esial
ÉCOLE SUPÉRIEURE D'INFORMATIQUE ET APPLICATIONS DE LORRAINE

Nancy-Université
Université Henri Poincaré

IA – 2008 / 2009

Programmation Orientée Objet

5ème Partie

Gérald Oster <oster@loria.fr>

Plan du cours

- Introduction
- Programmation orientée objet :
  - Classes, objets, encapsulation, composition
  - 1. Utilisation
  - 2. Définition
- Héritage et polymorphisme :
  - Interface, classe abstraite, liaison dynamique
- Exceptions
- Généricité

7ème Partie : Entrée/Sortie & Exceptions

Objectifs de cette partie

- Sauvegarder et lire des fichiers textes
- Apprendre à lever des exceptions
- Savoir définir ses propres exceptions
- Savoir différencier les exceptions vérifiées (*checked*) et les exceptions non vérifiées (*unchecked*)
- Apprendre à attraper des exceptions
- Savoir quand et où attraper les exceptions levées

Lecture de fichiers textes

- La manière la plus simple de lire du texte :
  - utiliser la classe `Scanner`
- Pour lire à partir d'un fichier, construire un objet `FileReader`
- Puis, utiliser le `FileReader` pour construire un objet `Scanner`

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

- Utiliser les méthodes de la classe `Scanner` pour lire les données du fichier

  `next`, `nextLine`, `nextInt`, et `nextDouble`

Ecriture de fichiers textes

- Pour écrire dans un fichier, construire un objet `PrintWriter`

```
PrintWriter out = new PrintWriter("output.txt");
```

- Si le fichier existe déjà, le contenu est écrasé
- Si le fichier n'existe pas, il est crée
- Utiliser `print` et `println` pour écrire dans le `PrintWriter`:

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

- Il faut fermer le fichier après l'avoir manipuler: `out.close();`

  Sinon, il se peut que tout ne soit pas écrit dans le fichier

FileNotFoundException

- Lorsque le fichier d'entrée (lecture) n'existe pas, il se peut qu'une exception `FileNotFoundException` soit levée
- Pour ne pas traiter l'exception, modifier le profil de la méthode main de la sorte :

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

Un programme exemple

- Lire toutes les lignes d'un fichier et les recopier dans un autre fichier en les numérotant.
- Exemple de fichier en entrée :

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

- Exemple de fichier en sortie :

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

ch11/fileio/LineNumberer.java

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

ch11/fileio/LineNumberer.java /2

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

Questions

Que se passe-t-il si l'on donne le même nom de fichier en entrée et en sortie du programme `LineNumberer`?

Réponse : Lors de la création de l'objet `PrintWriter`, le fichier de sortie est écrasé. Autrement dit, la boucle de lecture va s'arrêter dès le départ.

Questions

Que se passe-t-il si l'on fournit un nom d'un fichier qui n'existe pas comme nom de fichier à lire au programme `LineNumberer` ?

Réponse : Une exception de type `FileNotFoundException` est levée, un message d'erreur sera affiché et le programme termine.

Lever des exceptions

- Lever une exception signale qu'un événement exceptionnel s'est produit
- Exemple : `IllegalArgumentException:` une valeur non légale d'une paraètre

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

- Il n'est pas nécessaire de stocker l'exception dans une variable temporaire :

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

- Quand une exception est levée, la méthode termine immédiatement et le contrôle est rendu à la méthode appelante :

L'exécution se poursuit plus loin si il y a un bloc de code gérant l'exception.

Exemple

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

Hiérarchie des classes Exception

Syntaxe Levée d'une Exception

```
throw exceptionObject;
```

Exemple :

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

Objectif :

Lever une exception et transférer le contrôle à un bloc gérant l'exception (ou à défaut à la méthode appelante).

Questions

Comment doit-on modifier la méthode `deposit` pour s'assurer que le montant n'est jamais négatif ?

