{
		fputc('A', fichier); // Écriture du caractère A
		fclose(fichier);
	}
```

```
	return 0;
}
```

Voici le prototype de fputs : `char* fputs(const char* chaine, FILE* pointeurSurFichier);`

Elle prend deux paramètres :

- chaine : la chaîne à écrire.
- pointeurSurFichier : comme pour fputc, il s'agit de votre pointeur de type FILE* sur le fichier que vous avez ouvert.

La fonction renvoie EOF s'il y a eu une erreur, sinon c'est que cela a fonctionné.

Voici un autre exemplaire de la fonction printf. Celle-ci peut être utilisée pour écrire dans un fichier. Elle s'utilise de la même manière que printf d'ailleurs, excepté le fait que vous devez indiquer un pointeur de FILE en premier paramètre.

Exemple :

```
int main(int argc, char *argv[])
{
	FILE*  fichier = fopen("test.txt", "w");	// ouverture
	int    heure = 0;

	if ( fichier != NULL )
	{
		// On l'heure
		printf("indiquez l'heure : ");
		scanf("%d", &heure);

		// On l'écrit dans le fichier
		fprintf(fichier, "Le fichier a été écrit à %02d heures", heure);

		return fclose(fichier);	// fermeture
	}

	return -1;
}
```

2.3. Différentes méthodes de lecture

Nous pouvons utiliser quasiment les mêmes fonctions que pour l'écriture, le nom change juste un petit peu :

- fgetc : lit un caractère ;
- fgets : lit une chaîne ;
- fscanf : lit une chaîne formatée.

Voici le prototype de fgetc : `int fgetc(FILE* pointeurDeFichier);`

Cette fonction retourne un int : c'est le caractère qui a été lu.

Si la fonction n'a pas pu lire de caractère, elle retourne EOF.

fgetc avance dans le fichier en lisant séquentiellement caractère par caractère :

```
int main(int argc, char *argv[])
```

```
{
	FILE*  fichier = fopen("test.txt", "r");
	int    caractereActuel = 0;

	if ( fichier != NULL )
	{
		// Boucle de lecture des caractères un à un
		do
		{
			caractereActuel = fgetc(fichier);	// On lit le caractère
			printf("%c", caractereActuel);		// On l'affiche
		// tant que fgetc n'a pas retourné EOF (fin de fichier)
		} while ( caractereActuel != EOF );

		return fclose(fichier);
	}

	return -1;
}
```

Voici le prototype de fgets :

```
char* fgets(char* chaine, int nbreDeCaracteresALire, FILE* pointeurSurFichier);
```

Cette fonction a besoin du nombre de caractères à lire. Cela oblige la fonction fgets de s'arrêter de lire la ligne si elle contient plus de X caractères.

La fonction fgets renvoie NULL si elle n'est pas parvenue à lire le nombre de caractères demandé.

Ce code source lit et affiche tout le contenu de mon fichier, ligne par ligne :

```
#define TAILLE_MAX 1000

int main(int argc, char *argv[])
{
    FILE*    fichier = fopen("test.txt", "r");
    char     chaine[TAILLE_MAX] = "";

	if ( fichier != NULL )
	{
		// On lit le fichier tant qu'on ne reçoit pas d'erreur (NULL)
		while ( fgets(chaine, TAILLE_MAX, fichier) != NULL )
		{
			printf("%s", chaine); // On affiche la chaîne qu'on vient de lire
		}

		return fclose(fichier);
	}

	return -1;
}
```

La fonction fscanf lit dans un fichier qui doit avoir un format précis.

Exemple : un fichier de scores qui contient trois nombres séparés par un espace.

```
int main(int argc, char *argv[])
{
    FILE*    fichier = fopen("test.txt", "r");
```

```
	if ( fichier != NULL )
	{
		// On lit le fichier selon un format précis
		fscanf(fichier, "%d %d %d", &score[0], &score[1], &score[2]);
		printf("Les meilleurs scores sont : %d, %d et %d",
			score[0], score[1], score[2]);

		return fclose(fichier);
	}

	return -1;
}
```

2.4. Se déplacer dans un fichier

Chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier. Le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier.

Il existe trois fonctions à connaître :

- ftell : indique à la position actuelle dans le fichier ;
- fseek : positionne le curseur à un endroit précis ;
- rewind : remet le curseur au début du fichier.

Voici le prototype de ftell : `long ftell(FILE* pointeurSurFichier);`

Cette fonction renvoie la position actuelle du curseur sous la forme d'un long

Prototype de fseek : `int fseek(FILE* pointeurSurFichier, long deplacement, int origine);`

Cette fonction permet de déplacer le curseur d'un certain nombre de caractères (indiqué par deplacement) à partir de la position indiquée par origine. Le nombre deplacement peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).

Le nombre origine, peut avoir comme valeur l'une des trois constantes :

- SEEK_SET : indique le début du fichier ;
- SEEK_CUR : indique la position actuelle du curseur ;
- SEEK_END : indique la fin du fichier.

Remarque : Si vous écrivez après avoir fait un fseek qui mène à la fin du fichier, cela ajoutera vos informations à la suite dans le fichier (le fichier sera complété). En revanche, si vous placez le curseur au début et que vous écrivez, cela écrasera le texte.

Prototype de rewind : `void rewind(FILE* pointeurSurFichier);`

Cette fonction est équivalente à utiliser fseek pour renvoyer à la position 0 dans le fichier.

3. Langage C++

3.1. Ouvrir et fermer un fichier

La première chose à faire quand on veut manipuler des fichiers, c'est de les ouvrir. Pour cela, il faut commencer par inclure le fichier d'en-tête <fstream>. fstream correspond à file stream, « flux vers les fichiers » :

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ofstream monFlux("C:/Nanoc/data.txt");
   //Déclaration d'un flux permettant d'écrire dans le fichier
   // C:/Nanoc/data.txt
   return 0;
}
```

