```java
/**
 * Classe permetant la gestion d'un numero de compte
 *
 * @version 1.0
 * @author Laurent Constantin
 * @author Jonathan Gander
 */
public class Account {
	/**
	 * Renvoie le nombre de comptes maximum
	 * @return le nombre de comptes maximum
	 */
	public static int getMaxAccount(){
		return (int)(Math.pow(2, 25) - 1);
	}


	/**
	 * @param bank Numero de la banque
	 * @return Lit un numero de compte
	 */
	public static int readAccount(int bank) {
		int a = Toolbox.readInt(0, getMaxAccount());

		return serializeAccount(bank, a);
	}

	/**
	 * @param account An account number
	 * @return account id
	 */
	public static int getAccountId(int account){
		account =  account <<  8;
		return account >>> 8;
	}

	/**
	 * @param account An account number
	 * @return bankId
	 */
	public static int getBankId(int account){
		return account >> 24;
	}

	/**
	 * Retourne un numero de compte pour une banque donnee
	 * @param bank Numero de la banque
	 * @param account Numero du compte dans la banque
	 * @return Numero de compte pour la banque
	 */
	public static int serializeAccount(int bank, int account) {
		if (bank > (int)(Math.pow(2,9) - 1)) {
			throw new
			IllegalArgumentException("Le numero de banque est trop grand !");
		}
		if (account < 0 || account > getMaxAccount()) {
			throw new
			IllegalArgumentException("Le numero de compte n'est pas valide !");
		}

		bank = bank << 24;
		return bank + account;
```

```
63:             }
64: }
65:
```

```
001: import java.io.*;
002: import java.net.*;
003: import java.util.HashMap;
004: import java.util.Map;
005:
006: /**
007:  * Laboratoire No 2 PRR
008:  * Auteurs : L. Constantin & J.Gander
009:  * Date : octobre-novembre 2012
010:  * OS : Mac OS X 10.7 et 10.8
011:  *
012:  * DESCRIPTION
013:  * Dans ce laboratoire, il a ete demande de simuler une banque, representee par
014:  * deux succursales. Ces deux succursales stockent des comptes (num ro, solde)
015:  * et doivent les partager entre elles.
016:  * De plus, une application client permet de faire des requetes sur la banque
017:  * independemment de la succursale choisie.
018:  * La banque doit pouvoir offrir au client les fonctionnalites suivantes :
019:  * - Creer un compte
020:  * - Supprimer un compte
021:  * - Deposer un montant
022:  * - Retirer un montant
023:  * - Obtenir le solde d'un compte
024:  * Afin de s'assurer de la coherence des donnees, les deux succursales utilisent
025:  * l'algorithme de Lamport etudie en classe. Une petite modification de
026:  * l'algorithme a ete faite : lors du message de liberation, une succursale
027:  * communique a l'autre les changements effectues.
028:  * Les methodes "creer un compte" et "obtenir le solde d'un compte" ne neces-
029:  * sitent pas d'exclusion mutuelle entre les succursales contrairement aux
030:  * trois autres.
031:  * Les numeros de comptes sont representes par des entiers sur 32 bits avec 8
032:  * bits pour indiquer l'id de la succursale et 24 bits pour un id du compte au
033:  * sein de la succursale. Ceci permet d'eviter les communications inutiles lors
034:  * de la creation de compte.
035:  * Suppositions : le reseau et les succursales sont fiables. Les banques sont
036:  * lancees avant les clients.
037:  *
038:  * ANALYSE
039:  * Dans notre logique, la classe Client.java represente uniquement un terminal
040:  * permettant au client de faire des requetes. Ces dernieres sont gerees par un
041:  * guichetier (Teller.java). Celui-ci va gerer les communications avec la
042:  * succursale desiree.
043:  * La succursale traite les demandes clients une par une. Chaque succursale a
044:  * un thread Lamport permettant l'exclusion mutuelle si besoin avec l'autre
045:  * succursale.
046:  *
047:  * Lors de notre analyse, nous avons tout d'abord voulu mettre Lamport hors
048:  * d'un thread. Cependant, l'ecoute sur le socket doit se faire en tout temps
049:  * afin de gerer les replications. Si aucune demande client n'est faite,
050:  * l'ecoute ne serait pas effectuee et la replication ne fonctionnerait
051:  * donc pas.
052:  * Nous avons donc decide de separer la succursale (Bank.java) et le thread
053:  * Lamport (Lamport.java) afin de mieux correspondre avec l'algorithme Ada
054:  * etudie en cours.
055:  *
056:  *
057:  * COMMUNICATIONS
058:  * Afin de bien structurer les communications, nous avons utilise des types
059:  * enumeres pour representer les types de messages transitant sur le reseau.
060:  *
061:  * Les communications entre client et banque (Teller et Bank) ont la forme
062:  * suivante : (Type_Message [, Numero_Compte] [, Montant])
```

```
063:  *
064:  * Les communications Lamport en cas de Requete ou Quittance ont la forme
065:  * suivante : (Type_Message, Estampille, Id_Banque_Emettrice)
066:  *
067:  * Les communications Lamport en cas de Liberation ont la forme suivante afin
068:  * de repliquer les donnees a l'autre succursale :
069:  * (Type_Message, Estampille, Id_Banque_Emettrice, Type_Message
070:  * [, Numero_Compte] [, Montant])
071:  *
072:  * Lors de la creation de compte, la replication doit etre effectuee sans
073:  * utiliser Lamport. Le message de replication est tout de meme envoye en
074:  * utilisant cette classe afin de ne pas creer un thread supplementaire dans
075:  * les succursales. Ce message a la forme suivante :
076:  * (Type_Message, Numero_Compte, Montant)
077:  *
078:  * La creation des messages a ete faite par la classe Toolbox.
079:  *
080:  * REMARQUES
081:  * Le fait d'avoir utilise un thread pour Lamport nous a permis de bloquer le
082:  * traitement de la demande client a l'aide d'un wait() lorsque la succursale
083:  * n'a pas l'exclusion mutuelle. Et de la debloquer a l'aide d'un notify lorsque
084:  * l'acces est autorise. Il ne peut donc y avoir qu'une seule demande client
085:  * traitee a la fois.
086:  *
087:  * A comparer avec le laboratoire no 1, nous avons remarque qu'il nous a ete
088:  * plus difficile d'implementer l'architecture du probleme.
089:  *
090:  * TESTS
091:  * Afin de verifier le bon fonctionnement de notre programme, nous avons
092:  * realise des tests.
093:  * Dans une premier temps nous avons execute les programmes localement sur la
094:  * meme machines. Puis nous avons cree un reseau AdHoc entre nos deux machines.
095:  * Chacune hebergeait un client et une succursale.
096:  * Nous avons realise des tests simples sans reelle concurrence. Puis par la
097:  * suite, nous avons teste des cas plus complexe : par exemple, un client veut
098:  * supprimer un compte et un autre veut ajouter un montant sur ce meme compte.
099:  * Avec ce procede, nous avons pu corriger un probleme lors de la suppression
100:  * d'un compte et sa replication sur l'autre succursale.
101:  *
102:  * Afin de mieux tester la liberation de la SC par une succursale et la
103:  * continuite de l'autre, nous avons ajoute une pause qui se relance uniquement
104:  * a la pression d'une touche au clavier par nos soins. Ceci a permis de
105:  * realiser un debuggage "pas a pas".