Réponse : Lever une exception si le montant à déposer est négatif

Questions

Supposons alors que l'on construise un nouveau compte bancaire avec un solde nul et que l'on appelle la méthode `withdraw(10)`. Quel est la valeur de `balance` après cette exécution ?

Réponse: La balance reste nulle puisque la dernière instruction de la méthode `withdraw` n'a jamais été exécuté.

Exception testées (*checked*) et non testées (*unchecked*)

- Deux types d'exceptions :
  - *Testées (checked)*
    - Le compilateur vérifie que votre code ne les ignore pas
    - Elles sont généralement due à des circonstances externes que le développeur peut prévoir
    - Par exemple, `IOException`
  - *Non-testées (unchecked)*
    - Etendent la classe `RuntimeException` ou `Error`
    - Elles sont dues à une faute du développeur
    - Par exemple :
      `NumberFormatException`
      `IllegalArgumentException`
      `NullPointerException`
    - Exemple d'erreur :
      `OutOfMemoryError`

Exception testées (*checked*) et non testées (*unchecked*) /2

- Les catégories précédentes ne sont pas parfaites :
  - *`Scanner.nextInt` peut lever une exception non vérifiée `InputMismatchException`*
  - *Le développeur ne peut prévoir les données que l'utilisateur va saisir*
  - *Ce choix simplifie l'utilisation aux développeurs novices*
- Manipuler des exceptions vérifiées lorsque l'on programme avec des entrées-sorties vers des flux (fichiers).
- Par exemple, utiliser la classee `Scanner` pour lire un fichier

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

- Mais le constructeur de `FileReader` peut lever une exception `FileNotFoundException`

Exception testées (*checked*) et non testées (*unchecked*) /3

Deux choix :

1. *Gérer l'exception*
2. *Indiquer au compilateur que l'on souhaite que l'exécution de la méthode se termine lorsque l'exception survient*
   - Utiliser le mot clé `throws` pour que la méthode puisse lever l'exception vérifiée

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

   - Pour plusieurs exceptions :

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

Exception testées (*checked*) et non testées (*unchecked*) /4

- Garder à l'esprit l'arbre d'héritage :
  Si une méthode peut lever une exceptionI`IOException` et `FileNotFoundException`, utiliser uniquement IOException

- Il est préférable de déclarer des exceptions (pour les traiter plus tard) que de ne pas les gérer correctement

Syntaxe Spécification d'une Exception

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

Exemple :

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

Objectif :
Pour indiquer qu'une méthode peut lever des exceptions vérifiées.

Questions

Supposons qu'une méthode appelle le constructeur `FileReader` et la méthode `read` de la classe `FileReader`, qui peuvent lever une exception `IOException`. Quelle spécifications pour throws doit-on utiliser ?

Réponse : L'indication throws `IOException` est suffisante car `FileNotFoundException` est une sous classe de `IOException`.

Questions

Pourquoi `NullPointerException` n'est pas une exception vérifiée ?

Réponse : Parce que les développeurs peuvent généralement vérifier leur programme pour éviter ces erreurs au lieu d'être obligés d'écrire du code pour gérer une `NullPointerException`.

Attraper des Exceptions

- Installer un bloc gérant les exceptions en utilisant les instructions `try/catch`
- Un bloc `try` contient les instructions susceptibles de lever les exceptions
- La claude `catch` contient le code à exécuter en cas d'exception d'un type particulier

Attraper des Exceptions /2

- Exemple:

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

Attraper des Exceptions /3

- Les instructions dans le bloc `try` sont exécutées
- Si aucune exception n'est levée la clause `catch` est ignorée
- Si une exception d'un des types spécifiés est levée, l'exécution du bloc `try` s'arrête et la clause `catch` correspondante est exécutée
- Si une exception d'un autre type est levée, elle est propagée jusqu'à ce qu'un bloc `catch` la capture
- `catch (IOException exception) block`
  - *`exception` contient la référence vers l'exception qui est levée*
  - *`catch` peut analiser l'objet pour obtenir des détails sur l'erreur*
  - *`exception.printStackTrace()`: affiche la pile des appels de méthodes qui ont causé l'exception*