On a indiqué entre guillemets le chemin d'accès au fichier. Ce chemin doit prendre l'une ou l'autre des deux formes suivantes :

- Un chemin absolu, c'est-à-dire montrant l'emplacement du fichier depuis la racine du disque. Par exemple : C:/Nanoc/C++/Fichiers/data.txt.
- Un chemin relatif, c'est-à-dire montrant l'emplacement du fichier depuis l'endroit où se situe le programme sur le disque. Par exemple : Fichiers/data.txt si le programme se situe dans le dossier C:/Nanoc/C++/.

Remarque : si le fichier n'existe pas, le programme le crée automatiquement !

Des problèmes peuvent survenir lors de l'ouverture d'un fichier, si le fichier ne vous appartient pas ou si le disque dur est plein par exemple. C'est pour cela qu'il faut toujours tester si tout s'est bien passé.

```
Ofstream     monFlux;                  //Un flux sans fichier associé
monFlux.open("C:/Nanoc/data.txt");     //On ouvre le fichier C:/Nanoc/data.txt

if ( monFlux )  //On teste si tout est OK
{
      //Tout est OK, on peut utiliser le fichier
}
else
{
      cout << "ERREUR : Impossible d'ouvrir le fichier." << endl;
      exit(-1);
}

monFlux.close();  //On ferme le fichier
```

Remarque : les fichiers ouverts sont automatiquement refermés lorsque l'on sort du bloc où le flux est déclaré. Il n'est pas nécessaire d'utiliser la méthode close().

3.2. Écrire dans un flux

On parle de flux pour désigner les moyens de communication d'un programme avec l'extérieur.

Nous parlerons donc de flux vers les fichiers et nous utiliserons l'objet ofstream.

Tout se passe comme pour cout. C'est donc sans surprise que le moyen d'envoyer des informations dans un flux sont les chevrons (<<).

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
	string const nomFichier("C:/Nanoc/data.txt");
	ofstream	monFlux(nomFichier.c_str());

	if ( monFlux )
	{
		const unsigned int  pi(3.14159);
		monFlux << "La valeur approchée de pi est : " << pi << endl;
	}
	else
	{
		cout << "ERREUR : Impossible d'ouvrir le fichier." << endl;
		return -1;
	}

	monFlux.close();

	return 0;
}
```

Le langage C++ fournit également deux autres méthodes pour écrire dans un fichier :

- `put(char)` : fournit une alternative à << pour insérer un caractère dans le flot
- `write(const char* str,int length)` : permet d'insérer les length caractères de la chaîne str sur le flot de sortie.

3.3. Lire un fichier

Le principe est exactement le même : on va utiliser un objet ifstream au lieu d'un ofstream. Il y a trois manières différentes de lire un fichier :

- Ligne par ligne, en utilisant la fonction getline() ;
- Mot par mot, en utilisant les chevrons >> ;
- Utiliser les méthodes de l'objet.

1. La première méthode permet de récupérer une ligne entière et de la stocker dans une chaîne de caractères.

   ```
   string ligne;
   getline(monFlux, ligne); //On lit une ligne complète
   ```

2. La deuxième manière consiste à utiliser les propriétés des flux :

   ```
   double nombre;
   monFlux >> nombre; //Lit un nombre à virgule depuis le fichier
   ```

```
string mot;
monFlux >> mot;    //Lit un mot depuis le fichier
```

Cette méthode lit ce qui se trouve entre l'endroit où l'on se situe dans le fichier et l'espace suivant. Ce qui est lu est alors traduit en double, int ou string selon le type de variable dans lequel on écrit.

3. Le langage C++ fournit également deux autres méthodes pour lire un fichier :

   `get(char)` : fournit une alternative à >> pour extraire un caractère du flot. Cette méthode lit tous les caractères. Les espaces, retours à la ligne et tabulations sont, entre autres, lus par cette méthode.

   `getline(char *buf, int limit, char delim='\n')` : permet de lire et d'affecter à buf une ligne entière, jusqu'au caratère '\n' (par défaut) lorsque l'on ne veut pas sauter de blancs. getline() extrait au plus limit-1 caractères et ajoute un caractère NULL à la fin.

Pour savoir si l'on peut continuer à lire un fichier, il faut tester la valeur renvoyée par la fonction getline() ou la méthode get(). Si la valeur renvoyée vaut true, tout va bien, la lecture peut continuer. Si la valeur est false, c'est qu'on est arrivé à la fin du fichier ou qu'il y a eu une erreur. Dans les deux cas, il faut s'arrêter de lire.

Exemple :

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
	ifstream	fichier("C:/Nanoc/fichier.txt");

	if ( fichier )
	{
		//L'ouverture s'est bien passée, on peut donc lire
		string ligne; //Une variable pour stocker les lignes lues

		while ( getline(fichier, ligne) ) //Tant qu'on n'est pas à la fin, on lit
		{
			//Et on l'affiche dans la console
			cout << ligne << endl;
		}
	}
	else
	{
		cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << endl;
		return -1;
	}

	return 0;
}
```

3.4. Les différents modes d'ouverture

Les constructeurs des objets ofstream et ifstream admettent un deuxième paramètre dans leur constructeur qui spécifie le mode d'ouverture du fichier, c'est-à-dire une indication qui mentionne ce

que vous voulez faire : seulement écrire dans le fichier, seulement le lire, ou les deux à la fois.