106:  *
107:  * STRUCTURE DU PROGRAMME
108:  * Account.java : methodes statiques pour gerer les comptes
109:  * Bank.java : represente la banque
110:  * Client.java : represente le client
111:  * Config.java : valeurs par defaut pour utiliser le programme
112:  * ConfigParser.java : parser la ligne de commande pour creer la config
113:  * ErrorServerClient.java : erreurs du serveur au client
114:  * Lamport.java : algorithme de Lamport
115:  * LamportMessages.java : messages transitants avec Lamport
116:  * LamportState.java : etats utilises dans Lamport
117:  * LamportUnlockMessage.java : messages envoyes lors de la liberation de la SC
118:  * Menu.java : differents menus du client
119:  * Teller.java : guichetier qui fait la communication client->banque
120:  * Toolbox.java : utilitaires (saisies et conversions en bytes)
121:  *
122:  * UTILISATION
123:  * Il faut tout d'abord modifier le fichier Config.java avec les bonnes IPs.
124:  *
```

```
125:  * Le laboratoire se decompose en deux executables JAR pour le client et les
126:  * succursale. Les jars peuvent etre crees a l'aide de l'utilitaire ANT.
127:  *
128:  * Les fichiers se lancent avec la commande java -jar NomJar Parametres
129:  * Pour le client :
130:  * java -jar Client.jar
131:  * Pour la succursale :
132:  * java -jar Bank.jar id_de_la_succursale (0 ou 1)
133:  *
134:  * @version 1.0
135:  * @author Laurent Constantin
136:  * @author Jonathan Gander
137:  */
138: public class Bank {
139:         /**
140:          * Classe interne pour gerer une requete client
141:          */
142:         class Bank2ClientTeller {
143:                 private final Bank bank;
144:                 private final DatagramPacket packet;
145:
146:                 /**
147:                  * Constructeur
148:                  *
149:                  * @param bank
150:                  *            La banque qu'il doit gerer
151:                  * @param packet
152:                  *            Paquet envoye par le client
153:                  */
154:                 public Bank2ClientTeller(Bank bank, DatagramPacket packet) {
155:                         this.bank = bank;
156:                         this.packet = packet;
157:
158:                         handleClientRequests();
159:                 }
160:
161:                 /**
162:                  * Envoie des donnees au client
163:                  *
164:                  * @param code
165:                  *            Le code d'erreur
166:                  * @param data
167:                  *            [] Les donnees
168:                  * @throws IOException
169:                  *             Si erreur
170:                  */
171:                 public void sendDataToClient(ErrorServerClient code, int... data)
172:                                 throws IOException {
173:
174:                         byte[] message = Toolbox.buildMessage(code.getCode(), data);
175:
176:                         // Cree les donnees
177:                         DatagramPacket packet = new DatagramPacket(message, message.length,
178:                                         this.packet.getAddress(), this.packet.getPort());
179:                         // Cree le socket et envoi
180:                         DatagramSocket sendToClientSocket = new DatagramSocket();
181:                         sendToClientSocket.send(packet);
182:                 }
183:
184:                 /**
185:                  * Gere une requete client (appel des bonnes methodes)
186:                  */
```

```
187:                private void handleClientRequests() {
188:                        // Lit les donnees
189:                        Menu action = Menu.fromCode(Toolbox.getDataCode(this.packet));
190:                        int val[] = Toolbox.buildData(this.packet);
191:
192:                        // Debug
193:                        System.out.print("La banque " + bankId + " recoit: > " + action);
194:                        System.out.print("(");
195:                        for (int i = 0; i < val.length; i++)
196:                                System.out.print(val[i] + (i < val.length - 1 ? " " : ""));
197:                        System.out.println(")");
198:
199:                        // Lance une action suivant le message recu
200:                        try {
201:
202:                                switch (action) {
203:                                case ADD_ACCOUNT:
204:                                        if (val[0] < 1) {
205:                                                this.sendDataToClient(
206:                                                  ErrorServerClient.MONTANT_INCORRECT);
207:                                                return;
208:                                        }
209:
210:                                        int accountNumber = bank.addAccount(val[0]);
211:                                        if (accountNumber == -1) {
212:                                                // Renvoi erreur 4 (autre)
213:                                                this.sendDataToClient(ErrorServerClient.AUTRE);
214:                                                return;
215:                                        }
216:
217:                                        // Renvoie le numero de compte au client
218:                                        this.sendDataToClient(ErrorServerClient.OK, accountNumber);
219:
220:                                        // Prevenir les autres banques
221:                                        lamport.accountCreated(accountNumber, val[0]);
222:
223:                                        break;
224:                                case DELETE_ACCOUNT: {
225:
226:                                        ErrorServerClient ret = bank.deleteAccount(val[0]);
227:                                        if (ret != ErrorServerClient.OK) {
228:                                                // Erreur au client
229:                                                this.sendDataToClient(ret);
230:                                                return;
231:                                        }
232:
233:                                        // Reponse au client
234:                                        this.sendDataToClient(ErrorServerClient.OK);
235:
236:                                }
237:                                        break;
238:                                case ADD_MONEY: {
239:
240:                                        ErrorServerClient ret = bank.addMoney(val[0], val[1]);
241:
242:                                        if (ret != ErrorServerClient.OK) {
243:                                                // Erreur au client
244:                                                this.sendDataToClient(ret);
245:                                                return;
246:                                        }
247:
248:                                        // Reponse au client
```

```
249:                                        this.sendDataToClient(ErrorServerClient.OK);
250:
251:                                }
252:                                        break;
253:                                case TAKE_MONEY: {
254:
255:                                        ErrorServerClient ret = bank.takeMoney(val[0], val[1]);
256:                                        if (ret != ErrorServerClient.OK) {
257:                                                // Erreur au client
258:                                                this.sendDataToClient(ret);
259:                                                return;
260:                                        }
261:
262:                                        // Reponse au client
263:                                        this.sendDataToClient(ErrorServerClient.OK);
264:
265:                                }
266:                                        break;
267:                                case GET_BALANCE:
268:                                        int money = bank.getBalance(val[0]);
269:
270:                                        if (money < 0) {
271:                                                // Erreur au client
272:                                                this.sendDataToClient(
273:                                                  ErrorServerClient.COMPTE_INEXISTANT);
274:                                                return;
275:                                        }
276:
277:                                        // Reponse au client
278:                                        this.sendDataToClient(ErrorServerClient.OK, money);
279:                                        break;
280:                                default:
281:                                        throw new IllegalStateException("Unimplemented action");
282:                                }
283:                        } catch (IOException e) {
284:                                // Erreur d'envoi au client
285:                                e.printStackTrace();
286:                        }
287:                }
288:        }
289:
290:        // Comptes (n,montant)
291:        private Map<Integer, Integer> accounts = new HashMap<Integer, Integer>();
292:        private int bankId;
293:        private final Lamport lamport;
294:        private DatagramSocket listenFromClientSocket;
295:
296:        /**
297:         * Constructeur d'une banque
298:         *
299:         * @param id
300:         *            Id de la banque
301:         * @throws SocketException
302:         */
303:        public Bank(int id) throws SocketException {
304:                this.bankId = id;
305:                listenFromClientSocket = new DatagramSocket(
306:                                Config.banks2ClientPorts[id]);
307:
308:                this.lamport = new Lamport(this);
309:
310:                // 0. Receptionne une commande client
```

```
311:                 while (true) {
312:                         try {