Syntaxe Bloc Try/Catch

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

Syntaxe Bloc Try/Catch /2

Exemple:

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

Objectif :

Exécuter une ou plusieurs instructions susceptibles de lever des exceptions. Capturer et traiter les exceptions

Questions

Existe-t-il une différence entre attraper une exception non vérifiée et une exception vérifiée ?

Réponse : Non – on peut attraper les deux types d'exceptions de la même manière

La clause `finally`

- Une levée d'exception termine l'exécution courante
- Danger : on peut ignorer du code essentiel
- Exemple :

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close(); // Peut ne jamais arriver ici
```

- Il faut absolument exécuter `reader.close()` même si une exception est levée lors de la lecture
- Utiliser une clause `finally` pour exécuter du code "dans tous les cas"

La clause `finally` /2

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

La clause `finally` /3

- Exécuté lors de la sortie du bloc `try` dans un des 3 cas suivants :
  - *Après la dernière instructions du bloc `try`*
  - *Après la dernière instructions de la clause catch, si le bloc `try` a levé une exception*
  - *Lorsqu'une exception a été levée dans le bloc `try` mais n'a pas été capturée par une clause catch*

Syntaxe clause `finally`

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

Syntaxe clause `finally` /2

Exemple:

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

Objectif :

Garantit que les instructions de la clause`finally` sont toujours exécutées qu'une exception soit ou ne soit pas levée dans le bloc `try`.

Questions

Pourquoi la variable `reader` est-elle déclarée hors du bloc `try` ?

Réponse : Si elle avait été déclarée dans le bloc `try`, sa portée serait limitée à ce bloc, et la clause `catch` ne pourrait l'utiliser pour fermer le flux.

Concevoir ses propres exceptions

- On peut concevoir ses propres types d'exceptions – sous classes de `Exception` ou `RuntimeException`
-
```
if (amount > balance)
{
   throw new InsufficientFundsException(
      "withdrawal of " + amount + " exceeds balance of "
            + balance);
}
```
- La transformer en une exception non vérifiée – le développeur aurait pu l'éviter en appelant la méthode `getBalance` en premier
- Etendre `RuntimeException` ou une des ses sous-classes
- Fournir deux constructeurs
  1. *Constructeur par défaut*
  2. *Un constructeur qui accepte une chaîne de caractères décrivant la raison de l'exception*

Concevoir ses propres exceptions /2

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

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

Questions

Quel est le but de l'appel à `super(message)` dans le second constructeur de `InsufficientFundsException` ?

Réponse : Passer le message d'exception au constructeur de la super classe `RuntimeException`.

Un exemple complet

- Programme
  - *Demander à l'utilisateur un nom de fichier*
  - *Le fichier escompté contient des valeurs*
  - *La première ligne contient le nombre de valeurs du fichier*
  - *Les autres lignes contiennent les données*
  - *Par exemple :*

    3
    1.45
    -2.1
    0.05

Un exemple complet /2

- Qu'est-ce qui peut échouer ?
  - *Le fichier peut ne pas exister*
  - *Le fichier peut contenir des données érronées*
- *Qui peut détecter les erreurs ?*
  - *Le constructeur* `FileReader` *levera une exception si le fichier n'existe pas*
  - *Les méthodes qui manipulen les entrées doivent pouvoir lever des exceptions si elles détectent une erreur de format dans les données*
- Quelles sont les exceptions qui peuvent être levées ?
  - `FileNotFoundException` *peut être levé par le constructeur* `FileReader`
  - `IOException` *peut être levé par la méthode close de* `FileReader`
  - `BadDataException`*, une exception "à nous"*

Un exemple complet /3

- Qui peut remedier aux erreurs ?
  - *Seulement la méthode main du programme* `DataSetTester` *interagit avec l'utilisateur*
  - *Attraper les execptions*
  - *Afficher des messages d'erreur appropriés*
  - *Donne une autre chance à l'utilisateur de donner un fichier correct*

ch11/data/DataAnalyzer.java

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

ch11/data/DataAnalyzer.java /2

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

La méthode `readFile` de la classe `DataSetReader`

- Construit un objet `Scanner`
- Appelle la méthode `readData`
- N'est pas concernée pas aucune des exceptions
- Si il y a un problème avec le fichier, elle doit simplement propager l'exception à l'appelant

La méthode `readFile` de la classe `DataSetReader` /2

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

La méthode `readData` de la classe `DataSetReader`

- Les les nombres
- Construit un tableau
- Appelle la méthode `readValue` pour chaque valeur

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

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

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

La méthode `readData` de la classe `DataSetReader` /2

- Vérifie deux erreurs potentielles
  - *Le fichier peut ne pas débuter par un entier*
  - *Le fichier peut contenir plus (ou moins) de valeurs que indiqué*
- N'essaye pas d'attraper une exception

La méthode `readValue` de la classe `DataSetReader` /2

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

Animation 11.1 –