Voici les modes d'ouverture possibles :

ios::app	Si le fichier existe, son contenu est conservé et toutes les données insérées seront ajoutées en fin de fichier (app est l'abréviation de append).
ios::nocreate	Si le fichier n'existe pas, il ne sera pas créé, mais il y aura échec de l'ouverture.
ios::noreplace	Si le fichier existe déjà il y aura erreur d'ouverture (sauf si le mode ios::app est également spécifié).
ios::binary	Si ce mode n'est pas spécifié, le fichier est ouvert en mode "texte", ce qui signifie que divers ajustements seront automatiquement effectués, en fonction des nécessités imposées par le système d'exploitation. Ces ajustements concernent notamment la représentation du passage à la ligne, de façon à ce que l'insertion du caractère '\n' se traduise toujours par la présence dans le fichier du (ou des) caractères que le système d'exploitation considère être la bonne représentation d'un saut de ligne. Attention à ne pas confondre le mode texte avec le format texte. Le mode texte peut être appliqué aussi bien à des fichiers au format texte qu'à d'autres types de fichiers. Notez cependant que les ajustements réalisés en mode texte ont peu de chances de donner des résultats intéressants si le fichier concerné n'est pas au format texte !

3.5. Se déplacer dans un fichier

Chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier. Le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier.

Il existe une fonction permettant de savoir à quel octet du fichier on se trouve. Autrement dit, elle permet de savoir à quel caractère du fichier on se situe. Cette fonction n'a pas le même nom pour les flux entrant et sortant :

- Pour ifstream : tellg()
- Pour ofstream : tellp()

En revanche, elles s'utilisent toutes les deux de la même manière :

```
ofstream fichier("C:/Nanoc/data.txt");

int    position = fichier.tellp(); //On récupére la position
cout << "Nous nous situons au " << position << "eme caractere du fichier." << endl;
```

Pour se déplacer, il existe deux fonctions, une pour chaque type de flux :

- Pour ifstream : seekg()
- Pour ofstream : seekp()

Ces fonctions reçoivent deux arguments : une position dans le fichier et un nombre de caractères à ajouter à cette position.

Ex : `flux.seekp(nombreCaracteres, position);`

Les trois positions possibles sont :

- le début du fichier : ios::beg ;

- la fin du fichier : ios::end ;
- la position actuelle : ios::cur.

Remarque : pour connaître la taille d'un fichier, on se déplace à la fin et on demande au flux de nous dire où il se trouve.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
	ifstream fichier("C:/Nanoc/data.txt");  //On ouvre le fichier
	fichier.seekg(0, ios::end);  //On se déplace à la fin du fichier

	//On récupère la position qui correspond donc a la taille du fichier !
	int taille = fichier.tellg();

	cout << "Taille du fichier : " << taille << " octets." << endl;

	return 0;
}
```

4. Qt

4.1. La lecture et les différents modes d'ouverture

Les programmeurs de Qt ont créé la classe QFile pour manipuler les fichiers.

On récupère le contenu d'un fichier le plus couramment dans une instance de QString (ou QByteArray pour la manipulation des fichiers binaires). On peut aussi utiliser un objet flux QtextStream.

Nous allons déclarer un objet QFile en lui précisant le chemin du fichier qui doit se trouver dans le dossier de l'exécutable :

```
QFile fichier("test.txt");
```

Pour l'ouverture du fichier, on utilisera la fonction QFile::open() et on enverra en paramètre un flag de QIODevice pour définir le mode d'ouverture. Il en existe huit :

Constante	Valeur correspondante	Description
QIODevice::ReadOnly	0x0001	Ouverture en lecture seule.
QIODevice::WriteOnly	0x0002	Ouverture en écriture seule.
QIODevice::ReadWrite	0x0001 | 0x0002	Ouverture en lecture/écriture.
QIODevice::Truncate	0x0008	- À utiliser avec WriteOnly ou ReadWrite ; - Spécifie que le contenu sera au préalable supprimé.
QIODevice::Append	0x0004	- Peut s'utiliser avec tout ; - Spécifie que le curseur sera placé à la fin du fichier.

QIODevice::Text	0x0010	- Peut s'utiliser avec tout ; - Apporte quelques optimisations quand il s'agit d'un texte.
QIODevice::NotOpen	0x0000	N'ouvre tout simplement pas le fichier.
QIODevice::Unbuffered	0x0020	Désactive l'utilisation d'un buffer.

La fonction open revoie true si le fichier a pu être ouvert et dans le cas contraire, false.

Si le fichier a pu être ouvert, il faut récupérer son contenu avant de fermer le fichier à l'aide de la fonction membre close(). La méthode la plus simple consiste à utiliser la fonction readAll() et à stocker la chaîne renvoyée dans le Qstring.