313:                                 // 1. Ecoute et gere la demande d'un client
314:                                 byte[] buffer = new byte[Config.bufferSize];
315:                                 DatagramPacket data = new DatagramPacket(buffer, buffer.length);
316:                                 listenFromClientSocket.receive(data);
317:                                 new Bank2ClientTeller(this, data);
318:
319:                         } catch (IOException e) {
320:                                 e.printStackTrace();
321:                         }
322:                 }
323:         }
324:
325:         /**
326:          * Renvoie l'identifiant de la banque
327:          *
328:          * @return L'identifiant de la banque
329:          */
330:         public int getId() {
331:                 return bankId;
332:         }
333:
334:         /**
335:          * Cree un nouveau compte
336:          *
337:          * @param montant
338:          *            initial
339:          * @return numero du compte
340:          */
341:         public int addAccount(int money) {
342:                 // Boucle sur tous les comptes possibles
343:                 for (int i = 0; i < Account.getMaxAccount(); i++) {
344:                         // Recupere le numero de compte de la banque
345:                         int accountNumber = Account.serializeAccount(this.bankId, i);
346:
347:                         // Si le compte n'existe pas on le cree avec le montant initial
348:                         if (!accounts.containsKey(accountNumber)) {
349:                                 accounts.put(accountNumber, money);
350:                                 return accountNumber;
351:                         }
352:                 }
353:                 // Tous les comptes sont pris
354:                 return -1;
355:         }
356:
357:         /**
358:          * Supprime un compte (en section critique)
359:          *
360:          * @param account
361:          *            Compte a supprimer
362:          * @return Code d'erreur
363:          * @throws IOException
364:          */
365:         public ErrorServerClient deleteAccount(int account) throws IOException {
366:                 if (!accounts.containsKey(account)) {
367:                         return ErrorServerClient.COMPTE_INEXISTANT;
368:                 }
369:
370:                 if (accounts.get(account) != 0) {
371:                         return ErrorServerClient.SOLDE_INVALIDE;
372:                 }
```

```
373:
374:                lamport.lock();
375:
376:                if (!accounts.containsKey(account)) {
377:                        return ErrorServerClient.COMPTE_INEXISTANT;
378:                }
379:
380:                if (accounts.get(account) != 0) {
381:                        return ErrorServerClient.SOLDE_INVALIDE;
382:                }
383:
384:                accounts.remove(account);
385:
386:                lamport.unlock(LamportUnlockMessage.DELETE_ACCOUNT, account);
387:
388:                return ErrorServerClient.OK;
389:        }
390:
391:        /**
392:         * Ajoute un montant a un compte (en section critique)
393:         *
394:         * @param account
395:         *            Compte a crediter
396:         * @param money
397:         *            Montant a ajouter
398:         * @return Code d'erreur
399:         * @throws IOException
400:         */
401:        public ErrorServerClient addMoney(int account, int money)
402:                        throws IOException {
403:                if (money < 0) {
404:                        return ErrorServerClient.MONTANT_INCORRECT;
405:                }
406:
407:                if (!accounts.containsKey(account)) {
408:                        return ErrorServerClient.COMPTE_INEXISTANT;
409:                }
410:
411:                lamport.lock();
412:
413:                if (!accounts.containsKey(account)) {
414:                        return ErrorServerClient.COMPTE_INEXISTANT;
415:                }
416:
417:                accounts.put(account, accounts.get(account) + money);
418:
419:                lamport.unlock(LamportUnlockMessage.UPDATE_MONEY, account,
420:                                accounts.get(account));
421:
422:                return ErrorServerClient.OK;
423:
424:        }
425:
426:        /**
427:         * Debite un montant a un compte (en section critique)
428:         *
429:         * @param account
430:         *            Compte a crediter
431:         * @param montant
432:         *            a supprimer
433:         * @return Code d'erreur
434:         * @throws IOException
```

```
435:          */
436:         public ErrorServerClient takeMoney(int account, int money)
437:                         throws IOException {
438:                 if (money < 0) {
439:                         return ErrorServerClient.MONTANT_INCORRECT;
440:                 }
441:
442:                 if (!accounts.containsKey(account)) {
443:                         return ErrorServerClient.COMPTE_INEXISTANT;
444:                 }
445:
446:                 if (accounts.get(account) - money < 0) {
447:                         return ErrorServerClient.SOLDE_INVALIDE;
448:                 }
449:
450:                 lamport.lock();
451:
452:                 if (!accounts.containsKey(account)) {
453:                         return ErrorServerClient.COMPTE_INEXISTANT;
454:                 }
455:
456:                 if (accounts.get(account) - money < 0) {
457:                         return ErrorServerClient.SOLDE_INVALIDE;
458:                 }
459:
460:                 accounts.put(account, accounts.get(account) - money);
461:
462:                 lamport.unlock(LamportUnlockMessage.UPDATE_MONEY, account,
463:                                 accounts.get(account));
464:
465:                 return ErrorServerClient.OK;
466:
467:         }
468:
469:         /**
470:          * Obtient le solde d'un compte
471:          *
472:          * @param account
473:          *            compte a qui obtenir le solde
474:          * @return Solde du compte
475:          */
476:         public int getBalance(int account) {
477:                 if (accounts.containsKey(account))
478:                         return accounts.get(account);
479:
480:                 return -1;
481:         }
482:
483:         /**
484:          * Suppression d'un element lorsqu'une autre banque libere le mutex
485:          *
486:          * @param account
487:          *            Le compte
488:          * @param money
489:          *            le nouveau montant
490:          */
491:         public void handleOnUpdate(int account, int money) {
492:                 System.out.println("Mutex distant lache: la banque maj le compte "
493:                                 + account + " avec le montant :" + money);
494:                 if (!accounts.containsKey(account)) {
495:                         System.out.println("handleOnUpdate : le compte n'existe plus");
496:                         return;
```

```
497:                 }
498:                 accounts.put(account, money);
499:
500:         }
501:
502:         /**
503:          * Suppression d'un element lorsqu'une autre banque libere le mutex
504:          *
505:          * @param account
506:          *             Le compte a supprimer
507:          */
508:         public void handleOnDelete(int account) {
509:                 System.out.println("Mutex distant lache: la banque supprime le compte "
510:                                 + account);
511:
512:                 if (!accounts.containsKey(account)) {
513:                         return;
514:                 }
515:
516:                 if (accounts.get(account) != 0) {
517:                         return;
518:                 }
519:                 accounts.remove(account);
520:         }
521:
522:         /**
523:          * Replication d'un compte quand une autre banque en cree un.
524:          *
525:          * @param account
526:          *             Le compte a creer
527:          * @param money
528:          *             L'argent
529:          */
530:         public void handleOnCreate(int account, int money) {
531:                 System.out.println("Banque " + bankId + " : "
532:                                 + LamportMessages.NEW_ACCOUNT + " n: " + account + ", " + money
533:                                 + "CHF");
534:
535:                 if (money < 1)
536:                         return;
537:
538:                 accounts.put(account, money);
539:         }
540:
541:         /**
542:          * Permet d'instancier une banque
543:          *
544:          * @param args
545:          *             MulticastHost, port serveur, port client, nombre de clients
546:          */
547:         public static void main(String[] args) {
548:                 // java -jar Bank.jar BankId
549:                 try {
550:                         new Bank(ConfigParser.getBankIdFromArg(args, 0));
551:                 } catch (Exception e) {
552:                         e.printStackTrace();
553:                 }
554:         }
555: }
556:
```

-- Néant --

```java
import java.net.SocketException;
import java.net.UnknownHostException;

/**
 * Client.
 *
 * @version 1.0
 * @author Laurent Constantin
 * @author Jonathan Gander
 */
public class Client {

	public static void main(String[] args) {

		Menu choice;

		// Initialisation des guichetiers
		Teller tellers[] = new Teller[Config.banksAddresses.length];
		try {
			for (int i = 0; i < tellers.length; i++) {
				tellers[i] = new Teller(i);
			}
		} catch (UnknownHostException uhe) {
			System.err.println("Impossible de contacter une des banque.");
		} catch (SocketException se) {
			System.err.println("Impossible d'ouvrir la connexion.");
		}

		System.out.println("D marrage du client ");

		do {

			// Choisir une banque
			int bankChoice;
			do {
				System.out.print("Veuillez entrer le numero de la banque (0.."