```
21    {
22       FileReader reader = new FileReader(filename);
23       try
24       {
25          Scanner in = new Scanner(reader);
26          readData(in);
27       }
28       finally
29       {
30          reader.close();
31       }
32       return data;
33    }
34
35    /**
36       Reads all data.
37       @param in the scanner that scans the data
38    */
```

This animation walks through an exception handling scenario with the `DataAnalyzer` class from Chapter 11. You will learn about throwing exceptions, catching exceptions, and the `finally` clause.

Scenario

1. `DataSetTester.main` appelle `DataSetReader.readFile`
2. `readFile` appelle `readData`
3. `readData` appelle `readValue`
4. `readValue` ne trouve pas la valeur attendu et lève `BadDataException`
5. `readValue` ne traite pas l'exception et termine
6. `readData` ne traite pas l'exception et termine
7. `readFile` ne traite pas l'exception et termine après l'exécution de la clause `finally`
8. `DataSetTester.main` traite l'exception de type `BadDataException` ; Il affiche un message et donne une autre chance à l'utilisateur de saisir un nom de fichier.

ch11/data/DataSetReader.java

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

ch11/data/DataSetReader.java /2

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

ch11/data/DataSetReader.java /3

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

Questions

Pourquoi la méthode `DataSetReader.readFile` n'attrape-t-elle aucune exception ?

Réponse : Elle ne saurait pas trop quoi faire avec. La classe `DataSetReader` est réutilisable et peut être utilisée pour des systèmes avec des messages dans d'autres langues ou avec d'autres types d'interface utilisateur (graphique). Elle ne doit donc pas communiquer avec l'utilisateur directement

Questions

Supposons que l'utilisateur saisisse un nom de fichier qui existe dont le contenu du fichier est vide. Tracer le flot d'exécution.

Réponse : `DataSetTester.main` appelle `DataSetReader.readFile`, qui appelle `readData`. L'appel `in.hasNextInt()` retourne `false`, et `readData` lève une exception `BadDataException`. La méthode `readFile` n'attrape pas cette exception et elle est propagée à la méthode appelante `mai`, ou elle est traitée.

8ème Partie : Généricité

Objectifs de cette partie

- Comprendre l'intérêt de la généricité
- Pourvoir implémenter des méthode génériques et des classes génériques
- Comprendre l'exécution des méthodes générique par la machine virtuelle
- Connaître les limitations de la généricité en Java
- Comprendre les relations en généricité et héritage
- Apprendre à contraindre des types paramètrés

Types paramètrés

- *Programmation générique : "bouts de programme" qui peuvent être utilisés avec différents types de données*
  - *En Java, accompli en utilisant l'héritage et des types paramètrés*
- Par exemple:
  - *Types paramètrés:* `ArrayList` *(ex:* `ArrayList<String>`*)*
  - *Héritage :* `LinkedList` *peuvent stocker n'importe quel type d'objet*
- *Classe générique*: déclarée avec un type paramètré E
- Le type paramètré indique le type des éléments :