```
int main(int argc, char **argv)
{
	Qapplication a(argc, argv);
	QtextEdit    zoneTexte;

	zoneTexte.setGeometry(100,100,400,200);
	zoneTexte.setReadOnly(true);

	Qstring      texte;
	Qfile  fichier("test.txt");
	if ( fichier.open(QIODevice::ReadOnly | QIODevice::Text) )
	{
		texte = fichier.readAll();
		fichier.close();
	}
	else
		texte = "Impossible d'ouvrir le fichier !";

	zoneTexte.setText(texte);
	zoneTexte.show();

	return a.exec();
}
```

Il existe encore un autre moyen plus précis de récupérer le contenu d'un fichier. C'est cette méthode qui utilise un objet flux de type QTextStream en procédant ainsi :

```
if ( fichier.open(QIODevice::ReadOnly | QIODevice::Text) )
{
	QTextStream flux(&fichier);

	while ( !flux.atEnd() )
		// attention : les lignes sont mises à la chaîne sans retour à la ligne
		texte += flux.readLine() + '\n';       // '\n' permet le retour à la ligne
	texte.resize(texte.size()-1);                // Élimine le '\n' en trop

	fichier.close();
}
```

Ici, on déclare un objet flux QTextStream en lui envoyant l'adresse de l'objet QFile "fichier" en lui permettant ainsi de lire son contenu. Tant qu'on n'est pas arrivé au bout du fichier, on continue à récupérer les lignes.

4.2. L'écriture dans les fichiers

Les classes de Qt sont déjà tellement simplifiées qu'elles nous permettent de ne pas trop s'attarder sur ce concept. Il existe deux méthodes principales pour écrire dans un fichier :

- avec un objet flux de type QTextStream, très pratique et simple d'utilisation ;
- sans QTextStream en utilisant directement les méthodes de Qt.

4.2.1. objet flux QTextStream

Exemple : dans le programme ci-dessous

1. On demande à l'utilisateur le chemin absolu ou relatif du fichier dans lequel écrire, si ce dernier n'existe pas déjà, on le crée. On se servira de QInputDialog::getText et de QString.
2. Si le QString récupéré n'est pas vide, on continue.
3. On demande à l'utilisateur de saisir dans un QInputDialog::getText (encore) le texte à écrire dans le fichier.
4. On crée un objet de type QFile pour l'écriture dans le fichier.
5. On ouvre le fichier en écriture en utilisant les flags QIODevice::WriteOnly et QIODevice::Text.
6. On crée un objet flux de type QTextStream pour manipuler le flux entrant du fichier.
7. On écrit le texte saisi par l'utilisateur dans le fichier en nous servant de l'objet flux précédemment déclaré.
8. On ferme le fichier, on quitte le programme.

```
int main(int argc, char** argv)
{
	Qapplication app(argc, argv);
	Qstring	chemin, texte;

	// saisie du chemin du fichier
	while ( (chemin = QInputDialog::getText(NULL,"Fichier","Chemin du
fichier ?")).isEmpty() )
		QMessageBox::critical(NULL,"Erreur","Aucun chemin spécifié !");

	// saisie du texte à enregistrer
	while ( (texte = QInputDialog::getText(NULL, "Texte", "Texte à écrire dans
"+chemin.toLatin1())).isEmpty() )
		QMessageBox::critical(NULL,"Erreur","Aucun texte spécifié !");

	// ouvre  le fichier est ouvert en écriture seule
	QFile fichier(chemin);
	fichier.open(QIODevice::WriteOnly | QIODevice::Text);

	QTextStream flux(&fichier);	// création  objet flux QTextStream
	flux << texte;			// envoie du texte dans le flux

	fichier.close();

	exit(0);	// permet de quitter le programme
}
```

4.2.2. méthodes Qt

Voici trois des fonctions membres définies dans QFile les plus utilisées :

Fonction	Paramètre(s)	Description
write	const char * data, qint64 maxSize	Cette fonction écrit un texte brut dans le fichier. On doit préciser le nombre de caractères de la chaîne en second paramètre.
Write	const QByteArray & byteArray	Cette fonction (surchargée) écrit le contenu d'un objet QByteArray dans le fichier.
PutChar	char c	Cette fois, la fonction n'écrit qu'un seul caractère dans le fichier. Le curseur est placé après le caractère écrit.

Pour écrire le texte spécifié à l'aide de la fonction write(), on écrira :

```
fichier.write(texte.toLatin1(),texte.size());
```

Si on passe par un objet QByteArray, on doit écrire :

```
QByteArray bytes(texte.toLatin1());
fichier.write(bytes);
```

NB : la méthode toLatin1() de QString renvoie un objet QbyteArray.

Si on passe par la méthode putChar, on doit écrire :

```
fichier.putChar(texte.at(0).toLatin1());
```

Pour enregistrer un texte caractère par caractère :

```
for (QString::iterator it = texte.begin(); it != texte.end(); it++)
      fichier.putChar((*it).toLatin1());
```

5. Langage Python

Voici quelques utilisations habituelles du type file en python. Dans la pratique, il est recommandé de toujours utiliser l'instruction with :

```
# de cette manière, on garantit la fermeture du fichier
with open("test.txt", "w") as sortie:
      for i in range(2):
            sortie.write("{}\n".format(i))
```

Depuis son introduction dans python-2.5, cette forme est devenue très populaire car elle présente le gros avantage de garantir que le fichier sera bien fermé, et cela même si une exception devait être levée à l'intérieur du bloc with. Et marginalement le code est plus lisible dans cette forme.

5.1. Les modes d'ouverture

Les modes d'ouverture les plus utilisés sont :

- 'r' (la chaîne contenant l'unique caractère r) pour ouvrir un fichier en lecture seulement;
- 'w' en écriture seulement; le contenu précédent du fichier, s'il existait, est perdu;
- 'a' en écriture seulement, mais pour ajouter du contenu à la fin de fichier.

Voici par exemple comment on pourrait ajouter du texte dans le fichier test.txt qui devrait, à ce stade contenir 2 entiers :

```
# on ouvre le fichier en mode 'a' comme append - ou ajouter
with open("test.txt", "a") as sortie:
	for i in range(100, 102):
		sortie.write("{}\n".format(i))