						+ (Config.banksAddresses.length - 1) + ") > ");
				bankChoice = Toolbox.readBank();

			} while (bankChoice < 0
					|| bankChoice > Config.banksAddresses.length - 1);

			// Affichage du menu
			for (Menu m : Menu.values()) {
				System.out.println(m.ordinal() + ": " + m);
			}

			// Lecture du choix
			System.out.println("Votre choix > ");
			choice = Toolbox.readMenu();

			if (choice == null) {
				System.out.println("Erreur de saisie, veuillez recommencer.");
				continue;
			}

			// Lance la bonne operation
			switch (choice) {
			case ADD_ACCOUNT: {
				System.out.print("Entrer le montant initial > ");
				int money = Toolbox.readInt(1, Integer.MAX_VALUE);
				int accountNumber = tellers[bankChoice].addAccount(money);
```

```
063:                                    if (accountNumber < 0) {
064:                                            System.out.println("Il n'y a plus de compte disponible !");
065:                                    } else {
066:                                            System.out.println("Compte cree : " + accountNumber);
067:                                    }
068:                            }
069:                                    break;
070:                            case DELETE_ACCOUNT: {
071:                                    System.out.print("Entrer le numero du compte a supprimer > ");
072:                                    int account = Toolbox.readInt(0, Account.getMaxAccount());
073:
074:                                    ErrorServerClient ret = tellers[bankChoice]
075:                                                    .deleteAccount(account);
076:
077:                                    System.out.println(handleResponse(ret));
078:                            }
079:                                    break;
080:                            case ADD_MONEY: {
081:                                    System.out.print("Entrer le numero du compte a crediter > ");
082:                                    int account = Toolbox.readInt(0, Account.getMaxAccount());
083:                                    System.out.print("Entrer le montant a crediter > ");
084:                                    int money = Toolbox.readInt(1, Integer.MAX_VALUE);
085:
086:                                    ErrorServerClient ret = tellers[bankChoice].addMoney(account,
087:                                                    money);
088:                                    System.out.println(handleResponse(ret));
089:
090:                            }
091:                                    break;
092:                            case TAKE_MONEY: {
093:                                    System.out.print("Entrer le numero du compte a debiter > ");
094:                                    int account = Toolbox.readInt(0, Account.getMaxAccount());
095:                                    System.out.print("Entrer le montant a debiter > ");
096:                                    int money = Toolbox.readInt(1, Integer.MAX_VALUE);
097:
098:                                    ErrorServerClient ret = tellers[bankChoice].takeMoney(account,
099:                                                    money);
100:                                    System.out.println(handleResponse(ret));
101:
102:                            }
103:                                    break;
104:                            case GET_BALANCE: {
105:                                    System.out.print("Entrer le numero du compte > ");
106:                                    int account = Toolbox.readInt(0, Account.getMaxAccount());
107:                                    int balance = tellers[bankChoice].getBalance(account);
108:                                    if (balance < 0) {
109:                                            System.out.println("Le compte " + account
110:                                                            + " n'existe pas !");
111:                                    } else {
112:                                            System.out.println("Solde du compte " + account + " : "
113:                                                            + balance);
114:                                    }
115:                            }
116:                                    break;
117:                            case QUIT:
118:                                    break;
119:                            default:
120:                                    throw new UnsupportedOperationException("Menu inconnu");
121:                            }
122:
123:                    } while (choice != Menu.QUIT);
124:
```

```
125:                 System.out.println("Fin du client");
126:         }
127:
128:         /**
129:          * Permet d'afficher un message d'erreur correspondant a une reponse du
130:          * serveur au client
131:          *
132:          * @param response
133:          *            La reponse recue du client
134:          * @return Le message d'erreur
135:          */
136:         public static String handleResponse(ErrorServerClient response) {
137:                 switch (response) {
138:                 case OK:
139:                         return "Operation reussie !";
140:                 case COMPTE_INEXISTANT:
141:                         return "Le compte entre n'existe pas !";
142:                 case SOLDE_INVALIDE:
143:                         return "Le solde est invalide !";
144:                 case MONTANT_INCORRECT:
145:                         return "Le montant fourni est incorrect !";
146:                 case AUTRE:
147:                         return "Une erreur inconnue est survenue !";
148:                 default:
149:                         return "Une erreur non geree est survenue !";
150:                 }
151:         }
152: }
153:
```

-- Néant --

```java
/**
 * Configuration par defaut des adresses et ports pour les
 * communications
 *
 * @version 1.0
 * @author Laurent Constantin
 * @author Jonathan Gander
 */
public interface Config {
        /**
         * Ports utilises pour les communications
         */
        public static final int banks2ClientPorts[] = {1515,1516};
        /**
         * IP utilises pour les communications
         */
        public static final String banksAddresses[] = {"192.168.2.15","192.168.2.10"};

        /**
         * Taille du buffer pour la recetion des donnees
         */
        public static final int bufferSize = 256;
        /**
         * Port utilise pour la communication entre banques (p.ex lamport)
         */
        public static final int bank2bankLamportPort[] = {1517,1518};

}

```

-- Néant --

```java
import java.util.regex.Pattern;

/**
 * Permet de parser la configuration depuis la ligne de commande En cas
 * d'erreur, on utilise les valeurs par defaut
 *
 * @version 1.0
 * @author Laurent Constantin
 * @author Jonathan Gander
 */
public class ConfigParser implements Config {
	/**
	 * Parse une chaine et renvoie l'entier qu'elle contient.
	 * En cas d'erreur renvoie defaultInt
	 * @param args La chaine a parser
	 * @param defaultInt La valeur par default
	 * @return L'entier contenu dans la chaine
	 */
	private static int getIntFromString(String args, int defaultInt){
		if(Pattern.matches("[0-9]*", args)){
			return Integer.parseInt(args);
		}else{
			System.err.printf("Invalid number %s using %d \n",args,defaultInt );
		}
		return defaultInt;
	}
	/**
	 * Renvoie l'entier contenu a la position "index" de la ligne de commande
	 * @param args Parametres de la ligne de commande
	 * @param index Index dans la ligne de commande
	 * @param def Valeur par defaut (en cas d'erreur)
	 * @return Entier contenu dans args[index] ou def.