```
public class ArrayList<E> // could use "ElementType"
instead of E
{
   public ArrayList() { . . . }
   public void add(E element) { . . . }
   . . .
}
```

Types paramètrés /2

Peut être instancié avec le type d'une classe ou d'une interface

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

Ne peut être instancié avec un type primitif

```
ArrayList<double> // Faux!
```

Utiliser la classe enveloppe (*wrapper*) correspondante

```
ArrayList<Double>
```

Types paramètrés /3

- Le type fournie remplace le type paramètré dans l'interface de la classe
- Exemple: `add` dans `ArrayList<BankAccount>` a la variable `E` remplacée par `BankAccount`:

```
public void add(BankAccount element)
```

- Contrairement à la méthode `LinkedList.add`:

```
public void add(Object element)
```

Les types paramètrés améliore la correction

Les Types paramètrés rendent le code générique plus sûre et plus facile à lire

Impossible d'ajouter un String dans un `ArrayList<BankAccount>`
On peut ajouter un String dans une `LinkedList` où l'on souhaitait conserver uniquement des comptes bancaires

```
ArrayList<BankAccount> accounts1 =
                 new ArrayList<BankAccount>();
LinkedList accounts2 = new LinkedList();
     // ne devrait pas contenir des objets BankAccount
accounts1.add("my savings");
           // Erreur à la compilation
accounts2.add("my savings");
           // Non détecté à la compilation
. . .
BankAccount account = (BankAccount)
accounts2.getFirst(); // Erreur à l'exécution
```

Syntaxe Instanciation d'une classe générique

GenericClassName<Type1, Type2, . . .>

Exemple :

```
ArrayList<BankAccount>
HashMap<String, Integer>
```

Objectifs :
Fournir un type spécifique à la place du type paramètré d'une classe générique.

Questions

La librairie standard founie une classe `HashMap<K, V>` dont le type de la clé est e `K` et le type de la valeur est `V`. Déclarer une table de hachage qui associe des chaînes de caractères à des entiers.

Réponse : `HashMap<String, Integer>`

Implémenter des classes génériques

- Exemple: une simple classe qui stocke des paires de valeurs

```
Pair<String, BankAccount> result =
 new Pair<String, BankAccount>("Harry Hacker”,
                                        harrysChecking);
```

- Les méthodes `getFirst` et `getSecond` retournent la première et la seconde valeur de la paire

```
String name = result.getFirst();
BankAccount account = result.getSecond();
```

- Exemple d’utilisation : retourner deux valeurs à la fois (une méthode qui retourne un objet `Pair`)
- La classe générique `Pair` requiert deux types paramètrés `Pair<T, S>`

De bons noms de types paramètrés

Type paramètré	Signification
E	Type d’un élément d’une collection (**E**lement)
K	Type de la clé d’une valeur (**K**ey)
V	Type d’une valeur (**V**alue)
T	Type général (**T**ype)
S, U	D’autres types généraux

Classe `Pair`

```
public class Pair<T, S>
{
   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }

   private T first;
   private S second;
}
```

Transformer la classe `LinkedList` en une classe générique

```
public class LinkedList<E>
{
   . . .
   public E removeFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      E element = first.data;
      first = first.next; return element;
   }
   . . .
   private Node first;

   private class Node
   {
      public E data;
      public Node next;
   }
}
```

Implémenter des classes génériques

- Utiliser le type `E` quand on reçoit, retourne ou stocke un objet de type element