# maintenant on regarde ce que contient le fichier
with open("test.txt") as entree: # remarquez que sans 'mode', on ouvre en lecture seule
	for line in entree:
		# comme line contient deja la fin de ligne
		# on ajoute une virgule pour eviter une deuxieme fin de ligne
		print line,
```

Il existe de nombreuses variantes au mode d'ouverture, pour par exemple :

- ouvrir le fichier en lecture et en écriture,
- ouvrir le fichier en mode binaire,
- utiliser le mode dit universal newlines qui permet de s'affranchir des différences de fin de ligne entre les fichiers produits sur, d'une part linux et MacOS, et d'autre part Windows.

5.2. Lire un contenu – haut niveau

Les fichiers textuels classiques se lisent en général, comme on vient d'ailleurs de le faire, avec une simple boucle for sur l'objet fichier, qui itère sur les lignes du fichier. Cette méthode est recommandée car elle est efficace, et n'implique pas notamment de charger l'intégralité du fichier en mémoire.

On trouve aussi, dans du code plus ancien, l'appel à la méthode readlines qui renvoie une liste de lignes :

```
# il faut éviter cette forme qu'on peut trouver dans du code ancien
with open("test.txt") as entree:
	for line in entree.readlines():
		print line,
```

Cette méthode implique de charger l'intégralité du fichier en mémoire alors que l'utilisation du fichier comme un itérateur est de loin préférable.

5.3. Un fichier est un itérateur

Un fichier - qui donc est itérable puisqu'on peut le lire par une boucle for - est aussi son propre itérateur :

```
# un fichier est son propre itérateur
with open("s1.txt") as entree:
	print entree.__iter__() is entree
```

Les autres types de base ont leur itérateurs implémentés comme des objets séparés. Ce choix permet notamment de réaliser deux boucles imbriquées sur la même liste :

```
# deux boucles imbriquées sur la même liste fonctionnent comme attendu
liste = [1, 2]
for i i n liste:
	for j i n liste:
```

```
            print i, "x", j
```

Par contre, écrire deux boucles for imbriquées sur le même objet fichier ne fonctionnerait pas comme on pourrait s'y attendre :

```
# Si on essaie d'écrire deux boucles imbriquées
# sur le même objet fichier, le résultat est inattendu
with open("s1.txt") as entree:
      for l1 i n entree:
            # on enleve les fins de ligne
            l1 = l1.strip()
            for l2 i n entree:
                  # on enleve les fins de ligne
                  l2 = l2.strip()
                  print l1, "x", l2
```

5.4. Lire un contenu – bas niveau

Vous pouvez également accéder à des fonctions de beaucoup plus bas niveau, notamment celle fournies directement par le système d'exploitation ; nous allons en décrire deux parmi les plus utiles.

La méthode read() permet de lire dans le fichier un buffer d'une certaine taille:

```
# lire dans le fichier deux blocs de 4 caractères
with open("test.txt") as entree:
      for bloc i n xrange(2):
            print "Bloc {} >>{}<<".format(bloc, entree.read(4))
```

Le premier bloc contient bien 4 caractères si on compte les deux sauts de ligne : `Bloc1 = "0\n1\n"`

Le second bloc contient quant à lui : `Bloc2 = "100\n"`

Avec la méthode read mais sans argument, on peut lire tout le fichier d'un coup - mais là encore prenez garde à l'utilisation de la mémoire :

```
with open("s1.txt") as entree:
      contenu = entree.read()
      print "Dans un contenu de longueur {} " \
            "on a trouvé {} occurrences de 0" \
            .format(len(contenu), contenu.count('0'))
```

5.5. La méthode flush

Les entrées-sortie sur fichier sont bien souvent bufferisées par le système d'exploitation. Cela signifie qu'un appel à write ne provoque pas forcément une écriture immédiate, car pour des raisons de performance on attend d'avoir suffisamment de matière avant d'écrire sur le disque.

Il y a des cas où ce comportement peut s'avérer gênant, et où on a besoin d'écrire immédiatement (et donc de vider le buffer), et c'est le propos de la méthode flush().

6. Langage Java

Pour réaliser une entrée/sortie, Java emploie un stream[1]. Celui-ci joue le rôle de médiateur entre la source des données et sa destination. Toute opération sur les entrées/sorties doit suivre le schéma suivant : ouverture, lecture, fermeture du flux.

1 flux

Java a décomposé les objets traitant des flux en deux catégories :

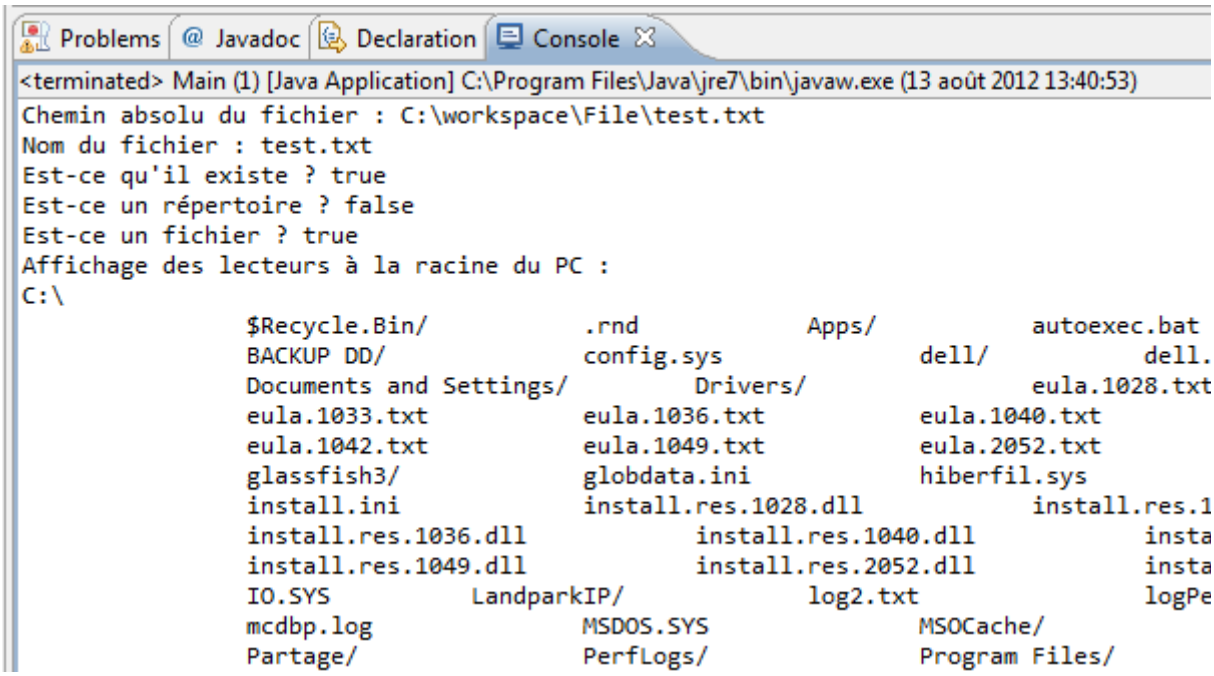
- les objets travaillant avec des flux d'entrée (in), pour la lecture de flux ;
- les objets travaillant avec des flux de sortie (out), pour l'écriture de flux.

6.1. L'objet File

Cet objet est très simple à utiliser et ses méthodes sont très explicites :