	 */
	private static int getIntFromArgs(String args[], int index, int def){
		if(args.length > index){
			return getIntFromString(args[index], def);
		}
		return def;
	}

	/**
	 * Renvoie l'id de la banque obtenu depuis la ligne de commande
	 * @param args Ligne de commande
	 * @param index Position dans la ligne de commande
	 */
	public static int getBankIdFromArg(String args[],int index){
		return ConfigParser.getIntFromArgs(args, index, 0);
	}
}
```

-- Néant --

```java
/**
 * Modelise les erreurs entre le serveur et le client
 * Attention : 127 elements maximum
 * @author Laurent Constantin
 * @author Jonathan Gander
 */

public enum ErrorServerClient {
        OK,
        MONTANT_INCORRECT,
        SOLDE_INVALIDE,
        COMPTE_INEXISTANT,
        AUTRE;

        /**
         * Permet d'obtenir le code de l'element pour un transfert reseau
         * @return Le code en byte
         */
        public byte getCode(){
                return (byte)this.ordinal();
        }
        /**
         * Reconsruit l'element depuis le code (sans verifications)
         * @param code Le code (ordinal)
         * @return L'element de l'enum
         */
        public static ErrorServerClient fromCode(byte code){
                return ErrorServerClient.values()[(int)code];
        }
}

```

-- Néant --

```
001: import java.io.IOException;
002: import java.net.DatagramPacket;
003: import java.net.DatagramSocket;
004: import java.net.InetAddress;
005: import java.net.SocketException;
006:
007: /**
008:  * Implementation de l'algorithme de Lamport avec replication de donnees lors
009:  * d'une liberation
010:  *
011:  * @author Constantin Laurent
012:  * @author Gander Jonathan
013:  * @version 1.0
014:  *
015:  */
016: public class Lamport implements Runnable {
017:         // Pour la communication
018:         private DatagramSocket socket;
019:         private final int port;
020:
021:         private final boolean DEBUG = true;
022:         // Pour le mutex
023:         private int localTimestamp; // horloge logique
024:         private Boolean hasMutex = false;
025:
026:         // Id de la banque courante
027:         final Bank bank;
028:         // Tableau des etats
029:         private LamportState[] state;
030:
031:         /**
032:          * Constructeur avec la banque
033:          *
034:          * @param bank
035:          *             Banque associee
036:          * @throws SocketException
037:          */
038:         public Lamport(Bank bank) throws SocketException {
039:                 this.bank = bank;
040:                 this.port = Config.bank2bankLamportPort[bank.getId()];
041:                 System.out.println("La banque " + bank.getId() + " ecoute sur le port "
042:                                 + port);
043:                 socket = new DatagramSocket(port);
044:
045:                 // Initialise le tableau d'etat
046:                 state = new LamportState[Config.banksAddresses.length];
047:                 for (int i = 0; i < state.length; i++)
048:                         state[i] = new LamportState();
049:
050:                 new Thread(this).start();
051:
052:         }
053:
054:         /**
055:          * Indique si la banque courante peut entrer en section critique
056:          *
057:          * @return True si la banque peut entrer en section critique
058:          */
059:         private boolean localAccesGranted() {
060:                 // Il peut si etat[bankid]=requete
061:                 // et que son estampille est la plus ancienne !
062:                 if (state[bank.getId()].type != LamportMessages.REQUEST)
```

```
063:                    return false;
064:
065:            int myTimeStamp = state[bank.getId()].timestamp;
066:            for (int i = 0; i < state.length; i++) {
067:                    if (myTimeStamp > state[i].timestamp) {
068:                            return false;
069:                    } else if (myTimeStamp == state[i].timestamp && i != bank.getId()) {
070:                            if (bank.getId() > i)
071:                                    return false;
072:                    }
073:            }
074:            return true;
075:    }
076:
077:    /**
078:     * Pour obtenir le mutex
079:     *
080:     * @throws IOException
081:     */
082:    public void lock() throws IOException {
083:            System.out.println("Lamport.lock()");
084:            // Mise a jour de l'estampille
085:            localTimestamp++;
086:            // Envoi d'une requete
087:            state[bank.getId()].set(LamportMessages.REQUEST, localTimestamp);
088:            sendToAllOthersBank(state[bank.getId()].toByte(this.bank.getId()));
089:            // Indique si l'on peut avoir le mutex
090:
091:            // Si on a pas le mutex, on est en attente !
092:            synchronized (this) {
093:                    hasMutex = localAccesGranted();
094:
095:                    while (!hasMutex) {
096:                            try {
097:                                    if (DEBUG) {
098:                                            System.out.println("Wait() sur la banque "
099:                                                            + bank.getId());
100:                                    }
101:                                    wait();
102:                            } catch (InterruptedException e) {
103:                                    e.printStackTrace();
104:                            }
105:                            hasMutex = localAccesGranted();
106:
107:                    }
108:            }
109:    }
110:
111:    /**
112:     * Replication quand un compte est cree
113:     *
114:     * @param account
115:     *            Compte cree
116:     * @param money
117:     *            Argent verse initialement sur le compte
118:     * @throws IOException
119:     */
120:    public void accountCreated(int account, int money) throws IOException {
121:
122:            // Infos de creation de compte
123:            byte[] temp = Toolbox.buildMessage(
124:                            LamportMessages.NEW_ACCOUNT.getCode(), account, money);
```

```
125:
126:                 // Envoi
127:                 sendToAllOthersBank(temp);
128:         }
129:
130:         /**
131:          * Libere le mutex et envoie des donnees dans le message de liberation
132:          *
133:          * @param code
134:          *             Le type de message.
135:          * @param data
136:          *             Les donnes.
137:          * @throws IOException
138:          */
139:         public void unlock(LamportUnlockMessage unlockType, int... data)
140:                         throws IOException {
141:                 System.out.println("Lamport.unlock()");
142:
143:                 // Construction du message a envoyer
144:                 state[bank.getId()].set(LamportMessages.RELEASE, localTimestamp);
145:                 byte[] messageData = state[bank.getId()].toByte(bank.getId());
146:                 // Ajout des infos de liberation
147:                 byte[] temp = Toolbox.buildMessage(unlockType.getCode(), data);
148:                 // Envoi
149:                 sendToAllOthersBank(Toolbox.concat(messageData, temp));
150:                 synchronized (this) {
151:                         hasMutex = false;
152:                 }
153:         }
154:
155:         /**
156:          * Envoie un message a toutes les banques excepte soi-meme
157:          *
158:          * @param data
159:          *             Le message
160:          * @throws IOException
161:          *              En cas d'erreur
162:          */
163:         public void sendToAllOthersBank(byte[] data) throws IOException {
164:                 for (int i = 0; i < state.length; i++) {
165:                         if (i != bank.getId()) {
166:                                 send(i, data);
167:                         }
168:                 }
169:         }
170:
171:         /**
172:          * Envoie un message a une banque
173:          *
174:          * @param bankId
175:          *             Id de la banque ou envoyer le message
176:          * @param data
177:          *             Le message
178:          * @throws IOException
179:          *              En cas d'erreur
180:          */
181:         public void send(int bankId, byte[] data) throws IOException {
182:                 // Construction de l'adresse et du datagramme
183:                 int port = Config.bank2bankLamportPort[bankId];
184:                 InetAddress host = InetAddress.getByName(Config.banksAddresses[bankId]);
185:                 DatagramPacket packet = new DatagramPacket(data, data.length, host,
186:                                 port);
```

```
187:                 // Envoi
188:                 socket.send(packet);
189:         }
190:
191:         /**
192:          * Action du Lamport
193:          */
194:         public void run() {
195:                 byte[] buffer = new byte[Config.bufferSize];
196:                 DatagramPacket data = new DatagramPacket(buffer, buffer.length);
197:
198:                 while (true) {
199:                         try {
200:                                 // 1. Reception d'un message
201:                                 socket.receive(data);
202:
203:                                 // 2. Regarde si une autre banque replique un nouveau compte
204:                                 LamportMessages type = LamportMessages
205:                                                 .fromCode(data.getData()[0]);
206:                                 if (type == LamportMessages.NEW_ACCOUNT) {
207:                                         // 2b. Si oui, on l'ajout a la banque
208:                                         int[] newAccountData = Toolbox.buildData(data.getData(),
209:                                                         data.getLength(), 0);