```
public class ListNode<E>.
```

Syntaxe Définition d'une classe générique

```
accessSpecifier class GenericClassName<TypeVariable1,
TypeVariable2, . . .>
{
   constructors
   methods
   fields
}
```

Exemple :

```
public class Pair<T, S>
{
   . . .
}
```

Objectifs :

Définir une classe générique dont les méthodes et les variables dépendent des types paramètrés.

ch17/genlist/LinkedList.java

```
001: import java.util.NoSuchElementException;
002:
003: /**
004:    A linked list is a sequence of nodes with efficient
005:    element insertion and removal. This class
006:    contains a subset of the methods of the standard
007:    java.util.LinkedList class.
008: */
009: public class LinkedList<E>
010: {
011:    /**
012:       Constructs an empty linked list.
013:    */
014:    public LinkedList()
015:    {
016:       first = null;
017:    }
018:
019:    /**
020:       Returns the first element in the linked list.
021:       @return the first element in the linked list
022:    */
```

ch17/genlist/LinkedList.java /2

```
023:    public E getFirst()
024:    {
025:       if (first == null)
026:          throw new NoSuchElementException();
027:       return first.data;
028:    }
029:
030:    /**
031:       Removes the first element in the linked list.
032:       @return the removed element
033:    */
034:    public E removeFirst()
035:    {
036:       if (first == null)
037:          throw new NoSuchElementException();
038:       E element = first.data;
039:       first = first.next;
040:       return element;
041:    }
042:
043:    /**
044:       Adds an element to the front of the linked list.
045:       @param element the element to add
046:    */
```

ch17/genlist/LinkedList.java /3

```
047:    public void addFirst(E element)
048:    {
049:       Node newNode = new Node();
050:       newNode.data = element;
051:       newNode.next = first;
052:       first = newNode;
053:    }
054:
055:    /**
056:       Returns an iterator for iterating through this list.
057:       @return an iterator for iterating through this list
058:    */
059:    public ListIterator<E> listIterator()
060:    {
061:       return new LinkedListIterator();
062:    }
063:
064:    private Node first;
065:
066:    private class Node
067:    {
```

ch17/genlist/LinkedList.java /4

```
068:       public E data;
069:       public Node next;
070:    }
071:
072:    private class LinkedListIterator implements ListIterator<E>
073:    {
074:       /**
075:          Constructs an iterator that points to the front
076:          of the linked list.
077:       */
078:       public LinkedListIterator()
079:       {
080:          position = null;
081:          previous = null;
082:       }
083:
084:       /**
085:          Moves the iterator past the next element.
086:          @return the traversed element
087:       */
088:       public E next()
089:       {
```

ch17/genlist/LinkedList.java /5

```
090:          if (!hasNext())
091:             throw new NoSuchElementException();
092:          previous = position; // Remember for remove
093:
094:          if (position == null)
095:             position = first;
096:          else
097:             position = position.next;
098:
099:          return position.data;
100:       }
101:
102:       /**
103:          Tests if there is an element after the iterator
104:          position.
105:          @return true if there is an element after the iterator
106:          position
107:       */
108:       public boolean hasNext()
109:       {
110:          if (position == null)
111:             return first != null;
```

ch17/genlist/LinkedList.java /6

```
112:          else
113:             return position.next != null;
114:       }
115:
116:       /**
117:          Adds an element before the iterator position
118:          and moves the iterator past the inserted element.
119:          @param element the element to add
120:       */
121:       public void add(E element)
122:       {
123:          if (position == null)
124:          {
125:             addFirst(element);
126:             position = first;
127:          }
128:          else
129:          {
130:             Node newNode = new Node();
131:             newNode.data = element;
132:             newNode.next = position.next;
133:             position.next = newNode;
```

ch17/genlist/LinkedList.java /7