```
//Package à importer afin d'utiliser l'objet File
import java.io.File;

public class Main
{
	public static void main(String[] args)
	{
		//Création de l'objet File
		File f = new File("test.txt");
		System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());
		System.out.println("Nom du fichier : " + f.getName());
		System.out.println("Est-ce qu'il existe ? " + f.exists());
		System.out.println("Est-ce un répertoire ? " + f.isDirectory());
		System.out.println("Est-ce un fichier ? " + f.isFile());

		System.out.println("Affichage des lecteurs à la racine du PC : ");
		for (File file : f.listRoots())
		{
			System.out.println(file.getAbsolutePath());
			try
			{
				int i = 1;
				//On parcourt la liste des fichiers et répertoires
				for (File nom : file.listFiles())
				{
					//S'il s'agit d'un dossier, on ajoute un "/"
					System.out.print("\t\t" + ((nom.isDirectory())
						? nom.getName()+"/"
						: nom.getName()));

					if ((i%4) == 0)
						System.out.print("\n");

					i++;
				} //end for
			System.out.println("\n");
			} //end try

			catch (NullPointerException e)
			{
				//L'instruction peut générer une NullPointerException
				//s'il n'y a pas de sous-fichier !
			}
		}//end for
	}
}
```

Le résultat :



6.2. Les objets FileInputStream et FileOutputStream

C'est par le biais des objets FileInputStream et FileOutputStream que nous pouvons :

- lire dans un fichier ;
- écrire dans un fichier.

Ces classes héritent des classes abstraites InputStream et OutputStream, présentes dans le package java.io.

Il existe une hiérarchie de classes pour les traitements in et une autre pour les traitements out. Les classes héritant d'InputStream sont destinées à la lecture et les classes héritant d'OutputStream se chargent de l'écriture.

Exemple : lecture du contenu d'un fichier et copie dans un autre.

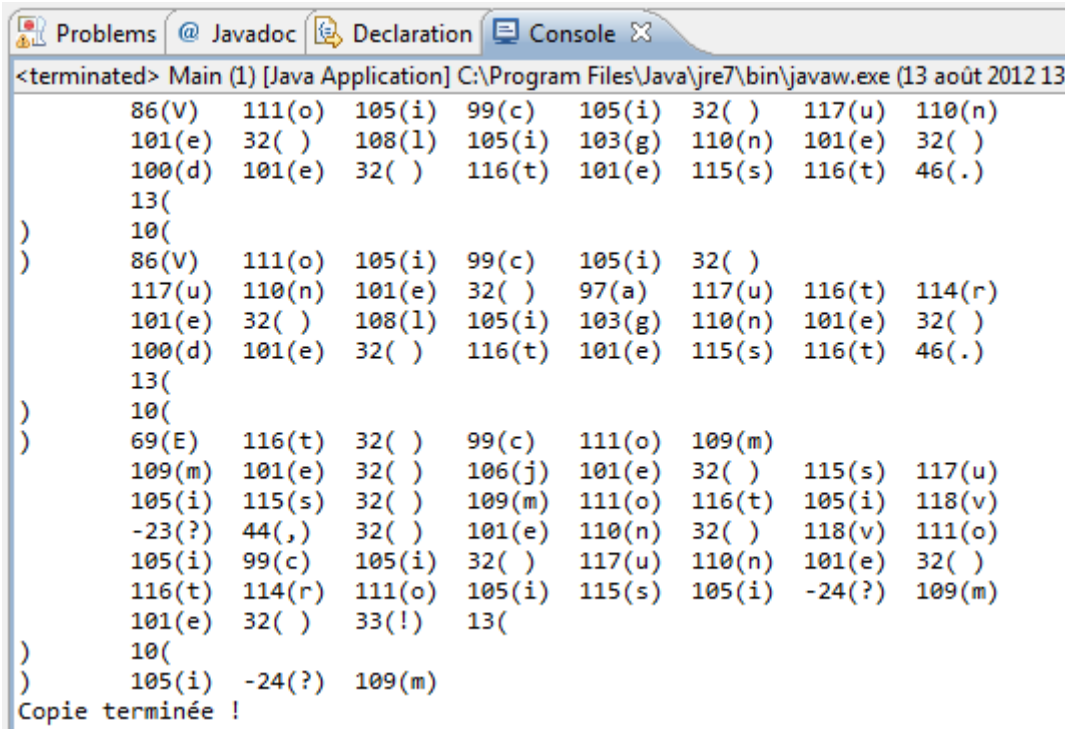
```
//Packages à importer afin d'utiliser les objets
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
	public static void main(String[] args) {
		// Nous déclarons nos objets en dehors du bloc try/catch
		FileInputStream	fis = null;
		FileOutputStream	fos = null;