210:                                         bank.handleOnCreate(newAccountData[0], newAccountData[1]);
211:
212:                                 } else {
213:                                         // 3. Sinon on a un message lamport normal
214:                                         LamportState state = LamportState.fromByte(data.getData(),
215:                                                         data.getLength());
216:                                         acceptReceive(state, data);
217:                                 }
218:
219:                         } catch (IOException e) {
220:                                 e.printStackTrace();
221:                         }
222:                 }
223:         }
224:
225:         /**
226:          * Implementation du Receive. Permet de gerer les messages de Lamport
227:          *
228:          * @param remoteBankId
229:          *             L'Id de la banque distante
230:          * @param state
231:          *             Etat de la requete
232:          * @throws IOException
233:          *              En cas d'erreur
234:          */
235:         private void acceptReceive(LamportState state, DatagramPacket data)
236:                         throws IOException {
237:
238:                 int remoteBankId = state.remoteBankId;
239:
240:                 // 1. Met a jour l'estampille
241:                 localTimestamp = Math.max(localTimestamp, state.timestamp) + 1;
242:
243:                 // 2. Effectue les actions suivant le type de message recu
244:                 switch (state.type) {
245:                 case NEW_ACCOUNT:
246:                         // Deja traite car pas un message lamport standard
247:                         break;
248:                 case REQUEST:
```

```
249:                    // Mise a jour de la table locale
250:                    this.state[remoteBankId].set(LamportMessages.REQUEST,
251:                                    state.timestamp);
252:
253:                    // Envoi du recu
254:                    LamportState data2send = new LamportState(LamportMessages.RECEIPT,
255:                                    localTimestamp);
256:
257:                    send(remoteBankId, data2send.toByte(bank.getId()));
258:                    break;
259:            case RELEASE:
260:                    // Mise a jour de la table locale
261:                    this.state[remoteBankId].set(LamportMessages.RELEASE,
262:                                    state.timestamp);
263:                    // Recupe des donnees en plus dans le message de release
264:                    if (state.type == LamportMessages.RELEASE && data.getLength() >= 9){
265:                            // Construction des donnees
266:                            byte code = data.getData()[9]; // Code
267:                            int[] releaseData = Toolbox.buildData(data.getData(),
268:                                            data.getLength() - 9, 9); // Donnee repliquee
269:
270:                            // Conversion en enum
271:                            LamportUnlockMessage lum = LamportUnlockMessage.fromCode(code);
272:
273:                            // Mise a jour de la banque suivant les donnees recues
274:                            switch (lum) {
275:                            case DELETE_ACCOUNT:
276:                                    bank.handleOnDelete(releaseData[0]);
277:                                    break;
278:                            case UPDATE_MONEY:
279:                                    bank.handleOnUpdate(releaseData[0], releaseData[1]);
280:                                    break;
281:                            default:
282:                                    System.err
283:                                      .println("LamportRelease: non implementee");
284:                                    break;
285:                            }
286:
287:                    }
288:
289:                    break;
290:            case RECEIPT:
291:                    // Met a jour si on avait pas de requete dans le tableau
292:                    if (this.state[remoteBankId].type != LamportMessages.REQUEST) {
293:                            this.state[remoteBankId].set(LamportMessages.RECEIPT,
294:                                            state.timestamp);
295:                    }
296:                    break;
297:
298:            }
299:            // Indique si l'on peut avoir le mutex
300:            synchronized (this) {
301:                    hasMutex = (this.state[bank.getId()].type ==
302:                                    LamportMessages.REQUEST) && localAccesGranted();
303:                    if (DEBUG) {
304:                            System.out.println("Lamport.acceptReceive()");
305:                    }
306:                    if (hasMutex) {
307:                            if (DEBUG)
308:                                    System.out
309:                                      .println("Notify() sur la banque " + bank.getId());
310:                            notify();
```

```
311:                                }
312:                            }
313:
314:                }
315: }
316:
```

```
01:
02: /**
03:  * @author Constantin Laurent
04:  * @author Jonathan Gander
05:  * @version 1.0
06:  *
07:  * Messages utilises pour le transfert dans l'algorithme de Lamport
08:  * Attention, 127 elements au maximum
09:
10:  */
11: public enum LamportMessages {
12:         RELEASE,
13:         REQUEST,
14:         RECEIPT,
15:         NEW_ACCOUNT;
16:
17:
18:         /**
19:          * Permet d'obtenir le code de l'element pour un transfert reseau
20:          * @return Le code en byte
21:          */
22:         public byte getCode(){
23:                 return (byte)this.ordinal();
24:         }
25:         /**
26:          * Reconsruit l'element depuis le code (sans verifications)
27:          * @param code Le code (ordinal)
28:          * @return L'element de l'enum
29:          */
30:         public static LamportMessages fromCode(byte code){
31:                 return LamportMessages.values()[(int)code];
32:         }
33: }
34:
```

-- Néant --

```
001: /**
002:  * Modelise les messages de fonctionnement pour Lamport : le type du message,
003:  * l'estampille et l'id de la banque emettrice sont utilises.
004:  * La conversion en octet est aussi geree dans cette classe.
005:  *
006:  * @author Laurent Constantin
007:  * @author Jonathan Gander
008:  */
009: public class LamportState {
010:
011:         public LamportMessages type;
012:         public int timestamp;
013:         // Utilise pour l'envoi et la reception uniquement
014:         public int remoteBankId;
015:
016:         /**
017:          * Construit un message de type RELEASE et d'estampille 0.
018:          *
019:          * @param type Le type du message
020:          * @param timestamp Le timestamp du message
021:          */
022:         public LamportState() {
023:                 this(LamportMessages.RELEASE, 0);
024:         }
025:
026:         /**
027:          * Construit un message
028:          *
029:          * @param type Le type du message
030:          * @param timestamp L'estampille du message
031:          */
032:         public LamportState(LamportMessages type, int timestamp) {
033:                 this.type = type;
034:                 this.timestamp = timestamp;
035:         }
036:
037:         /**
038:          * Definit un message
039:          *
040:          * @param type Le type du message
041:          * @param timestamp L'estampille du message
042:          */
043:         public void set(LamportMessages type, int timestamp) {
044:                 this.type = type;
045:                 this.timestamp = timestamp;
046:         }
047:
048:         /**
049:          * Convertit le message en un tableau de byte.
050:          *
051:          * @return Le tableau de byte
052:          */
053:         public byte[] toByte(int bankId) {
054:                 byte[] data = new byte[1 + 4 + 4];
055:                 // Type [1]
056:                 data[0] = type.getCode();
057:                 // Timestamp [4]
058:                 byte[] temp = Toolbox.int2Byte(timestamp);
059:                 for (int i = 0; i < temp.length; i++) {
060:                         data[1 + i] = temp[i];
061:                 }
062:                 // Bank id [4]
```

```
063:                 temp = Toolbox.int2Byte(bankId);
064:                 for (int i = 0; i < temp.length; i++) {
065:                         data[5 + i] = temp[i];
066:                 }
067:
068:                 return data;
069:         }
070:
071:         /**
072:          * Lit un message LAMPORT et renvoie ses infos.
073:          *
074:          * @param data Donnees
075:          * @param length Longueur des donnees
076:          * @return type et entropie du message
077:          */
078:         public static LamportState fromByte(byte[] data,int length) {
079:                 assert (length >= 9);
080:                 // type [1]
081:                 LamportState state = new LamportState();
082:                 state.type = LamportMessages.fromCode(data[0]);
083:                 // timestamp[4]
084:                 byte[] tempInt = new byte[4];
085:                 for (int i = 0; i < tempInt.length; i++) {
086:                         tempInt[i] = data[i + 1];
087:                 }
088:                 state.timestamp = Toolbox.byte2int(tempInt);
089:                 // Bank ID[4]
090:                 tempInt = new byte[4];
091:                 for (int i = 0; i < tempInt.length; i++) {
092:                         tempInt[i] = data[i + 5];
093:                 }
094:                 state.remoteBankId = Toolbox.byte2int(tempInt);
095:
096:
097:                 return state;
098:         }
099:
100: }
101:
```

```
01:
02: /**
03:  * Donnees aditionnelles envoyees lorsque on relache un mutex Lamport.