```
134:             position = newNode;
135:          }
136:          previous = position;
137:       }
138:
139:       /**
140:          Removes the last traversed element. This method may
141:          only be called after a call to the next() method.
142:       */
143:       public void remove()
144:       {
145:          if (previous == position)
146:             throw new IllegalStateException();
147:
148:          if (position == first)
149:          {
150:             removeFirst();
151:          }
152:          else
153:          {
154:             previous.next = position.next;
155:          }
```

ch17/genlist/LinkedList.java /8

```
156:          position = previous;
157:       }
158:
159:       /**
160:          Sets the last traversed element to a different
161:          value.
162:          @param element the element to set
163:       */
164:       public void set(E element)
165:       {
166:          if (position == null)
167:             throw new NoSuchElementException();
168:          position.data = element;
169:       }
170:
171:       private Node position;
172:       private Node previous;
173:    }
174: }
```

ch17/genlist/ListIterator.java

```
01: /**
02:    A list iterator allows access of a position in a linked list.
03:    This interface contains a subset of the methods of the
04:    standard java.util.ListIterator interface. The methods for
05:    backward traversal are not included.
06: */
07: public interface ListIterator<E>
08: {
09:    /**
10:       Moves the iterator past the next element.
11:       @return the traversed element
12:    */
13:    E next();
14:
15:    /**
16:       Tests if there is an element after the iterator
17:       position.
18:       @return true if there is an element after the iterator
19:       position
20:    */
21:    boolean hasNext();
```

ch17/genlist/ListIterator.java /2

```
22:
23:    /**
24:       Adds an element before the iterator position
25:       and moves the iterator past the inserted element.
26:       @param element the element to add
27:    */
28:    void add(E element);
29:
30:    /**
31:       Removes the last traversed element. This method may
32:       only be called after a call to the next() method.
33:    */
34:    void remove();
35:
36:    /**
37:       Sets the last traversed element to a different
38:       value.
39:       @param element the element to set
40:    */
41:    void set(E element);
42: }
```

ch17/genlist/ListTester.java

```
01: /**
02:    A program that tests the LinkedList class
03: */
04: public class ListTester
05: {
06:    public static void main(String[] args)
07:    {
08:       LinkedList<String> staff = new LinkedList<String>();
09:       staff.addFirst("Tom");
10:       staff.addFirst("Romeo");
11:       staff.addFirst("Harry");
12:       staff.addFirst("Dick");
13:
14:       // | in the comments indicates the iterator position
15:
16:       ListIterator<String> iterator = staff.listIterator(); // |DHRT
17:       iterator.next(); // D|HRT
18:       iterator.next(); // DH|RT
19:
20:       // Add more elements after second element
21:
```

ch17/genlist/ListTester.java /2

```
22:       iterator.add("Juliet"); // DHJ|RT
23:       iterator.add("Nina"); // DHJN|RT
24:
25:       iterator.next(); // DHJNR|T
26:
27:       // Remove last traversed element
28:
29:       iterator.remove(); // DHJN|T
30:
31:       // Print all elements
32:
33:       iterator = staff.listIterator();
34:       while (iterator.hasNext())
35:       {
36:          String element = iterator.next();
37:          System.out.print(element + " ");
38:       }
39:       System.out.println();
40:       System.out.println("Expected: Dick Harry Juliet Nina Tom");
41:    }
42: }
```

ch17/genlist/ListTester.java /3

Output:

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```

Questions

Comment utiliseriez-vous la classe générique `Pair` pour construire une paire de chaînes de caractères `"Hello"` et `"World"` ?

Réponse : `new Pair<String, String>("Hello", "World")`

Méthodes génériques

- *Méthode générique* : méthode utilisant un type paramètré
- Peut être définie dans une classe générique ou ordinaire
- Une mérhode ordinaire (non générique) :