		try {
			// On instancie nos objets :
			// fis va lire le fichier
			// fos va écrire dans le nouveau !
```

```
			fis = new FileInputStream(new File("test.txt"));
			fos = new FileOutputStream(new File("test2.txt"));

			// On crée un tableau de byte pour indiquer le nombre de bytes lus
			// à chaque tour de boucle
			byte[] buf = new byte[8];

			// On crée une variable de type int pour y affecter le résultat de
			// la lecture vaut -1 quand c'est fini
			int n = 0;

			// Tant que l'affectation dans la variable est possible, on boucle
			// Lorsque la lecture du fichier est terminée l'affectation n'est
			// plus possible et on sort donc de la boucle
			while ((n = fis.read(buf)) >= 0) {
				// On écrit dans notre deuxième fichier avec l'objet adéquat
				fos.write(buf);

				// On affiche ce qu'a lu notre boucle au format byte et au
				// format char
				for (byte bit : buf) {
					System.out.print("\t" + bit + "(" + (char) bit + ")");
					System.out.println("");
					}

				//Nous réinitialisons le buffer à vide au cas où
				//les derniers byte lus ne soient pas un multiple de 8
				//Ceci permet d'avoir un buffer viergec à chaque lecture
				//et ne pas avoir de doublon en fin de fichier
				buf = new byte[8];
				}
			System.out.println("Copie terminée !");
			}
		catch (FileNotFoundException e) {
			// Cette exception est levée
			// si l'objet FileInputStream ne trouve aucun fichier
			e.printStackTrace();
			}
		catch (IOException e) {
			// Celle-ci se produit lors d'une erreur d'écriture ou de lecture
			e.printStackTrace();
			}
		finally {
			// On ferme nos flux de données dans un bloc finally pour s'assurer
			// que ces instructions seront exécutées dans tous les cas même si
			// une exception est levée !
			try {
				if (fis != null)
				fis.close();
				}
			catch (IOException e) {
				e.printStackTrace();
				}
			try {
				if (fos != null)
				fos.close();
```