04:  * Attention, 127 elements au maximum
05:  * @author Laurent Constantin
06:  * @author Jonathan Gander
07:  * @version 1.0
08:  */
09: public enum LamportUnlockMessage {
10:         DELETE_ACCOUNT,
11:         UPDATE_MONEY;
12:
13:         /**
14:          * Permet d'obtenir le code de l'element pour un transfert reseau
15:          * @return Le code en byte
16:          */
17:         public byte getCode(){
18:                 return (byte)this.ordinal();
19:         }
20:         /**
21:          * Reconsruit l'element depuis le code (sans verifications)
22:          * @param code Le code (ordinal)
23:          * @return L'element de l'enum
24:          */
25:         public static LamportUnlockMessage fromCode(byte code){
26:                 return LamportUnlockMessage.values()[(int)code];
27:         }
28: }
29:
```

-- Néant --

```java
/**
 * Classe permettant la modelisation du menu client.
 * Attention, 127 elements au maximum
 *
 * @version 1.0
 * @author Laurent Constantin
 * @author Jonathan Gander*
 */
public enum Menu {

	ADD_ACCOUNT("Creer un compte"),
	DELETE_ACCOUNT("Supprimer un compte"),
	ADD_MONEY("Ajouter un montant a un compte"),
	TAKE_MONEY("Retirer un montant  a un compte"),
	GET_BALANCE("Obtenir le solde d'un compte"),
	QUIT("Quitter");

	private final String text;
	/**
	 * Action utilisateur avec texte descriptif
	 * @param text La descritpion
	 */
	Menu(String text) {
		this.text = text;
	}

	/**
	 * Representation du menu
	 * @return la description
	 */
	public String toString() {
		return text;
	}
	/**
	 * Permet d'obtenir le code de l'element pour un transfert reseau
	 * @return Le code en byte
	 */
	public byte getCode(){
		return (byte)this.ordinal();
	}
	/**
	 * Reconsruit l'element depuis le code (sans verifications)
	 * @param code Le code (ordinal)
	 * @return L'element de l'enum
	 */
	public static Menu fromCode(byte code){
		return Menu.values()[(int)code];
	}
}

```

-- Néant --

```java
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;

/**
 * Represente un guichetier qui fait les requetes aux banques pour le client
 *
 * @version 1.0
 * @author Laurent Constantin
 * @author Jonathan Gander
 */
public class Teller  {
        private int port;
        private InetAddress host;

        private DatagramSocket socket;

        /**
         * Constructeur
         *
         * @param bankId
         *            Id de la banque a creer
         * @throws UnknownHostException
         *             Si la banque ne peut pas etre trouvee sur le reseau
         * @throws SocketException
         *             Si la socket ne peut etre ouverte
         */
        public Teller(int bankId) throws UnknownHostException, SocketException {
                if (bankId < 0 || bankId > Config.banksAddresses.length - 1)
                        throw new IllegalArgumentException(
                                        "No de banque invalide pour le guichetier !");

                port = Config.banks2ClientPorts[bankId];
                host = InetAddress.getByName(Config.banksAddresses[bankId]);

                socket = new DatagramSocket();

        }

        /**
         * Permet d'envoyer un tampon de donnees au serveur
         *
         * @param tampon
         *            Tampon a envoyer
         * @throws IOException
         *             Erreur lors de l'envoi
         */
        private void sendPacket(byte[] tampon) throws IOException {
                DatagramPacket packet = new DatagramPacket(tampon, tampon.length, host,
                                port);

                socket.send(packet);
        }

        /**
         * Attend la reception d'un paquet (bloquant)
         *
         * @return Le paquet recu
         * @throws IOException
```

```
063:          *             En cas d'erreur
064:          */
065:         private DatagramPacket receivePacket() throws IOException {
066:                 byte[] tampon = new byte[Config.bufferSize];
067:
068:                 DatagramPacket packet = new DatagramPacket(tampon, tampon.length);
069:                 //System.out.println("Attente de la reponse de la banque");
070:                 socket.receive(packet);
071:
072:                 return packet;
073:         }
074:
075:         /**
076:          * Permet d'ajouter un compte
077:          *
078:          * @param money
079:          *             Montant initial
080:          */
081:         public int addAccount(int money) {
082:                 try {
083:                         // Envoi de la requete
084:                         sendPacket(Toolbox.buildMessage(Menu.ADD_ACCOUNT.getCode(), money));
085:
086:                         // Reception de la reponse
087:                         DatagramPacket p = receivePacket();
088:                         ErrorServerClient code = ErrorServerClient.fromCode(p.getData()[0]);
089:
090:                         // Si aucune erreur
091:                         if (code == ErrorServerClient.OK) {
092:                                 // Renvoie le numero de compte
093:                                 int[] data = Toolbox.buildData(p);
094:                                 return data[0];
095:                         } else {
096:                                 return -1;
097:                         }
098:
099:                 } catch (IOException e) {
100:                         e.printStackTrace();
101:                 }
102:                 return -1;
103:
104:         }
105:
106:         /**
107:          * Permet de supprimer un compte
108:          *
109:          * @param account
110:          *             compte a supprimer
111:          */
112:         public ErrorServerClient deleteAccount(int account) {
113:                 try {
114:                         // Envoi de la requete
115:                         sendPacket(Toolbox.buildMessage(Menu.DELETE_ACCOUNT.getCode(),
116:                                         account));
117:
118:                         // Reception de la reponse
119:                         DatagramPacket p = receivePacket();
120:                         return ErrorServerClient.fromCode(p.getData()[0]);
121:
122:                 } catch (IOException e) {
123:                         e.printStackTrace();
124:                 }
```

```
125:                return ErrorServerClient.AUTRE;
126:        }
127:
128:        /**
129:         * Ajout de l'argent a un compte
130:         *
131:         * @param account
132:         *            Compte a crediter
133:         * @param money
134:         *            Montant a ajouter
135:         */
136:        public ErrorServerClient addMoney(int account, int money) {
137:                try {
138:                        // Envoi de la requete
139:                        sendPacket(Toolbox.buildMessage(Menu.ADD_MONEY.getCode(),
140:                                        account, money));
141:
142:                        // Reception de la reponse
143:                        DatagramPacket p = receivePacket();
144:                        return ErrorServerClient.fromCode(p.getData()[0]);
145:
146:                } catch (IOException e) {
147:                        e.printStackTrace();
148:                }
149:                return ErrorServerClient.AUTRE;
150:        }
151:
152:        /**
153:         * Debite de l'argent a un compte
154:         *
155:         * @param account
156:         *            Compte a debiter
157:         * @param money
158:         *            Montant a retirer
159:         */
160:        public ErrorServerClient takeMoney(int account, int money) {
161:                try {
162:                        // Envoi de la requete
163:                        sendPacket(Toolbox.buildMessage(Menu.TAKE_MONEY.getCode(),
164:                                        account, money));
165:
166:                        // Reception de la reponse
167:                        DatagramPacket p = receivePacket();
168:                        return ErrorServerClient.fromCode(p.getData()[0]);
169:
170:                } catch (IOException e) {
171:                        e.printStackTrace();
172:                }
173:                return ErrorServerClient.AUTRE;
174:        }
175:
176:        /**
177:         * Obtenir le solde du compte
178:         *
179:         * @param account
180:         *            Compte
181:         * @return Solde du compte
182:         */
183:        public int getBalance(int account) {
184:                try {
185:                        // Envoi de la requete
186:                        sendPacket(Toolbox.buildMessage(Menu.GET_BALANCE.getCode(),
```

```
187:                                        account));
188:
189:                        // Reception de la reponse