```
/**
   Prints all elements in an array of strings.
   @param a the array to print
*/
public static void print(String[] a)
{
   for (String e : a)
      System.out.print(e + " ");
      System.out.println();
}
```

Méthodes génériques /2

- Que faire si l'on souhaite afficher un tableau d'objets de type Rectangle à la place ?

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Méthodes génériques /3

- Lorsque l'on appelle une méthode générique, il n'est pas nécessaire d'instancier le type paramètré :

```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

- Le compilateur déduit que `E` doit être `Rectangle`
- On peut aussi définir des méthodes génériques qui ne sont pas des méthodes de classes (static)
- On peut définir des méthodes génériques dans des classes génériques
- On ne peut pas remplacer les types paramètrés par des types primitifs
  ex.: la méthode générique `print` ne peut être utilisée pour afficher un tableau de type `int[]`

Syntaxe Définir une méthode générique

```
modifiers <TypeVariable1, TypeVariable2, . . .> returnType
methodName(parameters)
{
   body
}
```

Exemple :

```
public static <E> void print( E[] a)
{
   . . .
}
```

Objectifs :
Définir une méthode générique qui dépend de types paramètrés.

Questions

Est-ce que la méthode `getFirst` de la classe `Pair` est une méthode générique ?

Réponse : Non – la méthode n'a pas de paramètre. C'est une méthode ordinaire définie dans une classe générique.

Généricité contrainte

- Les types paramètrés peuvent être contraints

```
public static <E extends Comparable> E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

- On peut appeler `min` avec un tableau `String[]` mais pas avec une tableau `Rectangle[]`
- `Comparable` implique que l'on puisse appeler la méthode `compareTo`
  Sinon la méthode `min` ne pourrait être compilée

Généricité contrainte /2

- Parfois (rarement), on souhaite fournir plusieurs contraintes

```
<E extends Comparable & Cloneable>
```

- `extends` appliqué à un type paramètré signifie aussi bien "extends" que "implements"
- Les contraintes peuvent être aussi bien des classes que des interfaces
- Les types paramètrés peuvent être remplacés aussi bien par des classes que par des interfaces

Questions

Modifier la méthode min pour calculer le minimum d'un tableau d'élements qui implémente l'interface `Measurable` définie dans un cours précédent .

Réponse :

```
public static <E extends Measurable> E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
   if (a[i].getMeasure() < smallest.getMeasure()) < 0)
      smallest = a[i];
   return smallest;
}
```

Types "étoilés"

Nom	Syntaxe	Signification
Wildcard with lower bound	? extends B	Tout type sous classe de B
Wildcard with higher bound	? super B	Tout type super type de B
Unbounded wildcard	?	Tout type

Types "étoilés"

- `public void addAll(LinkedList<? extends E> other)`

```
{
   ListIterator<E> iter = other.listIterator();
   while (iter.hasNext()) add(iter.next());
}
```

- `public static <E extends Comparable<E>> E min(E[] a)`
- `public static <E extends Comparable<? super E>> E min(E[] a)`
- `static void reverse(List<?> list)`

Vous pouvez voir cette déclaration comme un raccourci pour la déclaration :

```
static void <T> reverse(List<T> list)
```

Types "bruts"

- La machine virtuelle fonctionne avec des types "bruts" et non pas des types génériques
- Le type "brut" d'un type générique et obtenu en effacant les types paramètrés
- Par exemple, la classe générique `Pair<T, S>` est transformée en la classe "brute" suivante :

Types “bruts” /2

```
public class Pair
{
   public Pair(Object firstElement, Object secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public Object getFirst() { return first; }
   public Object getSecond() { return second; }

   private Object first;
   private Object second;
}
```

Types “bruts” /3

- Le même principe est appliqué aux méthodes génériques :

```
public static Comparable min(Comparable[] a)
{
   Comparable smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

- Connaître cette notion de type “brut” peut vous permettre de comprendre certaines limitations de la généricité en Java
- Par exemple pourquoi on ne peut remplacer un type paramètré par un type primitif

Conclusion