```
                }
            catch (IOException e) {
                e.printStackTrace();
                }
            }// end finally
        }
    }
```

Pour que l'objet FileInputStream fonctionne, le fichier doit exister ! Sinon l'exception FileNotFoundException est levée. Par contre, si vous ouvrez un flux en écriture (FileOutputStream) vers un fichier inexistant, celui-ci sera créé automatiquement !

La figure suivante représente le résultat de ce code :



Le bloc finally permet de s'assurer que nos objets ont bien fermé leurs liens avec leurs fichiers respectifs, ceci afin de permette à Java de détruire ces objets pour ainsi libérer la mémoire de l'ordinateur.

Remarque : chaque caractère que vous saisissez ou que vous lisez dans un fichier correspond à un code binaire, et ce code binaire correspond à un code décimal. Voyez la table de correspondance (on parle de la table ASCII).

Les objets que nous venons d'utiliser emploient la première version d'UNICODE 1 qui ne comprend pas les caractères accentués, c'est pourquoi ces caractères ont un code décimal négatif et ils sont représentés par des « ? » ; de plus, il y a des caractères invisibles (les 32 premiers caractères de la table ASCII sont invisibles !) dans le fichier :

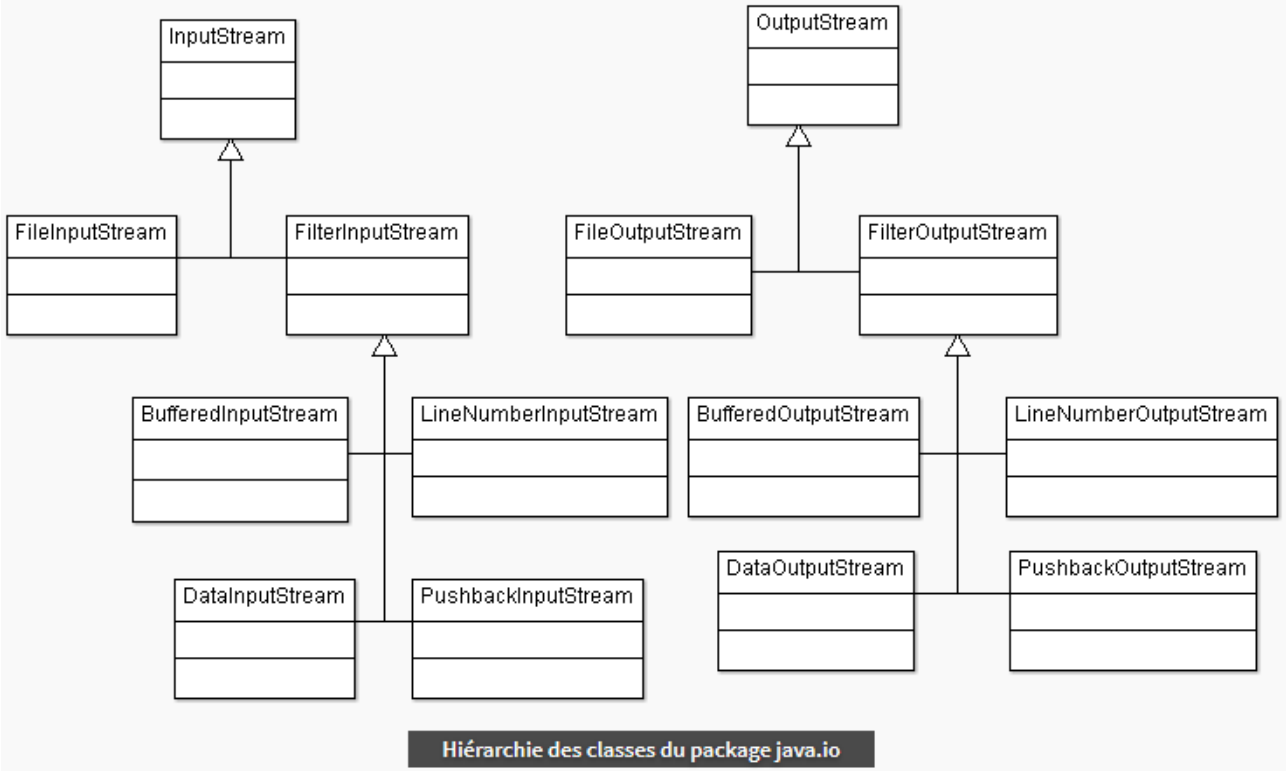
- les espaces : SP pour « SPace », code décimal 32 ;
- les sauts de lignes : LF pour « Line Feed », code décimal 13 ;
- les retours chariot : CR pour « Carriage Return », code décimal 10.

Les traitements des flux suivent une logique et une syntaxe précises. Lorsque nous avons copié

notre fichier, nous avons récupéré un certain nombre d'octets dans un flux entrant que nous avons passé à un flux sortant. À chaque tour de boucle, les données lues dans le fichier source sont écrites dans le fichier défini comme copie.

6.3. Les objets FilterInputStream et FilterOutputStream

Ces deux classes sont des classes abstraites qui permettent d'ajouter des fonctionnalités aux flux d'entrée/sortie.



Hiérarchie des classes du package java.io

- DataInputStream : offre la possibilité de lire directement des types primitifs (double, char, int) grâce à des méthodes comme readDouble(), readInt()…
- BufferedInputStream : cette classe permet d'avoir un tampon à disposition dans la lecture du flux. En gros, les données vont tout d'abord remplir le tampon, et dès que celui-ci est plein, le programme accède aux données.
- PushbackInputStream : permet de remettre un octet déjà lu dans le flux entrant.
- LineNumberInputStream : cette classe offre la possibilité de récupérer le numéro de la ligne lue à un instant T.

Ces classes prennent en paramètre une instance dérivant des classes InputStream(pour les classes héritant de FilterInputStream) ou de OutputStream (pour les classes héritant de FilterOutputStream).

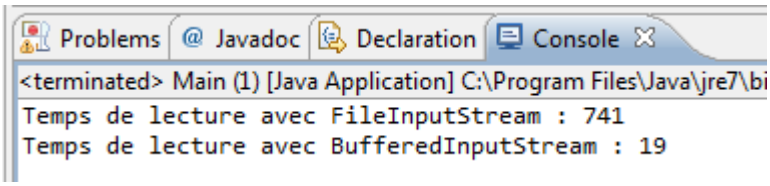
Ainsi, il est possiblede cumuler les filtres :

```
BufferedInputStream  bis = new BufferredInputStream(
                           new DataInputStream(
                             new FileInputStream(
                               new File("test.txt"))));
```

Exemple de code qui permet de tester le temps d'exécution de la lecture :

```
//Packages à importer afin d'utiliser l'objet File
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
	public static void main(String[] args) {
		//Nous déclarons nos objets en dehors du bloc try/catch
		FileInputStream     fis;
		BufferedInputStream bis;

		try {
			fis = new FileInputStream(new File("test.txt"));
			bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));
			byte[] buf = new byte[8];

			//On récupère le temps du système
			long startTime = System.currentTimeMillis();

			//Inutile d'effectuer des traitements dans notre boucle
			while ( fis.read(buf) != -1 );

			//On affiche le temps d'exécution
			System.out.println("Temps de lecture avec FileInputStream : " +
				(System.currentTimeMillis() - startTime));

			//On réinitialise
			startTime = System.currentTimeMillis();

			//Inutile d'effectuer des traitements dans notre boucle
			while ( bis.read(buf) != -1 );

			//On réaffiche
			System.out.println("Temps de lecture avec BufferedInputStream : " +
				System.currentTimeMillis() - startTime));

			//On ferme nos flux de données
			fis.close();
			bis.close();
			}
		catch (FileNotFoundException e) {
			e.printStackTrace();
			}
		catch (IOException e) {
			e.printStackTrace();
			}
		}
	}
```

Le résultat :



La différence de temps est énorme : 1,578 seconde pour la première méthode et 0,094 seconde pour la deuxième ! L'utilisation d'un buffer permet une nette amélioration des performances.