190:                        DatagramPacket p = receivePacket();
191:                        ErrorServerClient code = ErrorServerClient.fromCode(p.getData()[0]);
192:
193:                        if (code == ErrorServerClient.OK) {
194:                                return Toolbox.buildData(p)[0];
195:                        }
196:
197:                } catch (IOException e) {
198:                        e.printStackTrace();
199:                }
200:                return -1;
201:        }
202:
203: }
204:
```

```java
import java.net.DatagramPacket;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.Scanner;

/**
 * Classe utilitaire - Conversion en byte - Saisies utilisateur
 *
 * @version 1.0
 * @author Laurent Constantin
 * @author Jonathan Gander
 */
public class Toolbox {
	/**
	 * @return Saisie d'une banque
	 */
	public static int readBank() {
		int n;
		try {
			Scanner in = new Scanner(System.in);
			n = in.nextInt();
		} catch (Exception e) {
			return -1;
		}

		if (n < 0 || n > Config.banksAddresses.length - 1)
			return -1;
		else
			return n;
	}

	/**
	 * @return Menu en fonction d'un entier
	 */
	public static Menu readMenu() {
		int n;
		try {
			Scanner in = new Scanner(System.in);
			n = in.nextInt();
		} catch (Exception e) {
			return null;
		}

		if (n < 0 || n > Menu.values().length)
			return null;
		else
			return Menu.values()[n];
	}

	/**
	 * Lit la saisie d'un int compris entre min et max
	 *
	 * @param min valeur min
	 * @param max valeur max
	 * @return int compris entre min et max (inclus)
	 */
	public static int readInt(int min, int max) {
		if (min > max) {
			int tmp = max;
			max = min;
			min = tmp;
		}
```

```
063:
064:                int n;
065:
066:                do {
067:                        try {
068:                                Scanner in = new Scanner(System.in);
069:                                n = in.nextInt();
070:
071:                                if (n >= min && n <= max) {
072:                                        return n;
073:                                }
074:                                String msg = "Veuillez entrer une valeur entre " + min + " et "
075:                                                + max + " > ";
076:
077:                                if (max == Integer.MAX_VALUE)
078:                                        msg = "Veuillez entrer une valeur superieur a " + min
079:                                                        + " > ";
080:
081:                                System.out.print(msg);
082:                        } catch (Exception e) {
083:                                System.out.println("Erreur de saisie");
084:                        }
085:                } while (true);
086:        }
087:
088:        /**
089:         * Permet de convertir un long en un tableau de byte
090:         *
091:         * @param l long
092:         * @return Tableau de byte.
093:         */
094:        public static byte[] long2Byte(long l) {
095:                ByteBuffer boeuf = ByteBuffer.allocate(8);
096:                boeuf.order(ByteOrder.BIG_ENDIAN);
097:                boeuf.putLong(l);
098:                return boeuf.array();
099:        }
100:
101:        /**
102:         * Permet de convertir un tableau de byte en long
103:         *
104:         * @param b le tableau de byte
105:         * @return Le long qu'il contient
106:         */
107:        public static long byte2Long(byte[] b) {
108:                ByteBuffer bb = ByteBuffer.wrap(b);
109:                return bb.getLong();
110:        }
111:
112:        /**
113:         * Permet de convertir un int en un tableau de byte
114:         *
115:         * @param i int
116:         * @return Tableau de byte.
117:         */
118:        public static byte[] int2Byte(int i) {
119:                ByteBuffer boeuf = ByteBuffer.allocate(4);
120:                boeuf.order(ByteOrder.BIG_ENDIAN);
121:                boeuf.putInt(i);
122:                return boeuf.array();
123:        }
124:
```

```
125:         /**
126:          * Permet de convertir un tableau de byte en int
127:          *
128:          * @param b le tableau de byte
129:          * @return Le int qu'il contient
130:          */
131:         public static int byte2int(byte[] b) {
132:                 ByteBuffer bb = ByteBuffer.wrap(b);
133:                 return bb.getInt();
134:         }
135: 
136:         /**
137:          * Permet de construire un message avec la methode et les donnees a envoyer
138:          *
139:          * @param code Code de la methode
140:          * @param datas Data a envoyer
141:          * @return Message a envoyer au serveur
142:          */
143:         public static byte[] buildMessage(byte code, int... datas) {
144:                 byte[] message = new byte[1 + datas.length * 4];
145: 
146:                 message[0] = code;
147: 
148:                 int indice = 1;
149:                 for (int i = 0; i < datas.length; i++) {
150:                         byte[] data = Toolbox.int2Byte(datas[i]);
151: 
152:                         for (int j = 0; j < data.length; j++) {
153:                                 message[indice++] = data[j];
154:                         }
155:                 }
156: 
157:                 return message;
158:         }
159: 
160:         /**
161:          * Permet de re-construire les donnees a partir du message recu Les donnes
162:          * sont obtenue a partir d'un DatagramPacket
163:          *
164:          * @see buildData(byte[], int)
165:          *
166:          * @param packet Un datagramPacket
167:          * @return les donnees du message
168:          */
169:         public static int[] buildData(DatagramPacket packet) {
170:                 return buildData(packet.getData(), packet.getLength(),0);
171:         }
172:         /**
173:          * Permet de re-construire les donnees a partir du message recu Le code est
174:          * ignore (message[0]);
175:          *
176:          * @param message les donnes recu
177:          * @param taille du message
178:          * @return les donnees du message
179:          */
180:         public static int[] buildData(byte[] message, int length,int offset) {
181:                 // Attention: Ne pas utiliser message.length,
182:                 // renvoie la taille du buffer et pas du contenu
183: 
184:                 if (length <= 1)
185:                         throw new IllegalArgumentException();
186: 
```

```
187:                // le code du message se trouve dans message[0];
188:                // Le reste des donnees sont des entiers..
189:
190:                // Si pas de donnees, renvoie un tableau vide (eviter le null)
191:                if ((length - 1) % 4 != 0) {
192:                        return new int[0];
193:                }
194:                // Si des donnees
195:                int nbInt = (length - 1) / 4;
196:                int data[] = new int[nbInt];
197:
198:                if (message.length > 1) {
199:                        for (int index = 0; index < nbInt; index++) {
200:                                byte temp[] = new byte[4];
201:                                temp[0] = message[offset + 1 + index * 4];
202:                                temp[1] = message[offset + 2 + index * 4];
203:                                temp[2] = message[offset + 3 + index * 4];
204:                                temp[3] = message[offset + 4 + index * 4];
205:                                data[index] = Toolbox.byte2int(temp);
206:                        }
207:                }
208:                return data;
209:        }
210:
211:        /**
212:         * Renvoie le code d'un message
213:         *
214:         * @param p Un datagramme
215:         * @return Le code
216:         */
217:        public static Byte getDataCode(DatagramPacket p) {
218:                if (p == null || p.getLength() == 0)
219:                        return null;
220:
221:                return p.getData()[0];
222:        }
223:
224:        /**
225:         * Concatener deux tableaux de byte
226:         *
227:         * @param d1 Le premier tableau
228:         * @param d2 Le deuxieme tableau
229:         * @return la concatenation
230:         */
231:        public static byte[] concat(byte[] d1, byte[] d2) {
232:                byte[] data = new byte[d1.length + d2.length];
233:                for (int i = 0; i < d1.length; i++) {
234:                        data[i] = d1[i];
235:                }
236:                for (int i = 0; i < d2.length; i++) {
237:                        data[i + d2.length] = d2[i];
238:                }
239:                return data;
240:        }
241:
242: }
243:
```